Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Automatisierte Generierung von Quantum Workflows

Tim-Julian Ehret

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Dr. h.c. Frank Leymann
Betreuer/in:	Benjamin Weder, M.Sc.
Beginn am:	12. April 2021
Beendet am:	12. Oktober 2021

Kurzfassung

Der Einsatz von Quantum-Computing verspricht für viele Probleme effizientere Lösungen zu bieten, als es mit klassischen Computern möglich wäre. Ferner liegen die heute bereits implementierten Quantenalgorithmen häufig in Form von Skripts vor. Auf der anderen Seite kann eine mithilfe eines Workflows modellierte Applikation von einer Reihe von Vorzügen, wie Skalierbarkeit und Robustheit, profitieren. Die Integration des Quantum-Computing in klassische Applikationen stellt jedoch eine große Herausforderung dar. Insbesondere ist die manuelle Modellierung der durch diese Integration entstandenen Quantum Workflows fehleranfällig und zeitaufwändig. In dieser Arbeit wird daher ein Verfahren zur automatisierten Generierung von Quantum Workflows konzipiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

1 Einleitung

Das Quantum-Computing ist ein neues Computing-Paradigma. Als solches verspricht es, für viele Probleme effizientere Lösungen zu bieten, als es mit klassischen Computern möglich wäre [Pre18]. Die Anwendungsgebiete des Quantum-Computing sind hierbei sehr vielfältig [NM+19]. So ist beispielsweise eine Anwendung bei der Simulation von Teilchen in der Physik oder bei der Erforschung neuer Medikamente möglich. Heute ist bereits eine Vielzahl von Quantencomputer verfügbar und sogar über Cloud-Angebote wie IBMQ oder Rigetti Forest öffentlich zugänglich [LBF+20]. Ferner werden aktuell Software Plattformen für die Zusammenarbeit im Bereich des Quantum-Computing entwickelt [BLF+20]. Denn die Entwicklung von Quantenalgorithmen und passender Software Komponenten stellt eine komplexe Aufgabe dar [LB20] [WBL+20]. Folglich ist die Wiederverwendbarkeit dieser Algorithmen beziehungsweise Komponenten erstrebenswert. Mithilfe eines gemeinsamen Marktplatzes soll daher für Wissenschaft und auch Unternehmen der Einsatz von Quantenalgorithmen erleichtert werden [BLF+20].

Nichtsdestotrotz stellt die Integration des Quantum-Computing in klassische Applikationen eine große Herausforderung dar [WBLW20]. Durch eine solche Integration kann die Implementierung insbesondere von den Vorteilen von Workflows, wie Skalierbarkeit, Robustheit und Fehlerbehandlung, profitieren [WBLW20]. Die Ausführung eines bereits implementierten Quantenalgorithmus bedarf jedoch zusätzlicher Schritte für Pre- und Postprocessing [WBL+20]. Diese Schritte müssen daher mit dem eigentlichen Quantenalgorithmus zusammen orchestriert werden. Ein Blick in die Praxis zeigt zusätzlich, dass Quantenalgorithmen heutzutage häufig in einer einzigen Datei beziehungsweise einem einzelnen Skript implementiert werden [LB21]. Diese Skripts sind typischerweise nicht skalierbar und können nicht unterbrochen werden. Der Fokus einer solchen Skript-basierten Implementierung liegt klar auf der Funktionalität. Die Implementierung eines komplexen Quantenalgorithmus muss schließlich von einem Experten aus dem Bereich des Quanten-Computing übernommen werden. Andererseits erfordert die Modellierung von Workflows Wissen aus einer vollkommen anderen Domäne. In diesem Zusammenhang wurde QuantME als eine Modellierungserweiterung für imperative Workflow-Sprachen entwickelt [WBLW20]. Es ist also möglich Artefakte des Quantum-Computing in einem Workflow zu modellieren. Auf der anderen Seite existieren jedoch bereits viele open-source Implementierungen in Form von Skripts. Diese Implementierungen manuell in Workflows zu überführen ist jedoch nicht trivial und birgt großen Zeitaufwand. Eine Möglichkeit diese Lücke zu schließen ist es, Quantum Workflows aus solchen Skripten zu Generieren.

