QuRest: A RESTful Approach for Hybrid Quantum-Classical Circuit Modeling

Daniel Fink
Institute of Architecture of Application Systems
University of Stuttgart
Germany

May 27, 2021

***Abstract*—Utilizing quantum technologies these days starts mostly with choosing a specific quantum platform. Due to the strong dependency of the hardware, SDK and programming language, switching to different providers can be challenging. Moreover, a comparison of certain quantum algorithms, circuits or tools require simultaneous development of the same techniques on multiple platforms. In order to overcome the lack of cross-platform and cross-backend support of current Quantum SDKs, a RESTful approach for modeling and executing quantum circuits is proposed. The main idea lies in the fact, that the interface specification can be used as an intermediate representation for quantum circuits. While the specification is based on the Quantum Circuit Model, another contribution of this work is the extension of it to include classical control-flow elements like loops, conditions and placeholders, leading to a Hybrid Quantum-Classical Circuit Model. Moreover, a prototype is implement as proof-of-concept.**

***Index Terms*—quantum-computing, rest-api, hybrid-quantum-classical-algorithms**

I. Introduction

Since Richard Feynman proposed a basic model for simulating quantum systems on computers using quantum mechanical laws in 1982 [1] much research has been done in this area, today known as Quantum Computing. The development of sophisticated algorithms for specific purposes led to sub-disciplines like Quantum Chemistry, Quantum Finance and Quantum Machine Learning. For a large number of classical algorithms, quantum counterparts have been developed, offering a polynomial or even exponential speedup. Besides the development of several algorithms, many milestones have been achieved within the last years in terms of building and accessing the desired hardware for making these algorithms generally applicable. After IBM released the first publicly available gate-based quantum computer in 2016 [2] many other quantum hardware vendors also gained publicity. With the releases of AWS Braket in 2019 [3] and Azure Quantum in 2021 [4], two more big cloud-based services for accessing quantum hardware entered the market. With these services, vendors without own *Software Development Kids* (SDKs) like IonQ and Honeywell are able to offer access to their quantum backends.

On top of the hardware and algorithms, many libraries and tools were created, most notably proprietary SDKs from hardware vendors to support their own Quantum Processing Units (QPUs). However, current proprietary SDKs like Qiskit [5], PyQuil/Forest [6], Cirq [7], QDK [8] and Braket [9] only offer a limited native *cross-backend* support. This means, that only quantum backends from very few vendors can be used. Hence, developers and algorithm designers need to decide beforehand on which backends their software should run. Moreover, the comparison and benchmark of a particular quantum algorithm on different backends or simulators need separate implementations that target different backends. On the other side, accessing quantum resources is currently almost only possible from within the Python software platform, as the proprietary SDKs are build on top of it. Hence, the integration into preexisting software based on other platforms is challenging due to the lack of *cross-platform* support.

In order to overcome these problems, a *holistic RESTful approach* for modeling and executing quantum circuits is proposed in this work. Furthermore, a prototypical implementation of it is presented. The main idea of this approach lies in the fact that the interface specification can serve as an intermediate representation for quantum circuits. Thus, cross-backend support is achieved by translating the intermediate representation to work with proprietary SDKs. In addition, cross-platform support is realized by taking advantage of source code generators to generate clients for different software platforms. One crux of this approach is the definition of the interface. On the client-side, it is desirable to have a natural way of using the API, similar to a Quantum SDK like Qiskit. Thus, an obvious idea would be to design the specification based on the Quantum Circuit Model. However, some major disadvantages of this approach are the fixed number of qubits, a limited set of gates and less flexibility. Hence, the Quantum Circuit Model is extended in this work by including classical control flow elements like loops, conditions and placeholders leading to a *Hybrid Quantum-Classical Model*. This allows the design of generic quantum circuits, similar to the usage of a Quantum SDK but with cross-backend and cross-platform support.

The paper is approach as follows: First, an overview of the related work is given in Chapter II. After that, in Chapter III, two example quantum algorithms are introduced. Based on their implementations, requirements for the Hybrid Quantum-Classical Circuit Model and for the API specification are for-

TABLE I
PROPRIETARY QUANTUM SDKS

Proprietary SDK	Programming Language	Backends
Qiskit	OpenQASM, Python	IBM
PyQuil/Forest	Quil, Python	Rigetti
Cirq	Python	AQT
Braket	Python	Rigetti, IonQ
QDK	Q#	Honeywell, IonQ, QCI
Strawberry Fields	Python	Xanadu

mulated. In Chapter IV the Quantum Circuit Model extension is introduced and in Chapter V the prototypical implementation of the RESTful Quantum Service described. Finally, the entire RESTful Hybrid Quantum-Classical approach is summarized and discussed in Chapter VI.

II. RELATED WORK

Most proprietary Quantum SDKs use different HTTP protocols to send locally modeled circuits to a cloud for execution. Examples are WebSocket[1], ZeroMQ[2], REST[3] or plain Request/Response[4]. While quantum hardware vendors employ these protocols for the communication between their own SDKs and cloud services, the interfaces are also used to offer cross-backend support for other SDKs. This is done, for example, in Cirq to support AQT [7]. At this point, it should be noted that most vendors offer no documentation for their interfaces. Hence, writing applications that access those services involves reverse-engineering of the proprietary SDKs. A subset of famous proprietary Quantum SDKs and their natively supported backends is presented in Table I. Besides the native support for several backends, many community projects have been created in recent years to further support cross-backend support. Thus, Qiskit, for example, offers adapters for AQT [10], Honeywell [11] and IonQ [12] backends. Moreover, additional libraries have been created with the intend to support a wide range of backends. One example is the Quantum Machine Learning library PennyLane [13] that supports backends from IBM [14], Rigetti [15], IonQ [16], Honeywell [17], AQT [18] and Xanadu [19]. Besides PennyLane, the quantum libraries PyTket [20] and Tequila [21] offer a wide range of backends due to a plugin architecture. In contrast to the integration of AQT in Braket with an HTTP-based API, the plugins wrap the respective SDKs and thus offer a uniform access to multiple backends.

