Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Service-Based Translation of Quantum Circuits

Maximilian Jakob Johannes Kuhn

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Felix Truger, M.Sc., Martin Beisel, M.Sc.
Commenced:	December 1, 2021
Completed:	June 1, 2022

Abstract

Recently, quantum advantage has started to attract attention to the field of quantum computing. While current devices are still noisy and error-prone, numerous vendors have already established themselves, each offering their various approaches with different characteristics and optimizations. In the era of Noisy Intermediate-Scale Quantum (NISQ) computers, quantum circuits must be compiled and executed as efficiently as possible, to best utilize the limited quantum resources available. Therefore, selecting a fitting vendor is a major part of programming for quantum devices. However, different vendors offer different, often incompatible frameworks. Compiled circuits are also highly complex, making manual comparison non-trivial. The NISQ Analyzer has been presented as a solution to this issue. It automates the compilation process of a circuit over a subset of usually incompatible providers. For this purpose it utilizes translation, allowing it to access multiple frameworks even with a circuit only provided in one language. In this thesis, we extend upon this functionality. We make new frameworks available for translation, employing existing translation functionality where possible. For proof of concept, we also implement compilation for a new vendor using the NISQ Analyzer, utilizing our translations. Additionally, we include a detailed evaluation of the reliability of translation frameworks, as well as a case study showing how our extensions can be put to use.

Kurzfassung

In letzter Zeit hat das Versprechen des Quantenvorteils Aufmerksamkeit auf den Bereich der Quanteninformatik gezogen. Während die derzeitigen Geräte noch verrauscht und fehleranfällig sind, gibt es bereits eine große Anzahl von Anbietern, die ihre verschiedenen Ansätze anbieten. Diese weisen alle unterschiedliche Merkmale und Optimierungen auf. In der Ära der NISQ-Computer ist es wichtig, dass die Quantenschaltkreise so effizient wie möglich kompiliert und ausgeführt werden, um die limitierten Quantenresourcen bestmöglich zu nutzen. Daher ist die Auswahl eines geeigneten Anbieters ein wichtiger Bestandteil der Programmierung für Quantencomputer. Die verschiedenen Anbieter bieten jedoch unterschiedliche, oft inkompatible Frameworks an. Außerdem sind kompilierte Schaltungen hoch komplex, was einen manuellen Vergleich schwierig macht. Der NISQ Analyzer wurde als Lösung für dieses Problem vorgestellt. Er automatisiert den Prozess der Kompilierung eines Schaltkreises über eine Teilmenge von normalerweise inkompatiblen Anbietern. Zu diesem Zweck nutzt er Übersetzung, welche es ihm ermöglicht, auf mehrere Frameworks zuzugreifen, selbst wenn ein Schaltkreis nur in einer Sprache vorliegt. In dieser Arbeit erweitern wir diese Funktionalität, indem wir neue Frameworks für die Übersetzung verfügbar machen. Hierbei nutzen wir, wenn vorhanden, bestehende Übersetzungsfunktionen. Um zu zeigen, wie unsere Übersetzungen weiterführend eingebunden werden können, implementieren wir auch das Kompilieren für einen der neuen Anbieter mit dem NISQ Analyzer. Außerdem führen wir eine detaillierte Auswertung der Zuverlässigkeit der Übersetzungsframeworks durch, sowie eine Fallstudie, die zeigt, wie unsere Erweiterungen eingesetzt werden können.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

API Application Programming Interface. 18

clbit Classical Bit. 15

CLI Command Line Interface. 18

GUI Graphical User Interface. 18

IR Intermediate Representation. 32

JS JavaScript. 18

NISQ Noisy Intermediate-Scale Quantum. 3

OpenQASM Open Quantum Assembly Language. 18

PaaS Platform as a Service. 15

QCaaS Quantum Computing as a Service. 15

QFT Quantum Fourier Transform. 36

QPU Quantum Processing Unit. 15

QuAntiL Quantum Application Lifecycle Management. 5, 15

qubit Quantum Bit. 15

REST Representational State Transfer. 35

SDK Software Development Kit. 15

1 Introduction

One of the most exciting technologies that have been emerging in the late 20th and early 21st century is quantum computing. It promises to solve essential problems in numerous areas of research significantly faster and sometimes more precisely than current classical computers [AHY20; MEA+20; NSB+19]. The development in this field has reached a point where multiple quantum computers are available to quantum software engineers. They are developed by different research groups and utilize distinct technologies, each with unique benefits and limitations [LJL+10]. These machines are so-called NISQ computers [LB20; Pre18]. As the name suggests, they are noisy and have limited quantum resources. Still, multiple vendors have started granting access to their devices to the public via a cloud. This way, quantum computing has recently become available to a broad range of users [FBW18; LBF+20]. Since this is highly related to the Platform as a Service (PaaS) concept, these offerings are called Quantum Computing as a Service (QCaaS) [RI15].

There are different models for quantum computation, based on concepts such as quantum automata, quantum neural networks, and quantum annealing [NWA21]. However, we focus on gate-based quantum circuits utilizing Quantum Bits (qubits), the quantum variant of Classical Bits (clbits) [NC01], which is the currently most frequently used model. With this new form of computation come new languages for the representation of quantum circuits. And even in this space of similar concepts, a system of competitiveness between QCaaS providers has developed, where each one only supports a small subset of languages for execution on their Quantum Processing Units (QPUs).

This diversity brings some issues with it. Users are offered a variety of languages, Software Development Kits (SDKs), and QCaaS providers, that all rely on custom software and offer different features. This includes everything from high-level language features offered by the SDK to the gate set the quantum circuit is compiled to on the QPU [LBF+20]. Due to this heterogeneity, quantum programs are distributed over multiple platforms. This means that finding required quantum circuits in the preferred language might prove difficult. In the case that the circuit needs to be compatible with multiple providers, supported features such as the native gate set and the specification of supported hardware must be considered extra carefully. As a result, the comparison and selection of fitting providers is a cumbersome process. While there have been efforts to provide an analysis of the features and restrictions of the different languages, which are presented, for example, in Qverview [VN21], these results remained purely theoretical and have yet to be put to use.

To solve these and other hindrances in the development of quantum programs, the University of Stuttgart has instantiated the Quantum Application Lifecycle Management (QuAntiL) project [Wil22]. QuAntiL aims to enable service-based quantum computing. This includes the translation of quantum circuits into different formats and languages, compilation for multiple devices, analysis and comparison of compilations, and execution of quantum circuits. The translation is offered by the Quantum Circuit Transpiler [22j; Wan20] and the compilation and analysis by the NISQ Analyzer [SBB+20]. Another part of the QuAntiL repository is the project Quokka, which aims to offer API-based quantum circuit generation, translation, execution, and error mitigation all in one place [BT22b]. Generation of circuits is implemented in the Quantum Circuit Generator [BT22a]. Currently, these projects mostly address a small selection of the most popular languages. The goal of this project is to widen the range of available platforms.

To achieve this, we first analyze the current state-of-the-art in both quantum SDKs and quantum circuit translation, selecting components suitable for our implementation in the process. The existing services are then extended with new functionality. Both the quantum transpiler and consequently the NISQ Analyzer support two vendors with their respective SDKs and languages, IBMQ using Qiskit and OpenQASM as well as Rigetti using PyQuil and Quil. Using translation paths between languages presented alongside suitable translation framework options in Qverview [VN21], we expand the translation service to support six languages using mostly existing translation functionality. It is then used to allow compilation and analysis for a new provider in the NISQ Analyzer. We also add a new circuit generator to the Quantum Circuit Generator, which we use to evaluate the aforementioned translation functionality.

The remainder of this thesis is structured as follows: Chapter 2 discusses related work in the field of quantum circuit translation, as well as similarities and differences to this project. Chapter 3 gives an overview of the projects this thesis extends upon. Chapter 4 details the process we use to analyze current frameworks and presents our results, giving an overview of our selection. Chapter 5 shows how the different features are implemented. Chapter 6 describes the evaluation process and presents its results. In Chapter 7, we provide a case study of our integration into the NISQ Analyzer and Chapter 8 concludes this thesis, discussing possible future work in the process.

2 Related Work

In this chapter, we give a short overview of related projects in the field of quantum circuit translation. We take a look at frameworks that allow translation of circuits, discussing their functionalities, by what means circuit translation plays into it, as well as differences and similarities to this project.

2.1 t|ket⟩

t|ket⟩ is a platform for developing quantum software [SDC+20]. The unique characteristic of t|ket⟩ is that it is platform-agnostic, i.e. it is not developed by or specifically tailored towards one quantum hardware provider. This means it supports a variety of languages, both for the import of circuits into its internal format and for the export back into the provider-specific languages, allowing circuits to be executed on a wide range of devices. The core of t|ket⟩ is formed by a C++ library, however, it is available as the python module pytket, which provides the programming interface to interact with t|ket⟩. This root component offers mostly circuit creation, while the capability to interact with vendors and their languages is offered in the form of plug-ins. These offer functionalities such as import and export to that vendor's quantum circuit representation, as well as execution on their devices. At the time of writing, pytket offers a total amount of 14 plug-ins. Using import and export of different vendors in combination allows translation between them. In our project, we want to offer translation in a service-based way, which t|ket⟩ currently does not provide.

2.2 PennyLane

PennyLane is a python framework developed by the quantum hardware vendor Xanadu, with its focus on machine learning and optimization of quantum circuits [BIS+18]. This means that its main feature is not the compilation for multiple vendors or translation. However, similar to t|ket⟩, PennyLane comes with a plug-in system that makes it compatible with an assortment of vendors. The main focus of these plug-ins is to add devices to PennyLane that circuits can be executed on. Importing circuits on the other hand is not useful for PennyLane, so it is rarely included. Still, PennyLane is compatible with 12 plug-ins, each supporting one provider, which means its translation functionality is significant, even if not the main focus. The obvious difference in our project is that the core functionality of PennyLane is completely different. Also, due to only having very limited import, the translation paths it offers are mostly unidirectional. In addition, it similarly to t|ket⟩, is mostly available as a python package, while we want to offer service-based translation.

2.3 staq

Another example of a quantum toolkit not developed by a vendor is staq [AG20]. It is developed in C++ and its core functionality is reading and manipulating circuits written exclusively in the Open Quantum Assembly Language (OpenQASM), even though it is not maintained by the corresponding vendor. Its methodology is based on UNIX. It offers several small Command Line Interface (CLI) tools, providing functionalities from optimizations and transformations to physical mappings. They operate on an OpenQASM representation using staq's internal syntax extensions. Most important to this project, these tools also include translators, which allow staq to output circuits not only to OpenQASM but also to the languages of other providers. staq is also available via the python wrapper pystaq. While staq's main focus is compilation, akin to t|ket⟩, it only supports a single input language, which makes it similar to PennyLane in the amount of functionality it offers. Concerning our project, we again have the main differences in the way staq is provided and how it only offers translations in one direction.

2.4 qconvert

Lastly, we take a look at qconvert, which is part of the Quantum Programming Studio developed by Quantastica [22h]. The Quantum Programming Studio is a web-based quantum programming IDE, which allows both simulation and execution on real QPUs directly from the UI. qconvert is the service that allows the translation of quantum circuits between languages [22e]. Currently, it is available as a CLI tool or online[1], both as an Application Programming Interface (API) and a Graphical User Interface (GUI), hosted on the Quantum Programming Studio website and written in JavaScript (JS). There is also a python package in development, but currently, it only supports a vanishingly small number of languages in comparison to the 27 formats the JS version does [21]. However, it needs to be mentioned that qconvert, while supporting an extensive amount of formats, does not support many providers. Some of the formats are only graphical or universal JSON representations, but more importantly, qconvert supports multiple versions of the same languages, which all count as separate formats. It is also important to note that some of the supported versions are outdated. In terms of import, the JS version of qconvert supports exactly two languages, OpenQASM, and Quil. Out of the works listed, qconvert might seem like the one most closely related to our project, however, there is still a major difference. While both use a service-based access model, qconvert, similarly to staq and PennyLane, mostly offers unidirectional translation, while we want a bidirectional translation to be available for all languages.

[1] `https://quantum-circuit.com/qconvert`

3 Foundations

In this section, we give an overview of the projects that we work on during this thesis. We discuss the concepts behind them as well as their functionalities.

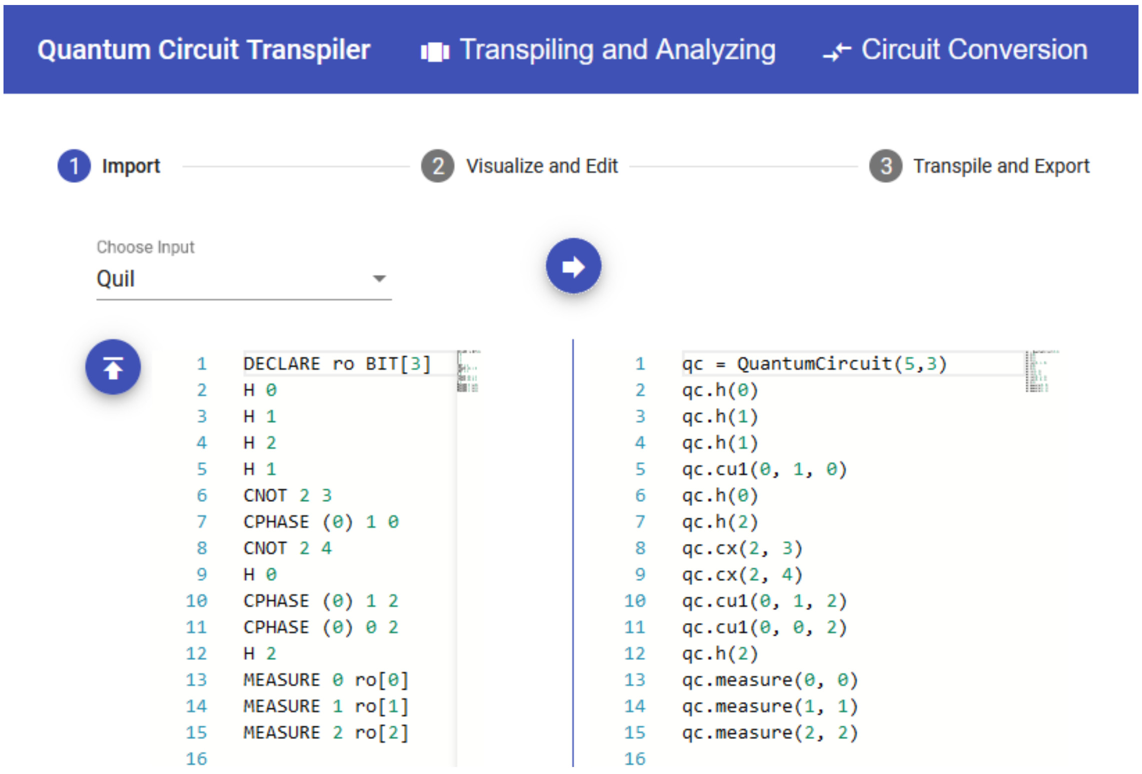
3.1 Quantum Application Lifecycle Management

QuAntiL by the University of Stuttgart has the goal of providing tools for the implementation, deployment, execution, and monitoring of quantum applications [22g]. It comprises several services, which implement different functionalities. One of the main components of QuAntiL is the QC Atlas, which allows the documentation of quantum circuits and also offers a GUI that can be used to access other services of QuAntiL such as the NISQ Analyzer. This work is largely concerned with the NISQ Analyzer and the Quantum Circuit Transpiler services, which are discussed in more detail in the following sections.

3.2 Quantum Circuit Transpiler

The Quantum Circuit Transpiler[1] [22j; Wan20] is responsible for converting circuits between languages, either by using its GUI or via API if requested by other services, such as the NISQ Analyzer or the Quokka API. The term *transpile* here refers to the process of creating *compiled* code by translating source code from one language to another. Compiled code is code on the hardware instruction level, usually created from high-level language code using a compiler program. In our case, the languages are quantum circuits on the gate level, compiled using quantum SDKs. Thus, translation between these languages is transpilation. This also results in the terms *translate* and *transpile* referring to the same process in our specific case. Figure 3.1 displays how the Quantum Circuit Transpiler allows a circuit to be imported from any supported language for usage in transpilation and analysis. It is translated to a Qiskit Quantum Circuit object, which is the internal format used by the Quantum Circuit Transpiler. The user is then provided with a graphical representation of the circuit after it has been translated to Qiskit, which is presented in Figure 3.2. It can be adjusted by moving, adding, and deleting gates. Figure 3.3 shows results of the local simulation functionality. Lastly, the circuit is mapped onto the basis gate sets of different providers' SDKs and different depths of the resulting circuit are presented to the user. The depth is the longest path of gates that needs to be transversed when executing the circuit. The transpiled circuit can then be exported to the different supported languages. Both the analysis result as well as the export functionality are depicted in Figure 3.4.