In einem konkreten Szenario hat sich ein Benutzer dazu entschlossen, eine Problemstellung mit einem Quantenalgorithmus zu lösen. Hierzu verfügt er über ein Quantum Skript, welches auf seine Problemstellung zugeschnitten ist. Als Experte für Quantum Computing ist er

zuversichtlich, dass es sich um eine valide Implementierung handelt. Um Skalierbarkeit und Robustheit zu gewährleisten, soll diese Lösung nun in einem Workflow umgesetzt werden. Hier treffen folglich zwei komplett verschiedene Domänen aufeinander. Die Modellierung des gewünschten Workflows soll deshalb durch ein automatisches Verfahren übernommen werden. Ferner soll das Verfahren aus einer zusammenhängenden, Skript-basierten Implementierung einen Workflow generieren.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Zunächst werden in Kapitel 2 einige grundlegende Begriffe aus dem Quantum-Computing und dem Bereich von Workflows eingeführt und erläutert. Anschließend wird in Kapitel 3 die Problemstellung und eine Reihe von Annahmen formuliert. In Kapitel 4 wird das Konzept des hier entwickelten Verfahrens zur automatisierten Generierung von Quantum Workflows beschrieben. Kapitel 5 widmet sich anschließend den Details der prototypischen Implementierung des Verfahrens. Am Ende dieses Kapitels wird außerdem ein Anwendungsbeispiel vorgestellt. Daraufhin werden in Kapitel 6 verwandte Arbeiten besprochen. Die Arbeit endet mit der Zusammenfassung und einem Ausblick auf zukünftige Entwicklungen beziehungsweise mögliche Erweiterungen in Kapitel 7.

2 Grundlagen

Dieses Kapitel ist der Erklärung von grundlegenden Begriffen und Konzepten gewidmet. Zunächst werden dazu die Grundlagen des Quantum-Computings geschildert. Als zentrales Element der Arbeit wird zusätzlich der Begriff Quantum Skript eingeführt. Um ein grundlegendes Verständnis für die Struktur und die interne Darstellung von Quellcode zu schaffen, werden daraufhin Syntax Bäume näher erläutert. Abschließend werden die Grundlagen von Workflows beschrieben.

2.1 Quantum-Computing

Beim *Quantum-Computing* werden die verwendeten Informationen in einem Quanten-System kodiert [Pre18]. Anstelle von klassischen Bits werden dabei *qubits* verwendet. Der fundamentale unterschied besteht nun darin, dass ein *qubit* sich nicht nur im Zustand 0 oder 1 befinden kann, sondern in beiden Zuständen gleichzeitig. Dies bezeichnet man als Superposition [RP11]. Typischerweise werden mehrere *qubits* zu einem Quanten-Register kombiniert [LBF+20]. Auch hier gilt nun das Prinzip der Superposition. Somit kann sich ein Quanten-Register mit n *qubits* in einer Superposition von 2^n Zuständen befinden. Um Berechnungen anzustellen müssen letzten Endes Manipulationen an den Quanten-Registern durchgeführt werden. Eine solche Manipulation des Zustands eines Quanten-Registers wird stets durch eine unitäre Transformation erreicht [LBF+20]. Die 2^n Zustände des Quanten-Registers werden dabei gleichzeitig modifiziert. Dieses zweite wichtige Prinzip bezeichnet man als Quanten-Parallelismus. Insbesondere ist es dadurch möglich, manche Probleme effizienter zu lösen als dies mithilfe von klassischen Methoden möglich wäre [DJ92]. Das Ergebnis einer Berechnung kann nun festgestellt werden, indem eine Messung auf dem Quanten-Register ausgeführt wird [LBF+20].