A different approach is used by Quantastica for their *Quantum Programming Studio* (QPS) [22], where the quantum circuits are modeled online. The QPS offers therefore several components such as a quantum circuit modeling UI, a JavaScript based quantum circuit simulator, a quantum programming language converter and a client module to connect the UI with the local machine in order to execute circuits. Supported is a local simulator, IBM and Rigetti backends. The quantum programming language converter is accessible from the UI and offers a rich set of export formats. Some of them are Quil, PyQuil, OpenQASM, Qiskit, Qobj, Braket, Cirq, TensorFlow Quantum, Q#, AQASM, PyAQASM and QuEST. For importing circuits, OpenQASM or Quil can be used. Thereby, QPS allows to convert a circuit written in a low-level programming language to executable code using a specific SDK. Although a REST-API exists for QPS, it cannot be used for the quantum programming language converter. Therefore, quantum circuits need to be converted by hand, which significantly limits the usage.

In contrast to the cross-backend support, only little research has been done in terms of cross-platform support. In analogy to classical software platforms, intermediate languages have been created to compile code from high-level quantum programming languages to a common representation. Examples are the Quantum Intermediate Representation (QIR) [23] and the Common Quantum Assembly Language (cQASM) [24]. A different approach is done in [25], where an *Extensible Markup Language (XML)* based format for quantum circuits is proposed. The format is called QIS-XML and is used for representing quantum circuits in a structured way, rather then in a low-level programming language. Starting from the well-known XML format, an implementation of compilers for specific software platforms can be done easily with the use of source code generators.

While all types of intermediate representations in general allow cross-platform support, a major drawback lies in the fact that compilers need to be implemented for every software platform. Additionally, plugins need to be created to further compile the intermediate representation to executable code for specific quantum backends. Using a RESTful approach can significantly reduce the overhead for achieving cross-platform support. This is because the compiler is implemented only once at the server-side and clients are created with the use of source code generators. Moreover, different backends can be used on the server-side due to the support of proprietary SDKs.

With this in mind, achieving cross-platform support with the use of an intermediate representation is inspired by the QIS-XML format [25]. Besides that, the idea of utilizing different proprietary SDKs for the server-side is taken from PennyLane [13]. Before the RESTful approach is discussed in more detail, the Quantum Circuit Model and two example algorithms are introduced first.

III. FUNDAMENTALS AND MOTIVATING EXAMPLES

In the following, a brief overview of the Quantum Circuit Model is given. Afterwards, two concrete quantum algorithms, namely Grover's Search Algorithm [26] and a Quantum Embedding Algorithm [27], their implementations as well as their limits are discussed. Based on that, requirements for the Hybrid Quantum-Classical Model and thus for the prototype are formulated.

[1]Used by IBM: https://api.quantum-computing.ibm.com/v2
[2]Used by Rigetti: https://github.com/rigetti/rpcq
[3]Used by IonQ: https://github.com/Qiskit-Partners/qiskit-ionq
[4]Used by AQT: https://www.aqt.eu/aqt-json-tutorial/

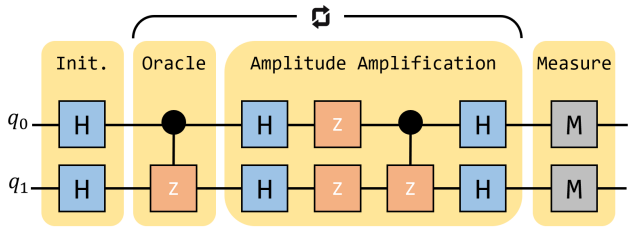

Fig. 1. A quantum circuit representing Grover's Search Algorithm from [35] using a `CZ` gate as oracle on two qubits. The application of the oracle and amplitude amplification is done in a loop, while the amount of iterations depends on the size of the search space.

A. Quantum Circuit Model

The core concept of a quantum circuit is to define the order and type of operations that act on a number of qubits. Such a quantum circuit is visualized as a special tensor network [28]. The amount of qubits associated with a quantum circuit is called the quantum register. Operations that manipulate qubits are the building blocks of every quantum circuit and can be distinguished into unitary and hermitian gates. The latter contains measurements or resets of certain qubits. Depending on the platform, different types of gates are supported that are either natively implemented in the backend or can be decomposed into a sequence of native components. The application of the Quantum Circuit Model can be seen in the following two algorithm examples.

B. Grover's Search Algorithm

Grover's Search Algorithm solves the problem of finding the input of a black box function given the output via $\mathcal{O}(\sqrt{n})$ function evaluations, while n is the size of the search space. The black box function in such a setting is defined as an arbitrary mapping

$$f : \{0,1\}^n \rightarrow \{0,1\}$$

and the quantum circuit implementation of it is called an oracle. Grover's Search Algorithm serves as an example because it has many applications such as estimating the mean and median of statistical distributions [29], solving the collision problem [30] and np-complete problems in general [31]. Additionally, many quantum algorithms use this algorithm as a subroutine [32], [33], [34].

A feasible implementation of the algorithm with two qubits is shown on the Qiskit website [35]. The corresponding quantum circuit is visualized in Fig. 1 and works as follows: First, a superposition of all possible solutions is created by applying a Hadamard gate (`H` gate) on every qubit in the initialization step. Second, the oracle is performed. Afterwards, the so-called amplitude amplification is applied. Here, the amplitude of the state of interest is maximized, while all other amplitudes are minimized. Depending on the size of the search domain, those two steps consist of several iterations. For the given example of two qubits, a single iteration is sufficient to decrease the amplitude of the state of interest to 1 and all others to 0. The last step is the measurement of all qubits, which results in the state of interest.

