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

FELIX GEMEINHARDT, Johannes Kepler University Linz, Institute of Business Informatics - Software Engineering, CDL-MINT, Austria
ANTONIO GARMENDIA, Johannes Kepler University Linz, Institute of Business Informatics - Software Engineering, Austria
MANUEL WIMMER, Johannes Kepler University Linz, Institute of 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 needs to understand the 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 of quantum circuit design not only reduces the development effort 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 to advocate a higher-level quantum algorithm design, together with automated code generation for the circuit execution. To demonstrate the benefits of the proposed approach, coined *Composition-Based Quantum Circuit Designer*, we applied it for realizing the Quantum Counting algorithm and the Quantum Approximate Optimization Algorithm. Our evaluation results show 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. In particular, the proposed approach shows constant scaling when increasing the size of the investigated quantum circuits and a lower change criticality when evolving existing quantum circuits.

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:
Felix Gemeinhardt, Antonio Garmendia, Manuel Wimmer, and Robert Wille. 2018. A Model-Driven Framework for Composition-Based Quantum Circuit Design. *ACM Trans. Quantum Comput.* 37, 4, Article 111 (August 2018), 36 pages. https://doi.org/10.1145/1122445.1122456

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

2643-6817/2018/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

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 [10, 22, 45].

Computations performed on a quantum computer are implemented with operations in terms of quantum gates, in analogy to classical gates for conventional computation [15]. 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 as the most commonly used realistic model to run quantum programs [62].

A universal fault-tolerant quantum computer would require millions of qubits of highest quality [27]. 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 [70]. Hybrid quantum-classical algorithms, called *Variational Quantum Algorithms* (VQAs), have been proposed to cope with the limitations given in the NISQ era [10], where the parameters of the quantum circuit are optimized with classical means of computation. Therefore, quantum algorithms are considered and developed which utilise either perfect or noisy qubits [8].

Nowadays, quantum programming languages, such as 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 [4]. However, code is usually written at the qubit level and requires to understand basic fundamental concepts of quantum physics, such as 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 to raise the level of abstraction by using higher-level quantum operations (e.g., *Quantum Fourier Transform* (QFT) [62]) which occur frequently in quantum algorithms. One example is the *Quantum Phase Estimation* (QPE) [62], 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 gate, i.e., it is composed of lower-level quantum gates. A controlled version of the U-gate 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 is another example of a higher-level, composite operation which is applied to the circuit, before the quantum state is measured.

Therefore, utilizing high-level design concepts and composite operations 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. The current lack of abstraction and reuse of

[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

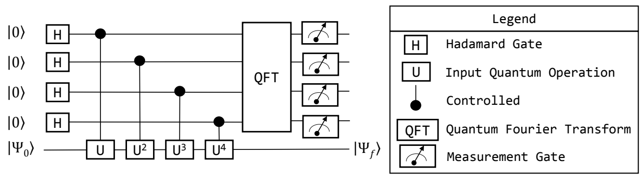


Fig. 1. Quantum circuit for QPE based on [6]

code based on components has been pointed out in [73, 74] as major problems of current tools and platforms.

Within the 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. 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) [12], and *Software Language Engineering* (SLE) [18] and transfer it to QSE. We go beyond recent proposals of applying means of MDE to quantum circuit design (e.g., [2]) and 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 modeling 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 such as 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 clearly separates the semantics concerning the quantum circuit itself and the specific quantum operations, which allows to add novel quantum operations without the need of conducting changes on the language level. Therefore, we present two declarative modeling languages to account for this separation of concerns. Note, that the proposed framework is by design modular concerning the utilized backends, the target quantum programming language for lower-level code generation, and the editor that is built on top as a front-end. Therefore, it allows for future extensions of the rapidly evolving field of QC.

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. We find that the proposed approach allows circuit design with substantially reduced development effort. Furthermore, it shows constant scaling when increasing the size of the investigated quantum circuits and a reduced change criticality when evolving existing programs to larger sizes.

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 and evaluate the proposed approach using the realization of the Quantum Counting algorithm [62] and the *Quantum Approximate*

Optimization Algorithm (QAOA) [26]. 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 Braket). Furthermore, vendor-agnostic tools have emerged for higher portability (e.g., XACC [58], Project Q [77], QuantumPath [44]) with an steadily increasing number of upcoming tools.

Current quantum software technologies have been classified in [73], ranging from quantum programming languages to software optimizers and quantum error correction tools. Based on the presented quantum software layers [73], our proposed approach addresses the quantum application layer. The latter is characterized by (i) the combination of the used design tool as well as (ii) the underlying programming language. The most useful quantum simulators and design tools have been found to be the ones of major quantum computing manufacturers, such as IBM, along with some exceptions such as the web-based Quirk[5] quantum circuit editor [73]. The quantum programming languages can be classified as *imperative*, *functional*, and *other* (e.g., circuit-based and declarative) programming languages. Our approach represents a circuit design tool providing a declarative modelling language, which supports domain-specific concepts for developing quantum circuits. This language is categorised as an external language because we create a dedicated custom syntax and an independent parser [28] for the domain-specific concepts. Instead of embedding these domain-specific concepts in an already available general-purpose language (as done by Qiskit, Cirq, etc.), external languages are designed to solely provide domain-specific concepts to shield the user from the complexity of general-purpose programming languages. Usually, this is accomplished by providing powerful concepts and an intuitive syntax suitable for domain experts. The combination of external declarative languages with design environments and simulators allows for a visual design of quantum circuits. The latter has been highlighted in the Talavera Manifesto [68, 74] as a requirement for agnostic quantum software development, together with hiding implementation and platform details from the user. This is also the goal of low-code development platforms, that aim to reduce the development effort and the amount of code required to implement the system [11, 24], and thus, allow the inclusion of domain experts who are not highly skilled in programming. Thus, our approach focuses on the quantum application layer by proposing a combination of an external declarative language with a quantum circuit design environment and, thereby, allows for visual representation of circuits. This class of quantum circuit editors is discussed in detail in the following.

The IBM Quantum Composer[6] 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 [56], and the QPS quantum circuit modeler which supports circuit execution on multiple platforms[7]. The Quirk[8] 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[9] 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 quantum circuits (`Automation`),

[5]https://algassert.com/quirk
[6]https://quantum-computing.ibm.com/composer/files/new
[7]https://quantum-circuit.com/docs
[8]https://algassert.com/quirk
[9]https://quantil.readthedocs.io/en/latest/user-guide/circuit-transformer

- grouping quantum gates to composite gates (`Grouping`), and
- defining abstract composite gates which require further configuration before execution (`Configuration`).

The latter feature allows the definition of reusable quantum software components. For example, considering Figure 1, the `QFT` represents a composite gate the implementation of which can be defined in a flexible manner by keeping the number of qubits as an unspecified parameter. Thus, the abstract composite gate shows a higher level of reusability and has to be configured by providing the number of qubits before its application and execution.

Table 1. Supported features of current graphical editors (yes (✓), no (✗))

Graphical Editor	`Automation`	`Grouping`	`Configuration`
IBM Quantum Composer [06.11.2023]	✓	✓	✗
QI editor [v1.0]	✓	✗	✗
QPS modeler [0.9.53]	✓	✗	✗
Quirk [v2.3]	✗	✓	✗
QuAntiL [v1.0.1]	✓	✗	✗

In Table 1, `Automation` has been evaluated as ✓if at least one code generator is provided. The support of pure static definitions (`Grouping`) would be sufficient for a certain fully specified oracle but not, e.g., for the general QFT. In contrast, the support of abstract composite gates (`Configuration`) has to comprise the possibility of defining such gates in a manner which is not specific to a certain application in a quantum circuit, but rather offers further configuration possibilities (e.g., the unspecified number of qubits) to raise the level of reusability. Note, that the number of qubits represents just one example for the configuration of abstract composite gates. The latter may also comprise, a.o., (potentially many) control qubits, building the inverse of an operation, or the definition of iterative patterns (e.g., the controlled U-gates in Figure 1). Table 1 illustrates that the majority of available graphical editors do not support composite gate definitions, particularly for configurable 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.

Above, we have argued for the class of graphical editors to be the design tools for comparison of our proposed approach and figured out the IBM Quantum Composer to be the most advanced tool when it comes to abstraction and automation features.

For completeness, we want to note that there are even higher-level quantum software development platforms available which allow to design quantum software above the circuit level, i.e., design and run whole quantum applications and workflows in a unique environment [73]. These platforms usually integrate classical as well as quantum resources and consider not only gate-based quantum computing but also other models like quantum annealing. Thus, these platforms utilize and integrate existing design tools for the respective quantum hardware, i.e., are located on top of the layer that is addressed with the proposed approach. Related challenges for such platforms are the integration of these heterogeneous approaches [29] as well as the lack of quality quantum software development [79]. Prominent examples for these higher-level platforms are Orchestra from Zapata[10], the Quantum Algorithm Design (QAD) platform from Classiq[11], and the QPath software tool[12] [44]. Zapata Orchestra focuses on the definition and execution of hybrid quantum-classical

[10]https://www.zapatacomputing.com/orquestra-platform
[11]https://www.classiq.io
[12]https://www.quantumpath.es

workflows. For the composition of the latter, quantum-specific (e.g., Qiskit) as well as classical domain-specific open-source libraries (e.g., Psi4[13], Tensorflow[14], Quantlib[15]) are integrated into a unified environment. The Quantum Algorithm Design (QAD) platform of Classiq focuses on the automatic synthesis of complete quantum circuits from high-level textual inputs. Thus, whole quantum algorithms are created automatically from ideas without coding at the quantum gate level. The QPath platform implements a complete quantum software lifecycle. It integrates tools for quantum and classical software development for gate-based and quantum annealing based approaches. For example, the Quirk software tool (cf. Table 1) represents the integrated circuit editor to allow for the graphical design of quantum circuits.

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. [66] who proposed a *Unified Modeling Language (UML)* [63] extension to allow for the addition of basic quantum elements. Furthermore, the use of UML-profiles has been suggested by Pérez-Castillo et al. [64]. 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 [33]. A model-based approach to quantum circuit design comprising code generation features and a meta-model for modeling quantum circuits has been proposed in [2]. However, in contrast to our work the proposed method does neither provide abstraction in terms of gate composition, nor does it allow to add novel quantum operations without changing the underlying meta-model. Finally, the role of MDE for software modernization towards quantum software has been investigated [47, 65], and it has also been discussed and envisioned in the context of Model-Driven Architecture [59]. Finally, we would like to mention reviews on quantum programming frameworks (e.g., [30, 55, 76]) and quantum software engineering in general [85].