Im Gate-basierten Modell des Quantum-Computing werden Quantenalgorithmen durch eine Reihe von unitären Transformationen dargestellt [LBF+20]. Der Quantenalgorithmus überführt dabei also einen Anfangszustand in einen Endzustand. Das Ergebnis des Algorithmus liegt, nach der entsprechenden Messung, in Form eines Bit-Strings vor [LBF+20]. Dieser repräsentiert daher einen der 2^n möglichen Zustände des Quanten-Registers. Da Quantenalgorithmen inhärent probabilistisch sind, müssen diese mehrfach ausgeführt werden, um ein brauchbares Resultat zu liefern [RP11]. Genauer entsteht hierbei eine Wahrscheinlichkeitsverteilung, wobei schließlich das am Häufigsten vorkommende Ergebnis ausgewählt wird [LBF+20].

Quantenalgorithmen können auch auf andere Art und Weise beschrieben werden [BBD+09]. So kann zum Beispiel das Messungs-basierte Modell gewählt werden, um Berechnungen mithilfe einer Reihe von einfachen Messungen umzusetzen [BBD+09]. Es wurde jedoch gezeigt, dass die verschiedenen Modelle formal äquivalent sind [BBD+09]. Zusätzlich basieren mehrere prominente Vertreter von Quantum Software Plattformen, wie IBMQ und Rigetti Forest auf dem Gate-basierten Modell. Wenngleich also im Bereich des Quantum-Computing noch weitere Modelle existieren, begnügt sich diese kurze Einführung daher mit dem Gate-basierten Modell.

Das Quantum-Computing bietet nun nicht nur neue Möglichkeiten sondern auch neue Herausforderungen. Denn die heute verfügbaren Quantencomputer sind typischerweise verrauscht beziehungsweise *noisy* [BBD+09] [Pre18]. Das bedeutet, dass die Gates und auch die *qubits* eines Quantencomputers durch verschiedene Einflüssen gestört werden können. So können beispielsweise unbeabsichtigte Interaktionen zwischen den *qubits* und der Umgebung auftreten [BBD+09]. Bei der Entwicklung und Implementierung von Quantenalgorithmen muss diese Einschränkung daher stets bedacht und entsprechend behandelt werden [WBL+20].

Quantum Skript

Für die vorliegende Arbeit sind Skript-basierte Implementierungen von Quantenalgorithmen von überragender Bedeutung. Innerhalb dieser Arbeit wird eine solche Implementierung als Quantum Skript bezeichnet. Um ein einheitliches Verständnis über diese Begrifflichkeit zu erlangen soll nun ein Beispiel betrachtet werden. Hierfür wurde der Algorithmus von Deutsch-Jozsa ausgewählt [DJ92]. Dieser Quantenalgorithmus bietet eine effiziente Lösung um zu entscheiden, ob eine gegebene boolesche Funktion balanciert oder konstant ist. Mathematisch kann die gegebene Funktion f durch

$$f(\{x_1, x_2, x_3, ...\}) \rightarrow \{0, 1\} | x_i \in \{0, 1\}$$

beschrieben werden. Hierbei ist garantiert, dass f entweder konstant oder balanciert ist. Konstant bedeutet, dass f jeden Eingabe-String konstant auf 0 beziehungsweise 1 abbildet. Im balancierten Fall wird hingegen genau die Hälfte der Eingaben auf 0 und die andere Hälfte auf 1 abgebildet. Der Deutsch-Jozsa Algorithmus kann nun durch einen einzigen Aufruf der Funktion f herausfinden, ob diese konstant oder balanciert ist [DJ92]. Ohne die mathematischen beziehungsweise physikalischen Hintergründe zu vertiefen, soll nun eine mögliche Umsetzung des Quantenalgorithmus mithilfe von Qiskit betrachtet werden. In Listing 2.1 ist dazu ein Ausschnitt einer möglichen Implementierung des Deutsch-Jozsa Algorithmus angegeben. In den einzelnen Schritten dieses Quantum Skripts werden hierzu ein Quantenschaltkreis mit verschiedenen Gates versehen. Insbesondere werden dabei sogenannte Hadamard-Gates und die in einem Orakel implementierte Funktion f angewandt. Wie für einen Quantenalgorithmus üblich, findet zum Abschluss eine Messung des Zustands statt. Der Skript-basierte Charakter der Implementierung ist hierbei klar zu erkennen.