Listing 1. Grover's Search Algorithm

```
1  from pyquil import Program
2  from pyquil.gates import H, Z, CZ
3  from pyquil.gates import MEASURE
4
5  def my_oracle(q0, q1):
6    oracle = Program()
7    oracle += CZ(q0, q1)
8    return oracle
9
10 def grover_algo(oracle):
11   grover = Program()
12   grover += H(0)
13   grover += H(1)
14   grover += oracle(0, 1)
15   grover += H(0)
16   grover += H(1)
17   grover += Z(0)
18   grover += Z(1)
19   grover += CZ(0, 1)
20   grover += H(0)
21   grover += H(1)
22   ro = grover.declare('ro', 'BIT', 2)
23   grover += MEASURE(0, ro[0])
24   grover += MEASURE(1, ro[1])
25   return grover
26
27 algo = grover_algo(my_oracle)
```

At the design-time of such an oracle-based algorithm, the implementation of the oracle is not specified. This is because it depends on the problem specification and will be set when executing the algorithm accordingly to the concrete problem. Hence, every oracle-based algorithm can be seen as a parameterized algorithm with the oracle as a parameter or placeholder that needs to be chosen before execution.

In Qiskit, for example, the implementation of Grover's Search Algorithm accepts an oracle as input parameter. This oracle can be a quantum circuit Python object or a statevector object, when using a simulator backend. In general, oracles can be used as placeholders with lambda expressions or function pointers. Such an example implementation in PyQuil is given in Listing 1. The oracle is defined as the function `my_oracle` in Line 5 and returns a `program` object. The circuit construction function `grover_algo` in Line 10 accepts a function pointer as argument. This expression is evaluated in Line 14 and creates the oracle on demand.

In OpenQASM or Quil, such a behavior can not be realized. This is because placeholders are not part of the specification. In order to design generic algorithms, placeholders are crucial. Hence, a requirement can be formulated as follows:

Requirement A: *It should be possible to use placeholders during the design-time which are replaced by concrete instances before the execution.*

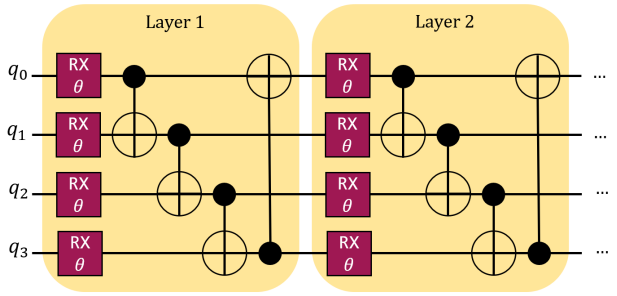

Fig. 2. A two layer Quantum Embedding circuit acting on 4 qubits from [27] using Pauli-X rotations and a circular entanglement strategy.

Besides placeholders, loops are also crucial for designing generic algorithms. They can be found for example in Quantum Embedding algorithms, that are discussed in the following.

C. Quantum Embedding Algorithm

Quantum Embeddings refer to quantum algorithms that are designed to learn the representation of classical data (or a compressed version of it) on quantum computers [36]. A Quantum Embedding algorithm is a Variational Quantum Algorithm (VQA), i.e. it is a parameterized quantum circuit that is trained in a loop via classical optimizers [37]. VQAs in Quantum Machine Learning often use different entanglement strategies to increase the expressivity and performance of the model. Examples can be found in many Quantum Machine Learning algorithms, such as Quantum Autoencoders [38], general Quantum Neural Networks [39], Quantum Support Vector Machines [40] and also Quantum Embeddings [27]. The Quantum Embedding circuit from [27] is visualized in Figure 2. Here, a circular entanglement strategy is implemented, i.e. qubit i is entangled with qubit $i+1$ and the last with the first one. The circuit works as follows: First, a set of rotational gates (here Pauli-X rotations) is applied to every qubit. Second, the entanglement strategy is applied. This procedure is done l times, with l being the number of layers. The example in Fig 2 shows a two layered circuit using four qubits. The rotational gates are parameterized with angles θ that will be optimized within the classical optimization loop.

Quantum Embedding algorithms need to be able to load arbitrary datasets. Hence, it is desirable to model such algorithms based on the amount of qubits. While the structure of the circuit remains the same, increasing the number of qubits results in an increase of rotational gates and extension of the entanglement strategy. A generic version of the Quantum Embedding circuit from Figure 2 is implemented in Listing 2. Here, Qiskit is used as the SDK.

The two different building blocks are implemented as separate Python functions. The rotational block is defined as `pauli_x_rotations` in Line 4 and gets the quantum circuit `qc`, the number of qubits `n` and the current layer index `l` as argument. Parameters using the Qiskit Parameter object are created in the loop in Line 6. The entanglement strategy

Listing 2. Quantum Embedding Algorithm

```
1  from qiskit.circuit import Parameter
2  from qiskit.circuit import QuantumCircuit
3
4  def pauli_x_rotations(qc, n, l):
5    for i in range(0, n):
6      p = Parameter(f'p{l}{i}')
7      qc.rx(p, i)
8    return qc
9
10 def circular_entanglement(qc, n):
11   for i in range(0, n-1):
12     qc.cx(i, i+1)
13   qc.cx(n-1, 0)
14   return qc
15
16 def quantum_model(n, L):
17   qc = QuantumCircuit(n, n)
18   for l in range(0, L):
19     pauli_x_rotations(qc, n, l)
20     circular_entanglement(qc, n)
21   return qc
22
23 qm = quantum_model(4, 2)
```

is applied in the function `circular_entanglement` in Line 10. Another classical for-loop is used here to iterate over the number of qubits `n`. The entire quantum circuit is constructed with calling the `quantum_model` function in Line 23. Here, the number of qubits `n` and the amount of layers `L` are specified. It should be emphasized that all functions use a for-loop to dynamically create a specific amount of gates, depending on the input parameters. While such a behavior can be easily implemented with classical control-flow elements defined in a high-level programming language like Python, it is not possible to achieve a similar result in low-level quantum programming languages like OpenQASM or Quil. This is simply due to the fact, that the language specifications miss classical control-flow elements. Hence, a requirement can be formulated as

> **Requirement B**: *It should be possible to model quantum circuits that scale with the input parameters.*

The two example algorithms have shown which techniques are used to construct generic quantum algorithms. Moreover, it exhibited how the Quantum Circuit Model is used within a high-level programming language like Python. In the next chapter, the focus is turned upside down. Instead of integrating the Quantum Circuit Model into a classical programming language, classical programming elements are integrated into the Quantum Circuit Model, leading to a Hybrid Quantum-Classical Model.