Overall, there exists a variety of graphical as well as non-graphical solutions for the manipulation of quantum circuits where currently only the latter kind promotes high-level design features and automation. Furthermore, first attempts have been made to apply 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 abstract composite operations, and (ii) provides automated code generation. Besides that, the proposed approach also comes with a clear separation between the quantum circuit syntax and the definitions of the quantum operations which allows to build and use customized libraries.

3 COMPOSITION-BASED QUANTUM CIRCUIT DESIGNER

In the following, we provide a condensed overview of the proposed approach for composition-based quantum circuit design (Section 3.1) and discuss our long-term vision for abstract quantum circuit design based on software libraries of reusable components (Section 3.2).

3.1 Overview

This section describes the proposed approach to develop modelling environments for quantum circuits. Figure 2 provides a corresponding overview on the basic setting. 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

[13]https://psicode.org/
[14]https://www.tensorflow.org/
[15]https://www.quantlib.org/

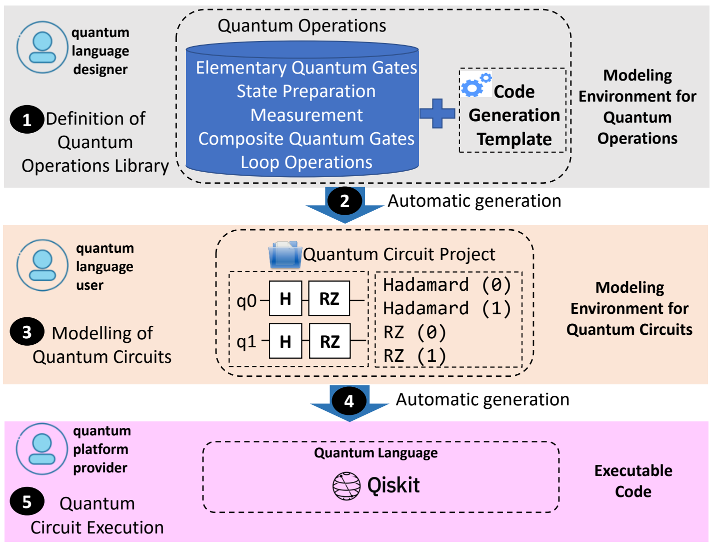


Fig. 2. Overview of the proposed process to build custom quantum operations, model quantum circuits, and generate executable code

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). Note, that the target languages for circuit execution may come with various levels of abstraction. For example, Qiskit shows abstraction levels ranging from individual pulses to whole quantum algorithms, i.e., even above the circuit level. However, our approach builds on the circuit-level abstractions of these target languages. Thus it represents a declarative manner to model quantum circuits and provides automated code generation to the target languages for execution of these circuits. Thereby, it harnesses concepts of the low-code paradigm to modelling quantum circuits, which shields the user from the complexity of general-purpose programming languages. This feature is in contrast to most of the current quantum programming languages, which as internal languages, require knowledge of general-purpose programming languages used as host language (e.g., Python). At the same time, quantum circuit designers need scalable modelling concepts to build realistic circuits. Thus, our approach addresses different target users, i.e., domain experts who are not necessarily also highly skilled in programming. However, our tool allows automated code generation to general-purpose programming languages, which can be used as the basis for further development by expert programmers.

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 in the upcoming sections.

3.2 Perspectives and potential impact

With the proposed composition based circuit designer, we lay the foundation for our envisioned quantum software reuse system. Alternative reuse schemes are either unsystematic copying, pasting & modifying, as well as systematic reuse approaches, such as universal modules, templates, libraries, or design patterns [61].

For classical software systems, libraries are advocated for abstracting, selecting, specializing and integrating software artifacts [53]. Furthermore, first attempts already exist for vendor-specific quantum software libraries (e.g., Qiskit Circuit Library[16]), and software libraries have been demonstrated to be highly successful for managing complexity of reuse intensive software for automated production systems [80, 81]. Similarly as for quantum software, the latter require flexible, reusable control software that can be easily maintained and evolved [81].

As with our proposed approach, to enhance scalability to large library systems, classical library-based software reuse systems usually foresee a role separation regarding the definition, creation, and use of software libraries [61, 80]. Furthermore, techniques from MDE are extensively used due to the required high degree of abstraction [81].

Thus, we envision purpose-specific library-based quantum software reuse systems which account for all related issues regarding, e.g., system structure, management, categorization, component selection, and library release. For a reuse system to be effective, the intellectual effort for the user to select, specify, and implement a certain software component has to be minimal [53]. This requirement is particularly fulfilled for application generators [53], i.e., the language category of our proposed approach, since domain-specific abstractions and automated code generation for executing the programs are provided and, thus, many of the required design and implementation steps are eliminated.

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 we show how quantum circuits can be represented using the proposed framework with a simple example (Section 4.4). Finally, we discuss extension aspects of the proposed approach (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 [62], as well as identified patterns in quantum computing[17].

[16]https://qiskit.org/documentation/apidoc/circuit_library.html

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

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 [14]. However, the quantum register object does not have to be created explicitly by the user. Our approach would automatically create a default register transparently by just specifying the number of qubits. 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*. Thus, the concept of *Layers* ensures that gates are not simultaneously applied to the same qubit, i.e., within one layer, each qubit can only be addressed once as a target or control qubit. 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). Moreover, it can be observed that all containment references that allow objects of type *Selector*, except *classicSelector*, may contain many selector objects, i.e., *RangeSelector* and *ElementSelector*. In this way, it is possible to address different selector objects, because it could be that several qubits are not in range or not even in the same register. The reference to the abstract *Register* class allows to address different *QuantumRegisters*.

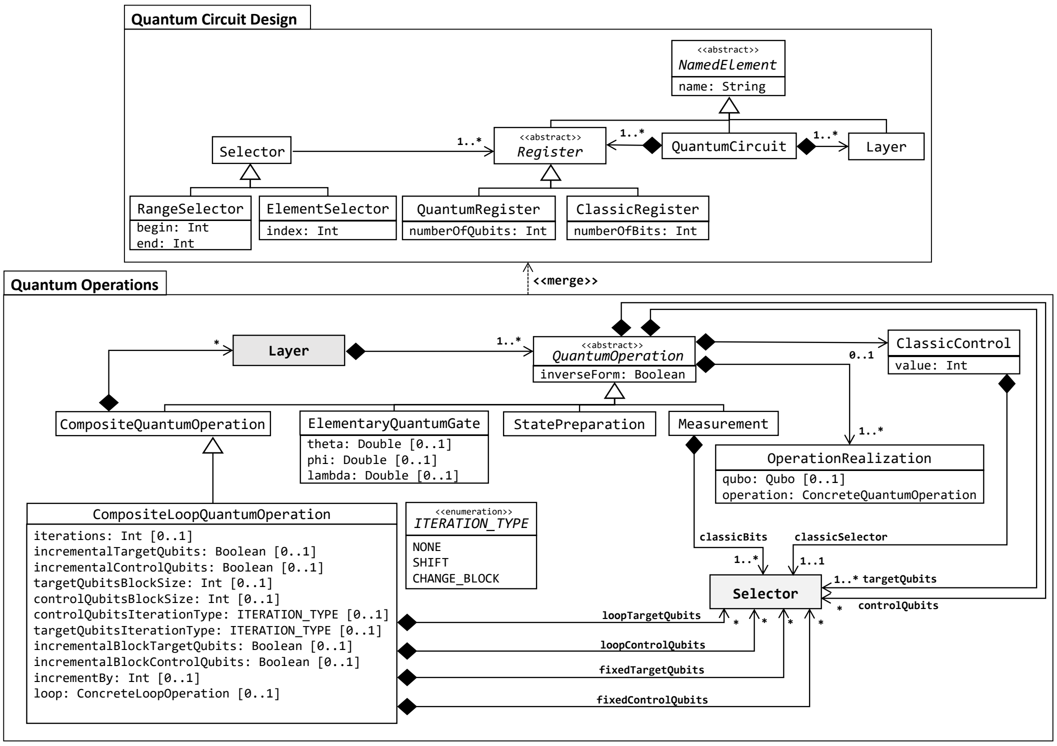


Fig. 3. Meta-model for quantum circuit design

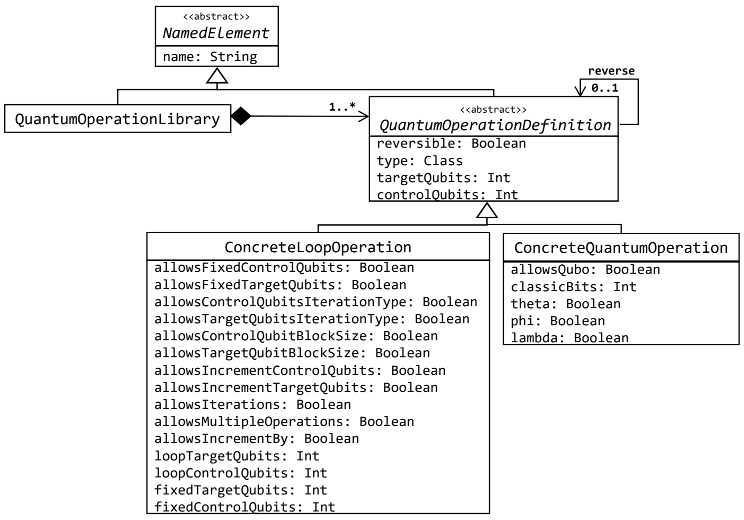
Regarding the *QuantumOperation*, stating one *controlQubit* means that the respective gate is converted to its single-controlled version, whereas a size of *controlQubits*, which is greater than 1, results in a multi-controlled gate. Furthermore, this class takes the *inverseForm* attribute, which causes 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*. Furthermore, the relation to the *OperationRealization* 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 4.5.

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 [62]. 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 [57].

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*.