Zusätzlich zur eigentlichen *Quantenberechnung*, benötigt man zur Ausführung eines Quantenalgorithmus typischerweise zusätzliche Schritte [WBL+20]. Üblicherweise bezeichnet man diese als *Preprocessing* beziehungsweise *Postprocessing*. Beispielsweise kann hierbei eine Vorbereitung der Eingabedaten beziehungsweise ein Verfahren zur Fehlerkorrektur angewandt werden.

Listing 2.1 Code-Ausschnitt der Skript-basierten Implementierung eines Quantenalgorithmus entsprechend der Qiskit Dokumentation [ACB+20]

```
1  dj_circuit = QuantumCircuit(n+1, n)
2
3  # Apply H-gates
4  for qubit in range(n):
5     dj_circuit.h(qubit)
6
7  # Put qubit in state |->
8  dj_circuit.x(n)
9  dj_circuit.h(n)
10
11 # Add oracle
12 dj_circuit += balanced_oracle
13
14 # Repeat H-gates
15 for qubit in range(n):
16    dj_circuit.h(qubit)
17 dj_circuit.barrier()
18
19 # Measure
20 for i in range(n):
21    dj_circuit.measure(i, i)
```

2.2 Syntax Bäume

Ein (abstrakter) Syntax Baum ist ein Baum, welcher darauf zugeschnitten ist, die (abstrakte) syntaktische Struktur von Quellcode abzubilden [BYM+98]. Wie jeder Baum in der Informatik besteht der Syntax Baum aus Knoten und Kanten. Beginnend an der Wurzel, verweist jeder interne Knoten auf seine Kind-Knoten. Hat ein Knoten keine Kind-Knoten, so handelt es sich um einen Blatt-Knoten. Die Knoten des Syntax Baums entsprechen hierbei den Konstrukten beziehungsweise den Symbolen eines gegebenen Quellcodes. Mithilfe der Kanten wird die Struktur des Quellcodes dargestellt. Um dieses Konzept zu verdeutlichen, befindet sich in Listing 2.2 ein kleiner Code-Ausschnitt. Genauer handelt es sich hierbei um eine Zuweisung. In der Variable *result* soll das Ergebnis einer Addition abgespeichert werden. Die Argumente der Addition sind die Variablen *a* und *b*.

Listing 2.2 Code-Ausschnitt einer Zuweisung

```
1        result = a + b
```

Der daraus abgeleitete Syntax Baum ist in Abbildung 2.1 angegeben. Von der Wurzel aus beginnt der Baum mit einem Knoten für die Zuweisung. Eine Zuweisung besteht typischerweise aus einem Wert und einem Ziel. Das Ziel ist in diesem Beispiel die Variable *result*. Daher befindet sich im Baum ein Blatt-Knoten mit dem entsprechenden Namen. Der Wert der Zuweisung setzt sich hier hingegen aus mehreren Knoten zusammen. Es liegt schließlich eine Addition vor. Diese wird mithilfe eines binären Operators repräsentiert. Also einem Operator mit genau zwei Argumenten. Neben den zwei Knoten für die Argumente wird ein zusätzlicher Wert-Knoten benötigt, um verschiedene binäre Operatoren auseinander zu halten. Der entsprechende Knoten verfügt folglich über insgesamt drei Kind-Knoten. Hierbei gibt der Wert-Knoten die Art der Operation, wie beispielsweise Addition oder Multiplikation, an. In diesem Fall erhält dieser schließlich einen zur Addition passenden Blatt-Knoten. Die verbleibenden Knoten repräsentieren die Argumente des Operators. Entsprechend dem Quellcode sind dies also die Bezeichner *a* beziehungsweise *b*.