IV. Hybrid Quantum-Classical Circuit Modeling

Within this Chapter, the components of the Hybrid Quantum-Classical Circuit Model are first introduced in a graphical representation and then described in an OpenQASM like

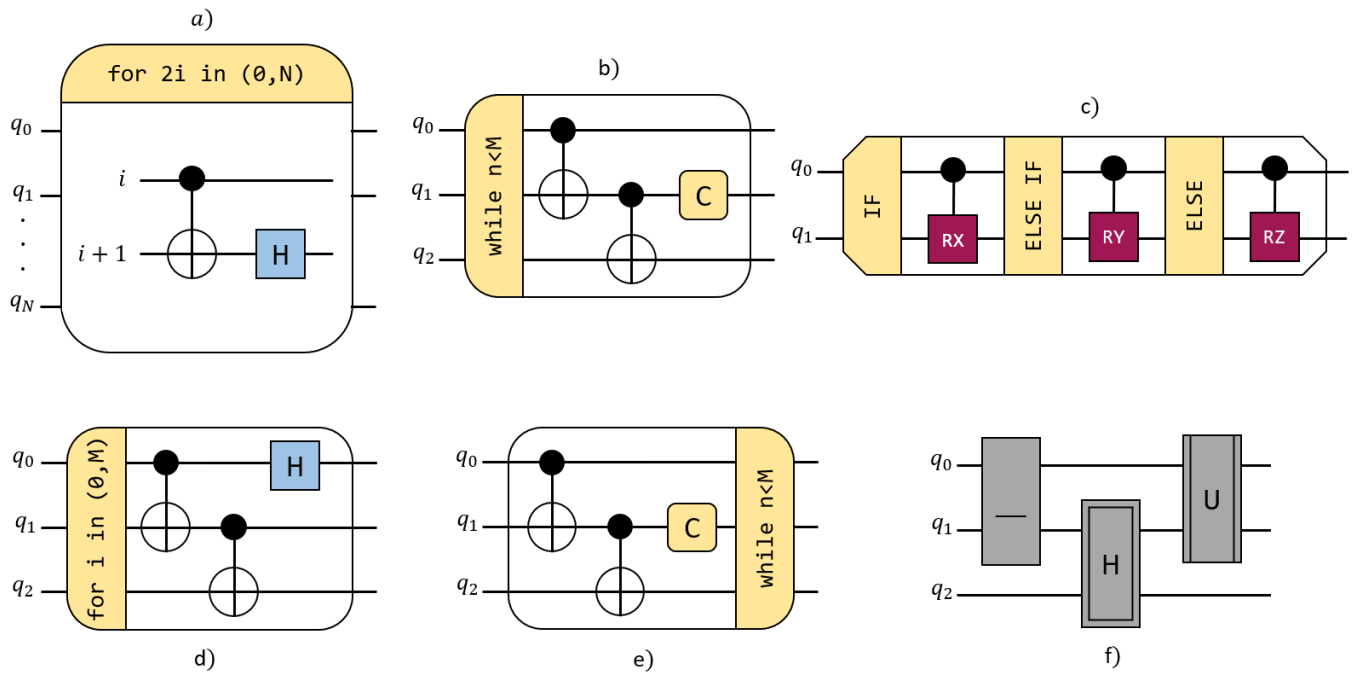

Fig. 3. Visual representation of the extension components. The variable `N` is the variable size of the circuit and `M` an arbitrary design variable. The C gate is a classical operation used for updating design variables. a) For-loop with step-width 2, depending on the size of the circuit `N`. b) While-loop acting on three qubits with the stop-condition $n < N$. c) If-else-if-else conditions acting on two qubits. d) For-loop acting on three qubits independent of the circuit size `N`, but dependent on the design variable `M`. e) Do-while-loop acting on three qubits with the condition $n < M$. f) From left to right: placeholder, hermitian reference and unitarian reference acting on two qubits each.

pseudo-code. After giving an overview of all components, each group is discussed individually.

A. Components Overview

The extension components can be distinguished into control-flow and functional components, see Table II. The first one is further subdivided into loops like for-loops, and conditions such as if-statements. Control-flow components are only feasible with the support of functional components. These components can be split into reference components like placeholders and memory components like variables. The realization of control flow components is possible not least only because of the introduction of so called *design variables* that can be modified at the design-time.

B. Loop Components

In Fig. 3 a), b), d) and e) different types of loops in a quantum circuit are visualized. They all share the concept of repeatedly executing a particular block of code or, in the language of circuit modeling, a sub-circuit. In terms of visualization, their shape differs from quantum gates with having rounded corners instead of sharp ones. Moreover, for the visual representation, loops can be further distinguished into horizontal and vertical loops. The latter one is a loop that is dependent on the (variable) size of the circuit, e.g. the for-loop in Fig. 3 a), while the horizontal loops are independent (e.g. Fig. 3 b), d) and e)). The amount of iterations for for-loops can directly be seen in the header due to the step-size, see Fig. 3 a). In contrast, for while- and do-while-loops, classical operations are needed to determine the length. These classical operations are the "C-Gates", that are discussed later. In Listing 3, a pseudo-code implementation of the four different loops is given. The code-block starts with introducing the variables `N`, `M` and `n`, while the first one is the amount of qubits. The for-loop starting in Line 5 iterates over all even numbers and applies a `CX` gate as well as an `H` gate onto the odd numbered qubits. This makes it a vertical loop, as the iteration is going vertically in contrast to the others that iterate horizontally. The while- and do-while loops in Line 9 and 19 use the variable update operation `n++` to control the amount of iterations. It should be noted here, that the distinction between horizontal and vertical loops is only needed for the visual representation but not necessary on a code level.