[1] sometimes also known as *Circuit Transformer* or simply *Quantum Transpiler*



```
DECLARE ro BIT[3]
H 0
H 1
H 2
H 1
CNOT 2 3
CPHASE (0) 1 0
CNOT 2 4
H 0
CPHASE (0) 1 2
CPHASE (0) 0 2
H 2
MEASURE 0 ro[0]
MEASURE 1 ro[1]
MEASURE 2 ro[2]
```

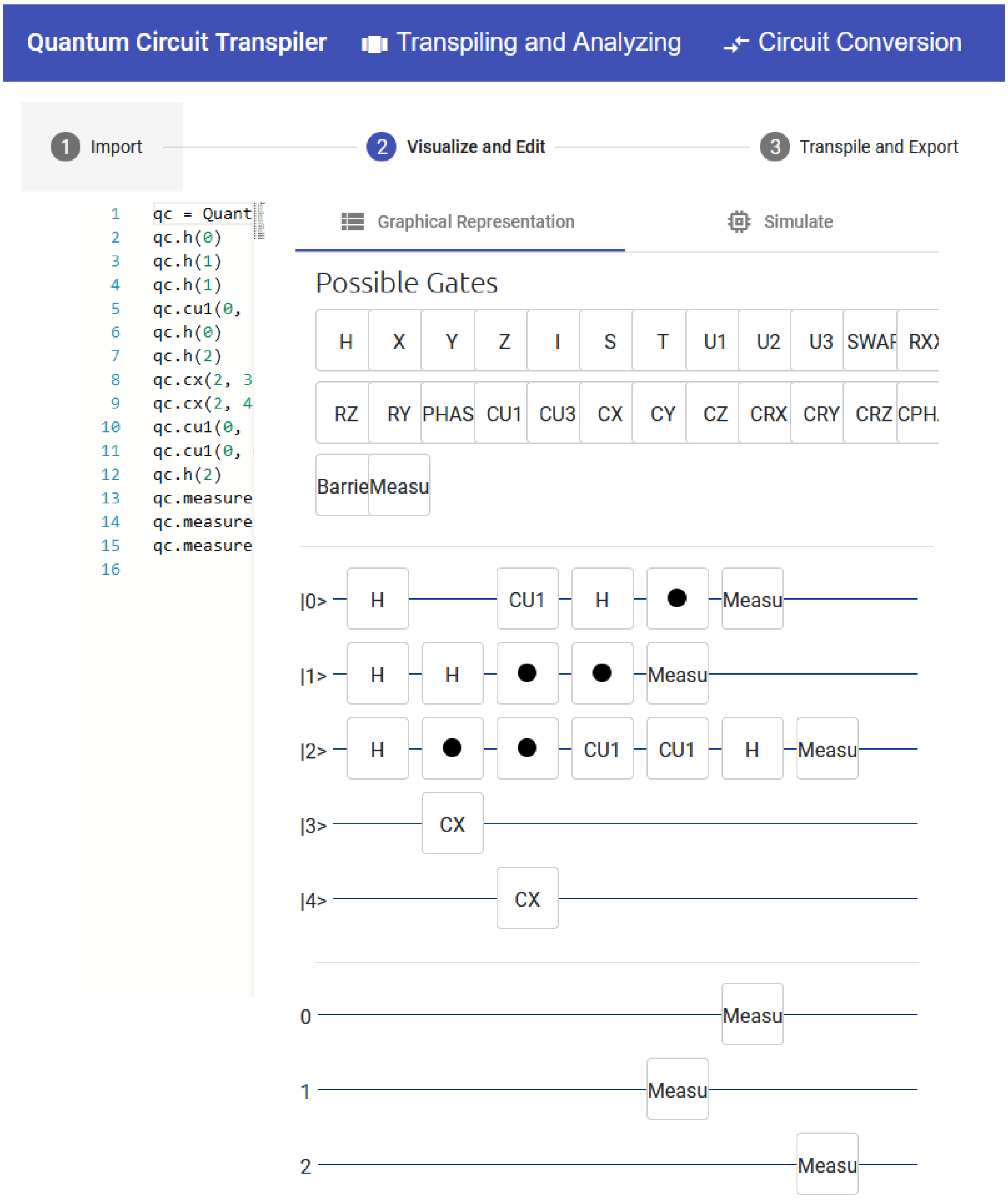
```
qc = QuantumCircuit(5,3)
qc.h(0)
qc.h(1)
qc.h(1)
qc.cu1(0, 1, 0)
qc.h(0)
qc.h(2)
qc.cx(2, 3)
qc.cx(2, 4)
qc.cu1(0, 1, 2)
qc.cu1(0, 0, 2)
qc.h(2)
qc.measure(0, 0)
qc.measure(1, 1)
qc.measure(2, 2)
```

Figure 3.1: Importing a Circuit

While the analysis functionality of the Quantum Circuit Transpiler is very useful, as part of QuAntiL it is mostly used for translation purposes. For this, it offers an API, which combines import and export into a single conversion endpoint. As shown in Figure 3.5, its user interface can be accessed using the second tab of the GUI. Circuits are translated by first importing them to the intermediate format used by the transpiler, a Qiskit Quantum Circuit, and then exporting that circuit to the target language. The Quantum Circuit Transpiler does not use any external frameworks for its translations yet. Instead, it uses a list of mappings of gates from Qiskit and PyQuil to each other to convert the circuits.

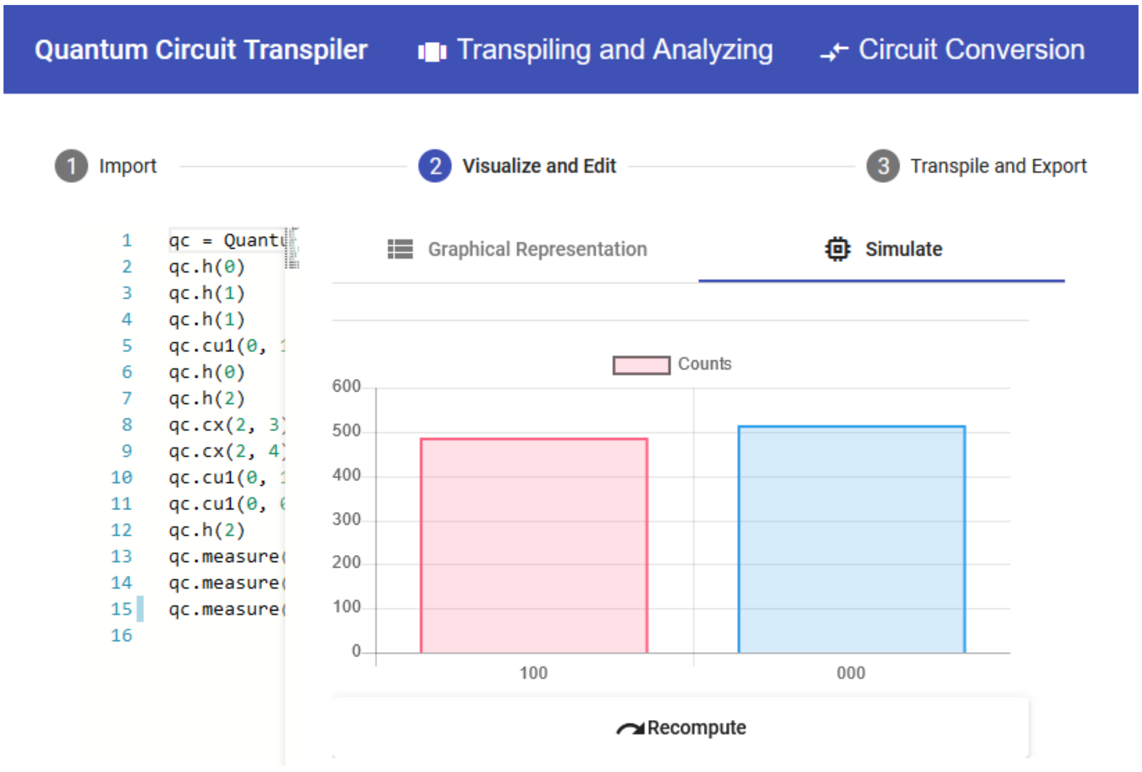
3.3 NISQ Analyzer

While the Quantum Circuit Transpiler offers some analysis functionality, the main service for analysis and comparison of compilations in QuAntiL is the NISQ Analyzer [SBL+21]. Its main goal is to assist in the selection of the device best suited for the execution of a quantum circuit. For this purpose, it offers an automated comparison of suited compilers and QPUs for an algorithm implementation, independently of the language in which the algorithm is currently implemented. This is done in a multi-step process, which is depicted in Figure 3.6.
First, a quantum circuit is supplied as input to the API. Furthermore, the user chooses whether SDKs or QPUs should be compared. Based on the selected mode, either a QPU needs to be supplied and a list of suitable SDKs is created, or an SDK is given and its list of supported QPUs is returned. This can be done since each SDK defines which vendors and thus which QPUs it supports. After



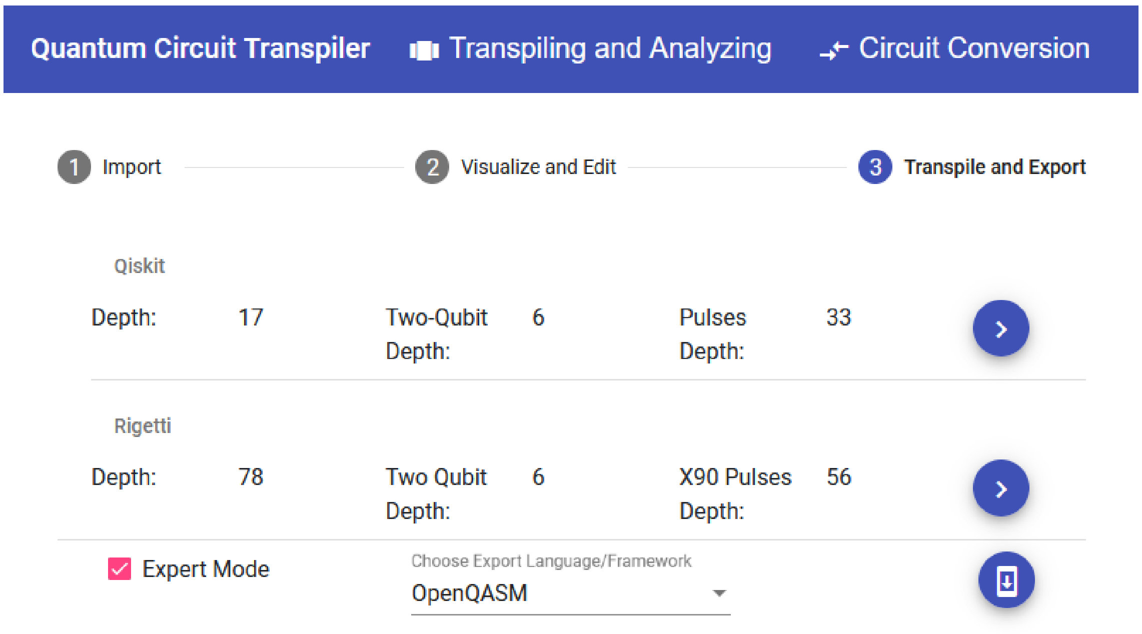
```
qc = Quant
qc.h(0)
qc.h(1)
qc.h(1)
qc.cu1(0,
qc.h(0)
qc.h(2)
qc.cx(2, 3
qc.cx(2, 4
qc.cu1(0,
qc.cu1(0,
qc.h(2)
qc.measure
qc.measure
qc.measure
```

Figure 3.2: Editing a Circuit



```
qc = Quantu
qc.h(0)
qc.h(1)
qc.h(1)
qc.cu1(0, 1
qc.h(0)
qc.h(2)
qc.cx(2, 3)
qc.cx(2, 4)
qc.cu1(0, 1
qc.cu1(0, 6
qc.h(2)
qc.measure(
qc.measure(
qc.measure(
```

Figure 3.3: Simulation Results



Qiskit

Depth:	17	Two-Qubit Depth:	6	Pulses Depth:	33

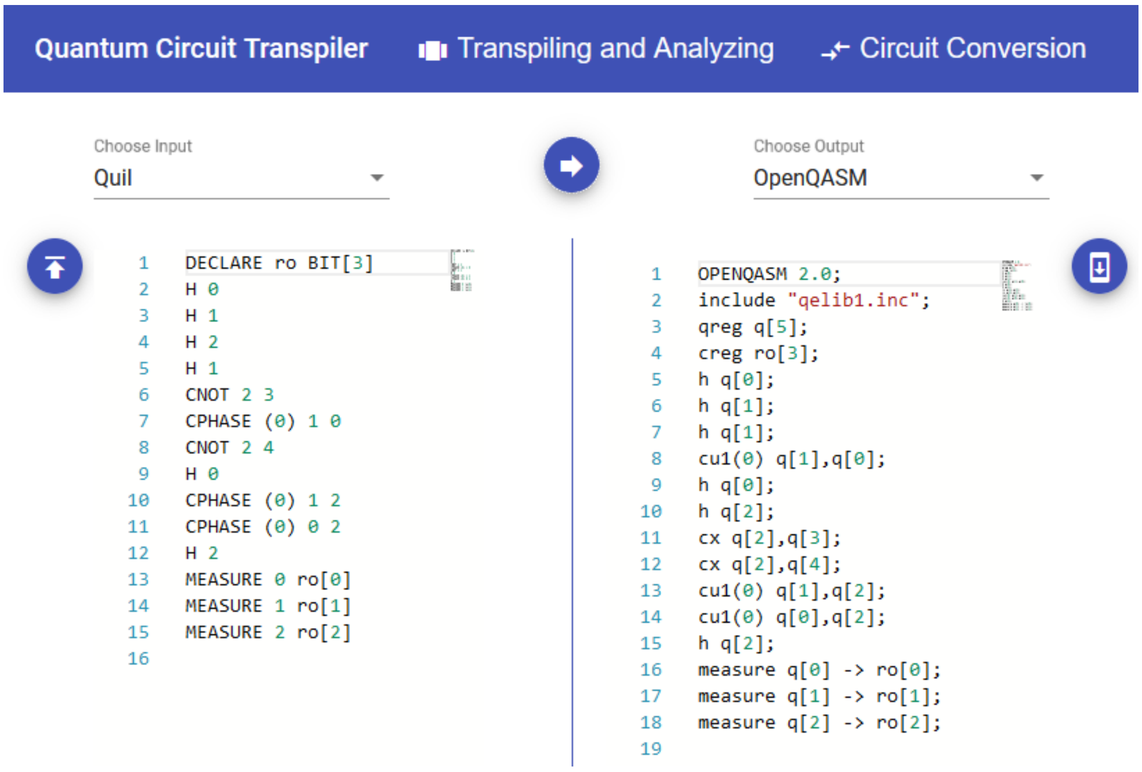
Rigetti

Depth:	78	Two Qubit Depth:	6	X90 Pulses Depth:	56

✓ Expert Mode

Choose Export Language/Framework
OpenQASM

Figure 3.4: Transpilation Results and Export



```
DECLARE ro BIT[3]
H 0
H 1
H 2
H 1
CNOT 2 3
CPHASE (0) 1 0
CNOT 2 4
H 0
CPHASE (0) 1 2
CPHASE (0) 0 2
H 2
MEASURE 0 ro[0]
MEASURE 1 ro[1]
MEASURE 2 ro[2]
```