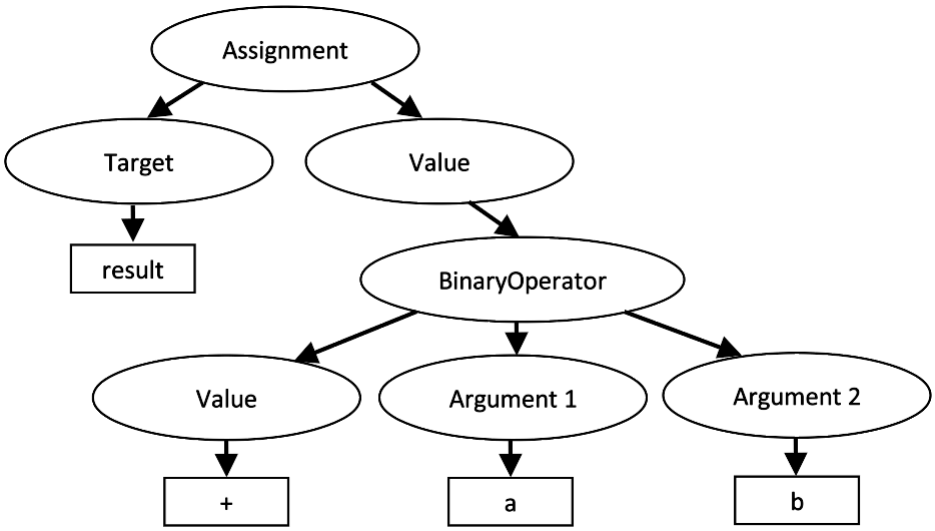

Abbildung 2.1: Syntax Baum einer Zuweisung

Syntax Bäume werden in vielen Programmiersprachen und Werkzeugen des Software-Engineering eingesetzt [ZWZ+19]. Die genaue Implementierung des Syntax Baumes ist dabei insbesondere von der betroffenen Programmiersprache abhängig. So wurde beispielsweise im Rahmen des Open-Source-Projekts *RedBaron* ein Syntax Baum für die Programmiersprache Python implementiert [Psy]. Es kann also auf eine bereits existierende Implementierung zurückgegriffen werden, um diese bewährte Methode zur Analyse von Quellcode einzusetzen. In dieser Arbeit wird daher ein Syntax Baum als interne Datenstruktur genutzt. Insbesondere

stützt sich das in Kapitel 4 entwickelte Konzept auf die Darstellung mithilfe eines Syntax Baumes.

2.3 Workflows

Eine zentrale Aufgabe der IT in der Wirtschaft ist die Integration verschiedener Software-Artefakte. Es wurden daher bereits verschiedene Ansätze entwickelt, um Software-Artefakte gemeinsam zu modellieren und insbesondere die von diesen bereitgestellte Funktionalität zu orchestrieren [Ell99]. Ein *Workflow* beziehungsweise allgemeiner die *Workflow Technologie* stellt einen solchen praxiserprobten Ansatz dar [LR99]. Hierbei können beispielsweise Business Prozesse oder wissenschaftliche Workflows modelliert werden. Um die einzelnen Artefakte miteinander zu verbinden, werden hierbei Workflow Modelle spezifiziert, welche mithilfe einer Workflow Engine automatisiert ausgeführt werden können [LR99]. So kann die Funktionalität heterogener Anwendungen zu neuen Anwendungen beziehungsweise Services kombiniert werden [MM04]. In einem Workflow Modell werden demnach die einzelnen Schritte beziehungsweise Aufgaben und deren Ausführungsreihenfolge abgebildet. Gleichzeitig wird hierbei der Datenfluss zwischen den einzelnen Schritten definiert.