TABLE II
EXTENSION COMPONENTS

Control-Flow		Functional	
Loop	**Condition**	**Reference**	**Memory**
FOR	IF	Placeholder	Design Variable
WHILE	ELSE-IF	Unitarian Reference	Runtime Variable
DO-WHILE	ELSE	Hermitian Reference	Variable Update

C. Condition Components

Conditions are also realized with the use of design variables. They can be visualized using rectangles with snipped corners for the "open" sides, i.e. before the if-statement and after the

Listing 3. Loops

```
1  size(N);            # define the size as N
2  par(M);             # add parameter M
3  par(n);             # add parameter n
4
5  for 2i in (0,N):               # Fig 3, a)
6    cx q[i], q[i+1];
7    h q[i+1];
8
9  while n < M:                   # Fig 3, b)
10   cx q[0], q[1];
11   cx q[1], q[2];
12   n++;
13
14 for i in (0,M):                # Fig 3, d)
15   cx q[0], q[1];
16   cx q[1], q[2];
17   h q[0];
18
19 do:                           # Fig 3, e)
20   cx q[0], q[1];
21   cx q[1], q[2];
22   n++;
23 while n < M;
24
```

Listing 4. Conditions, References and Placeholders

```
1  size(3);       # define the size as 3
2  par(n);        # add parameter n
3  par(theta);    # add parameter theta
4  par(myPlace);  # add parameter myPlace
5  ref(myHermit);# add reference to myHermit
6  ref(myUnit);   # add reference to myUnit
7
8  if n == 0:                      # Fig 3, c)
9    rx(theta) q[0], q[1];
10
11 else if n == 1:
12   ry(theta) q[0], q[1];
13
14 else:
15   rz(theta) q[0], q[1];
16
17 plc(myPlace)  q[0], q[1];      # Fig 3, f)
18 cir(myHermit) q[1], q[2];
19 cir(myUnit)   q[0], q[1];
20
```

else-statement, see Fig. 3 c). It should be noted, that the if-else- and else-statements are optional. In Listing 4, a pseudo-code implementation of the condition-chain is given. The parameter `n` is used as a flag and applies either the X-, Y- or Z-Rotation gate with the parameter `theta`.

D. Memory Components

In contrast to classical computing, no variables are allowed in the Quantum Circuit Model. All values need to be set before executing the circuit and the values are fixed once a circuit is running. Exceptions are the upcoming mid-circuit-measurements [41], that allow to change the state of classical registers, where the measurement outputs are stored. These variables are called *run-time variables*, as their state can be changed at the run-time. In order to allow classical components in the Quantum Circuit Model, *design variables* are introduced. Those are variables that can be modified at the design-time, i.e. before a circuit is executed. The familiar rotation-gate parameters as in Listing 2 are examples of design variables. Additionally, meta-information such as the number of qubits for a circuit and the length of a for-loop are also design variables. With these variables, it is possible to model circuits based on the amount of qubits with loops and thus fulfill Requirement B.

In Listing 3 the three design variables `N`, `M` and `n` are introduced with the `size(·)` and the `par(·)` functions. When a circuit is executed, all design variables need to be mapped to real values, such that the final output is a quantum circuit, only containing constants and run-time variables. This is a requirement for executing circuits on real quantum hardware. Because loops can depend on design variables, a mechanism to update those variables is needed. Hence, a variable update gate is introduced, that updates a design variable. This variable update operation, also called "C-Gate", can be seen in Fig. 3 b) and e) to update the loop-index. It corresponds to the Lines 12 and 22 in Listing 3. It should be noted, that these gates do not have an associated qubit. They are classical and only operate on design variables.

E. Reference Components

Besides memory components, also references are part of the extension. It is possible to either reference directly to an existing unitarian or hermitian circuit, or using a placeholder instead. If the latter one is chosen, the corresponding sub-circuit needs to be mapped before execution. An illustration is given in Fig. 3 f). A placeholder is visualized with an underscore to indicate that something is still missing. Hermitian references use the identifier `H` like the Hadamard gate, but has another inner quadrilateral to indicate that it can be "zoomed in" to see the referenced subcircuit. The same holds true for the unitarian reference with the identifier `U`. However, to distinguish reference gates, the unitarian reference do not have a full inner quadrilateral, but just two sides of it instead. This indicates, that a unitarian itself should not be seen as a closed circuit, in contrast to a hermitian reference that includes measurements. In pseudo-code, references are introduced with the `ref(·)` function and placeholders with the `par(·)` function, but performed using the `cir(·)` respectively the `plc(·)` function, see Listing 4. Thus, Requirement A is fulfilled by a placeholder that will be replaced by a sub-circuit before execution.

Within this Chapter we have seen, that the introduction of design variables allows the use of classical control-flow elements, leading to a Hybrid Quantum-Classical Model. However, due to the hardware restrictions, design variables as well as placeholders need to be resolved before execution. This introduces an additional transpilation step between the mod-

eling and execution task, where values for the variables and placeholders need to be specified. How this transpilation step can be realized in a RESTful Quantum Service is discussed in the next Chapter.

V. RESTful Quantum Service

So far it was shown in Chapter III how generic quantum algorithms can be implemented currently. However, this approach lead to limited cross-backend support, depending on the SDK and a lack of cross-platform support, as the SDKs are based on Python. Using a REST-API specification as intermediate representation can overcome this issues. However, one crux lies in the fact that the API should offer a natural way on using the service, similar to a Quantum SDK. This is where the Hybrid Quantum-Classical Circuit Model comes into place, that serves as a guideline for the API specification. In the following, the basic principle of the Quantum Service is described first. Afterwards, some implementation details and the features of the prototype are presented. The Chapter ends with the definition of the intermediate representation and an overview of the components and operations of the prototype.

A. Basic Principle

The main idea of the Quantum Service lies in the fact that the interface specification serves as an intermediate representation for quantum circuits. This means, that the interface abstract away SDK and backend related details thus guaranteeing a uniform access for modeling and execution. Numerous proprietary SDKs can be used on the server-side to offer a wide range of supported backends. The use of an intermediate representation always introduces a certain overhead. This is, because different software platforms need to be able to work with the format, hence implementations for all supported platforms are needed. However, in the RESTful setting, several technologies exist that support the development of an API. Some of them are OpenAPI, GraphQL, RAML, gRPC and JsonAPI. A major benefit of those technologies is the support of source code generators that generate clients including the API contracts. Moreover, as the optimization and transpilation is done on the server-side, the client is only responsible for creating the intermediate representation and send it to the API. Thus, almost no manual code-writing is needed to support a specific software platform.