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[5];
creg ro[3];
h q[0];
h q[1];
h q[1];
cu1(0) q[1],q[0];
h q[0];
h q[2];
cx q[2],q[3];
cx q[2],q[4];
cu1(0) q[1],q[2];
cu1(0) q[0],q[2];
h q[2];
measure q[0] -> ro[0];
measure q[1] -> ro[1];
measure q[2] -> ro[2];
```

Figure 3.5: Conversion of Circuits

this, the languages supported by the selected SDKs are determined. In case the original language of the given circuit is not supported by all SDKs, it is translated to a supported language using the aforementioned Quantum Circuit Transpiler.
In the second phase, the circuits in their supported languages are compiled for the chosen QPU(s) using the selected compiler(s). The different compilers are executed in parallel. This compilation results in a list of compilations of the quantum circuit in the languages compatible with the chosen SDKs.
In the third phase, the compilations are analyzed and compared regarding their properties, such as depth and width. First, some automatic checks are executed to determine whether the circuit could even be executed on the device. Afterward, the user can manually choose a preferred compilation.
In the final phase, after a compilation has been chosen, it can be directly executed on the QPU using the SDK. Since not every QPU is directly accessible like this, simulators might be used instead. The NISQ Analyzer supports IBMQ and Rigetti QPUs, as well as the compilers Qiskit, PyQuil, and Pytket.

3.4 Quantum Circuit Generator

The last service that is extended is the Quantum Circuit Generator. Contrary to the other mentioned projects, while being part of the QuAntiL repository, it is not directly interconnected with the NISQ Analyzer or the QC Atlas. Instead, its purpose is to serve as the quantum circuit library for the project Quokka [BT22b]. To allows this, its API offers functionality to provide both encodings for

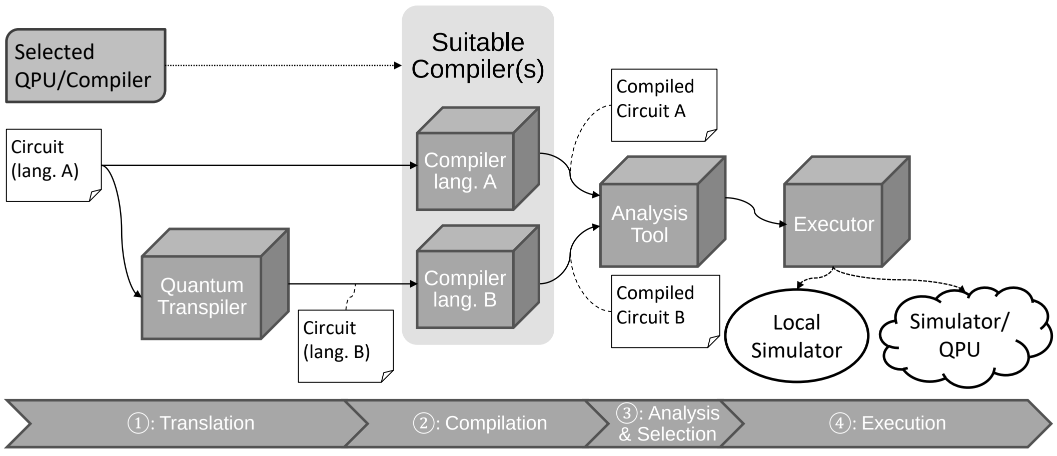


Figure 3.6: Working Scheme of the NISQ Analyzer - Adapted From [SBL+21]

input data, as well as algorithms. The first endpoint can be used to prepare a quantum state, using, for example, basis, angle, or amplitude encoding. The second endpoint provides the user with algorithm circuits or circuit fragments. This includes the algorithms HHL and QAOA. Since these circuits have real applications, using them for the evaluation of other services, like those extend in this project, gives meaningful results.

4 Analysis

As mentioned in Chapter 1, the variety of QCaaS providers is large, and so is the variety of quantum SDKs. Furthermore, Chapter 2 shows that the frameworks offering translation functionality between these languages are diverse, with the number of supported languages and features, as well as the quality of translations varying widely. Therefore we start with an analysis of the relevant space and select the vendors and frameworks most suitable for this project. After that, we give a more detailed look at the selected assets. This helps in getting a better understanding of the steps we are taking in Chapter 5.

4.1 Evaluating Provider-Specific SDKs

Before we can start analyzing translation frameworks, we first have to select which languages and frameworks we want to support utilizing them. Thus we provide an analysis of quantum SDKs in the following section, choosing which to support and giving reasoning to our choices.

4.1.1 Selection Criteria and Information Gathering

When it comes to selecting which SDKs and languages we want to support, we first need to define which criteria we evaluate them by. In the case of this project, we decided on the quantum cloud services we cover by supporting the selected SDKs, as well as whether they are still actively developed and upheld. Lastly, if multiple languages support the same native compiler format for the import and export of circuits, all are simultaneously supported. Languages are selected from the Quantum Open Source Foundation [FB22] and information on the languages is sourced from existing work [LaR19], the QCaaS specific documentation as listed in Table 4.1, as well as study results viewable using the Qverview tool [VN21]. The languages we analyze are summarized in Table 4.2.

Name	**URL**
IBMQ	`https://quantum-computing.ibm.com/docs/`
Forest	`https://docs.rigetti.com/qcs/`
Azure Quantum	`https://docs.microsoft.com/en-us/azure/quantum/`
Amazon Braket	`https://docs.aws.amazon.com/braket/`
ProjectQ	`http://projectq.ch/code-and-docs/`
Quantum Inspire	`https://www.quantum-inspire.com/kbase/`

Table 4.1: QCaaS Vendor Documentation

SDKs	Compiler Language	Hosts	Quantum Cloud Services	Active
Qiskit	OpenQASM	Python Java Script	IBM Quantum AQT Cloud	true
Cirq	Cirq-JSON	Python	Google Cloud AQT Cloud Pasqal	true
Forest	Quil	Python	Rigetti QCS	true
ProjectQ	OpenQASM	Python	IBM Quantum AQT Cloud	false
Quantum Development Kit	Q#	C# Python	Azure Quantum	true
Amazon Braket Python SDK	Braket IR	Python	AWS Braket	true
Quantum Inspire SDK	cQASM	Python	Quantum Inspire IBM Quantum	true

Table 4.2: Comparison of Different SDKs

4.1.2 Selection and Reasoning

The next step is to select the most suitable SDKs. To start with, Qiskit and Forest are already supported by the Quantum Transpiler. Consequently, IBM Quantum, Rigetti QCS, Quil, and OpenQASM are supported as well. Since ProjectQ is out of active development and overlaps with Qiskit in both compiler language and supported services, explicitly supporting it is not necessary. Cirq supports multiple unique cloud services and has a unique compiler language, so it is certainly worth supporting. This leaves us with the Quantum Development Kit, the Amazon Braket Python SDK, and the Quantum Inspire SDK. These all would allow access to one new cloud service, are in active development, and have a unique compiler language. However, due to time constraints, we need to limit ourselves to 3 main translations. As AWS Braket and Azure Quantum promise more future potential due to their financing and integration into web service frameworks by tech giants like Amazon and Microsoft, we decide in favor of the Quantum Development Kit and Amazon Braket Python SDK. Conveniently, all selected frameworks are also available in python.

4.2 Evaluating Translation Frameworks

Now that we are set on which SDKs we want to support, we next have to look into the implementation of the translation between them. This section is split into three parts. We start by specifying our approach to translation, then provide an analysis of the availability and compatibility of frameworks with our selected quantum SDKs and afterward give reasoning to our choices based on these results.

4.2.1 Approach to Translation

Before starting, we have to decide on which formats we want to use as the in- and output of our translation process. Currently, the quantum circuit transpiler supports both python code, which uses the SDKs to define circuits, as well as compiled gate level code. Despite that, we decide to not support translation to python code implementations due to our main goal for this project being to provide compiler code that can be compared and executed.
While it is possible to manually implement translations between all languages, this process is very cumbersome. Additionally, as mentioned in Chapter 2, there already exists a broad range of frameworks that implement the import and export of different formats and even packages dedicated solely to the translation and compilation of various languages. This is why in this project we only implement paths manually that are not yet available and otherwise make use of existing frameworks. On a different note, we do not implement direct translation between languages, since this would result in a superlinear increase in translation paths. Instead, the Quantum Circuit Transpiler, similar to pytket [SDC+20] and PennyLane [BIS+18], utilizes a plug-in architecture [Wan20]. We use Qiskit as the single intermediate format that all circuits are translated to and from. This means that for every additional SDK, only one new service and thus one additional path needs to be implemented. In contrast to pytket, we choose not to implement a unique format for this, as Qiskit is already supported by a large number of frameworks and for a new language no translations would be available. It is also the internal storage format of the Quantum Circuit Transpiler, allowing this functionality to be reused. This means that we only have to analyze translation paths from and to Qiskit for the selected languages.

4.2.2 Selection Criteria and Available Framework Options

Frameworks are chosen mainly based on restrictions on translation, as well as the quality of translated circuits. However, since both of these values cannot be determined purely in advance, it might be necessary to implement multiple paths and compare results before choosing a final framework. Available paths for the selected languages can be seen in Figure 4.1, which is an adjusted sub-graph of the conversion graph presented in Qverview [VN21]. If a compiler language can be both imported and exported using its native SDK, it is included using +. Otherwise, it is listed separately.

4.2.3 Selection and Reasoning

Analogous to the previous analysis, we now narrow down which paths to use in our implementation. The results of this selection are also shown in Figure 4.1. It represents selected paths by a solid arrow. The reasoning behind these choices is explained hereafter. For Cirq, there exists an import and export function for OpenQASM code. While we do use third-party translation frameworks in this project, functionality included as part of the original SDK is preferred, since it is more likely to be up-to-date and maintained. This is why we choose this approach for supporting Cirq. For Q#, the language associated with Microsofts Quantum Development Kit, we have an interesting case. While there are four ways to translate from Qiskit to Q#, there is none for the other direction. This means that the translation from Q# to Qiskit is implemented manually. For the other direction, we have the aforementioned pytket, PennyLane, pystaq, and qconvert. Out of these, we do not

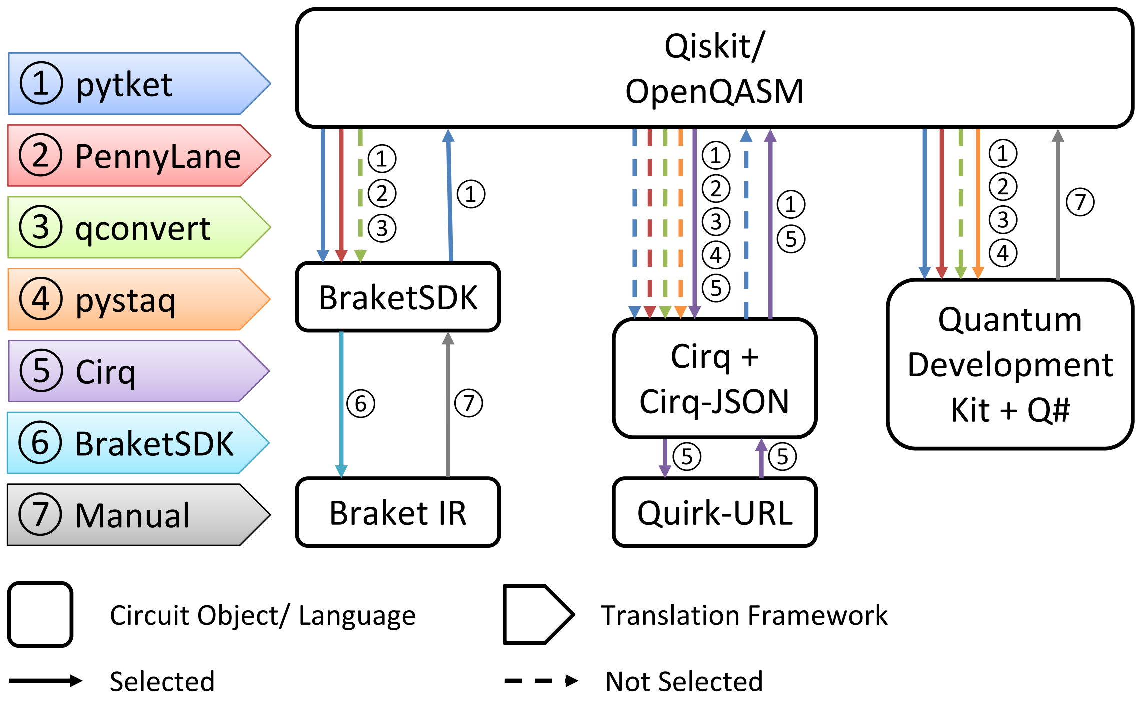


Figure 4.1: Translation Paths Between Selected Languages

implement qconvert [22e]. This is because it only supports an outdated version of Q#, 0.10. With the current Q# build being 0.24, this is not acceptable. Furthermore, the python version of qconvert does not support Q# at all [21], only the JS version does. With various superior alternatives present, we decide to omit qconvert. pytket, PennyLane, and staq do not have any obvious advantages over each other. Thus, all are implemented and evaluated. Next, we take a look at the Braket SDK. When translating from Braket, we are only given one option, which is pytket, so we are implementing it. In the other direction, we are given the options pytket and PennyLane. Again, both are integrated and tested. However, Braket comes with another issue. While all other chosen frameworks allow their exported compiler language to be imported again, Braket does not. Yet, there is no alternative way to export circuits created in Braket. This means that the direction from Brakets intermediate representation back to the SDK is implemented manually. Lastly, while not mentioned in our analysis of vendors, there is an online tool called Quirk, that allows for the easy creation of quantum circuits via a web GUI. Cirq can import and export Quirk URLs. Since we already integrate translation to Cirq, implementing translation to Quirk is very little effort and is supported as well.

4.3 Selected Quantum SDKs

After assessing the providers and SDKs we are working with, we now provide a more detailed look into their operating principles and how to interact with their quantum hardware and simulators.

4.3.1 Qiskit, Cirq, and Braket

The SDKs provided by IBMQ, Google, and Amazon work similarly for the most part and thus are discussed simultaneously [22b; 22d; 22f; AAA+22; Dev21]. All three quantum SDKs are offered in form of a python package that provides the user with the tools to build quantum circuits, as well as compile and then execute or simulate the circuit on their own or different hardware. In this section, we give a short overview of the base operation principles behind these frameworks: At the center of all of these SDKs is an object that represents a quantum circuit. While named differently, the circuit is handled similarly in all frameworks. On creation, a circuit is supplied with qubits and clbits. In Qiskit, this is either done by directly passing a list of bit registers or by simply defining the circuit size in the form of integers. In Cirq, qubits are also defined using registers, but they are independent of the circuit object. In Braket, neither bits need to be defined in advance.

After the creation of the circuit, operations need to be added. The different SDKs offer various ways of doing this, but all allow you to append instruction class objects to the end of the circuit. Qiskit and Braket even offer methods on their circuit object that allow you to skip the creation of the operation and just directly add it to the circuit. In the case of Cirq and Braket, the core data structure of a circuit is called *moments*, where each moment represents a single time slice, and operations are inserted into these slices using different strategies. They include adding to the earliest moment possible or always creating a new moment.

After the state transforming gates are added, the resulting states of the qubits need to be measured. Qiskit and Braket implement a *measure* operation that allows the evaluation of qubits and maps that measurement either to a tag or to some amount of clbits. Braket does not support this. Instead, similar to instructions, it is possible to add *result types* to a circuit. They define how the state of the quantum circuit is evaluated, such as the observable under which to measure the circuit.

While this allows creating the circuit, to retrieve a computation result, it needs to be compiled and executed. For this purpose, the frameworks offer compilers as well as simulators and quantum devices that can execute quantum circuit objects. These objects are called *backend* (Qiskit), *engine* (Cirq), or *device* (Braket) but all serve a similar purpose. They are created by defining which QPU or simulator the user wishes to use and can then be passed a quantum circuit object to execute as well as the number of shots to take and other meta parameters. In addition, local simulators that simulate the execution on local hardware and use a simplified creation process can be created. Note that while circuits can be compiled automatically before execution, the frameworks offer functionality that allows you to manually do the compilation for specific devices or even optimize the circuit before execution. The results of execution are the frequencies of measurements over the defined amount of shots. They can be represented, for example, in the form of a histogram. In the case of simulators, access to otherwise inaccessible information like the complete state vector can be granted. A demonstration of this process is shown in Listing A.1. Its results can be viewed in Listing A.2.

4.3.2 The Quantum Development Kit and Q#