Workflow Modelle können mithilfe einer Workflow-Sprache, wie zum Beispiel BPMN spezifiziert werden [OPM11]. Business Process Model and Notation (BPMN) ist eine standardisierte Modellierungssprache für die Modellierung von Geschäftsprozessen beziehungsweise Workflows [AK15] [OPM11]. Seit der ersten Veröffentlichung im Jahr 2004 wurde BPMN von der Object Management Group (OMG) überarbeitet und schließlich in der aktuellen Version BPMN 2.0, ebenfalls von OMG, veröffentlicht [OPM11]. Die Standardisierung ermöglicht insbesondere die Portabilität zwischen verschiedenen BPMN Engines.

Als zentrale Elemente enthalten die Modelle dabei verschiedene Aktivitäten beziehungsweise Tasks, welche mithilfe von Konnektoren verbunden werden. Es sollen nun einige der im Standard definierten Komponenten und insbesondere deren grafische Darstellung kurz vorgestellt werden. Die einzelnen Schritte innerhalb eines BPMN Modells werden von Aktivitäten repräsentiert [OPM11]. Atomare Aktivitäten werden im Standard ferner als Tasks bezeichnet und sind in verschiedene Arten eingeteilt. Wenn ein Task zur Erfüllung seiner Aufgabe eine beliebige Art von Service verwendet, spricht man von so beispielsweise von einem Service Task [OPM11]. Der Service Task kann also insbesondere auf eine automatisierte Anwendung zugreifen. Um die einzelnen Aktivitäten miteinander zu verbinden und deren Reihenfolge festzulegen, wird ein Kontrollfluss definiert. Zusätzlich können Gateways verwendet werden, um den Kontrollfluss aufzuspalten und wieder zusammenzuführen [OPM11]. Zur Veranschaulichung befindet sich in 2.2 ein kleines Workflow Beispiel. Hierbei wird die Auszahlung eines gewissen Geldbetrages an einem Geldautomat illustriert.

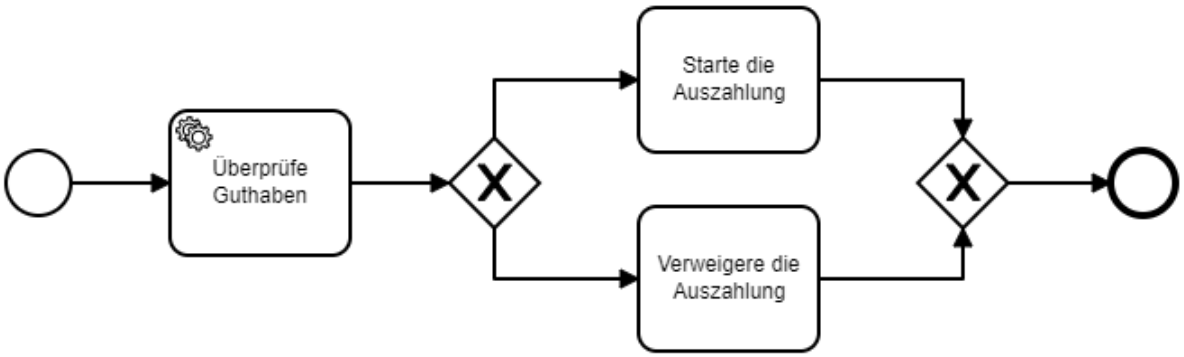

Abbildung 2.2: Beispiel eines BPMN Workflows

Die grafische Darstellung eines Tasks verwendet ein abgerundetes Rechteck mit einem speziellen Marker in der linken oberen Ecke [OPM11]. Entsprechend befinden sich in Abbildung 2.2 insgesamt drei Tasks. Folgt man dem Kontrollfluss entlang der abgebildeten Pfeile, so wird als erstes der Service Task erreicht. Im Beispiel wird hier automatisch überprüft, ob der Kunde über ausreichend Guthaben verfügt. Je nach Ergebnis der Prüfung trennen sich die Pfade am nachfolgenden Gateway. Es handelt sich hierbei um einen exklusiven Gateway, daher kann nur einer der ausgehenden Pfade weiterverfolgt werden. Im positiven Fall wird die Auszahlung gestartet. Falls nicht genug Guthaben verfügbar ist, wird die Auszahlung verweigert. Schließlich werden die Pfade durch einen weiteren Gateway wiedervereint und der Workflow kann beendet werden.