B. Implementation Background and Features

The implementation of the prototype is done using the programming language C# with the .NET Standard 2.1 SDK and can be found on GitHub [42], Docker Hub and the .NET package manager Nuget. The server is build as an ASP.NET Core REST-API, follows the Command–Query Separation (CQRS) principle and uses either an In-Memory database or a JSON-based local file storage. Moreover, the interface specification standard OpenAPI [43] is used to automatically generate a UI and an interface specification file for the. The .NET based code generator NSwag [44] is used to generate a .NET client, based on the interface specification file.

In principle, the prototype supports the Hybrid Quantum-Classical Circuit Model introduced in Chapter IV. As output, the API generates OpenQASM code, rather then executing a circuit on a QPU. Using the code to execute it, however, is not a difficult task and can be easily done, for example in Qiskit. Hence, the prototype is used as a proof-of-concept. However, while-, do-while-loops, references, runtime variables and variable-update operations are not implemented. This is due to the fact that a single loop is sufficient for a proof-of-concept, references can technically be implemented the same way like placeholders and runtime variables are yet not part of the OpenQASM specification.

C. Intermediate Representation

As the API specification serves as an intermediate representation much attention is given to the definition of the API contracts. While a quantum circuit can be represented in a low-level programming language like the pseudo-code used in Chapter IV, it does not fit with structured API contracts that are serialized to XML or JSON. Since the QIS-XML format [25] has already shown that an XML-based description is possible, the decision was made in favor of XML. In contrast to QIS-XML, where each gate needs to be defined as a matrix, a more simple XML-based format is created. This XML-based format, from now on named *QXML*, uses only an enumeration of strings for gates and is visualized in Fig. 4 a). Each QXML file contains a `Project` root element which itself contains a list of `QuantumCircuits`. The `QuantumCircuit` element contains meta-information such as the name, a description, the size and the parameters. Moreover, it has a list of `Steps`, that defines the order of operations, and a list of `Unitarians`, `Hermitians`, `Placeholders`, `Loops` and `Conditions`, that are used in the circuit. Each of the elements has an index, which is mapped to an entry of the steps list. An example is given in Fig. 4 b), which shows the QXML representation of the for-loop from Fig. 3 a), respectively from Listing 3. Step 0 is associated with a loop element of type `ForLoopStart`. This element contains meta-information about the loop such as the loop-variable, the start and end value as well as the increment. Step 1 is mapped to a `Unitarian`, more preciously to a `CX` gate and Step 2 to a `H` gate. The last Step is again mapped to a loop element, but this time to an element of type `ForLoopEnd`. The remaining components, i.e. `Conditions`, `Hermitians` and `Placeholders`, are defined analogously.

The implementation of the intermediate representation is done in a model-first approach. This means, that the XML schema definition is created first and a .NET based source code generator is used to generate the corresponding objects in C# [45]. Afterwards, an interface definition file is generated based on the source code.

D. Components and Operations of the Prototype

The core part of the prototype are two services called *QxmlTranspiler* and *QxmlTranslator*. The QxmlTranspiler takes a

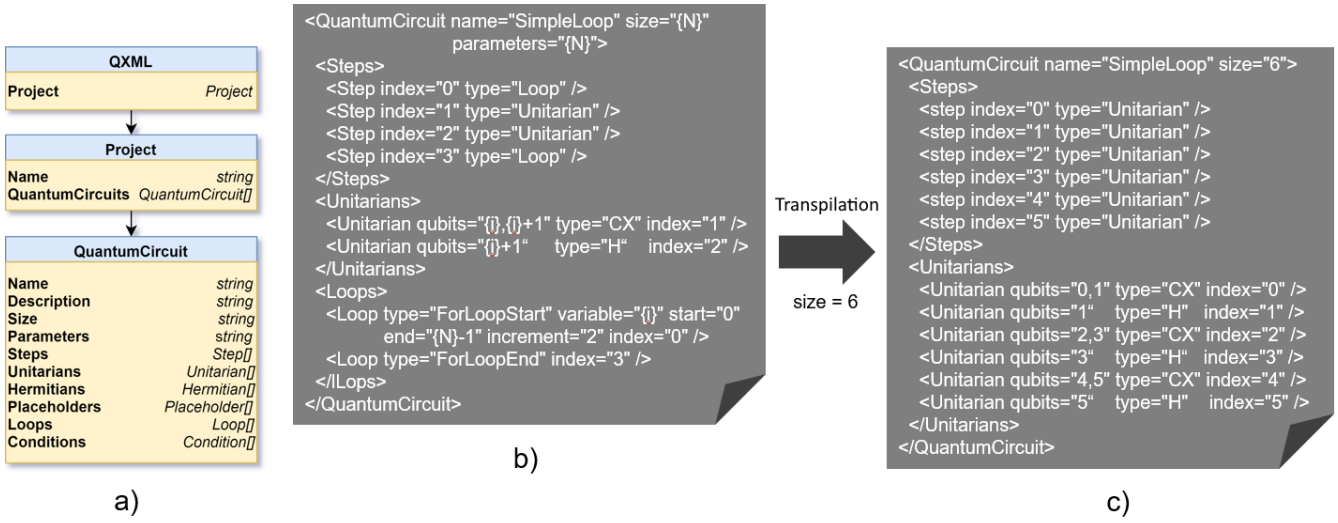

Fig. 4. a) Visualization of the XML Schema Definition of QXML. b) QXML representation of the for-loop from Fig. 3 a), respectively Listing 3. c) QXML representation of the compiler output.

QXML object together with input parameters and transpiles it to another QXML object. This is the additional transpilation step that is needed for the Hybrid Quantum-Classical Circuit Model. Together with the input parameters, all classical elements are resolved within this step and the resulting QXML object contains only `Unitarians` and `Hermitians`. The QXML output of the for-loop from Fig. 3 a) is shown in Fig. 4 c). Here, the size is set to 6 and the output shows 6 `Unitarians` after resolving the for-loop. When having a QXML object without design variables and classical control flow elements, the circuit can be translated to OpenQASM, which is done in the QxmlTranslator.

More complicated examples such as nested for-loops in conjunction with placeholders and conditions as well as real-world examples like the preparation of a GHZ state and Grover's Search Algorithm can be found on the GitHub repository [42].