The next framework we support is Microsoft's Quantum Development Kit, which is part of Azure Quantum [22c]. As its mode of operation is rather different from the other SDKs, it is discussed separately. The main reason for this is that it does not allow the definition of quantum circuits in object representation at all. Instead, it separates between a definition language, Q#, in which circuits are written and the package qsharp which allows host languages like python and C# to call and execute circuits. Circuits can be written exclusively in the Q# language. A host language, such as python with the qsharp package installed, can be used to execute the defined operations but is not able to add new gates or modify the circuits. There is no quantum circuit object and in general, the functionality available in the host language is rather sparse. Instead, at the center of the Quantum Development Kit is Q#. It is used to define *operations*, which are methods that can be called from inside and outside of the file. These methods can contain quantum operations, but also non-quantum operations. Base functionalities are imported from preexisting Q# libraries that provide everything from maths to quantum gates. Again, even in these files, no quantum circuit object is used. Instead, qubits are defined individually and operations are defined directly on them. An example circuit can be seen in Listing A.8

Q# files can be executed directly via dotnet using a CLI. However, we mostly use Q# in combination with its package qsharp in the host language python. Its main functionality is to import functions from Q# files and simulate them, as can be seen in Listing A.3. Different simulators are addressed by using different methods. Other functionalities include getting information on circuits, and, essential for this project, compiling operations from strings. This allows the use of the qsharp package without necessarily having to create Q# files.

4.3.3 Quirk

Finally, we take a look at Quirk [Gid16]. While it is not an SDK in the measure of the aforementioned candidates, it still is interesting to support, because of its value as an introductory simulator that allows experimentation with small quantum circuits. Quirk's main features are drag-and-drop circuit editing and a simulator that reacts to modifications of the circuit in real-time. It is mainly accessible using its GUI, which can be built locally, but is also available and hosted online[1]. It however does not have any integration with a host language like python, limiting its use cases. There is a function for exporting circuits in the shape of a JSON, but since Quirk encodes the currently created circuit in its URL, it also offers a feature to export this URL in a safe format for sharing and reusing. An example circuit is shown in Figure 4.2. Exporting it returns the URL `https://algassert.com/quirk#circuit=%7B%22cols%22%3A%5B%5B%22H%22%5D%2C%5B%22%E2%80%A2%22%2C%22X%22%5D%2C%5B%22Measure%22%2C%22Measure%22%5D%5D%7D` which can then be opened to view and edit the circuit again.

4.4 Selected Quantum Circuit Languages

While quantum circuits can be created using the aforementioned SDKs, having them in the form of objects, although useful for creation and usage in code, is not suitable for both saving and sharing applications or even execution. This is why most of the quantum computing providers offer a separate language in combination with their SDK that their objects can be exported to. And even though the frameworks are rather similar, these languages all differ, be it in the syntax used to define

[1] `https://algassert.com/quirk`

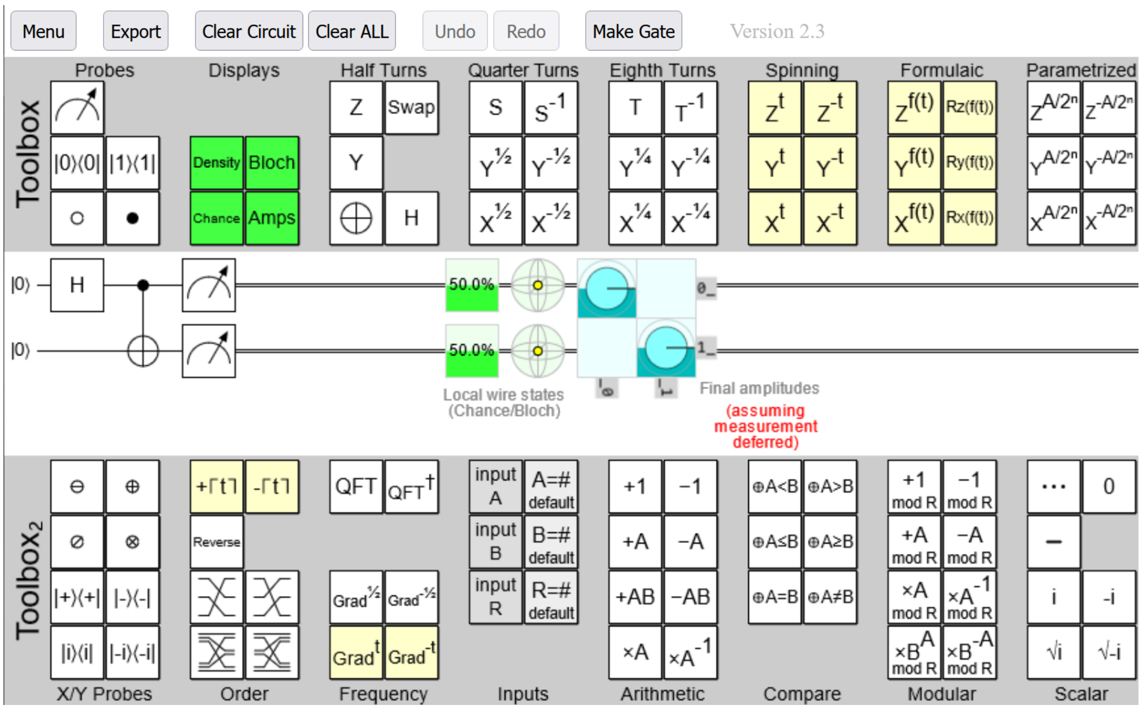


Figure 4.2: A Simple Circuit in Quirk

operations or the whole data format used. To us, they are very relevant, since they are the format that most users have quantum circuits available in. However, since most frameworks also allow the import of their languages into quantum circuit objects with one-to-one gate support, this translation is rather trivial and not a major problem. We still give a brief overview of the different languages, except Q# which has already been discussed.

4.4.1 OpenQASM

OpenQASM is the quantum circuit language released for use with IBMQ [CBSG17; CJA+21]. Thus, it can be easily ported in and out of Qiskit. It is used as the compiler code that circuits are sent to the QPU in. The current version is OpenQASM 2.0, with OpenQASM 3.0 being in pre-release since 2020. It uses a syntax that resembles parts of C and assembly languages. A file always starts by defining the current OpenQASM version. After that, other source files can be included. Qubits and clbits are defined either individually or as registers. Gates and operations such as measurements, resets or barriers are defined on these bits, with each line corresponding to one operation. Custom gates can also be defined as a sequence of other gates. A simple example of an OpenQASM circuit can be seen in Listing A.9.

4.4.2 Cirq JSON

While IBMQ has a unique language for its representation of compiled quantum circuits and Q# is purely based on such a language, this is not necessarily the standard for all providers. Google, for example, does not have a unique language that circuits created in Cirq can be exported to. However, some sort of serializable representation is necessary, at the latest when it comes to communicating with quantum hardware. In the case of Cirq, a simple JSON format is used [Dev21]. It represents the moment structure Cirq is based on, first listing all included moments. These then, in turn, contain a list of operations with corresponding targets. In addition, supplementary data such as the target device are defined as part of the JSON. As can be seen in Listing A.10, this sort of representation is rather spacious and in contrast to OpenQASM not easily readable for humans. However, as it can be directly ported in and out of Cirq, it is still the best representation of circuits for that vendor.

4.4.3 Braket Intermediate Representation

Similar to Google, Amazon also does not support a unique export language for their Braket SDK. Instead, there is something called the Braket Intermediate Representation (IR), which is the format quantum circuits are converted to before they are sent to the Amazon API for execution [22a]. This is again a JSON representation of the quantum circuit. As Braket's circuit objects are simpler than Cirq's, the resulting JSON file is also a bit simpler, as can be seen in Listing A.11. It consists of a simple list of used gates with their associated qubits. Since Braket has *result types* instead of measurement instructions, there is a separate section in the JSON describing how the circuit should be measured.

4.5 Selected Translation Frameworks

While we already mentioned all the frameworks we are using for translation in Chapter 2, we only described them superficially. Nonetheless, to follow how we implement translation using them, it is very advantageous to get a deeper understanding of their mode of operation. In the following sections, we describe how the packages associated with the frameworks work, with our focus for the largest part being the functionalities that allow us to use them for translation.

4.5.1 pytket

pytket is the package that allows accessing the t|ket⟩ compiler via python [SDC+20]. As mentioned in Chapter 2, translation from and to pytket is implemented in its plug-ins. The use of these plug-ins is twofold. On the one side, they allow the import of their associated language. On the other side, they allow pytket to simulate or execute on their vendor's devices, which means exporting to their language again. The pytket package itself works very similar to Qiskit and the other SDKs, with circuits being represented as python objects that can be compiled for devices and then executed. However, since pytket is platform-agnostic, circuits can be compiled for and executed on devices from all available plug-ins. For our project, the most important functionality lies within the plug-ins themselves and their methods which translate objects from the language or object representation supported by the plug-in to a pytket circuit and the other way around. As seen in Listing A.4, this translation process is easily accessible. It is important to note though that the pytket circuit has its own gate set that is not necessarily compatible with the source language and thus compilation to suitable gate sets might be required before translation. It is also important to note that not all extensions must support both directions. The pytket-qsharp package for example does only support translation to, but not from Q#.

4.5.2 PennyLane

As indicated in Chapter 2, PennyLane's core focus is not actually translation or even compilation for multiple vendors [BIS+18]. This results in a structure that is rather different from the one used in pytket and the aforementioned quantum SDKs. PennyLane uses special python functions with annotations to define its quantum functions. Each function includes a list of operations. *Wires* are used to represent the qubits of the circuit. Measurements can only be taken as the final step of the circuit and are represented by the return values of the function. To make a circuit executable, it needs a device. This device function pair is called a *node* and can be executed. An example node can be seen in Listing A.5.
While we do not want to manipulate circuits in PennyLane at all in this project, this principle is still very important to understand, because of the way importing and exporting works using it. In PennyLane, these two processes are very different, so we go over them one by one. Both however are reliant on the plug-ins of the languages that are supposed to be imported and exported being installed. For importing, PennyLane, if used with the PennyLane-Qiskit or PennyLane-Forest plug-ins, offers functions for converting circuits from these frameworks' objects or languages into PennyLane subcircuits. These can be called inside a PennyLane node to execute that circuit. If no other circuit is included in the node, it corresponds to the translated circuit. As measurements in PennyLane are the return values, it is important to note that no measurements can be imported into a node. On the other hand, since it is required to have at least one return value, each circuit needs to have some measurement added to it in the end. Listing A.6 shows an example implementation of this process.
Although importing a circuit is only supported for two SDKs, the process is straightforward and well documented. Exporting on the other hand supports a wider range of providers, but is not always as easily accessible. This is because exporting is purely reliant on the plug-in used. In contrast to pytket, plug-ins in PennyLane are not uniform. Outside of the base functionality of providing a device, they are rather distinct. The PennyLane-Qiskit plug-in, for example, offers a *compile* method on its devices that can be used to translate PennyLane nodes to Qiskit QuantumCircuits, but this is not the case for all plug-ins. This means that for other languages different approaches must be taken.

4.5.3 pystaq

Last in the list of translation frameworks we selected is pystaq, the python wrapper of the CLI tool staq [22i; AG20]. Since staq only supports a single import language, OpenQASM, importing in pystaq is done in a single function, either *.parse_str* or *.parse_file*, which returns a pystaq program. pystaq then offers functions that correspond to CLI commands, such as *.simplify* or *.estimate_resources*, that take as input a program, followed by the options the staq tool would normally have. Among these are functions named *.output_LANGUAGE*, that export the program to the target language. Thus, translation using pystaq is very simple, as can be seen in Listing A.7.

5 Design and Implementation

In Chapter 3 we described the projects this work is based on. Later in Chapter 4, we analyzed both what we want to implement, as well as what can be used to do so. In this chapter, we now discuss how we utilize these findings in our implementation. To accomplish this, we start by giving an overview of the project architecture, showing dependencies as well as what components we added or modified in the scope of this project. This facilitates understanding the relations the changes have to each other and the general structure we build upon. We then go into more detail on the individual changes, describing the ideas and structures on which our implementation is based.

5.1 Architectural Overview

Figure 5.1 presents the architecture and dependencies of the components relevant to our project. As is evident from the diagram, we not only extend about half of the components shown in some way but also add a new service to the architecture. The baseline is the extension of the Quantum Circuit Transpiler. For the other services to support more languages, translation from and to them must be made possible and this functionality is included in the circuit transformer. The new transpilation paths are implemented based on the results of Chapter 4. Unavailable paths are manually implemented by using new and existing gate mappings. The Representational State

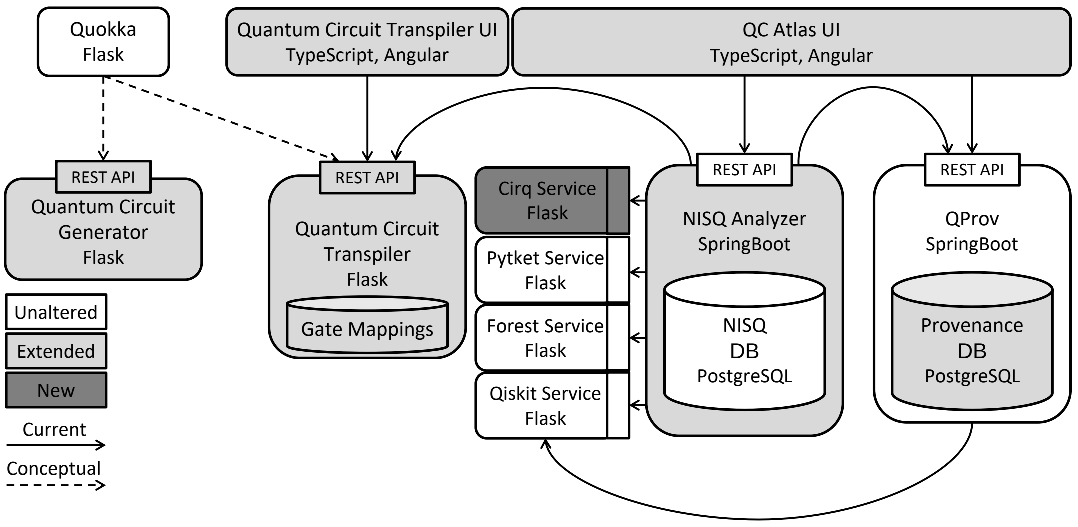


Figure 5.1: Architecture Diagram of Relevant Components Included in the QuAntiL Repository - Adapted From [22g]

Transfer (REST) API is also extended to make these new paths available for use. Similarly, we adapt the Quantum Circuit Transpiler's GUI to allow conversion of the new languages.
The next objective is to use these paths to add new supported languages to the NISQ Analyzer. This is first implemented for Cirq as a proof-of-concept. For this, we expand upon the analyzer's implementation. Since it outsources compilation and execution to multiple smaller services, we add a new one to the list, the Cirq Service, which is complementary to the other services and provides functionality to compile for and simulate execution on Google QPUs. In addition, the NISQ Analyzer also relies on QPU data sourced from the provenance database QProv. For the NISQ Analyzer to take the new QPUs into account, data regarding the devices supported by the new Cirq Service are injected into QProv. For the other languages, only translation, but neither compilation, analysis, nor execution are added, so their devices are not included in the database. The new functionalities are now available from the REST API provided by the analyzer, but not yet from any frontend. To achieve this, the QC Atlas UI, which offers access to the NISQ Analyzer's functionalities is also fitted to accommodate the new languages. Now all functionalities are integrated into QuAntiL's main structure.
For the last feature, the Quantum Circuit Generator is extended by adding the Quantum Fourier Transform (QFT) to its circuit library. Its API is also extended to allow access to this new circuit. Both the Quantum Circuit Generator and the Quantum Circuit Transpiler are also conceptually integrated into Quokka.

5.2 Implementation Details

After describing how the features we implement relate to each other in the project architecture and quickly going over what was implemented, we examine the different components and describe how we realize the aforementioned functionalities in this section.

5.2.1 Quantum Circuit Transpiler

Implementing the translation functionality of the Quantum Transpiler is done in accordance with the results of Chapter 4. During our selection, we further concluded that some paths are not implemented yet and thus need manual transpilation. We now outline the design principles we apply for our manual translations. Subsequently, we specify how both our earlier results and these new principles are applied to implement service-based translation.