<<abstract>>
NamedElement
name: String
reverse
0..1
QuantumOperationLibrary
1..*
<<abstract>>
QuantumOperationDefinition
reversible: Boolean
type: Class
targetQubits: Int
controlQubits: Int
ConcreteLoopOperation
allowsFixedControlQubits: Boolean
allowsFixedTargetQubits: Boolean
allowsControlQubitsIterationType: Boolean
allowsTargetQubitsIterationType: Boolean
allowsControlQubitBlockSize: Boolean
allowsTargetQubitBlockSize: Boolean
allowsIncrementControlQubits: Boolean
allowsIncrementTargetQubits: Boolean
allowsIterations: Boolean
allowsMultipleOperations: Boolean
allowsIncrementBy: Boolean
loopTargetQubits: Int
loopControlQubits: Int
fixedTargetQubits: Int
fixedControlQubits: Int
ConcreteQuantumOperation
allowsQubo: Boolean
classicBits: Int
theta: Boolean
phi: Boolean
lambda: Boolean

Fig. 4. Meta-model for the quantum library

The *CompositeQuantumOperation* is a composite gate to aggregate arbitrary elements in its composition. This gate may consist of multiple *Layers*, representing its decomposed form. These

Layers in turn comprise *QuantumOperation*s, which closes the cycle. Note that to avoid infinite loops, a constraint is defined that an operation cannot admit a layer that contains an operation equal 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 [10, 26, 67], Quantum Arithmetics [49], Shor's Algorithm [7], or QPE and QFT [62]. 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 [14]. 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 for the whole *CompositeLoopQuantumOperation* itself.

Note that this meta-model does not contain 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 [48], multi-level modeling [54], 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 [31].

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 language 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 Subsection 4.3. Furthermore, the number of *loopTargetQuibts*, *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 *QuantumOperationDefintion*. 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 $x \epsilon \mathbb{N}_0$, 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 [10, 26, 67], or QPE and QFT [10]. 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 in detail in Appendix A. The set of minimum required parameters has been distilled by investigating various

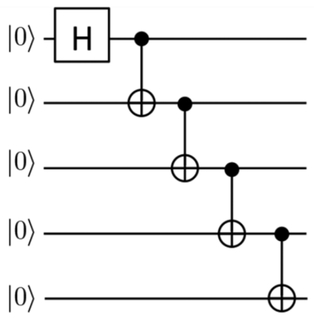


Fig. 5. Quantum Circuit for the generation of a 5-qubit GHZ-state (taken from [20])

loop patterns (e.g., from [49], [62], the PlanQK Pattern Atlas[18], the Qiskit Textbook[19]). Instead of going into the details of those parameters, the use of the *GeneralLoop* is subsequently discussed based on an illustrative example.

4.4 Representation of quantum circuits

The chosen example to demonstrate the application of the proposed approach is the standard circuit to generate the GHZ-state [38]. This fully entangled state is important, e.g., for distributed quantum information processing and quantum communication [23]. 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)]). Basically, within the *GeneralLoop* arbitrary blocks of control and target qubits can be iteratively applied according to some defined pattern with potentially varying sizes of the qubit blocks and potentially non-varying qubit indices between iterations.

The required parameters for the loop of the present example result from its definition as a *Loop* along with the *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*). Whereas the latter parameters are equally required for all loop operations, the following are specifically used in the *GeneralLoop*. 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 of the four *iterations*.

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* (cf.

[18]https://patterns.platform.planqk.de/pattern-languages/af7780d5-1f97-4536-8da7-4194b093ab1d
[19]https://qiskit.org/textbook/preface.html

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 {
       Loop {
            loop GeneralLoop
            targetQubits [(0-4)]
            operations (Pauli-X)
            loopTargetQubits [(1-4)]
            loopControlQubits [(0-3)]
            incrementTargetQubits
            incrementControlQubits
            targetQubitsBlockSize 1 -- default: 1
            controlQubitsBlockSize 1 -- default: 1
            targetQubitsIterationType SHIFT
            controlQubitsIterationType SHIFT
            iterations 4 -- default: maximum possible number of iterations
        }
    }
}
```

Listing 1). Thus, for example, in the *GeneralLoop* the number of *iterations* of the loop would default to the maximum possible number in a circuit, i.e., leveraging all available qubits. Overall, the *GeneralLoop* may be specified by 11 parameters, seven of which have been discussed in the presented example. The remaining four parameters would allow to specify (i) the change of target and control qubit blocks and (ii) target and control qubits indices which should vary between iterations (cf. Appendix A). Thus, the *GeneralLoop* with its 11 parameters is notably more complex to specify than, e.g., the *Power2Loop* (two parameters). However, once the parameters are mastered, it is expressive enough to be applied to the majority of existing iterative patterns in quantum circuits as they are encountered, e.g., in [49], [62], the PlanQK Pattern Atlas[20], the Qiskit Textbook[21].

4.5 Extension aspects

Extensibility by separation of syntax: As outlined in Section 3.2, the aim of this paper is to provide a model-driven approach as the basis for a potentially large library system for quantum software reuse. Particularly in this context, only modeling languages that encourage modularity can be considered for organizing reusable elements in large libraries [37]. Thus, in other domains that have to deal with extensive complexity and large libraries of reuse-intensive software, such as in automated production systems, the roles of library developers and application developers are usually separated [61, 80]. Usually, the application developer can not only reuse provided software components, but is also able to define and store new components according to the rules defined by the library extension mechanism. This enables scaling to thousands of software components stored within well-managed library systems. To realize this kind of scalability, it is required to customize existing and define new quantum operations and store them in the library system without the need to change the underlying meta-model. We realize this by separating the quantum circuit syntax

[20]https://patterns.platform.planqk.de/pattern-languages/af7780d5-1f97-4536-8da7-4194b093ab1d
[21]https://qiskit.org/textbook/preface.html

Listing 2. Addition of the Hadamard gate to CoQuaDe

```
1 {
2   "name": "Hadamard",
3   "reversible": true,
4   "type": "ElementaryQuantumGate",
5   "allowsQubo": false,
6   "theta": false,
7   "phi": false,
8   "lambda": false
9 }
```

from the concrete quantum operation syntax (cf. Section 4.1 and 4.2). Thus, our language does not define any specific quantum operation as, for example, the approach proposed in [2].

An example of gate addition may be seen in Listing 2. In this example, we add the features of the *Hadamard* gate conforming to the language defined in Figure 4. Likewise, elementary gates like Pauli-X, and Pauli-Y may be defined as well. Notice in line 4, that the type is *ElementaryQuantum-Gate*, but besides this type, our language supports the addition of *CompositeQuantumOperation*, *CompositeLoopQuantumOperation*, *StatePreparation* and *Measurement* (see Figure 3). Thus, in our approach novel quantum operations can be easily added to the quantum software library without touching the underlying language. Note, that we provide a detailed description regarding the technical implementation of the syntax separation and extension by novel quantum operators in our repository[22].

Language 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 [51], but rather be defined by problem-specific information like, e.g., the cost function in the case of QAOA [26]. 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. Because this matrix is symmetric, the framework requires an upper triangle matrix as input to avoid redundant information and input errors.

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 [10], 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.

Feature Summary: Overall, we propose a framework for quantum circuit modeling that allows for building dedicated libraries for reusable quantum software components as envisioned in Section 3.2. We go beyond mere aggregation of lower-level target code since 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.

[22]https://github.com/jku-win-se/composition-quantum-circuit

Furthermore, the proposed approach introduces the loop concept for implementing iterative patterns of quantum circuits and integrates automated code generation features for lower-level target quantum programming languages. Our declarative approach and the language-agnostic definitions of quantum circuits and quantum operations at a model level additionally allow for graphical user interfaces and error-checking at a higher design level. The separation of the operator and circuit syntax allows for sufficient extensibility of libraries to account for novel quantum subroutines, thus, laying the fundamentals regarding our envisioned system for quantum software component reuse.

5 TOOL SUPPORT

We implemented the proposed approach, called CoQuaDe, atop of the *Eclipse Modeling Framework* (EMF) [78] 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 [9], 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*[23] 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 uses 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 agnostic concerning the lower-level quantum programming language. However, for demonstration purposes, we rely on the Python-based Qiskit SDK [3] as described bellow. The tool architecture as well as an example for the procedure of the automated code generation to the Qiskit target language is provided in the the repository.

6 DEMONSTRATION AND EVALUATION

In the following, we will demonstrate and 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 compared to plain grouping of quantum gates?*

To assess *RQ1*, we apply the approach to the QPE algorithm, which is a prominent representative of quantum algorithms for fault-tolerant quantum computation [75], and a central building block of many other quantum algorithms (e.g., HHL algorithm [42], Shor's algorithm [7]). Specifically, we

[23]https://www.eclipse.org/

will treat the Quantum Counting algorithm [62] (cf. Subsection 6.1), which represents an instance of QPE. *RQ2* will be assessed by implementing the QAOA algorithm [26] 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 with the IBM Quantum Composer (*RQ3*) regarding (i) the number of required actions, (ii) the Halstead software metrics [41], and a metric to measure the criticality of evolving programs. 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[24]. However, the IBM Composer only supports plain grouping of gates for composite gate definitions. Thus, by comparing to the IBM Composer we assess the impact of those features of our proposed approach, which go beyond simple gate grouping (cf. Section 4.5).

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 potential reusability options would be lost. Therefore, we will justify the chosen level of composition for a fair comparison in Section 6.3.

6.1 Demonstration Case: 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 [62]. Being an application of the QPE procedure [62], 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 [62]. The inverse QFT is finally used to translate this information to the computational basis before the state is being measured.

6.1.2 Implementation of the Quantum Circuit. The described demonstration case is taken from the IBM Qiskit Textbook[25]. 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.

[24]https://quantum-computing.ibm.com/composer/docs/iqx/new
[25]https://qiskit.org/textbook/ch-algorithms/quantum-counting.html

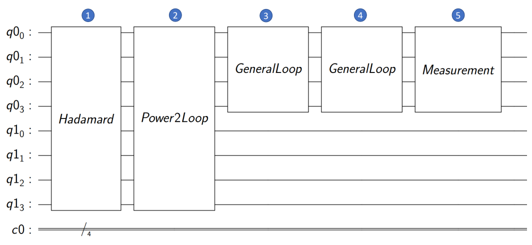


Fig. 6. High level view of generated quantum circuit for Quantum Counting (*Alternative 1*); visualization conducted with [3]

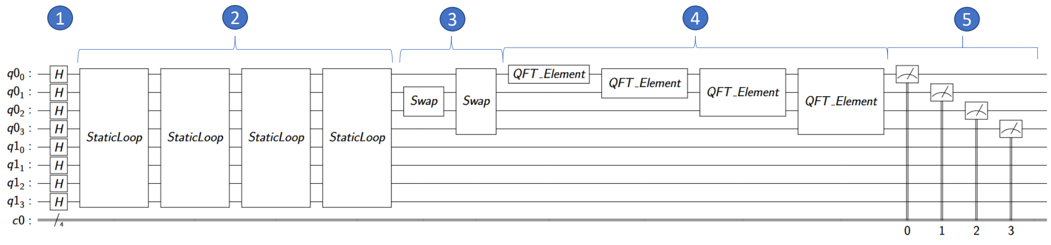


Fig. 7. First order decomposition of generated quantum circuit for Quantum Counting (*Alternative 1*); visualization conducted with [3]

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* (Appendix C: Figure 12).

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 3. 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 composite gate are possible: first, as an object which just utilizes concepts and methods from the Qiskit SDK [3] 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 *CompositeQuantumOperation* being specified, the rotation part of the inverse QFT is generated with the attributes for the *GeneralLoop* as presented in Listing 4 (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.

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

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

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

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

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 [13, 19] 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[26].

6.2 Demonstration Case: QAOA

The application of VQAs has been shown useful for exploiting the potential of current NISQ devices [10]. Such algorithms take a certain parameterized quantum circuit, called ansatz, where the

[26]https://research.ibm.com/blog/ibm-quantum-roadmap-2025

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 [26], 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 [36], 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., [84]) or the mixing unitary (e.g., [39]). 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 Textbook[27]. 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, and the implemented cost unitary is provided in Appendix C (Figure 13). The implementation of the quantum circuit for QAOA is presented in Listing 5.

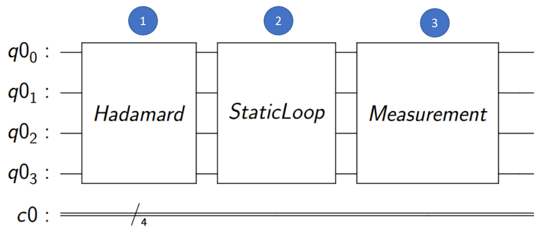


Fig. 8. High level view of generated quantum circuit for QAOA (*Alternative 2*); visualization conducted with [3]

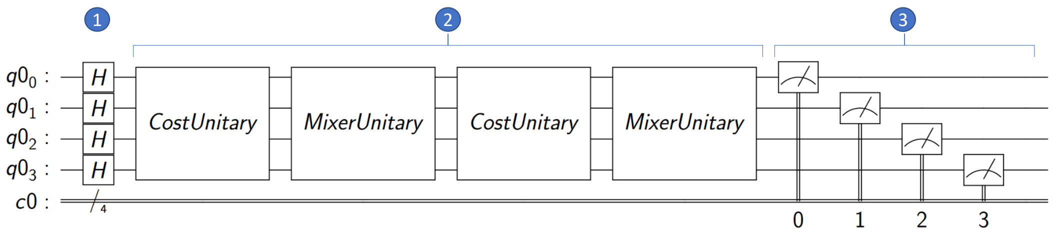


Fig. 9. First order decomposition of generated quantum circuit for QAOA (*Alternative 2*); visualization conducted with [3]

[27]https://qiskit.org/textbook/ch-applications/qaoa.html

Listing 5. Implementation of QAOA quantum circuit with CoQuaDe