VI. Conclusion

This work has shown that a RESTful implementation of a Quantum Service is generally possible. Moreover, due to the use of a REST-API, cross-platform support is achieved and the potential for the support of multiple backends exist. The Hybrid Quantum-Classical Model offers new elements for modeling quantum circuits and equip the API with a natural way of using it, like a Quantum SDK. While the prototype is implemented in C#, the intermediate representation is based on XML. Thus, source code generators can be used to easily re-implement the prototype on different software platforms.

Moving away the circuit modeling unit from a local machine to a web-server achieves another advantage for Variational Quantum Circuits. As those circuits need to be trained before usage, the two steps can be done on different software platforms. Hence, it would be possible to train the circuit via fine-tuned classical optimization techniques on C++ but use the trained circuit from a web-based software platform like JavaScript. Additionally, the designing part of a quantum algorithm can also be done separately utilizing software platforms that offer rich graphical support. Furthermore, with only using well-known HTTP technologies like REST, the Quantum Service can directly be integrated into workflow technologies without manually building wrappers. In addition, the contribution of the Hybrid Quantum-Classical Circuit Model can itself be used without a RESTful approach. Thus, the graphical representation can give more insights into complicated quantum algorithms.

However, besides the possibility to construct and execute quantum circuits, most Quantum SDKs do also provide a library of already implemented quantum algorithms. Those algorithms need to be converted to the intermediate representation manually in order to be used in the RESTful approach. Additionally, SDK support to analyze and benchmark quantum algorithms is missing.

On the other side, a major question is, if cross-platform support is crucial for the future of quantum computation. As quantum algorithms are mainly hybrid algorithms, a good interoperability with a classical software platform is desirable. In a microservice architecture, a software platform with adequate quantum support can be chosen for the quantum-subroutine, while the rest of the system can be build based on other software platforms.

In order to decide whether this approach is beneficial in practice, the prototype needs to be extend in order to execute quantum circuits and experiments with various quantum algorithms needs to be done. This shows whether the advantage of cross-platform and cross-backend support outweighs the disadvantages of the RESTful approach.

References

[1] R. P. Feynman, "Simulating Physics with Computers," *International Journal of Theoretical Physics*, vol. 21, no. 6, pp. 467–488, Jun 1982.