Concept for Transpilation

The foundation we work on when implementing transpilation is the fact that on both sides of the process are expressions that represent gate-based quantum circuits, and that every gate has a corresponding transformation matrix. At its core, a quantum circuit is nothing more than one large transformation matrix. Decomposition allows us to represent a matrix as a product of other matrices. Be it a file or a python object, if both represent a matrix and both provide several sub-matrices (gates) sufficient to decompose arbitrary translation matrices, they can also represent the same circuit [NC01]. This way, translation between them is possible. Even though both sides allow representations of the same circuit, it is in no way guaranteed that these representations are similar.

The types of gates provided, as well as their names, are most likely not identical. In addition, the base structure of circuits and qubits might vary from representation to representation. This means the same circuit in some cases has a different depth, width, and types of gates in different languages. Hence, to facilitate translation, we define equivalences of operations between our languages. There is a large number of standard gates that are supported by most frameworks that are directly translated. Similarly, operations such as resets, barriers, and measurements are also translated directly. For gates not supported by the target language, decomposition gives us a way to represent them in terms of standard gates that then in turn can be translated. Difficulties mostly arise when languages include higher-level features, that are not part of the quantum operation. These are likely not compatible with both languages and even if they are, there is no tool taking the role of the decomposition available, which would allow us to translate them. For quantum gates, we use a mapping structure to facilitate transpilation. If every operation is mapped to an operation with an equivalent transformation matrix, the resulting circuit is also equivalent. If gates or operations are included in both languages, they are mapped one to one trivially guaranteeing matrix equivalence. Given that the target language does not offer a gate, its transformation matrix is decomposed to a set of gates available in the target language. If the gate is translated, this replacement circuit takes its position. Since they have equivalent matrices, the resulting circuit is also equivalent. In the case that the target language allows custom gate definitions, we just define a new gate with the matrix of the gate we want to translate. Lastly, we map the qubits and clbits of the different representations to each other, so the targets of our operations remain unaltered.

Implementation of Transpilation

Transpilation by the aforementioned principle is already implemented in the Quantum Circuit Transpiler to facilitate Translation between Qiskit and PyQuil, complete with a mapping table of gates from Qiskit and PyQuil to each other. To realize the translation from Q# to our intermediate representation of Qiskit, we reuse this table. Since the Qiskit gates are already included, only one extra column is added. The complete translation process is laid out in a simplified form in Figure 5.2. Before we start the mapping process, the gates that represent the circuit are extracted from the Q# code. For this purpose, we employ the qsharp python library. It offers functionality to return a tree structure representing both a decomposition of the circuit to the set of gates necessary to execute it, as well as the qubits included in the circuit. Thereby, qubits can be directly adopted. The graph is then recursively traced, with recursion terminating when we find an operation that is included in our mapping. These gates are saved in a list. In the final step, we iterate over this list and map each gate to an equivalent one in Qiskit. Since the tree already is a decomposition, we do not need to manually decompose any gates.
For the translation from Braket IR to the Braket SDK's objects, we use a similar procedure, except that this time no mapping data structure is necessary. We work with two representations of the same vendor, which have almost the same structure, gates, and designations. Gate mapping thus is simply done by name. Braket's result types can also directly be taken over. The few exceptions are individually addressed.
The other paths are implemented and evaluated by integrating the frameworks selected for them in Chapter 4. If necessary, we also implement pre and postprocessing for the circuits. This includes compilation to a certain gate set to avoid unsupported gates, but also in some cases more specific adjustments. For Cirq and Braket, code written in python that creates a quantum circuit object can also be used as an input. A unique translation path is necessary for Quirk, since here also no direct

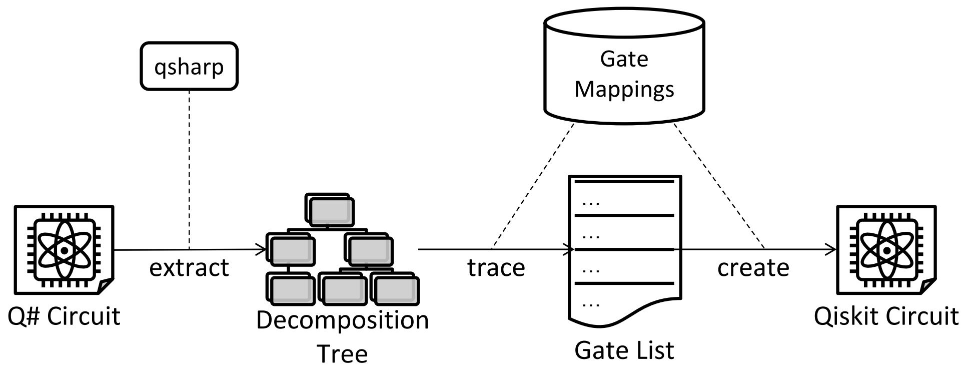


Figure 5.2: Translation From Q#

path is available. However, it is simply implemented by first translating to Cirq, and then exporting to a Quirk URL.
These new translation paths are then integrated into the API and thus made available to be called by other services like the Quantum Circuit Transpilers' Angular GUI. Here only some minor adjustments are necessary to make the new languages available for all import, export, and conversion.

5.2.2 NISQ Analyzer and QC Atlas UI

As is clear from the architectural overview, the NISQ Analyzer is the most interconnected component we are working with. It not only depends on the Quantum Circuit Transpiler but also on SDK-specific compiler services and QProv. In this section, we detail the relation of the NISQ Analyzer to its sub-components and then briefly describe our approach to extending it.

Design of the NISQ Analyzer

Figure 5.3 is an illustration of the NISQ Analyzer and the services it communicates with. For simplification purposes, we only cover services that are important to our project. To help understand how the tool deals with compilation and execution requests, we describe the process necessary to handle them. First, the NISQ Analyzer has a main control service that offers an API from which compilation can be requested. If such a request reaches the analyzer, it uses its QProv Connector to query QProv. It retrieves information on the QPUs that are available for the provider included in the request from its database. If the QPU is specifically defined, it only gets its information. Subsequently, it is reviewed what services can compile for the selected QPUs and in consequence which circuit languages they support. If the circuit currently is not in a language supported by one of the services, a translation request is formed using the Translator Service, which requests translation from the Quantum Transpiler's API. Now the circuit is present in a language compatible with the compiler. However, the NISQ Analyzer itself is implemented in Java, while almost all quantum frameworks are python-based. This is why the compilers are outsourced to their complimentary services. The analyzer has a connector for each service, that propagates the compilation request to the service. They compile the circuit for the chosen QPUs and return the compiled versions. These results are then made available via the analyzer's API. Subsequently, the user can request the

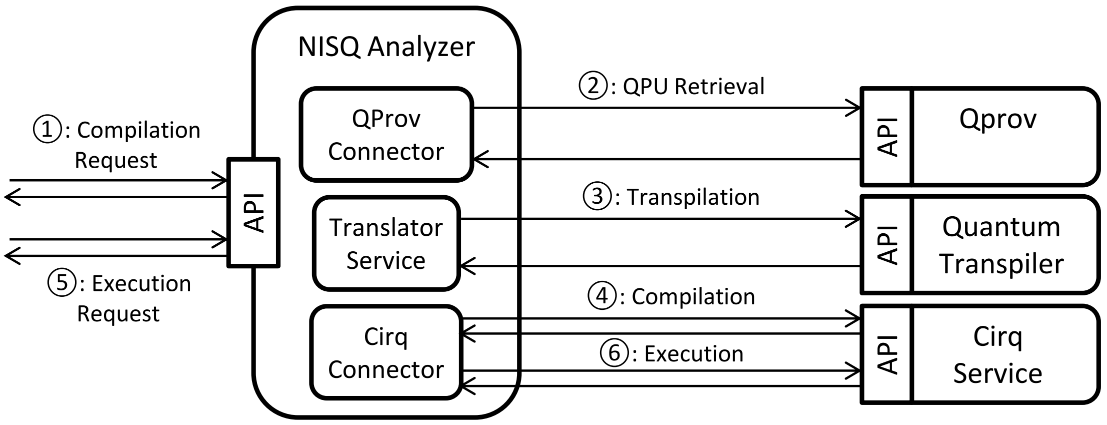


Figure 5.3: Detailed Working Scheme of the NISQ Analyzer

execution of the compilation results. Execution is also handled by the complementary services, so the request is again propagated using the associated connector, and results are made available in a similar manner to compilation.

Implementation of Cirq Compilation and Simulation

We extend the Quantum Circuit Transpiler by four languages and make all four languages available to the NISQ Analyzer to translate to and from. However, due to time constraints, we only implement compilation and execution functionality for one of the SDKs, namely Cirq. We choose Cirq since it provides the best analysis functionality out of the new frameworks, which comes in handy when returning details on the compilation results to the analyzer. For the NISQ Analyzer to even consider this new compiler, we first extend the QProv's database to include information on the four Google QPUs Sycamore, Sycamore23, Bristlecone, and Foxtail. Now, information on them can be retrieved by the Analyzer when querying the provider Cirq-Google. For compilation and execution, we create a new service analogous to the existing services, using the required functionality from the Cirq SDK. Importantly though, since we do not have access to the actual Google devices, execution is currently only simulated. Similarly, we also implement a new Cirq Connector that allows the analyzer to communicate with our new service. Now the new functionality is available via the REST API. Since the NISQ Analyzer receives requests from the QC Atlas UI, we implement minor adjustments for it to consider all new languages, Cirq as a compiler, and the Google QPUs as target devices.

5.2.3 Quantum Circuit Generator

For our last implementation goal, we extend the Quantum Circuit Generator's circuit library. In particular, we want the QFT and its inverted form to be available via the API. In the following section, we briefly discuss the QFT, followed by a summary of our implementation approach.

The Quantum Fourier Transform

The QFT is the quantum implementation of the discrete Fourier transform [NC01]. At its core, the Fourier transform is a basis transformation of a signal into the basis of its frequencies. In quantum computing, we transform from the computational basis (Z) to the Fourier basis. In the computational basis, we encode binary numbers as a sequence of the states $|0\rangle$ and $|1\rangle$. In the Fourier basis, we store them as rotations around the Z-axis. Consequently, the Quantum Fourier transform is an n qubit operation that operates on the amplitudes of qubits. It is usually implemented using Hadamard H and controlled rotation CROT_k gates. The Fourier transform is one of the most important quantum operations, with applications in quantum phase estimation and Shor's algorithm to name only a few.

Quantum Fourier Transform Implementation

We implement the QFT by using Qiskit's circuit library. It allows the creation of the circuit for a selected width and approximation degree. It can also be inverted by simply inverting the order of all gates. This new algorithm is added to the generator's API. A request needs to include information on the number of qubits, the approximation degree, and whether or not to invert the circuit. Returned is the generated circuit as a OpenQASM string, as well as information on its size, the generation input, and a timestamp.

6 Evaluation

While describing the concepts used in implementation is essential, it is equally important to evaluate the implemented features. This proves that they provide their intended function as described. In this chapter, we take a detailed look at the transpilation functionality provided by the Quantum Circuit Transpiler. It forms the core of most features implemented in this project and therefore is especially important to it. Therefore it is thoroughly evaluated to ensure the correct function of both itself and the components depending on it. In the following sections, we first describe the design of our evaluation regarding this core component and afterward detail the results we obtain when applying the aforementioned design.

6.1 Concept of Evaluation

When it comes to evaluating translation functionality, we are confronted with the question: "What is the correct translation of a quantum circuit?" Answers to this might include "The most one-to-one mapping of gates to the target language," or "The most compact representation of the circuit possible in the target language." For this project, we define equality of circuits as the equality of the quantum operations they represent. This means the gates and operations defining each circuit are currently not relevant to us, instead we only look at the circuit as a whole. This leads us to the next question: "How do we test, whether two circuits implement the same quantum operation?" The simple answer to this is to create the complete transformation matrices for both circuits and test their equivalence. However, there are multiple issues with this method. First, while there exist various approaches, all of them are rather complex and thus the effort necessary to apply them is not reasonable, especially since we want to evaluate a large number of circuits. The implementation of these algorithms is also not easily available. In addition, these algorithms check for the exact equality of the circuits. However, if both circuits approximate the same function to a very high degree, it would satisfy our criteria of equality of operations, as they would not be differentiable. In the past, there have been efforts to show that simulation is a powerful tool for checking the equivalence of quantum circuits [BW20]. This is due to the fact, that simulation is nothing more than the multiplication of the initial state vector with the transformation matrix that is the circuit. If the resulting vectors for both matrices are equal for a large number of initializations, then both matrices most likely define a sufficiently similar operation. While it is also stated that this comparison alone is not definitive proof of equality, for our purposes and constraints, we accept this limitation. Because we work with results of quantum circuits, which do not have to be deterministic, we compare the histograms of results that we obtain over a large amount of execution to compare the results of the circuits.
Our evaluation approach is based on random circuit generation. We use Qiskit to generate randomized circuits with varying width and depth. These circuits contain gates with up to three qubits. Initialization is not done manually, since by definition the circuits are random, and so is their initialization. For each implemented translation path, we translate the circuits to the new language

and then translate them back again to Qiskit. All of the original, translated, and restored circuits are then simulated. We measure all qubits, creating measurement histograms. For comparison, we use histogram intersection, a similarity measure usually applied to color histograms of images [SB91]. It is a simple comparison algorithm that adds the minima of the values of both histograms and divides it by the sum of all values of the original histogram to calculate similarity. This similarity is averaged over a certain amount of circuits to mitigate outliers. Of course, for the translation to be correct, it has to be possible first and foremost. So if the translation of a circuit leads to an error, it is noted, and the overall error rate is recorded.
As the description indicates, our approach is not static, but dependent on the number of circuits we create, the shots we use when executing, the size of the circuits we use, and which language we are evaluating. Since evaluating all of this together is not practical, we use an approach where we always only evaluate a subsection of parameters while keeping the others static or random.
Finally, we also carry out supplementary manual tests of all paths to detect issues not covered by the systematical evaluation.