```
QuantumCircuit QAOA {
    QuantumRegister qr {
        NumberOfQubits 4
    }
    ClassicRegister cr {
        NumberOfBits 4
    }
    Layer L1 {
        ElementaryQuantumGate {
            operation Hadamard
            targetQubits [(0-3)]
        }
    }
    Layer L2 {
        Loop {
            iterations 2
            operations (CostUnitary(SampleMatrix),MixerUnitaryQAOA)
            targetQubits [(0-3)]
            loop StaticLoop
            loopTargetQubits [(0-3)]
        }
    }
    Layer L3 {
        Measurement {
            operation MeasurementCompBasis
            targetQubits [(0-3)]
            classicBits [(0-3)]
        }
    }
}
```

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 5: *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 4.5. 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 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 5. Finally, the measurement is conducted by a single *Measurement* gate (Listing 5: *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 Comparison Study

6.3.1 Experimental Setup. We compare the proposed approach to the IBM Quantum Composer. The latter has been identified in Section 2 as the most powerful framework in the peer group of our proposed approach, i.e., quantum circuit designers based on external declarative programming languages. The IBM Composer has been identified as the only quantum circuit designer, which supports code generation as well as gate composition features. We evaluate the development effort to design quantum circuits for QPE and QAOA. The comparison of textual and graphical declarative languages requires the application of according metrics. Thus, we evaluate the approaches in terms of (i) the number of actions taken by the user, (ii) the Halstead software metrics [41], and (iii) the MICOSE4aPS metric [80]. Therefore, we evaluate the effort of creating novel quantum circuits using well-known quantitative software quality metrics (Halstead) and a newly defined metric (number of actions), as well as measures to assess the criticality of evolving existing circuits (MICOSE4aPS). Note, that the chosen metrics allow for a direct comparison of text-based (CoQuaDe) and graphical (IBM Composer) editors that are built on top of an external declarative programming approach. Whereas a basic explanation of these metrics and their application is provided in the following, we refer the interested reader to Appendix B for a detailed description.

Regarding the number of actions, we interpret the declaration of a quantum circuit as an attributed typed graph [43]. Based on this representation, the required number of actions is defined as the sum of (i) created objects (nodes of the graph), (ii) user-specified non-default attributes (attributes of the nodes), and (iii) links between objects (edges of the graph). The Halstead metrics [41] represent a family of common software metrics to assess the program quality by quantitative means and have also been proposed by [86] to be applied in the context of quantum programs. These metrics are based on the assumption, that each program can be viewed as a collection of tokens where each token may be either an operator or an operand. Based on the given unique operators and operands as well as overall operators and operands of a given program, the following metrics can be computed: vocabulary, length, volume, difficulty of understanding and writing the program, implementation effort, and implementation time. The MICOSE4aPS metric [80] has been proposed to measure the modularity, i.e., the ease of software reuse, specifically in the context of automated production systems. Thus, in contrast to the number of actions and the Halstead metrics, which are based on a static analysis of a single program, the MICOSE4aPS metric assesses the effort and criticality of changing and evolving existing programs related to software reuse. The MICOSE4aPS metric has been specifically designed to remove the insufficiency of other metrics, such as *Source Lines Of Code* (SLOC), for deriving the criticality of implementation changes [80]. Furthermore, we perceive the application of this metric particularly appropriate since we envision a quantum software reuse system based on software libraries similar to the available systems for the classical reuse-intensive software of automated production systems (cf. Section 3.2).

Having defined the metrics to be applied, sound composition levels of the circuits have to be chosen to allow for a fair comparison between CoQuaDe and the IBM Quantum Composer. 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. 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 [42], Shor's algorithm [7], and QPE [62]. 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.

Table 2. Metrics regarding number of actions regarding the QPE and QAOA use case for different number of qubits (Alt. 2 each); CoQuaDe / IBM Quantum Composer

	QPE8	**QPE16**	**QPE32**	**QAOA4**	**QAOA16**	**QAOA56**
objects	19/43	19/323	19/65731	14/70	14/826	14/9586
links	5/107	5/2391	5/1114415	6/100	6/1360	6/15960
non-default parameters	32/13	32/35	32/127	21/37	21/313	21/3313
actions	56/163	56/2749	56/1180273	41/207	41/2499	41/28859

We conducted analogously with elementary quantum gates that are not supported by the IBM Quantum Composer to avoid an artificially high number of actions in its evaluation. We want to highlight at this point, that the creation of controlled composite 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. We evaluate QPE circuits of different sizes to assess the scaling properties of the two approaches, i.e., QPE circuits of 8, 16, and 32 qubits are measured. Note that for the evaluation, we view the Grover unitary as an opaque gate with an arbitrary number of qubits. This is possible because we do not consider the specific implementation of the Grover unitary in the evaluation. Thus, the QPE circuit is scalable without further redefinitions.

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 4.5, 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., [5, 40, 71, 72, 84, 87]). 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 [25, 35]). 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. We evaluate the QAOA circuits for 4, 16, and 56 qubits. Note, that different circuit sizes solely require a redefinition of the QUBO-input. Since our evaluation metrics are invariant regarding the specific values of the QUBO input, we use random inputs of the respective circuit sizes.

6.3.2 Results. The results of the evaluation are summarized in Tables 2-4.

Table 2 shows the metrics related to the number of actions when using CoQuaDe and the IBM Quantum Composer.

For the small QPE8, i.e., the QPE circuit with 8 qubits, except for the number of non-default parameters, all number of actions components are smaller when using CoQuaDe compared to the IBM Composer. Notably, the gap increases significantly when scaling the number of qubits to 16 and 32. The lowest increase is observed for the number of non-default parameters, whereas the number of links goes up the most when scaling to larger circuits and using the IBM Composer. Regarding the number of actions, the respective figure remains constant when scaling to larger circuits for CoQuaDe and is increased 17-fold (QPE8-QPE16) and 429-fold (QPE16-QPE32) for the IBM Composer.

For the QAOA use case, similar trends can be observed. Again, the number of actions remains constant for CoQuaDe when evolving to larger circuits and increases 12-fold (QAOA4-QAOA16, QAOA16-QAOA56) for the IBM Composer. Note, that the latter increase is less pronounced for the

Table 3. Halstead metrics regarding the QPE and QAOA use case for different number of qubits (Alt. 2 each); CoQuaDe / IBM Quantum Composer

	QPE8	**QPE16**	**QPE32**	**QAOA4**	**QAOA16**	**QAOA56**
unique operands	25/16	25/28	25/52	22/34	22/298	22/3258
overall operands	44/120	44/2426	44/1114542	32/137	32/1673	32/19273
unique operators	13/8	13/8	13/8	11/13	11/13	11/13
overall operators	26/43	26/323	26/65731	19/72	19/828	19/9588
vocabulary	38/24	38/36	38/60	33/47	33/311	33/3271
length	70/163	70/2749	70/1180273	30/209	30/2501	30/28861
volume	367/747	367/14212	367/6971743	257/1161	257/20710	257/336967
difficulty	11/30	11/347	11/85734	8/26	8/36	8/38
effort	4203/22420	4203/4925516	4203/ $5.977*10^{11}$	2058/30406	2058/755749	2058/ $1.295*10^{8}$
time	233/1246	233/273639	233/ $3.206*10^{10}$	114/1689	114/41986	114/719824

Table 4. Modularity computed by MICOSE4aPS metric for circuit changes regarding number of qubits

	CoQuaDe	**IBM Quantum Composer**
QPE8-QPE16	0.58	0.30
QPE16-QPE-32	0.58	0.11
QPE4-QPE-32	0.58	0.11
QAOA4-QAOA16	0.57	0.16
QAOA16-QAOA56	0.57	0.16
QAOA4-QAOA56	0.57	0.16

QAOA use case than for the QPE use case although the number of qubits grows exponentially for QAOA and linearly for QPE. This highlights the strong dependence of the circuit size and scaling complexity on the respective use case.

Table 3 summarizes the obtained results for the Halstead software metrics. Since we are particularly interested in the development effort, among the various Halstead metrics, we specifically focus on the difficulty and effort in the following. Note that the time is per definition directly related to the effort by $time = \frac{effort}{18}$ as outlined in Appendix B. Regarding the difficulty and effort increase between CoQuaDe and the IBM Composer for the QPE use case, we observe a strong increase in difficulty and effort when using the IBM Composer, especially when going from QPE16 to QPE32. In contrast, as with the number of actions, all Halstead metrics remain constant for CoQuaDe when scaling to larger circuit sizes.

The observed trends for the QAOA use case are similar. Table 3 shows, that the advantages of CoQuaDe are less pronounced for the difficulty, compared to the effort. When using the IBM Composer, the difficulty and effort when scaling to larger circuit sizes show an 1.4-fold and 25-fold increase, respectively. Again, with CoQuaDe, the metrics remain the same.

Finally, Table 4 shows the modularity of the programs with respect to changes related to scaling to larger circuit sizes. For the QPE use case, CoQuaDe shows modularities that are 2-5 times higher than those of the IBM Composer. For the QAOA use case, the respective increase is 3-fold. Note, that the MICOSE4aPS represents a metric that is normalized, i.e., ranges from 0 to 1.

6.3.3 Discussion. In summary, by using CoQuaDe, we were able to develop quantum circuits for QPE (*RQ1*) as well as QAOA (*RQ2*) for different alternatives.

Regarding *RQ3*, the succinctness of our approach is evaluated against plain grouping of quantum gates by comparing to the IBM Quantum Composer. The succinctness is assessed by measuring the development effort using metrics for static analyses of single programs (number of actions, Halstead software metrics), and also for the criticality and modularity regarding the changes when scaling circuits to larger instances using the MICOSE4aPS metric. We found a significant increase between CoQuaDe and the IBM Composer regarding the number of actions, the difficulty, and the effort. Furthermore, whereas using CoQuaDe comes with constant scaling due to the use of composite gates, the scaling for the IBM Composer heavily relies on the structure of the iterative patterns and composite gates given in the respective circuit. Considering the underlying quantities for the Halstead metrics (cf. first four rows in Table 3), the advantages of using CoQuaDe are particularly pronounced for the overall number of operators and operands. This indicates that the unique entities of CoQuaDe and the IBM Composer are similar, and the reduced development effort originates from a reduced number of calls of these entities. We attribute the advantages of using CoQuaDe to its abstraction features which go beyond simple gate aggregation. Relating to the features summary provided in Section 4.5, particularly the concepts of variability (i.e., the possibility of defining configurable composite gates) and the loop concept allow to significantly reduce the development effort of building quantum circuits.

Regarding modularity, CoQuaDe shows a significant increase, highlighting the reduced change criticality and effort when scaling given circuits to larger instances. Here, the `Selector` concept allows to scale to larger instances by just changing the qubit indices that make up the range for the composite gate applications.

Overall, using the CoQuaDe allows for quantum circuit design on a higher level of abstraction where we find a significantly reduced development effort, particularly for creating large quantum circuits, and a reduced change criticality with respect to evolving existing ones.

However, we have to note that the Halstead metrics represent a common relative software quality metric for comparative purposes. Thus, only the underlying base quantities, i.e., number operators and operands, have an absolute meaning and the derived quantities may only be considered for a relative comparison. Therefore, although the Halstead metrics capture the difficulty and effort of understanding and implementing a program, these metrics do not substitute a thorough qualitative user study in terms of measuring the learnability, understandability, or usability of the proposed approach. Thus, dedicated exhaustive rigorous user studies are kept as future work.