[Online]. Available: https://doi.org/10.1007/BF02650179
[2] IBM Research, "IBM Makes Quantum Computing Available on IBM Cloud to Accelerate Innovation," http://www-03.ibm.com/press/us/en/pressrelease/49661.wss, 2016.
[3] Amazon Web Services, "Introducing Amazon Braket, a Service for Exploring and Evaluating Quantum Computing," https://aws.amazon.com/about-aws/whats-new/2019/12/introducing-amazon-braket, 2019.
[4] Krysta Svore, "Azure Quantum is now in Public Preview," https://cloudblogs.microsoft.com/quantum/2021/02/01/azure-quantum-preview, 2021.
[5] H. Abraham, AduOffei, R. Agarwal, I. Y. Akhalwaya, G. Aleksandrowicz *et al.*, "Qiskit: An Open-source Framework for Quantum Computing," 2019. [Online]. Available: https://zenodo.org/record/2562111
[6] R. S. Smith, M. J. Curtis, and W. J. Zeng, "A Practical Quantum Instruction Set Architecture," 2017. [Online]. Available: https://arxiv.org/abs/1608.03355
[7] Cirq Developers, "Cirq," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4586899
[8] Microsoft Azure, "Q# and the Quantum Development Kit," https://azure.microsoft.com/en-us/resources/development-kit/quantum-computing, 2021.
[9] Amazon Web Services, "Amazon Braket: Explore and Experiment with Quantum Computing," https://aws.amazon.com/braket, 2021.
[10] M. Treinish, P. Nation, A. Javadi-Abhari, L. Bello, A. Frisch *et al.*, "GitHub Repository: Qiskit AQT Provider," https://github.com/qiskit-community/qiskit-aqt-provider, 2021.
[11] M. Treinish and A. Javadi-Abhari, "GitHub Repository: Qiskit Honeywell Provider," https://github.com/qiskit-community/qiskit-honeywell-provider, 2021.
[12] C. Collins, P. Nation, A. Milstead, M. Treinish, and J. Gambetta, "GitHub Repository: Qiskit IonQ Provider," https://github.com/Qiskit-Partners/qiskit-ionq, 2021.
[13] V. Bergholm, J. Izaac, M. Schuld, C. Gogolin, M. S. Alam *et al.*, "PennyLane: Automatic Differentiation of Hybrid Quantum-Classical Computations," 2020. [Online]. Available: https://arxiv.org/abs/1811.04968
[14] C. Blank, J. Izaac, S. Boerakker, C. Gogolin, C. Lee *et al.*, "GitHub Repository: PennyLane-Qiskit Plugin," https://github.com/PennyLaneAI/pennylane-qiskit, 2021.
[15] J. Izaac, M. S. Alam, M. Schuld, O. D. Matteo, N. Killoran *et al.*, "GitHub Repository: PennyLane Forest Plugin," https://github.com/PennyLaneAI/pennylane-forest, 2021.
[16] J. Izaac, N. Killoran, V. Bergholm, and Z. Zabaneh, "GitHub Repository: PennyLane-IonQ Plugin," https://github.com/PennyLaneAI/PennyLane-IonQ, 2021.
[17] J. Izaac, N. Killoran, and M. Schuld, "GitHub Repository: PennyLane-Honeywell Plugin," https://github.com/PennyLaneAI/pennylane-honeywell, 2021.
[18] N. Killoran, J. Izaac, and M. Schuld, "GitHub Repository: PennyLane-AQT Plugin," https://github.com/PennyLaneAI/pennylane-aqt, 2021.
[19] J. Izaac, N. Killoran, M. Schuld, J. J. Meyer, T. Bromley *et al.*, "GitHub Repository: PennyLane Strawberry Fields Plugin," https://github.com/PennyLaneAI/pennylane-sf, 2021.
[20] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan, "t—ket¿: A Retargetable Compiler for NISQ Devices," *Quantum Science and Technology*, vol. 6, no. 1, p. 014003, nov 2020. [Online]. Available: https://doi.org/10.1088/2058-9565/ab8e92
[21] J. S. Kottmann, S. Alperin-Lea, T. Tamayo-Mendoza, A. Cervera-Lierta, C. Lavigne *et al.*, "TEQUILA: A Platform for Rapid Development of Quantum Algorithms," *Quantum Science and Technology*, vol. 6, no. 2, p. 024009, Mar 2021. [Online]. Available: http://dx.doi.org/10.1088/2058-9565/abe567
[22] P. Korponaic, "GitHub Repository: Quantum Programming Studio," https://github.com/quantastica/qps-client, 2021.
[23] A. Geller, "Introducing Quantum Intermediate Representation (QIR)," https://devblogs.microsoft.com/qsharp/introducing-quantum-intermediate-representation-qir, 2020.
[24] N. Khammassi, G. G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, and K. Bertels, "cQASM v1.0: Towards a Common Quantum Assembly Language," *arXiv e-prints*, p. arXiv:1805.09607, May 2018.
[25] P. Heus and R. Gomez, "QIS-XML: An Extensible Markup Language for Quantum Information Science," *arXiv e-prints*, p. arXiv:1106.2684, Jun. 2011.
[26] L. K. Grover, "A Fast Quantum Mechanical Algorithm for Database Search," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, ser. STOC '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 212–219. [Online]. Available: https://doi.org/10.1145/237814.237866
[27] M. Schuld, R. Sweke, and J. J. Meyer, "Effect of Data Encoding on the Expressive Power of Variational Quantum-Machine-Learning Models," *Physical Review A*, vol. 103, no. 3, Mar 2021. [Online]. Available: http://dx.doi.org/10.1103/PhysRevA.103.032430
[28] J. Biamonte and V. Bergholm, "Tensor Networks in a Nutshell," *arXiv e-prints*, p. arXiv:1708.00006, Jul. 2017.
[29] L. K. Grover, "A Framework for Fast Quantum Mechanical Algorithms," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 53–62. [Online]. Available: https://arxiv.org/abs/quant-ph/9711043v2
[30] G. Brassard, P. Hoyer, and A. Tapp, "Quantum Algorithm for the Collision Problem," *arXiv preprint quant-ph/9705002*, 1997. [Online]. Available: https://arxiv.org/abs/quant-ph/9705002
[31] M. Fürer, "Solving NP-Complete Problems with Quantum Search," in *LATIN 2008: Theoretical Informatics*, vol. 4957, 04 2008, pp. 784–792. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-78773-0_67
[32] D. Bulger, "Combining a Local Search and Grover's Algorithm in BlackBox Global Optimization," *Journal of Optimization Theory and Applications - J OPTIMIZ THEOR APPL*, vol. 133, pp. 289–301, 07 2007. [Online]. Available: https://link.springer.com/article/10.1007/s10957-007-9168-2
[33] E. Aïmeur, G. Brassard, and S. Gambs, "Quantum Clustering Algorithms," in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 1–8. [Online]. Available: https://doi.org/10.1145/1273496.1273497
[34] Y. Du, M.-H. Hsieh, T. Liu, and D. Tao, "A Grover-Search based Quantum Learning Scheme for Classification," *New Journal of Physics*, vol. 23, no. 2, p. 023020, feb 2021. [Online]. Available: https://doi.org/10.1088/1367-2630/abdefa
[35] IBM, "Qiskit Textbook: Grover's Algorithm," https://qiskit.org/textbook/ch-algorithms/grover.html, 2021.
[36] M. Schuld and N. Killoran, "Quantum Machine Learning in Feature Hilbert Spaces," *arXiv e-prints*, p. arXiv:1803.07128, Mar. 2018.
[37] M. Cerezo, A. Arrasmith, R. Babbush, S. C. Benjamin, S. Endo *et al.*, "Variational Quantum Algorithms," *arXiv e-prints*, p. arXiv:2012.09265, Dec. 2020.
[38] J. Romero, J. P. Olson, and A. Aspuru-Guzik, "Quantum Autoencoders for Efficient Compression of Quantum Data," *Quantum Science and Technology*, vol. 2, no. 4, p. 045001, Aug 2017. [Online]. Available: http://dx.doi.org/10.1088/2058-9565/aa8072
[39] A. Abbas, D. Sutter, C. Zoufal, A. Lucchi, A. Figalli, and S. Woerner, "The Power of Quantum Neural Networks," 2020.
[40] V. Havlíček, A. D. Córcoles, K. Temme, A. W. Harrow, A. Kandala *et al.*, "Supervised Learning with Quantum-Enhanced Feature Spaces," *Nature*, vol. 567, no. 7747, p. 209–212, Mar 2019. [Online]. Available: http://dx.doi.org/10.1038/s41586-019-0980-2
[41] K. Rudinger, G. J. Ribeill, L. C. G. Govia, M. Ware, E. Nielsen *et al.*, "Characterizing Mid-Circuit Measurements on a Superconducting Qubit using Gate Set Tomography," *arXiv e-prints*, p. arXiv:2103.03008, Mar. 2021.
[42] D. Fink, "GitHub Repository: QuRest: A RESTful Approach for Hybrid Quantum-Classical Circuit Modeling," https://github.com/StuttgarterDotNet/qurest, 2021.
[43] T. Tam, K. Hahn, M. Ralphson, R. Dolin, M. Gardiner *et al.*, "GitHub Repository: The OpenAPI Specification," https://github.com/OAI/OpenAPI-Specification, 2021.
[44] R. Suter, J. Groves, D. Bunting, P. Zawadzki, D. Gioulakis *et al.*, "GitHub Repository: NSwag: The Swagger/OpenAPI toolchain for .NET, ASP.NET Core and TypeScript," https://github.com/RicoSuter/NSwag, 2021.
[45] M. Ganss, M. J. B. Callaghan, P. Kranz, T. Southwick, M. Kacprzak, and D. Gardiner, "GitHub Repository: XmlSchemaClassGenerator," https://github.com/mganss/XmlSchemaClassGenerator, 2021.