6.2 Evaluation Results

Because quantum circuits are inherently not deterministic, the first parameter we vary is the number of shots we use when executing the circuits. As the simulation is done locally and all frameworks operate similarly, we only repeat this for one language. We choose Cirq and record similarities dependent on the number of shots used. The results are presented in Figure 6.1. The amount of circuits is fixed at 50 per run due to run time limitations. Circuits have random widths and depths between one and six. The success rate is not relevant here, since it is determined before the circuit is executed at all, so it is independent of shots.
The average similarity of the results for both the once and twice translated circuit to the original circuit rises with the increased shot amount. This is to be expected, as by the law of large numbers, with increasing shots, we approximate the actual distribution better and thus dissimilarity caused by variation is eliminated. We can also see that the overall similarity is very satisfactory, reaching 99% at about 3000 shots. This means that the results of our circuits are 99% equal and consequently our circuits are also satisfactorily similar. We have to note that this result is only relevant to the Cirq framework. Nonetheless, since shots work identical in most frameworks, their impact on similarity is also similar. Thus, we now know that fixing the shots at around 3000 is enough to reach a satisfactory similarity.
However, while we could just calculate the overall similarity at 3000 shots and 50 circuits for the other frameworks as well, these results are not really of much value. Instead, we want to see in which way the size of the circuit influences this similarity, to find more unique aspects of the different translation paths. This is why we choose to calculate the average similarity dependent on circuit depth and width. As we are still limited by run time, we take into consideration circuits up to the value of eight in both parameters. Simultaneously, we also evaluate the success rate of translation and execution. In the following sections, we go over the different SDKs, specifying the individual evaluation process, and afterward present and discuss the results.

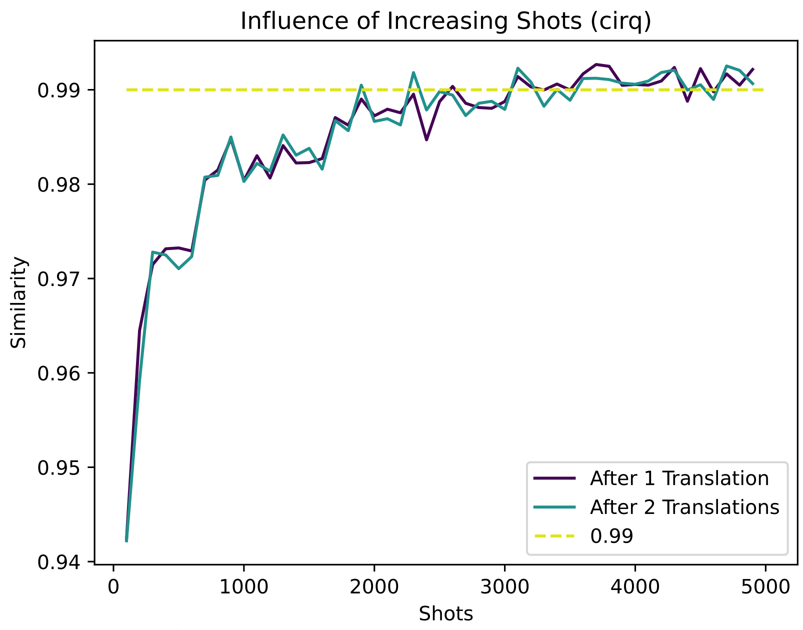


Figure 6.1: Average Similarity by Shot Amount

6.2.1 Evaluation of Translation to and From Cirq

When evaluating similarity for Cirq, we choose to use 3000 shots, since this is the value that previously allowed us to reach a 99% similarity. As before we use 50 random circuits for each size since we still cannot afford more due to run time limitations. As can be seen in Figure 6.2, the similarity is overall great, both after one and two translations. It decreases slightly when reaching larger circuits. The influence of depth and width is also very similar. Both graphs are also very akin to each other. Figure 6.2c shows the success rate of translation and execution to Cirq. Here we use twice the amount of circuits but only a single simulation since we do not care about the actual results. The rate is overall almost perfect, with small declines around the depth of 2.
Decreases in similarity can be explained by the inclusion of more and larger gates, which often are split up into smaller ones during translation, causing very minor differences during simulation. Since the change from one to two translations is minimal, we can also deduct that the restoration back to Qiskit is very reliable with minimum impact on the circuit. The declines in success rate are mostly caused by measurement issues. As translation can change the position of measurements, we add measurements manually in a final step after all translations are done. However, Qiskit random circuits can include empty qubits to which no operations are applied. They are ignored and thus lost during translation because they have no impact on the functionality and therefore lead to different measurement histograms, even if the operations of the circuits are identical. This happens mostly at around the depth of 2 since here we start getting empty qubits, but an even larger depth would mean more chances for operations to appear on them. Nonetheless, it might also happen in other sizes, since it is purely dependent on chance.
Manual testing mostly confirms the documentation of the import function, which states that neither barriers nor conditionals are supported for translation to Cirq [22d]. In addition, we also find that certain double-controlled gates are not compatible with translation back to Qiskit, but this is discussed in more detail in the evaluation regarding Quirk.

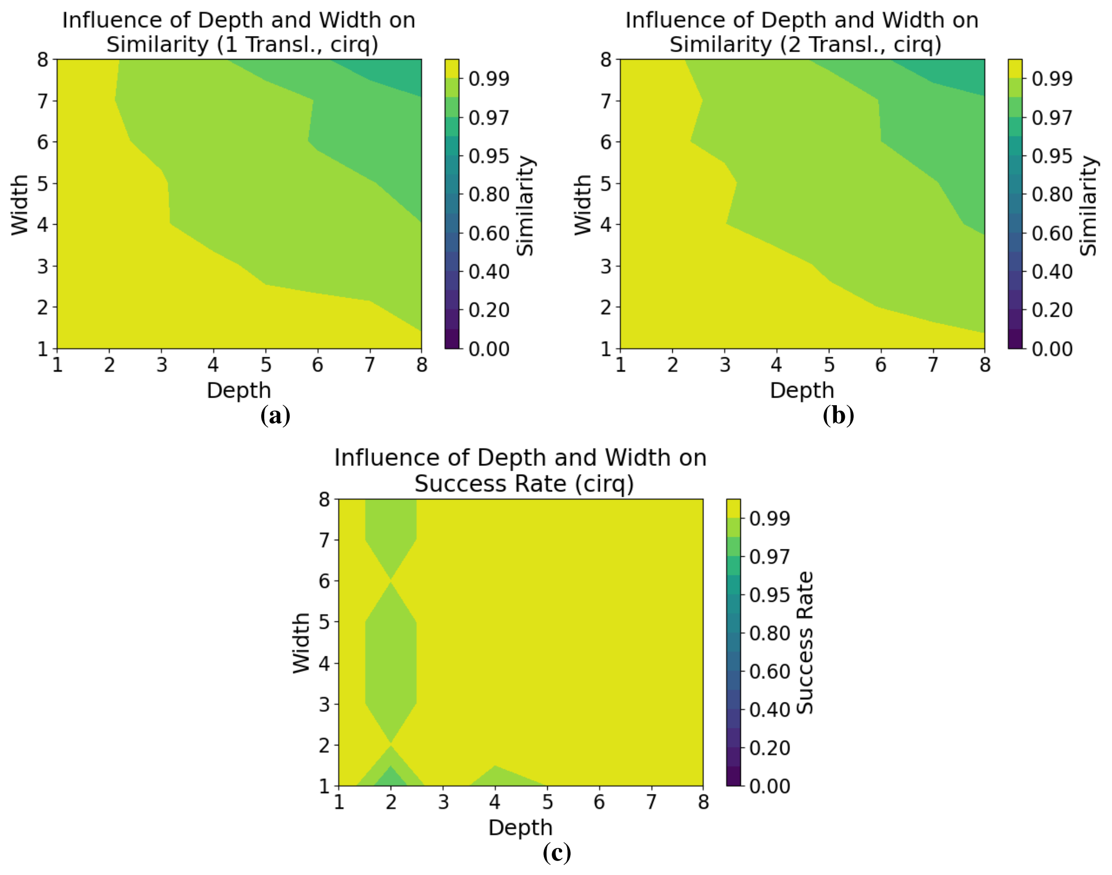


Figure 6.2: Similarities and Success Rates for Cirq

6.2.2 Evaluation of Translation to and From the Amazon Braket SDK

For Braket, we use parameters identical to those we use for Cirq. The main difference is that this time, we have two frameworks to work with. As discussed in Chapter 4, we evaluate both individually and then compare their results. It is noteworthy though, that the frameworks are not compatible. This is due to the fact, that PennyLane uses a large amount of custom unitary transformations in its definition of circuits, which pytket does not support. PennyLane itself also does not support translation back to Qiskit. So in the case of PennyLane, only the similarity between the original and translated circuits can be compared. As seen in Figure 6.3a, using pytket results in similar findings to Cirq. Similarity decreases slightly with size but is overall very high. Reasoning similar to Cirq applies here. Figure 6.3b also has a very similar pattern, so translation back to Qiskit works as intended as well.

In contrast, Figure 6.3c shows that using PennyLane results in unsatisfactory similarity as soon as the size starts to increase. Especially with increasing depth, it drops significantly. The width's influence is most relevant at greater depths. Upon inspection, this seems to be due to PennyLane mistranslating certain multi-qubit-gates. This is also the reason why at width one we still get high similarities at all depths. As for the success rate, circuits are translated and executed successfully 100% of the time. While Figure 6.3d is labeled *tket*, the results using PennyLane are identical under the assumption that we do not translate twice. If we do, the rate drops to 0% almost instantly, due to

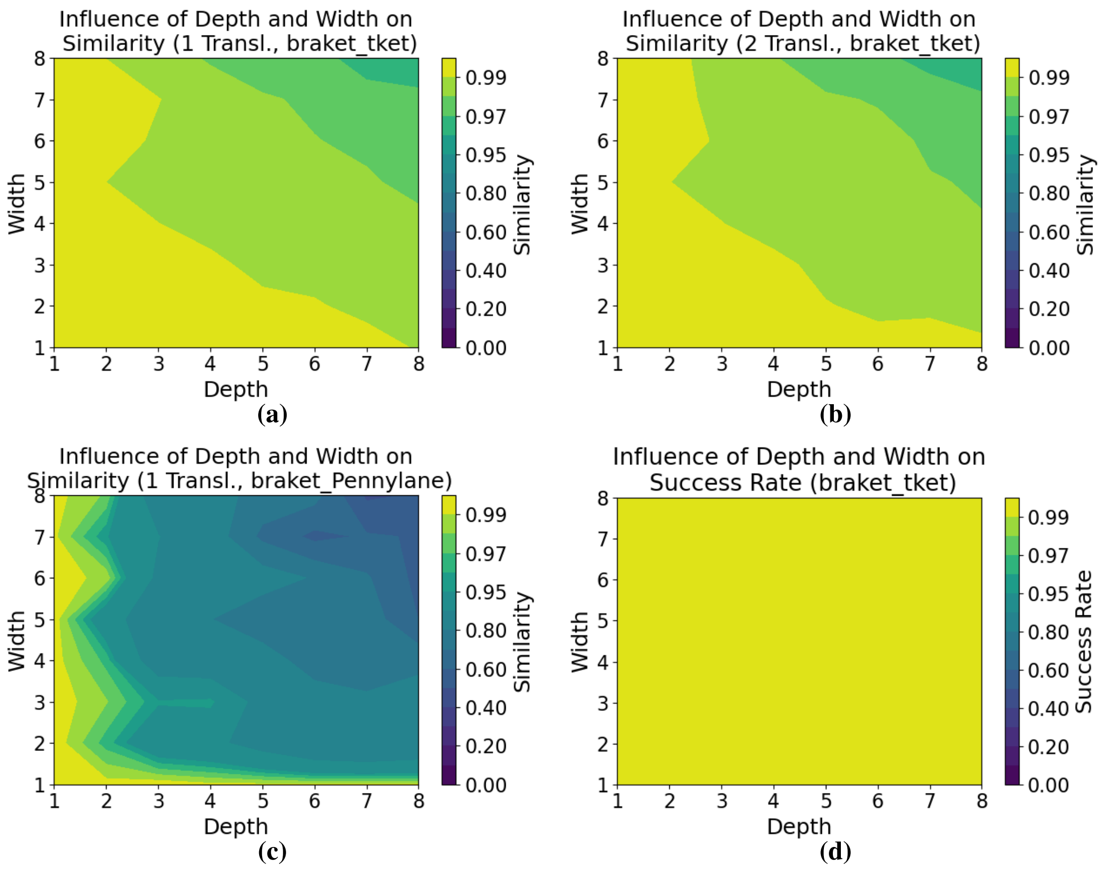


Figure 6.3: Similarities and Success Rates for Braket

the ubiquity of custom unitary gates in PennyLane translations.
Further findings which were obtained during further manual testing include the inability of both frameworks to translate measurements to Braket, as well as the incompatibility of pytket with Braket's Unitary, PPhaseshift, and PSwap gates. Nonetheless, we conclude that using pytket as the framework to translate to and from Braket is favorable.

6.2.3 Evaluation of Translation to and From Q#

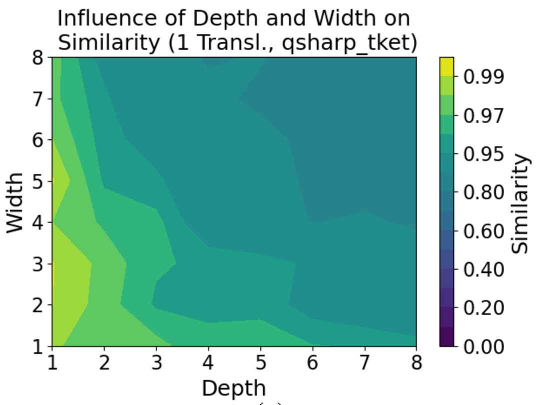
Next, we take a look at translation around Q#. This case is rather special since we have three different frameworks to work with for translation to Q# and the translation from Q# was implemented manually. Additionally, simulation in Q# is remarkably slow. Since both our manual translation and the translation using PennyLane require execution of the circuit as well, the time needed for just one circuit is significantly larger than with the previous paths. Due to time constraints, we, therefore, are limited to 20 circuits and 100 shots for our evaluation. Furthermore, while staq translates measurements, it does so in a way that does not allow accessing them directly. This means that for staq, we only provide measurements in the original and restored circuit.
Figure 6.4a-e show the similarities we recorded for translation to and from Q#. We directly notice that values are in general smaller than they were for translation to and from Braket. This however can be explained by the significant decrease in shots. Both once and twice translated similarities are

again very similar, so the translation back to Qiskit works sufficiently. On further inspection though, we notice that pytket has the best similarity by a large margin whereas staq's and PennyLane's similarities suffer tremendously at greater sizes.
Finding the reason for this however is not trivial, as both PennyLane and staq create greatly inflated circuits that are hard to compare by hand. We can nonetheless prove that the issue is indeed related to the translation and not to our approach to measurements for evaluation. We do this by examining a subset of histograms in detail. The histograms[1] in Figure 6.5 show the counts of an example circuit that resulted in similarity under 50%. If the issue was due to a swap of qubits or change of order in measurements, the histograms would need to be similar, just shifted in some way. What we see here though are completely different histograms. Consequently, the issue is indeed rooted somewhere in the frameworks producing flawed translations for gates of all sizes.
As for the success rates that can be seen in Figure 6.4f-h, we notice that both pytket and PennyLane have very good success rates in general, except for very large circuits, where it seems to drop significantly. In contrast, staq's success rate seems to drop rather randomly. It is to note that, because we are limited by rather small sample sizes, small break-ins might be purely coincidental. Further manual testing yielded only results that were already mentioned, such as PennyLane's inability to translate measurements, the fact that the circuit created by staq cannot be executed without adjustments, and that non-quantum operations cannot be translated from Q# to Qiskit. We conclude that due to its compatibility with measurements and the by far best results for translation similarity, pytket is the most suitable tool for translation in our case and thus is used as the standard. Though, since we are limited by rather small sample sizes, the results regarding the other frameworks are not as devastating as they seem, and they are still made available for access via a special request.

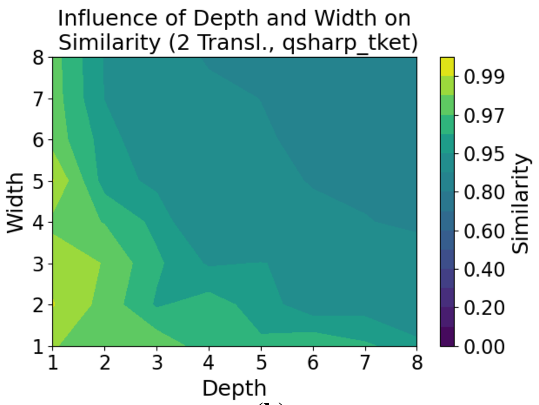
6.2.4 Evaluation of Translation to and from Quirk

Finally, we take a look at Quirk, where our approach is again mostly similar to Cirq, with 3000 shots and 50 circuits for each size. Since Quirk is imported as an URL, we have no means of simulating the actual circuit in Quirk. Instead, we only have the original and restored circuit to work with.
As can be seen from Figure 6.6a, we have a lot of zeroes in this similarity. This is because the success rate of translating from and to Quirk is almost 0 for larger circuits, as is evident from Figure 6.6c. If no circuit is ever successfully executed, it is shown as a similarity of 0. The issue this time is not the execution, but the translation itself. We use Cirq to translate to Quirk and this direction works without errors. Cirq exports circuits to Quirk and also imports them back in. Yet, the circuit gained from this process is different from the original, especially in the definition of three qubit operations. They use a special class that Cirq cannot export to OpenQASM and in turn, not to Qiskit. Since three qubit gates are responsible for this phenomenon, it happens exclusively at widths greater than three. If we turn off three qubits gates in circuit generation, a success rate very similar to the one when translating to Cirq can be observed, as presented in Figure 6.6c.
But this is not the only restriction. As Quirk is a purely graphical tool, it offers a lot of functionalities that are hard to translate. Manual testing shows that this includes anything with classical feedback, as well default values for arithmetic operations.

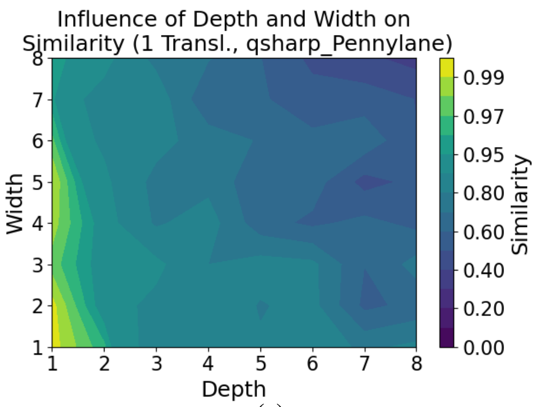
[1]The histograms of the translated circuits are very similar to the ones of the restored circuits, but since we can afford more shots if we skip the execution using qsharp, we choose to do so for this purpose.



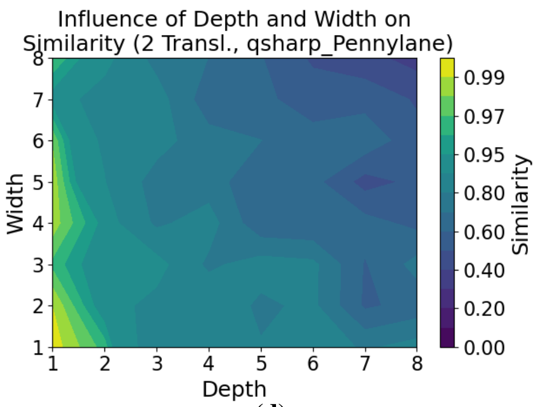
(a)



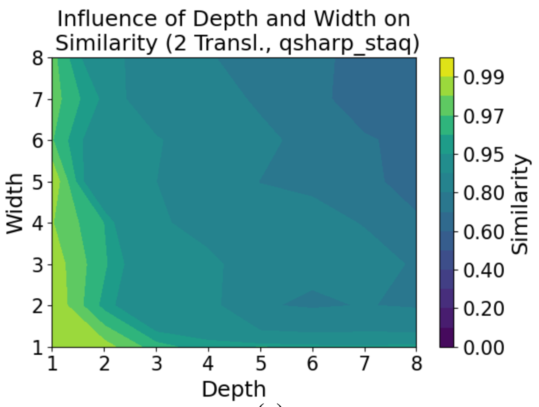
(b)



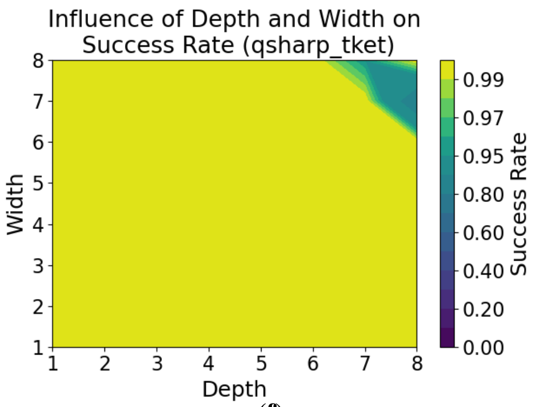
(c)



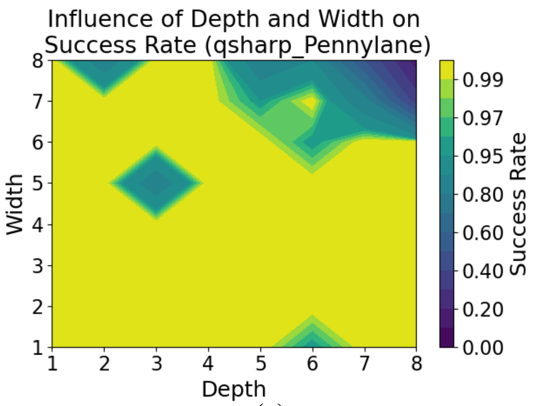
(d)



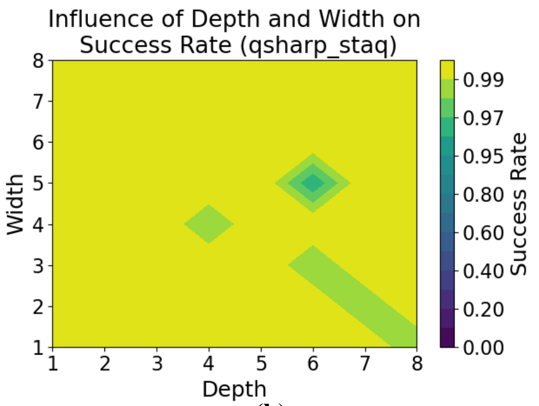
(e)



(f)

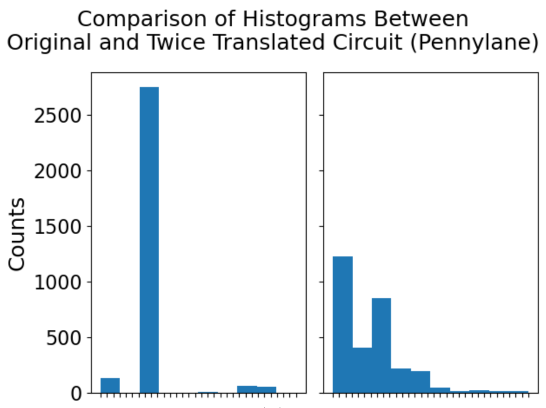


(g)

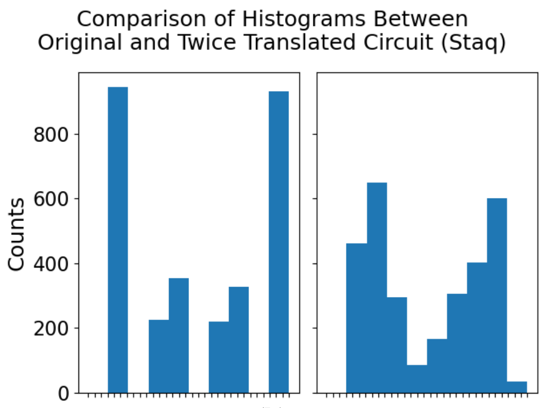


(h)

Figure 6.4: Similarities and Success Rates for Q#

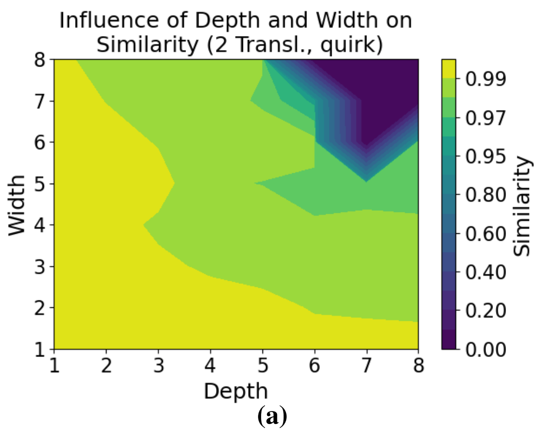


(a)

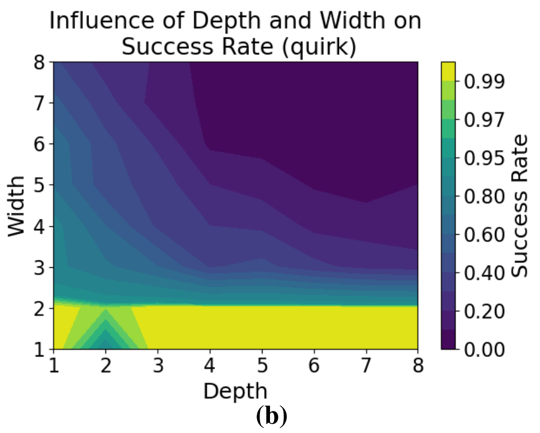


(b)

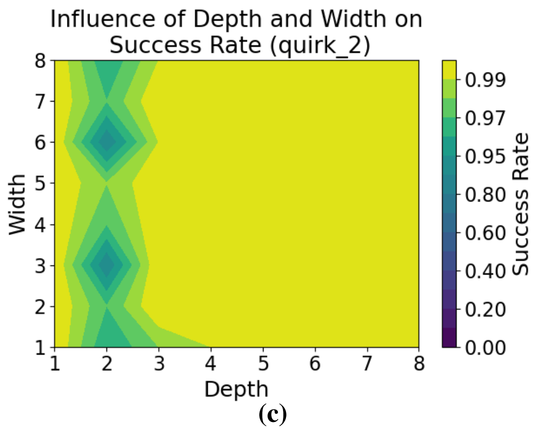
Figure 6.5: Exemplary Histogram Comparison for PennyLane and staq



(a)



(b)



(c)

Figure 6.6: Similarities and Success Rates for Quirk

7 Case Study

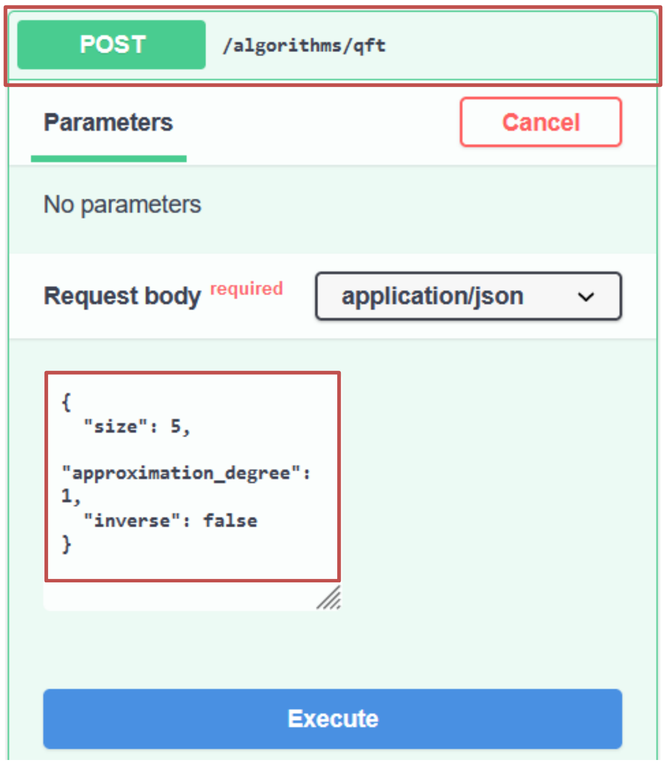
After evaluating the translation functionality and showing its correctness, we now demonstrate its integration into the NISQ Analyzer. As mentioned in Chapter 5, we offer the new SDKs as import options to the Quantum Circuit Transpiler. In addition, we also implemented a new component that allows the compilation and execution of Cirq circuits. To make all these extensions evident, in this chapter we provide an example process of using the NISQ Analyzer, utilizing the new features. To show how the new components play into the NISQ Analyzer's functionality, we use the example of the QFT. We generate a QFT circuit using the Quantum Circuit Generator, then use the NISQ Analyzer to compile it for Google QPUs and afterward execute our chosen compilation on the Cirq simulator.

7.1 Generation of the Quantum Circuit

For the first step, we need to acquire an implementation of the quantum circuit we want to analyze, in this case, the QFT. In this project, we implemented an extension to the Quantum Circuit Generator that adds QFT circuit generation to its functionality, so we use that to generate our circuit. For demonstration purposes, we use a size of 5 qubits with an approximation degree of one. Currently, the best way to access the Quantum Circuit Generator is its Swagger GUI. The corresponding request can be seen in Figure 7.1.
The resulting circuit is given as a OpenQASM string. We add measurements to the circuit to make the results gained upon execution more interesting. To make this circuit usable for the NISQ Analyzer, we upload it to a platform where it can be accessed as raw text. Such an URL is one type of input compatible with the NISQ Analyzer. Listing 7.1 shows the QFT with measurements as a OpenQASM string.

7.2 Creation and Configuration in the QC Atlas UI

For the next big step, we want to request the NISQ Analyzer to compile our circuit for a single QPU. While we could do so via the Swagger GUI, the NISQ Analyzer can be accessed via the QC Atlas UI, which is the intended and thus preferred approach. To use the NISQ Analyzer this way, we first create an algorithm named *Quantum Fourier Transform* and then add to it an implementation. We start by heading to the *Algorithms* tab via the sidebar and clicking the button marked in Figure 7.2 to add a new algorithm. In the algorithm view, we can add various metadata, but more importantly, we can add an implementation to the algorithm, which we can execute. To do so, we head to the *Implementation* tab seen in Figure 7.3 and click on a similar button as we did when creating the algorithm. We name it *QFT (Quantum Circuit Generator)* and can again add meta information if