6.4 Threats to validity

The following threats to validity [82] can be identified.

Construct validity. Threats to construct validity regard the extent to which the chosen evaluation accurately measures what is supposed to be assessed. First, the chosen metrics must be appropriate to assess the succinctness regarding text-based (CoQuaDe) as well as graphical (IBM Composer) editors that rely on a declarative programming approach. The Halstead metrics and the MICOSE4aPS metric represent standard quantitative approaches to measure software quality and change criticality, respectively, and can be applied according to the above requirement. However, we additionally define the number of actions as a further suitable metric to assess the succinctness of both types of editors. Second, there are no specific definitions and implementations of the chosen metrics available in the context of quantum software engineering. Although in [86], the use of Halstead metrics and definition of quantum operators and operands is proposed, no concrete implementations are suggested. We mitigated this threat by defining and applying the chosen metrics (cf. Appendix B) as similar as possible to the classical software domain and had intense discussions with experts in classical software quality evaluation. The third threat regards the choice of the IBM Quantum Composer as the state-of-the-art tool to compare our approach. We mitigated this threat by carefully studying existing literature on the categorization of the quantum software stack as outlined in Section 2. We find the IBM Quantum Composer to be the most feasible choice for comparison since it addresses the same layer of the stack and represents a declarative approach to circuit design. Among the identified tools of this class, the IBM Composer has been found to be the most advanced in terms of provided abstraction and automation (cf. Section 2). A final threat concerns the subject selection, i.e., whether the QPE and QAOA algorithms are representative examples of non-parameterized and parameterized quantum circuits. We mitigated this risk by a preceding literature search.

Conclusion validity. Threats to conclusion validity regard the adequacy of drawn conclusions with respect to the relationships in the observed data. First, the chosen metrics represent standard quantitative approaches to measure software quality and change criticality. However, the metrics only have relative meaning and do not directly assess the usability and development effort in practical settings. Thus, it may be possible that the actual effort is different due to user interface or tool support reasons. Thus, using quantitative methods only limited statements can be provided regarding understandability, learnability, usability, maintainability and portability of the proposed approach. Therefore, we plan qualitative user studies where the latter can be assessed in a methodologically rigorous way [69]. Second, we found that, by using sophisticated composite operators going beyond simple grouping concepts, our proposed approach shows the advantages of (*i*) a reduced development effort, (*ii*) constant scaling properties when scaling up the circuits, and (*iii*) a strong dependence on the chosen use case with respect to the scaling properties of the IBM Composer. Thus, it is possible that for certain small use cases, using the IBM Composer reduces the development effort. We anticipate use case dependent break-even-points between CoQuaDe and the IBM Composer with respect to the number of qubits, which are indicated but not rigorously assessed by our evaluation.

Internal validity. Threats to internal validity regard the question, of whether the proposed approach or any other factor led to the observed outcome. In general, the evaluation of an approach by the designers of this approach brings in considerable bias. Therefore, we followed a structured approach to define, design, and conduct the evaluation study to mitigate this risk. Future work plans are to involve external researchers and practitioners for replicating the evaluation study. Furthermore, regarding the causal relationship between our introduced concepts and the obtained results, the advantages compared to the IBM Composer may not necessarily be caused by our concepts in practical settings. For example, shortcuts could have been introduced in the form of available overly specific composite gates. We mitigated these risks as explained in Section 6.3.1, e.g., by the chosen level of composition for the used quantum operations.

External validity. Threats to external validity comprise generalization issues of the conducted experiments and evaluation. First, the evaluated demonstration cases represent 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. Furthermore, the chosen circuits may represent overly simplified problems. Thus, we not only considered small circuits of the given kind but also their up-scaled versions to larger qubit numbers. Second, the analysed use cases do not make use of more complex operations of CoQuaDe, such as the *GeneralLoop*, thus, prohibiting a generalization in this regard. Third, we provide automated code generation facilities for Qiskit as the target language. Although our approach is designed in a modular manner regarding the underlying target programming language, we cannot strictly ensure feasibility for arbitrary other target languages, e.g., the Intel SDK [83], but keep corresponding investigations as future work. Finally, we studied existing literature on the categorization of the quantum software stack as outlined in Section 2 and defined the quantum circuit designers based on a declarative programming approach as the class of tools to compare our approach. However, we cannot generalize our findings to other platforms for quantum software development.

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 and plain grouping of low-level quantum gates, the proposal allows for configurable and reusable composite quantum operations and automated code generation from the built quantum circuits. This allows hiding of low-level implementation details in the design of such circuits. Furthermore, we demonstrate the feasibility and succinctness benefits of the proposed approach via the application to the Quantum Counting algorithm and QAOA. We found significantly reduced development and evolution efforts compared to using existing state-of-the-art quantum circuit designers.

Future Work. The proposed approach offers several extension possibilities. First, we will further aim to generalize our results by performing applications to more quantum circuits and provide code generation features for other quantum programming languages besides Qiskit (e.g., the Intel Quantum SDK [83]). Additionally, we will conduct a user study to evaluate qualitative aspects of our proposed approach, such as usability, learnability, and understandability. Second, as the present work represents the basis for a library system that supports the reuse of quantum software components, we will further explore possible architectures and library structures that support proper management, maintenance, categorization, component selection, and library release. Third, we will investigate how our concepts can be transferred to circuit library systems based on internal quantum programming languages (e.g., Qiskit Circuit Library[28]). Fourth, we will explore frameworks like 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 built atop of the presented quantum languages [16] and, thus, further extend the low-code features of our proposed approach. In addition, we plan to enable the import of quantum circuits and subsequent manipulation of the circuit with our framework.

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 synthesis, utilizing techniques from genetic programming and reinforcement learning. Here, the goal is to automatically create a *CompositeQuantumGate* that yields a certain target output state. Regarding the latter, we aim for an integration of existing genetic programming approaches for quantum circuit synthesis [34] and their combination with existing model-based optimization frameworks [32].

Furthermore, the circuit synthesis may comprise model-based circuit aggregation and partitioning [21], and the framework may incorporate generic as well as NISQ-specific circuit optimization procedures (e.g., [60]). Applying concepts from MDE also allows to use well-known model-based transformation tools [50] for quantum circuit transformations to different representations. The later are required, for example, when using the ZX-calculus [52] and the LOv-calculus [17]. Finally, as

[28]https://qiskit.org/documentation/apidoc/circuit_library.html

the bespoke procedures may produce errors, a subsequent verification step [46] 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. Besides this, our work has been partially sponsored by the Spanish MICINN, with projects PID2021-122270OB-I00 and TED2021-129381B-C21. This work also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (DA QC, grant agreement No. 101001318), was part of the Munich Quantum Valley, which is supported by the Bavarian state government with funds from the Hightech Agenda Bayern Plus, and has been supported by the BMWK on the basis of a decision by the German Bundestag through project QuaST.

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.

REFERENCES

[1] Shaukat Ali and Tao Yue. 2020. Modeling Quantum programs: challenges, initial results, and research directions. In *Proc. of the 1st ACM SIGSOFT Int. Workshop on Architectures and Paradigms for Engineering Quantum Soft.* 14–21.

[2] Diego Alonso, Pedro Sánchez, and Francisco Sánchez-Rubio. 2022. Engineering the development of quantum programs: Application to the Boolean satisfiability problem. *Advances in Engineering Software* 173 (2022), 103216.

[3] MD SAJID ANIS et al. 2021. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2573505

[4] Ian Arawjo, Anthony DeArmas, Michael Roberts, Shrutarshi Basu, and Tapan Parikh. 2022. Notational Programming for Notebook Environments: A Case Study with Quantum Circuits. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. 1–20.

[5] Andreas Bärtschi and Stephan Eidenbenz. 2020. Grover mixers for QAOA: Shifting complexity from mixer design to state preparation. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, 72–82.

[6] Bela Bauer and Chetan Nayak. 2014. Analyzing many-body localization with a quantum computer. *Physical Review X* 4, 4 (2014), 041021.

[7] Stephane Beauregard. 2002. Circuit for Shor's algorithm using 2n+ 3 qubits. *arXiv preprint quant-ph/0205095* (2002).

[8] Koen Bertels, Aritra Sarkar, and Imran Ashraf. 2021. Quantum computing—from NISQ to PISQ. *IEEE Micro* 41, 5 (2021), 24–32.

[9] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd.

[10] Kishor Bharti, Alba Cervera-Lierta, Thi Ha Kyaw, Tobias Haug, Sumner Alperin-Lea, Abhinav Anand, Matthias Degroote, Hermanni Heimonen, Jakob S Kottmann, Tim Menke, et al. 2021. Noisy intermediate-scale quantum (NISQ) algorithms. *arXiv preprint* (2021).

[11] Alexander C Bock and Ulrich Frank. 2021. Low-code platform. *Business & Information Systems Engineering* 63 (2021), 733–740.

[12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice, Second Edition.* Morgan & Claypool Publishers.

[13] Lukas Burgholzer and Robert Wille. 2021. Towards verification of dynamic quantum circuits. *arXiv preprint arXiv:2106.01099* (2021).

[14] Jordi Cabot and Martin Gogolla. 2012. Object Constraint Language (OCL): A Definitive Guide. In *12th Int. School on Formal Methods for the Design of Computer, Communication, and Soft. Systems (SFM).* Springer, 58–90.

[15] G Chen, DA Church, BG Englert, MS Zubairy, et al. 2003. Mathematical models of contemporary elementary quantum computing devices. *Quantum Control: Mathematical and Numerical Challenges* 33 (2003), 79–117.

[16] Federico Ciccozzi, Matthias Tichy, Hans Vangheluwe, and Danny Weyns. 2019. Blended modelling-what, why and how. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 425–430.

[17] Alexandre Clément, Nicolas Heurtel, Shane Mansfield, Simon Perdrix, and Benoît Valiron. 2022. LOv-Calculus: A Graphical Language for Linear Optical Quantum Circuits. *arXiv preprint arXiv:2204.11787* (2022).

[18] Benoît Combemale, Ralf Lämmel, and Eric Van Wyk. 2018. SLEBOK: the software language engineering body of knowledge (Dagstuhl Seminar 17342). In *Dagstuhl Reports*, Vol. 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[19] Antonio D Córcoles, Maika Takita, Ken Inoue, Scott Lekuch, Zlatko K Minev, Jerry M Chow, and Jay M Gambetta. 2021. Exploiting dynamic quantum circuits in a quantum algorithm with superconducting qubits. *Physical Review Letters* 127, 10 (2021), 100501.

[20] Diogo Cruz, Romain Fournier, Fabien Gremion, Alix Jeannerot, Kenichi Komagata, Tara Tosic, Jarla Thiesbrummel, Chun Lam Chan, Nicolas Macris, Marc-André Dupertuis, et al. 2019. Efficient quantum algorithms for GHZ and W states, and implementation on the IBM quantum computer. *Advanced Quantum Technologies* 2, 5-6 (2019), 1900015.

[21] Omid Daei, Keivan Navi, and Mariam Zomorodi-Moghadam. 2020. Optimized Quantum Circuit Partitioning. *Int. Journal of Theoretical Physics* 59, 12 (2020), 3804–3820.

[22] Franklin de Lima Marquezino, Renato Portugal, and Carlile Lavor. 2019. *A primer on quantum computing*. Springer.