Schließlich bietet der Einsatz von Workflow Technologie eine Reihe von Vorteilen. Unter anderem werden Skalierbarkeit, Robustheit, Zuverlässigkeit und adäquate Fehlerbehandlung ermöglicht [Ell99][LR99]. Außerdem ist es, zumindest technisch, möglich Quanten-Schaltkreise in die Orchestrierung mit aufzunehmen [WBLW20].

3 Problemstellung und Annahmen

In diesem Kapitel wird noch einmal genauer auf die Problemstellung eingegangen. Hierzu wird zunächst aus den Beobachtungen der vorangestellten Kapiteln eine entsprechende Fragestellung abgeleitet. Im Anschluss werden einige Annahmen im Bezug auf die Beschaffenheit von Quantum Skripts formuliert.

3.1 Problemstellung

In den vorangestellten Kapiteln wird deutlich, dass die Entwicklung von Quantum Workflows eine komplexe Aufgabe darstellt. Mithilfe von designierten Software Plattformen können bereits implementierte Quantenalgorithmen unter Experten ausgetauscht und wiederverwendet werden [BLF+20]. Die Modellierungserweiterung QuantME ermöglicht es, Artefakte des Quantum-Computing in einem Workflow zu modellieren [WBLW20]. Dementsprechend können Quantum-Computing und Workflows zusammengeführt werden. Lösungsansätze, welche Quantenalgorithmen einsetzen, können somit von den in Kapitel 2 geschilderten Vorzügen von Workflows profitieren. Die manuelle Modellierung von Workflows ist und bleibt jedoch zeitaufwändig und fehleranfällig. Das Ziel der vorliegenden Arbeit ist es daher ein automatisiertes Verfahren zur Generierung von Quantum Workflows zu entwickeln. Insbesondere soll hierbei die folgende Forschungs-Frage beantwortet werden:
„Wie kann aus einer gegebenen, Skript-basierten Implementierung eines Quantenalgorithmus automatisiert ein Workflow generiert werden?“

3.2 Annahmen

Quellcode und insbesondere Skript-basierte Implementierungen lassen sich im Allgemeinen beliebig komplex gestalten. Ein zuverlässiges Analyseverfahren zu implementieren stellt daher eine ebenso komplexe Aufgabe dar. Es war daher im Rahmen dieser Arbeit notwendig, gewisse Annahmen über die Struktur des zu analysierenden Quellcodes zu treffen. Diese Annahmen sind in Tabelle 3.1 aufgelistet und sollen im folgenden kurz erläutert werden.

ID	Kurzbeschreibung
A1	Flache Skripts
A2	Programmiersprache Python
A3	Existenz von drei Teilen
A4	Valide Programme
A5	For Schleifen

Tabelle 3.1: Annahmen bezüglich der Beschaffenheit der Quantum Skripts

A1 - Flache Skripts Der zu analysierende Quellcode wird in Form von Quantum Skripts implementiert. Ein solches Skript wird hier als „ flach“ bezeichnet, wenn darin keine Hilfsfunktionen definiert und verwendet werden. Es wird davon ausgegangen, dass die Eingabe des Algorithmus stets flach ist.