```
{
  "size": 5,
"approximation_degree":
1,
  "inverse": false
}
```

Figure 7.1: Request a QFT Implementation From the Quantum Circuit Generator

Listing 7.1 Example Implementation of the QFT as an OpenQASM String

```
OPENQASM 2.0;
include "qelib1.inc";
gate qft q0,q1,q2,q3,q4 { h q4; cp(pi/2) q4,q3; cp(pi/4) q4,q2; cp(pi/8) q4,q1; h q3; cp(pi/2)
 q3,q2; cp(pi/4) q3,q1; cp(pi/8) q3,q0; h q2; cp(pi/2) q2,q1; cp(pi/4) q2,q0; h q1; cp(pi/2)
q1,q0; h q0; swap q0,q4; swap q1,q3; }
qreg q[5];
creg meas[5];
qft q[0],q[1],q[2],q[3],q[4];
measure q[0] -> meas[0];
measure q[1] -> meas[1];
measure q[2] -> meas[2];
measure q[3] -> meas[3];
measure q[4] -> meas[4];
```

we wish to. We then open the *Selection Criteria Tab* displayed in Figure 7.4. Here we configure the file location to be the URL of the OpenQASM file we generated and uploaded. Correspondingly we choose OpenQASM as the circuit language.

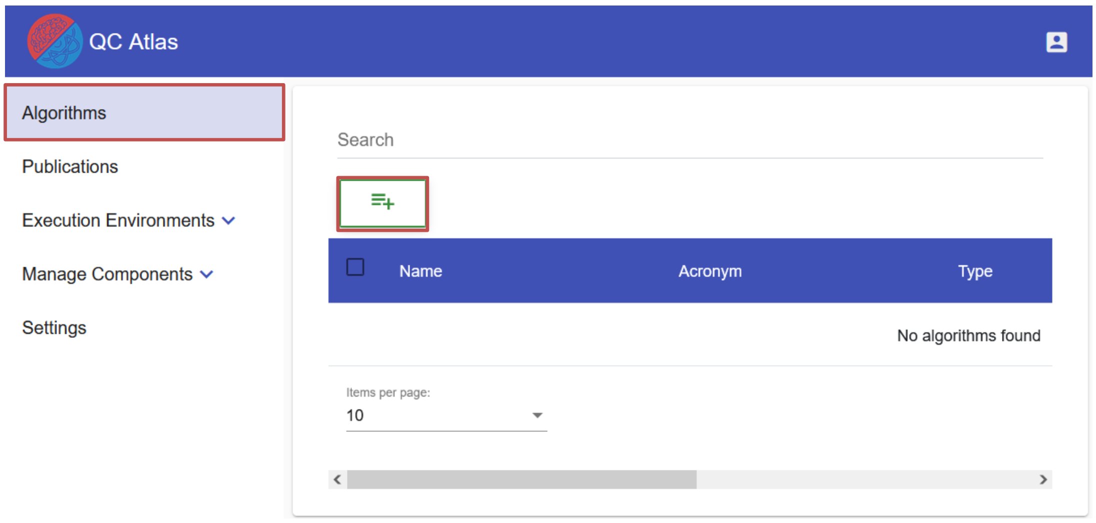


Figure 7.2: Creating a new Algorithm

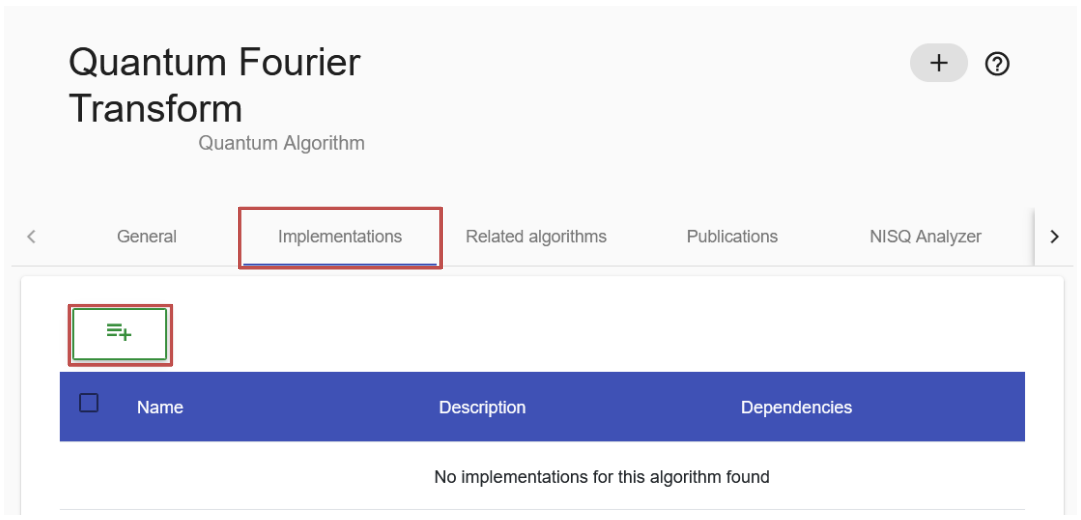
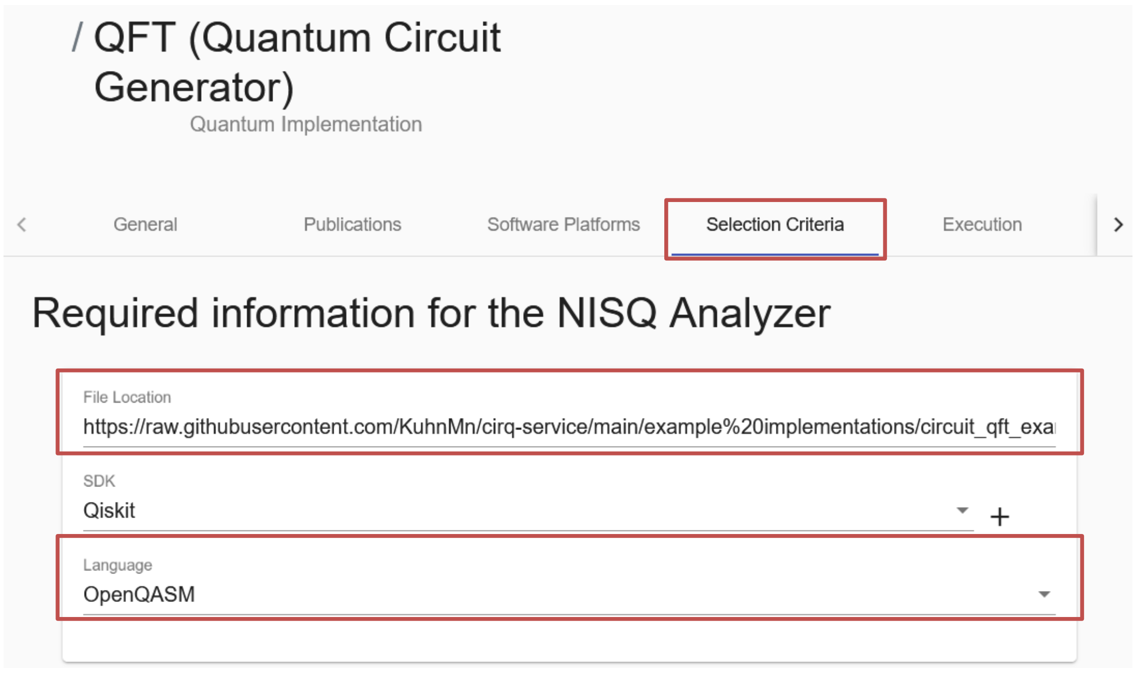


Figure 7.3: Creating a new Implementation for an Algorithm

7.3 Compilation and Execution for a Specific QPU

After the implementation is set up, we can start using the NISQ Analyzer. Its function to simply compile a circuit for a specific QPU is available under the tab *Execution* seen in Figure 7.5. We start a new compilation, choosing Cirq-Google as our vendor and sycamore as our target QPU from the creation dialog shown in Figure 7.6a. We then wait for the compilation to finish and deliver the results. The compiler currently implemented for Google Devices is Cirq, so we get a result compiled using it. As visible in Figure 7.6b, the characteristics of the compiled circuit are listed in the GUI. In addition to the ones shown, these include information on the number of specific operations or gate times and error rates, if available for the QPUs. Since Cirq is currently only simulated, this is not the case here yet. If we scroll through these characteristics on the far right,


/ QFT (Quantum Circuit Generator)
Quantum Implementation

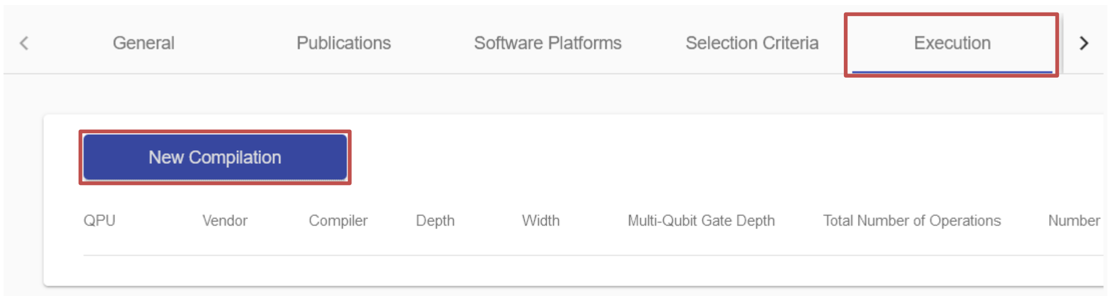
Required information for the NISQ Analyzer

File Location
https://raw.githubusercontent.com/KuhnMn/cirq-service/main/example%20implementations/circuit_qft_exa

SDK
Qiskit

Language
OpenQASM

Figure 7.4: Configuring File URL and Type



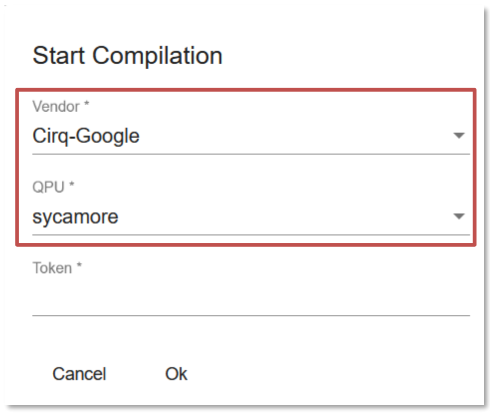
QPU	Vendor	Compiler	Depth	Width	Multi-Qubit Gate Depth	Total Number of Operations	Number

Figure 7.5: Execution Tab

we find a button labeled *Execute* that allows us to execute our selected compilation. Figure 7.7 represents how the button changes and reveals the retrieved results after the execution concludes. The results are the measurement histograms obtained from simulating the circuit using the Cirq service, listed in the structure of a JSON.

7.4 Analysis of Compilation for Multiple QPUs

We now showed how we can use the NISQ Analyzer to compile and execute our circuit. Nonetheless, we also want to show that we can apply its main functionality, the compilation for multiple QPUs at once with various compilers and a comparison of the compilation results. The tab titled *NISQ Analyzer* found in Figure 7.8 provides this function. We start a new analysis using the corresponding

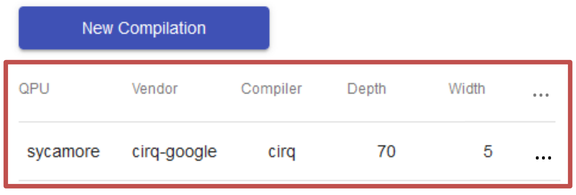

Start Compilation

Vendor *
Cirq-Google

QPU *
sycamore

Token *

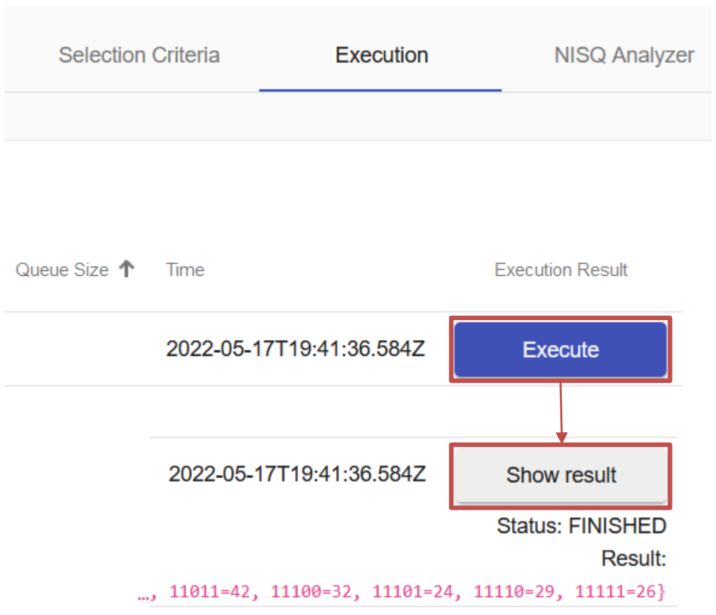
(a) Compilation Dialog



QPU	Vendor	Compiler	Depth	Width	...
sycamore	cirq-google	cirq	70	5	...

(b) Compilation Result

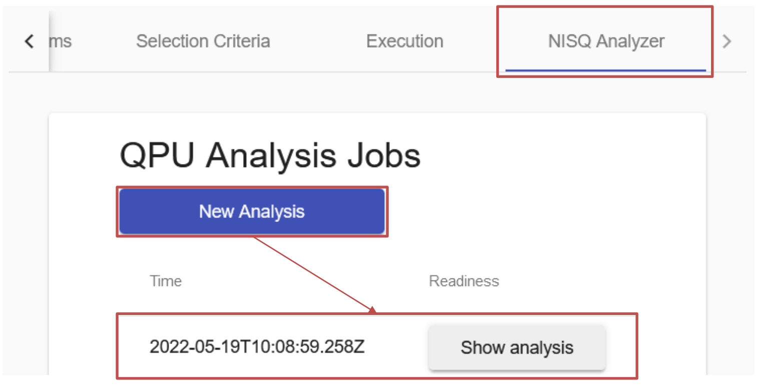
Figure 7.6: Compilation for a Single QPU



Queue Size ↑	Time	Execution Result
	2022-05-17T19:41:36.584Z	Execute
	2022-05-17T19:41:36.584Z	Show result Status: FINISHED Result: …, 11011=42, 11100=32, 11101=24, 11110=29, 11111=26}

Figure 7.7: Execution and Result Retrieval

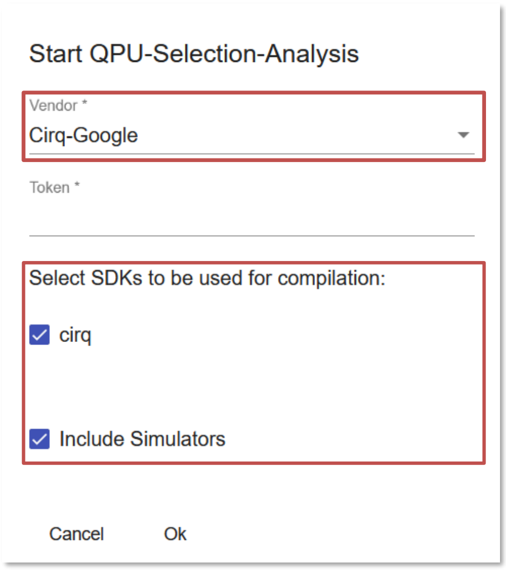
button which opens the dialog shown in Figure 7.9. Here we select the vendor whose QPUs we want to compile for, the compilers we want to use, and whether or not we want to include simulators. As the compilation for Google QPUs in the NISQ Analyzer is only supported by Cirq, we select it as our compiler. In addition, since we do not have access to the real Google devices yet, execution is currently only simulated. Thus the corresponding field is ticked as well. The results are available under the button *Show analysis* in the NISQ Analyzer tab. It leads to the view depicted in Figure 7.10 where results are listed similarly to the compilation results of the last section. This time, not only one but compilations for all QPUs are displayed right next to each other. If we had more compilers



QPU Analysis Jobs

Time	Readiness
2022-05-19T10:08:59.258Z	

Figure 7.8: NISQ Analyzer Tab



Start QPU-Selection-Analysis

Vendor *: Cirq-Google

Token *:

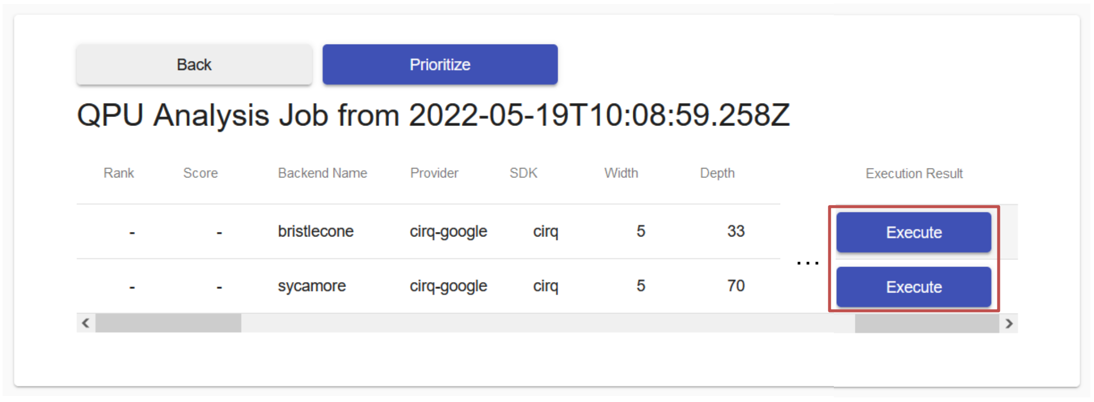
Select SDKs to be used for compilation:

✓ cirq

✓ Include Simulators

Figure 7.9: Analysis Creation Dialog

selected, these results would be listed here as well. Pressing the *Execute* button works identical to the one under the *Execute* tab and gives similarly structured results to the ones shown earlier in Figure 7.7.



QPU Analysis Job from 2022-05-19T10:08:59.258Z

Rank	Score	Backend Name	Provider	SDK	Width	Depth	…	Execution Result
-	-	bristlecone	cirq-google	cirq	5	33		Execute
-	-	sycamore	cirq-google	cirq	5	70		Execute

Figure 7.10: Analysis Creation Dialog

8 Conclusion and Outlook

In this thesis, we presented extensions to multiple components of the QuAntiL project, most prominently to the Quantum Circuit Transpiler and the NISQ Analyzer. Combining established translation functionalities with manual implementation, we were able to make all of Google's Cirq, Amazon's Braket SDK, Microsoft's Quantum Development Kit as well as Quirk available as translation endpoints and in consequence as import options for the NISQ Analyzer. Utilizing these results now also makes it possible to compile circuits for QPUs of these vendors, which we successfully implemented with the example of Cirq. Utilizing these results more than doubled the number of languages available for import to the NISQ Analyzer and made compilation for a completely new set of QPUs possible. The goal of the NISQ Analyzer to reduce dependencies between circuit languages and vendors was thus advanced further to complete fulfillment. The results obtained from analyzing and evaluating the reliability of various frameworks will also prove valuable when selecting such for future projects. We also added a new circuit to the Quantum Circuit Generators library, which together with the new translations will be valuable to Quokka in the future.
As mentioned before, when we selected which frameworks to support, we were limited by time constraints and thus had to cut down on certain choices. In future projects, these options can be revisited and implemented analogously to the previous languages. Additionally, adding services akin to the one we implemented for Cirq is now possible and will allow the NISQ Analyzer to utilize the translation functionalities to the full extent. Thus, their implementation will certainly be a focus of subsequent projects. Furthermore, once we gain access to Google QPUs, the local simulation will be replaced with the execution of the circuits on real quantum devices. Finally, since our evaluation of translation frameworks returned mixed results, it might be useful to further evaluate these translations and if necessary implement manual alternatives in a fashion similar to how translation to Quil and from Q# is currently implemented.

Bibliography

[21] *Quantum programming language converter*. https://github.com/quantastica/qconvert. 2021 (cit. on pp. 18, 28).

[22a] *Amazon Braket Python Schemas Documentation*. https://amazon-braket-schemas-python.readthedocs.io/en/latest/index.html. 2022 (cit. on p. 32).

[22b] *Amazon Braket Python SDK Documentation*. https://amazon-braket-sdk-python.readthedocs.io/en/stable/index.html. 2022 (cit. on p. 29).

[22c] *Azure Quantum Documentation*. https://docs.microsoft.com/en-us/azure/quantum/. 2022 (cit. on p. 29).

[22d] *Cirq Documentation*. https://quantumai.google/cirq/. 2022 (cit. on pp. 29, 43).

[22e] *Q-Convert-JS - Quantum Language Converter*. https://github.com/quantastica/qconvert-js. 2022 (cit. on pp. 18, 28).

[22f] *Qiskit Documentation*. https://qiskit.org/documentation/. 2022 (cit. on p. 29).

[22g] *QuAntiL Documentation*. https://quantil.readthedocs.io/en/latest/. 2022 (cit. on pp. 19, 35).

[22h] *quantum-circuit Documentation*. https://quantum-circuit.com/docs/quantum_circuit. 2022 (cit. on p. 18).

[22i] *staq, a full-stack quantum processing toolkit*. https://github.com/softwareQinc/staq. 2022 (cit. on p. 33).

[22j] *Vendor-Independent Quantum Transpiler*. https://github.com/UST-QuAntiL/QuantumTranspiler. 2022 (cit. on pp. 15, 19).

[AAA+22] M. S. ANIS et al. *Qiskit: An Open-source Framework for Quantum Computing*. Version 0.36.2. 2022. DOI: 10.5281/zenodo.2573505 (cit. on p. 29).

[AG20] M. Amy, V. Gheorghiu. “staq—A full-stack quantum processing toolkit”. In: *Quantum Science and Technology* 5.3 (2020), p. 034016. DOI: 10.1088/2058-9565/ab9359 (cit. on pp. 18, 33).

[AHY20] A. Ajagekar, T. Humble, F. You. “Quantum computing based hybrid solution strategies for large-scale discrete-continuous optimization problems”. In: *Comput. Chem. Eng.* 132 (2020). DOI: 10.1016/j.compchemeng.2019.106630 (cit. on p. 15).

[BIS+18] V. Bergholm, J. Izaac, M. Schuld, C. Gogolin, M. S. Alam, S. Ahmed, J. M. Arrazola, C. Blank, A. Delgado, S. Jahangiri, K. McKiernan, J. J. Meyer, Z. Niu, A. Száva, N. Killoran. *PennyLane: Automatic differentiation of hybrid quantum-classical computations*. 2018. DOI: 10.48550/ARXIV.1811.04968 (cit. on pp. 17, 27, 33).

[BT22a] M. Beisel, F. Truger. *Quantum Circuit Generator*. https://github.com/UST-QuAntiL/quantum-circuit-generator. 2022 (cit. on p. 16).

[BT22b] M. Beisel, F. Truger. *Quokka - The quantum API Gateway*. https://github.com/UST-QuAntiL/quokka. 2022 (cit. on pp. 16, 23).

[BW20] L. Burgholzer, R. Wille. "The power of simulation for equivalence checking in quantum computing". In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218563 (cit. on p. 41).

[CBSG17] A. W. Cross, L. S. Bishop, J. A. Smolin, J. M. Gambetta. *Open Quantum Assembly Language*. 2017. DOI: 10.48550/ARXIV.1707.03429 (cit. on p. 31).

[CJA+21] A. Cross, A. Javadi-Abhari, T. Alexander, N. de Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, P. Sivarajah, J. Smolin, J. M. Gambetta, B. R. Johnson. "OpenQASM 3: A broader and deeper quantum assembly language". In: *ACM Transactions on Quantum Computing* (2021). DOI: 10.1145/3505636 (cit. on p. 31).

[Dev21] C. Developers. *Cirq*. Version v0.12.0. See full list of authors on Github: https://github .com/quantumlib/Cirq/graphs/contributors. 2021. DOI: 10.5281/zenodo.5182845 (cit. on pp. 29, 32).

[FB22] M. Fingerhuth, T. Babej. *Quantum Open Source Foundation*. 2022. URL: https://qosf.org/project_list/ (cit. on p. 25).

[FBW18] M. Fingerhuth, T. Babej, P. Wittek. "Open source software in quantum computing". In: *PLOS ONE* 13.12 (2018), pp. 1–28. DOI: 10.1371/journal.pone.0208561 (cit. on p. 15).

[Gid16] C. Gidney. *My Quantum Circuit Simulator: Quirk*. 2016. URL: https://algassert.com/2016/05/22/quirk.html (cit. on p. 30).

[LaR19] R. LaRose. "Overview and Comparison of Gate Level Quantum Software Platforms". In: *Quantum* 3 (2019), p. 130. DOI: 10.22331/q-2019-03-25-130 (cit. on p. 25).

[LB20] F. Leymann, J. Barzen. "The bitter truth about gate-based quantum algorithms in the NISQ era". In: *Quantum Science and Technology* 5.4 (2020), p. 044007. DOI: 10.1088/2058-9565/abae7d (cit. on p. 15).

[LBF+20] F. Leymann, J. Barzen, M. Falkenthal, D. Vietz, B. Weder, K. Wild. "Quantum in the Cloud: Application Potentials and Research Opportunities". In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science, CLOSER 2020, Prague, Czech Republic, May 7-9, 2020* (2020), pp. 9–24. DOI: 10.5220/0009819800090024 (cit. on p. 15).

[LJL+10] T. D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. R. Monroe, J. L. O'Brien. "Quantum computers". In: *Nat.* 464.7285 (2010), pp. 45–53. DOI: 10.1038/nature08812 (cit. on p. 15).

[MEA+20] S. McArdle, S. Endo, A. Aspuru-Guzik, S. C. Benjamin, X. Yuan. "Quantum computational chemistry". In: *Rev. Mod. Phys.* 92 (1 2020), p. 015003. DOI: 10.1103/RevModPhys.92.015003 (cit. on p. 15).

[NC01] M. A. Nielsen, I. L. Chuang. "Quantum computation and quantum information". In: *Phys. Today* 54.2 (2001), p. 60. URL: https://csis.pace.edu/~ctappert/cs837-19spring/QC-textbook.pdf (cit. on pp. 15, 36, 40).

[NSB+19] E. National Academies of Sciences, D. Sciences, I. Board, C. Board, C. Computing, M. Horowitz, E. Grumbling. *Quantum computing: progress and prospects*. 2019. URL: https://books.google.de/books?id=jjiPDwAAQBAJ (cit. on p. 15).

[NWA21] P. Nimbe, B. A. Weyori, A. F. Adekoya. "Models in quantum computing: a systematic review". In: *Quantum Inf. Process.* 20.2 (2021), p. 80. DOI: 10.1007/s11128-021-03021-3 (cit. on p. 15).

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

[RI15] M. Rahaman, M. M. Islam. "A review on progress and problems of quantum computing as a service (QcaaS) in the perspective of cloud computing". In: *Global Journal of Computer Science and Technology* (2015). URL: https://computerresearch.org/index.php/computer/article/view/1279 (cit. on p. 15).

[SB91] M. J. Swain, D. H. Ballard. "Color indexing". In: *International journal of computer vision* 7.1 (1991), pp. 11–32. DOI: 10.1007/BF00130487 (cit. on p. 42).

[SBB+20] M. Salm, J. Barzen, U. Breitenbücher, F. Leymann, B. Weder, K. Wild. "The NISQ Analyzer: Automating the Selection of Quantum Computers for Quantum Algorithms". In: *Symposium and Summer School on Service-Oriented Computing*. Vol. 1310. 2020, pp. 66–85. DOI: 10.1007/978-3-030-64846-6_5 (cit. on p. 15).

[SBL+21] M. Salm, J. Barzen, F. Leymann, B. Weder, K. Wild. "Automating the Comparison of Quantum Compilers for Quantum Circuits". In: *Symposium and Summer School on Service-Oriented Computing*. Vol. 1429. 2021, pp. 64–80. DOI: 10.1007/978-3-030-87568-8_4 (cit. on pp. 20, 24).

[SDC+20] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, R. Duncan. "t| ket>: a retargetable compiler for NISQ devices". In: *Quantum Science and Technology* 6.1 (2020), p. 014003. DOI: 10.1088/2058-9565/ab8e92 (cit. on pp. 17, 27, 32).

[VN21] D. Vietz, T. Niederhausen. *Qverview*. https://github.com/UST-QuAntiL/Qverview. 2021 (cit. on pp. 15, 16, 25, 27).

[Wan20] T. Wangler. "Development of a vendor independent quantum computing transpiler". MA thesis. 2020. DOI: 10.18419/opus-11219 (cit. on pp. 15, 19, 27).

[Wil22] K. Wild. *QuAntiL - Quantum Application Lifecycle Management*. https://github.com/UST-QuAntiL. 2022 (cit. on p. 15).

All links were last followed on May 30, 2022.

A Appendix

A.1 Example Implementations of Frameworks

Listing A.1 Quantum Circuit Creation and Execution in Qiskit