[23] Ellie D'Hondt and Prakash Panangaden. 2004. The computational power of the W and GHZ states. *arXiv preprint quant-ph/0412177* (2004).

[24] Davide Di Ruscio, Dimitris Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. 2022. Low-code development and model-driven engineering: Two sides of the same coin? *Software and Systems Modeling* 21, 2 (2022), 437–446.

[25] Christoph Durr and Peter Hoyer. 1996. A quantum algorithm for finding the minimum. *arXiv preprint* (1996).

[26] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A quantum approximate optimization algorithm. *arXiv preprint* (2014).

[27] Austin G Fowler, Matteo Mariantoni, John M Martinis, and Andrew N Cleland. 2012. Surface codes: Towards practical large-scale quantum computation. *Physical Review A* 86, 3 (2012), 032324.

[28] Martin Fowler. 2010. *Domain-specific languages*. Pearson Education.

[29] J. L. Hevi G. Peterssen, M. Piattini. 2022. Practical quantum computing: Challenges of quantum software development. QuantumPath Blog. https://www.quantumpath.es/2022/01/31/practical-quantum-computing-challenges-of-quantum-software-development/

[30] Sunita Garhwal, Maryam Ghorani, and Amir Ahmad. 2021. Quantum programming language: A systematic review of research topic and top cited languages. *Archives of Computational Methods in Engineering* 28, 2 (2021), 289–310.

[31] Irene Garrigós, Manuel Wimmer, and Jose-Norberto Mazón. 2013. Weaving aspect-orientation into web modeling languages. In *Int. Conf. on Web Eng.* Springer, 117–132.

[32] F Gemeinhardt, M Eisenberg, S Klikovits, and M Wimmer. 2023. Model-Driven Optimization for Quantum Program Synthesis with MOMoT. In *5th Workshop on Artificial Intelligence and Model-driven Engineering at MODELS*. ACM/IEEE.

[33] Felix Gemeinhardt, Antonio Garmendia, and Manuel Wimmer. 2021. Towards Model-Driven Quantum Soft. Engineering. In *Second Int. Workshop on Quantum Soft. Engineering (Q-SE 2021) co-located with ICSE 2021*.

[34] Felix Günther Gemeinhardt, Stefan Klikovits, and Manuel Wimmer. 2023. Hybrid Multi-Objective Genetic Programming for Parameterized Quantum Operator Discovery. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*. 795–798.

[35] Austin Gilliam, Stefan Woerner, and Constantin Gonciulea. 2021. Grover adaptive search for constrained polynomial binary optimization. *Quantum* 5 (2021), 428.

[36] Fred Glover, Gary Kochenberger, and Yu Du. 2018. A tutorial on formulating and using QUBO models. *arXiv preprint arXiv:1811.11538* (2018).

[37] Jeff Gray and Bernhard Rumpe. 2021. Modeling in the large: model libraries. *Software and Systems Modeling* 20, 3 (2021), 591–593.

[38] Daniel M Greenberger, Michael A Horne, and Anton Zeilinger. 1989. Going beyond Bell's theorem. In *Bell's theorem, quantum theory and conceptions of the universe*. Springer, 69–72.

[39] Stuart Hadfield, Zhihui Wang, Bryan O'gorman, Eleanor G Rieffel, Davide Venturelli, and Rupak Biswas. 2019. From the quantum approximate optimization algorithm to a quantum alternating operator ansatz. *Algorithms* 12, 2 (2019), 34.

[40] Stuart Hadfield, Zhihui Wang, Bryan O'Gorman, Eleanor G Rieffel, Davide Venturelli, and Rupak Biswas. 2019. From the quantum approximate optimization algorithm to a quantum alternating operator ansatz. *Algorithms* 12, 2 (2019), 34.

[41] Maurice H Halstead. 1977. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc.

[42] Aram W Harrow, Avinatan Hassidim, and Seth Lloyd. 2009. Quantum algorithm for linear systems of equations. *Physical review letters* 103, 15 (2009), 150502.

[43] Reiko Heckel and Gabriele Taentzer. 2020. *Graph Transformation for Software Engineers*. Springer.

[44] Jose Luis Hevia, Guido Peterssen, and Mario Piattini. 2022. QuantumPath: A quantum software development platform. *Software: Practice and Experience* 52, 6 (2022), 1517–1530.

[45] Jack D Hidary. 2019. *Quantum Computing: An Applied Approach.* Springer.

[46] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A verified optimizer for quantum circuits. *Proc. of the ACM on Programming Languages* 5, POPL (2021), 1–29.

[47] Luis Jiménez-Navajas, Ricardo Pérez-Castillo, and Mario Piattini. 2021. KDM to UML Model Transformation for Quantum Soft. Modernization. In *Int. Conf. on the Quality of Information and Communications Technology*. Springer, 211–224.

[48] Ralph Johnson and Bobby Woolf. 1997. *Type Object.* Addison-Wesley, 47–65.

[49] Eric R Johnston, Nic Harrigan, and Mercedes Gimeno-Segovia. 2019. *Programming Quantum Computers: essential algorithms and code samples.* O'Reilly Media.

[50] Nafiseh Kahani, Mojtaba Bagherzadeh, James R Cordy, Juergen Dingel, and Daniel Varró. 2019. Survey and classification of model transformation tools. *Software & Systems Modeling* 18, 4 (2019), 2361–2397.

[51] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M Chow, and Jay M Gambetta. 2017. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature* 549, 7671 (2017), 242–246.

[52] Aleks Kissinger and John van de Wetering. 2019. Pyzx: Large scale automated diagrammatic reasoning. *arXiv preprint arXiv:1904.04735* (2019).

[53] Charles W Krueger. 1992. Software reuse. *ACM Computing Surveys (CSUR)* 24, 2 (1992), 131–183.

[54] Juan De Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2014. When and how to use multilevel modelling. *ACM Transactions on Soft. Eng. and Methodology (TOSEM)* 24, 2 (2014), 1–46.

[55] Ryan LaRose. 2019. Overview and comparison of gate level quantum Soft. platforms. *Quantum* 3 (2019), 130.

[56] Thorsten Last, Nodar Samkharadze, Pieter Eendebak, Richard Versluis, Xiao Xue, Amir Sammak, Delphine Brousse, Kelvin Loh, Henk Polinder, Giordano Scappucci, et al. 2020. Quantum Inspire: QuTech's platform for co-development and collaboration in quantum computing. In *Novel Patterning Technologies for Semiconductors, MEMS/NEMS and MOEMS 2020*, Vol. 11324. Int. Society for Optics and Photonics, 113240J.

[57] Frank Leymann. 2019. Towards a Pattern Language for Quantum Algorithms. In *Quantum Technology and Optimization Problems (Lecture Notes in Computer Science (LNCS), Vol. 11413).* Springer, 218–230.

[58] Alexander McCaskey, Eugene Dumitrescu, Dmitry Liakh, and Travis Humble. 2018. Hybrid programming for near-term quantum computing systems. In *2018 IEEE Int. Conf. on Rebooting Computing (ICRC).* IEEE, 1–12.

[59] Armin Moin, Moharram Challenger, Atta Badii, and Stephan Günnemann. 2021. MDE4QAI: Towards Model-Driven Engineering for Quantum Artificial Intelligence. *arXiv preprint* (2021).

[60] Beatrice Nash, Vlad Gheorghiu, and Michele Mosca. 2020. Quantum circuit optimizations for NISQ architectures. *Quantum Science and Technology* 5, 2 (2020).

[61] Eva-Maria Neumann, Birgit Vogel-Heuser, Michael Gnadlinger, Juliane Fischer, Laura Reimoser, Sebastian Diehm, Tobias Englert, and Michael Schwarz. 2022. Metric-based identification of target conflicts in the development of industrial automation software libraries. In *IEEE International Conference on Industrial Engineering and Engineering Management (IEEM).* IEEE, 1493–1499.

[62] Michael A Nielsen and Isaac Chuang. 2002. *Quantum computation and quantum information.* American Association of Physics Teachers.

[63] OMG. 2017. UML. https://www.omg.org/spec/UML/.

[64] Ricardo Pérez-Castillo, Luis Jiménez-Navajas, and Mario Piattini. 2021. Modelling Quantum Circuits with UML. *arXiv preprint* (2021).

[65] Ricardo Pérez-Castillo, Manuel A Serrano, and Mario Piattini. 2021. Soft. modernization to embrace quantum technology. *Advances in Engineering Soft.* 151 (2021), 102933.

[66] Carlos A Pérez-Delgado and Hector G Perez-Gonzalez. 2020. Towards a quantum Soft. modeling language. In *Proc. of the IEEE/ACM 42nd Int. Conf. on Soft. Eng. Workshops.* 442–444.

[67] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature communications* 5, 1 (2014), 1–7.

[68] Mario Piattini, Guido Peterssen, Ricardo Pérez-Castillo, Jose Luis Hevia, Manuel A Serrano, Guillermo Hernández, Ignacio García Rodríguez de Guzmán, Claudio Andrés Paradela, Macario Polo, Ezequiel Murina, et al. 2020. The talavera manifesto for quantum software engineering and programming. In *1st International Workshop on the QuANtum SoftWare Engineering & pRogramming (QANSWER).*

[69] Ildevana Poltronieri Rodrigues, Márcia de Borba Campos, and Avelino F Zorzo. 2017. Usability evaluation of domain-specific languages: a systematic literature review. In *Human-Computer Interaction. User Interface Design, Development and Multimodality: 19th International Conference, HCI International 2017.* Springer, 522–534.

[70] John Preskill. 2018. Quantum computing in the NISQ era and beyond. *Quantum* 2 (2018), 79.

[71] Eleanor Rieffel, Jason M. Dominy, Nicholas Rubin, and Zhihui Wang. 2020. XY-mixers: analytical and numerical results for QAOA. *Phys. Rev. A* 101 (2020), 012320.

[72] Yue Ruan, Samuel Marsh, Xilin Xue, Xi Li, Zhihao Liu, and Jingbo Wang. 2020. Quantum approximate algorithm for NP optimization problems with constraints. *arXiv preprint arXiv:2002.00943* (2020).

[73] Manuel A Serrano, Jose A Cruz-Lemus, Ricardo Perez-Castillo, and Mario Piattini. 2022. Quantum software components and platforms: Overview and quality assessment. *Comput. Surveys* 55, 8 (2022), 1–31.

[74] Manuel A Serrano, Ricardo Perez-Castillo, and Mario Piattini. 2022. *Quantum Software Engineering.* Springer Nature.

[75] Peter W Shor. 1996. Fault-tolerant quantum computation. In *Proceedings of 37th conference on foundations of computer science.* IEEE, 56–65.

[76] Balwinder Sodhi and Ritu Kapur. 2021. Quantum Computing Platforms: Assessing the Impact on Quality Attributes and SDLC Activities. In *2021 IEEE 18th Int. Conf. on Soft. Architecture (ICSA).* IEEE, 80–91.

[77] Damian S Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: an open source Soft. framework for quantum computing. *Quantum* 2 (2018), 49.

[78] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: Eclipse Modeling Framework.* Addison Wesley.

[79] Javier Verduro, Moisés Rodríguez, and Mario Piattini. 2021. Software Quality Issues in Quantum Information Systems.. In *Q-SET@ QCE.* 54–59.

[80] Birgit Vogel-Heuser, Eva-Maria Neumann, and Juliane Fischer. 2021. MICOSE4aPS: industrially applicable maturity metric to improve systematic reuse of control software. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–24.

[81] Birgit Vogel-Heuser and Felix Ocker. 2018. Maintainability and evolvability of control software in machine and plant manufacturing—An industrial survey. *Control engineering practice* 80 (2018), 157–173.

[82] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering.* Springer.

[83] Xin-Chuan Wu, Shavindra P Premaratne, and Kevin Rasch. 2023. A Comprehensive Introduction to the Intel Quantum SDK. In *Proceedings of the 2023 Introduction on Hybrid Quantum-Classical Programming Using C++ Quantum Extension.* 1–2.

[84] Shixin Zhang, Chang-Yu Hsieh, Shengyu Zhang, and Hong Yao. 2021. Differentiable Quantum Architecture Search. *Bulletin of the American Physical Society* 66 (2021).

[85] Jianjun Zhao. 2020. Quantum Soft. engineering: Landscapes and horizons. *arXiv preprint* (2020).

[86] Jianjun Zhao. 2021. Some size and structure metrics for quantum software. In *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE).* IEEE, 22–27.

[87] Linghua Zhu, Ho Lun Tang, George S Barron, FA Calderon-Vargas, Nicholas J Mayhall, Edwin Barnes, and Sophia E Economou. 2020. An adaptive quantum approximate optimization algorithm for solving combinatorial problems on a quantum computer. *arXiv preprint arXiv:2005.10258* (2020).

A GENERAL LOOP PARAMETERS

By investigating various loop patterns (e.g., from [49], [62], the PlanQK Pattern Atlas[29], the Qiskit Textbook[30]) 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 *targetQubitIterationType* 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.

[29]https://patterns.platform.planqk.de/pattern-languages/af7780d5-1f97-4536-8da7-4194b093ab1d
[30]https://qiskit.org/textbook/preface.html

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 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 bellow).

- *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* $(0, 1, 2, 3, 4)$, *incrementBlockTargetQubit*= $True$, 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.

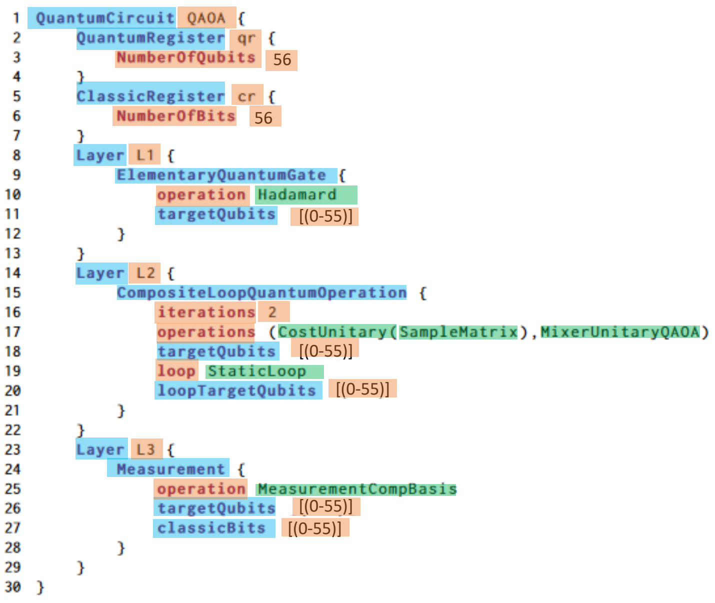
B EVALUATION - METRICS DEFINITION

In the following, a detailed description of our metrics definitions regarding the evaluation provided in Section 6 is presented. The considered metrics comprise the number of actions, the Halstead metrics, and the MICOSE4aPS metric.

Number of actions. We define the number of actions as the sum of (i) objects, (ii) links, and (iii) non-default parameters of a program. An example is shown in Figure 10 for the QAOA56 use case.

The definition is based on the quantum circuit meta-model (Figure 3 of the main text). Thus, whenever an object of a concrete class is created, or a parameter of this object is explicitly specified, we conduct the counting accordingly. The links are between objects related to the quantum circuit meta-model and instances related to the quantum operations library or QUBO meta-model.

Regarding the counting of objects, links, and non-default parameters using the IBM Quantum Composer, we arrive at the figures provided in Table 2 of the main text by applying the rules as illustratively described for the QAOA4 use case in the following. In this specific case, each analysis