```
from qiskit import QuantumCircuit
from qiskit.circuit.library import HGate
from qiskit import Aer, transpile
from qiskit import IBMQ

#Creation of the circuit object
circuit = QuantumCircuit(2, 2)

# Appending using append method
circuit.append(HGate(), [0])

# Appending using gate method
circuit.cnot(0,1)

# Measures qubit 0 to clbit 0
circuit.measure(0,0)

# Measures qubit 1 to clbit 1
circuit.measure(1,1)

# Execution on a simulator
simulator = Aer.get_backend('aer_simulator')
circuit = transpile(circuit, simulator)
result = simulator.run(circuit, shots=1024).result()
counts = result.get_counts(circuit)
print(counts)

# Execution on the IBMQ backend
IBMQ.enable_account('TOKEN')
provider = IBMQ.get_provider(hub='ibm-q')
print(provider.backends())
backend = provider.get_backend('ibmq_manila')
circuit = transpile(circuit, backend)
job = backend.run(circuit, shots=1024)

# Check for the job to finish
...

result = job.result()
counts = result.get_counts(circuit)
```

Listing A.2 Count Results

```
{'11': 496, '00': 528}
```

Listing A.3 Executing Q# Code Using the qsharp Library in Python

```
import qsharp

from Example import Circuit

# Simulating on full state simulator
result = Circuit.simulate()

# Simulating on toffoli simulator
result_tof = Circuit.toffoli_simulate()
```

Listing A.4 Translation From and to Qiskit using pytket

```
{
from pytket.extensions.qiskit import tk_to_qiskit, qiskit_to_tk

...

circuit_pytket = qiskit_to_tk(circuit_qiskit)
circuit_qiskit = tf_to_qiskit(circuit_pytket)
}
```

Listing A.5 Example Circuit in PennyLane

```
{
import pennylane as qml

dev = qml.device('default.qubit', wires=2, shots=1024)

@qml.qnode(dev)
def circuit():
    qml.H(wires=0)
    qml.CNOT(wires=[0,1])
    return qml.expval(qml.PauliZ(0)), qml.expval(qml.PauliZ(1))

result = circuit()
}
```

Listing A.6 Import in PennyLane

```
{
import pennylane as qml

...

dev = qml.device('default.qubit', wires=2, shots=1024)

@qml.qnode(dev)
def circuit():
    qml.from_qiskit(circuit_qiskit)
    return qml.expval(qml.PauliZ(0)))

result = circuit()
}
```

Listing A.7 Translation to Q# Using pystaq

```
{
import pystaq

...

p = pystaq.parse_str(qasm_str)
result = pystaq.output_qsharp(p)
}
```

Listing A.8 Example Circuit in Q#

```
namespace Example{
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Convert;
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Math;

     //Entry point defines where the file is executed from when called
     @EntryPoint()
    operation Circuit() : Unit {
            using (q = Qubit[2]) {
                mutable c = new Result[2];
                H(q[0]);
                CNOT(q[0], q[1]);
                set c w/= 0 <- M(q[0]);
                set c w/= 1 <- M(q[1]);
                ResetAll(q);
            }
        }
}
```

A.2 Example Circuits

Listing A.9 Example Circuit in OpenQASM

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg c[2];
h q[0];
cx q[0],q[1];
measure q[0] -> c[0];
measure q[1] -> c[1];
```

Listing A.10 Example Circuit in Cirq JSON

```
{
    "cirq_type": "Circuit",
    "moments": [
        {
            "cirq_type": "Moment",
            "operations": [
                {
                    "cirq_type": "GateOperation",
                    "gate": {
                        "cirq_type": "HPowGate",
                        "exponent": 1.0,
                        "global_shift": 0.0
                    },
                    "qubits": [
                        {
                            "cirq_type": "NamedQubit",
                            "name": "q_0"
                        }
                    ]
                }
            ]
        },
        {
            "cirq_type": "Moment",
            "operations": [
                {
                    "cirq_type": "GateOperation",
                    "gate": {
                        "cirq_type": "MeasurementGate",
                        "num_qubits": 1,
                        "key": "c_0",
                        "invert_mask": []
                    },
                    "qubits": [
                        {
                            "cirq_type": "NamedQubit",
                            "name": "q_0"
                        }
                    ]
                }
            ]
        }
    ],
    "device": {
        "cirq_type": "_UnconstrainedDevice"
    }
}
```

Listing A.11 Example Circuit in Braket IR

```
{
    "braketSchemaHeader": {
        "name": "braket.ir.jaqcd.program",
        "version": "1"
    },
    "instructions": [
        {
            "target": 0,
            "type": "h"
        },
        {
            "control": 0,
            "target": 1,
            "type": "cnot"
        }
    ],
    "results": [
        {
            "type": "statevector"
        }
    ],
    "basis_rotation_instructions": []
}
```

Declaration

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

place, date, signature