```
QuantumCircuit QAOA {
    QuantumRegister qr {
        NumberOfQubits 56
    }
    ClassicRegister cr {
        NumberOfBits 56
    }
    Layer L1 {
        ElementaryQuantumGate {
            operation Hadamard
            targetQubits [(0-55)]
        }
    }
    Layer L2 {
        CompositeLoopQuantumOperation {
            iterations 2
            operations (CostUnitary(SampleMatrix),MixerUnitaryQAOA)
            targetQubits [(0-55)]
            loop StaticLoop
            loopTargetQubits [(0-55)]
        }
    }
    Layer L3 {
        Measurement {
            operation MeasurementCompBasis
            targetQubits [(0-55)]
            classicBits [(0-55)]
        }
    }
}
```

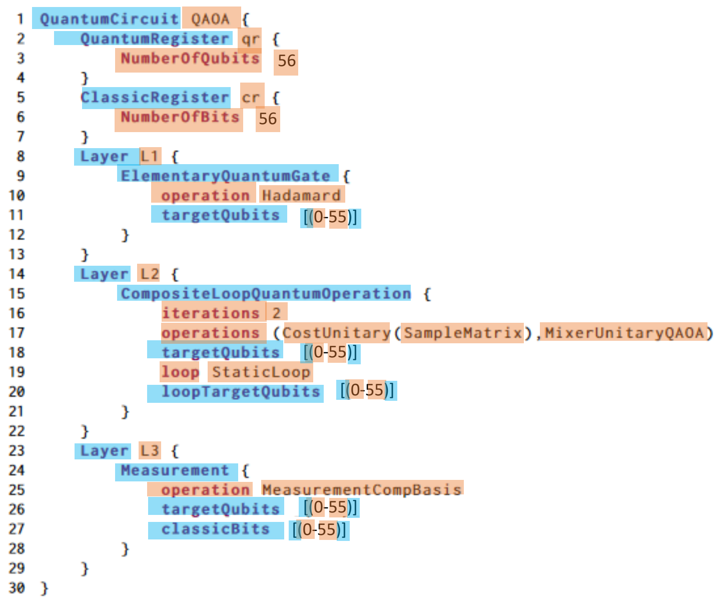
Fig. 10. Counting of objects (blue), links (green), and non-default parameters (orange) for the QAOA56 use case

has to be conducted for (i) the definition of the two cost unitaries, (ii) the two mixer unitaries, and (iii) the overall quantum circuit. The cost and mixer unitary have to be counted twice since the use case comprises two iterations with different angle parameters, also leading to two different quantum operation definitions in the QASM code. Thus, the number of objects is computed as $2 * (22 + 1) + 2 * (4 + 1) + 14 = 70$, where additional to each quantum gate, the +1 resulting from the generation of the cost and mixer unitary, are counted as an object. The links are defined as the qubits on which the gates are applied and are calculated as $2 * 34 + 2 * 4 + 24 = 100$. The non-default parameters comprise (i) the angle parameters of the quantum gates, (ii) the names of the registers, the circuit, and the grouped operations, and (iii) the number of the qubits and classical bits in each register. They are computed as $2 * (10 + 1) + 2 * (4 + 1) + (2 + 2 + 1) = 37$. The metrics for the larger QAOA use cases and the Quantum Counting use cases are calculated analogously.

Halstead software metrics. From the number of unique operators (μ_1), the number of unique operands (μ_2), the number of overall operators (N_1), and the number of overall operands (N_2), the Halstead metrics [41] for a program can be computed as:

- length: $N = N_1 + N_2$
- vocabulary: $\mu = \mu_1 + \mu 2$
- volume: $V = N * log_2 \mu$
- difficulty of writing and understanding: $D = \frac{\mu_1}{2} * \frac{N_2}{\mu_2}$
- effort: $E = D * V$
- implementation time: $T = \frac{E}{18}$

The use of the Halstead metrics requires a proper definition of operators and operands specific to the present languages. Regarding the use of CoQuaDe, an illustrative example for the QAOA56 use case is shown in Figure 11.



```
QuantumCircuit QAOA {
    QuantumRegister qr {
        NumberOfQubits 56
    }
    ClassicRegister cr {
        NumberOfBits 56
    }
    Layer L1 {
        ElementaryQuantumGate {
            operation Hadamard
            targetQubits [(0-55)]
        }
    }
    Layer L2 {
        CompositeLoopQuantumOperation {
            iterations 2
            operations (CostUnitary(SampleMatrix),MixerUnitaryQAOA)
            targetQubits [(0-55)]
            loop StaticLoop
            loopTargetQubits [(0-55)]
        }
    }
    Layer L3 {
        Measurement {
            operation MeasurementCompBasis
            targetQubits [(0-55)]
            classicBits [(0-55)]
        }
    }
}
```

Fig. 11. Counting of operators (blue) and operands (orange) for the QAOA56 use case

Thus, the object creators are treated as operands, i.e., QuantumCircuit, QuantumRegister, ClassicalRegister, Layer, ElementaryQuantumGate, targetQubits, CompositeLoopQuantumOperation, loopTargetQubits, Measurement, classicBits. Additionally, the [()] is considered as the range operator. Operands comprise the qubit numbers (0, 55, 56), the object names (QAOA, qr, cr, L1, L2, L3), the parameter names (iterations, 2, NumberOfQubits, NumberOfBits, operation, operations, loop), and the names of links (Hadamard, StaticLoop, MeasurementCompBasis, CostUnitary, SampleMatrix, MixerUnitaryQAOA). The larger QAOA use cases and Quantum Counting use cases are analyzed analogously. Note, however, that we differentiate between qubits of the different quantum registers for the Quantum Counting use case, i.e., qubit 0 of the first quantum register and qubit 0 of the second quantum register are two unique operands.

Regarding the use of the IBM Quantum Composer, we count the following as operators: circuit, quantum register, classical register, method definition for composed operators, the quantum gates (e.g., cx, rz, rx, h, cost_unitary, mixer_unitary, cgrover, measure). Similar to the use of CoQuaDe, we define the operands as: the name of the registers and the circuit, the number of qubits and bits for each register, the qubit indices, the names of composite operations, and the angle parameters of the quantum gates.

MICOSE4aPS. The MICOSE4aPS metric [80] has been developed to measure the program modularity with respect to conducted changes of the given program. Its computation, which is described further below, relies on several inputs which are defined for the present case in the following:

- Source Lines of Code (SLOC): For the IBM Quantum Composer, we rely on the SLOC of the generated QASM code. Note, that the IBM Composer provides almost no abstraction over QASM code and actions using the graphical editor match the lines of code in the QASM program. Examples where some abstraction is provided are represented by shortcuts when using the IBM Composer. For example, it is not required to state a classical bit for the measurement operation like in QASM, but rather is the index of the classical bit by default set to the index of the measured qubit.

- change category: According to [80], one can differentiate between functional, structural, and operator changes. For the present analysis, all changes are operator changes. Thus, an additional categorization criterion is introduced, namely, whether or not the change affects the quantum processing itself.
- change types: We define the following change types: (i) change of quantum operator, change of qubit, change of angle parameter as quantum processing changes, and (ii) change of number of qubits or bits for the registers, and name changes of the quantum circuit and quantum registers as non-quantum processing changes.

Using these definitions and following [80], the modularity is computed by

$$Maturity = 1 - \frac{1}{n}\sum_{i=1}^{n}\Delta_i \tag{1}$$

for n different changes Δ_i. The latter is given by

$$\Delta = k_l * w * \frac{\sum changes}{max(\sum before; \sum changes)} + k_e * w * (1 - exp(-p * \frac{\sum changes}{max(\sum before; \sum changes)})) \tag{2}$$

with $p = 5$ and

$$k_e = \begin{cases} 1, & \text{if } SLOC \geq 10^3, \\ \frac{1}{850} * (SLOC - 150), & \text{if } 150 < SLOC < 10^3, \\ 0, & \text{if } SLOC \leq 150 \end{cases}$$

and $k_l = 1 - k_e$, and $w = 0.8 * s_1 + 0.2 * s_2$ where s_1 and s_2 are weights related to the change category. The latter are set to $s_1^{qp} = 1$ for quantum processing changes, and $s_1^{nqp} = 0.5$ for non-quantum processing changes. The s_2^* represent specific values which are defaulted to 1 in our analysis. The $changes$ and $before$ relate to the number of items of a certain change type, e.g., the number of changed qubits and the number of qubits before the change, respectively. Note, that for CoQuade, the changed qubits denote the only quantum processing changes. Furthermore, we estimated the changes for the IBM Quantum Composer conservatively to be solely given by the added elements, i.e., we assumed that evolving to a larger circuit is possible without altering the existing items.

C DECOMPOSED QUANTUM OPERATORS OF INVESTIGATED USE CASES

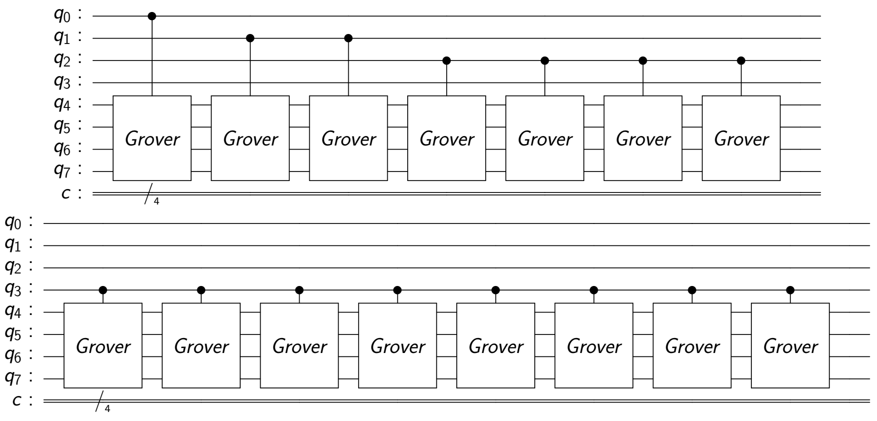


Fig. 12. Controlled Grover unitaries applied within the *Power2Loop* (label 2); visualization conducted with [3]

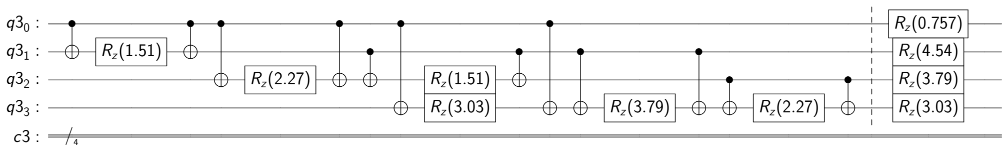


Fig. 13. Generated cost unitary for QAOA; visualization conducted with [3]