A2 - Programmiersprache Python Die automatische Generierung von Quantum Workflows stützt sich in erster Linie auf die Analyse einer gegebenen Implementierung beziehungsweise deren Quellcode. Das in dieser Arbeit entwickelte Verfahren soll zeigen, wie eine solche automatisierte Generierung ablaufen und umgesetzt werden kann. Insofern sollte es ausreichen, in der konkreten Umsetzung eine prominente Programmiersprache zu unterstützen. Es wird daher davon ausgegangen, dass der Quellcode in der Programmiersprache Python geschrieben ist. Es ist denkbar, die unterstützen Programmiersprachen im Zuge weiterer Arbeiten zu erweitern.

A3 - Existenz von drei Teilen Wie in Kapitel 2 bereits erläutert wurde, besteht eine Implementierung eines Quantenalgorithmus üblicherweise aus den drei Teilen *Preprocessing*, *Quantenberechnung* und *Postprocessing*. Der Algorithmus geht daher davon aus, dass sich jeder eingegebene Quellcode in drei Teile zerlegen lässt. Ein solcher Teil kann jedoch leer sein, falls ihm letzten Endes keine Statements zugeordnet werden können.

A4 - Valide Programme Um garantieren zu können, dass das Ergebnis des Generierungsprozesses korrekt ausführbar ist, muss die Eingabe ein valides Programm sein. Außerdem kann nur eine valide Eingabe überhaupt korrekt analysiert werden.

A5 - For Schleifen Zum jetzigen Zeitpunkt erstreckt sich die Analysefähigkeit des entwickelten Algorithmus noch nicht über alle in der Programmiersprache möglichen Konstrukte. Insbesondere wird nach aktuellem Stand nur eine Art von Schleifen unterstützt. Es wird daher angenommen, dass sich der Quellcode auf die Verwendung von For-Schleifen beschränkt.

4 Automatisierten Generierung von Quantum Workflows

Im Folgenden wird der in dieser Arbeit entwickelte Ansatz zur automatisierten Generierung von Quantum Workflows vorgestellt. Als Erstes wird hierzu die High-Level-Ansicht des Ansatzes dargelegt. Anschließend werden einige notwendige Annahmen getroffen und begründet. Um eine Gesamtübersicht über die verschiedenen Komponenten zu vermitteln, wird daraufhin eine Architektur konzipiert. Den Kern des Ansatzes bildet der hier entwickelte Script Splitting Algorithmus. Daher wird an dieser Stelle der Algorithmus im Detail präsentiert. Am Ende des Kapitels wird schließlich der automatisierte Generierungsprozess vorgestellt.

4.1 High-Level-Ansicht

In diesem Abschnitt soll eine Übersicht über das ganze Konzept gegeben werden. Das Ziel ist es, von einer Skript-basierten Implementierung eines Quantenalgorithmus zu einem Workflow zu gelangen. Abbildung 4.1 veranschaulicht den daraus abgeleiteten automatisierten Generierungsprozess. Der Prozess besteht im Wesentlichen aus drei Schritten. Zunächst muss im ersten Schritt eine Implementierung gefunden werden. Diese Aufgabe muss vom Benutzer beziehungsweise von einem Experten des Quantum Computing übernommen werden. Die Implementierung kann dabei selbst entwickelt oder aus einer Datenbank beziehungsweise einem Repository entnommen werden. Das somit gegebene Quantum Skript kann nun als Eingabe für die Generierung dienen. Der zweite Schritt des Prozesses dient der Analyse des Quantum Skripts. Hierbei werden die für das Quantum-Computing relevanten Teile des Skripts identifiziert. Im dritten Schritt wird der zunächst zusammenhängende Quellcode in mehrere Teile zerlegt. Jeder Teil liegt danach wiederum als unabhängiges Skript vor. Aus diesen Teilen wird im vierten Schritt ein Workflow generiert. In Abbildung 4.1 ist leicht zu erkennen, dass der generierte Workflow große strukturelle Ähnlichkeit zu den entstandenen Teilen des Skripts hat. Für den generierte Workflow kann anschließend im fünften Schritt ein Deployment stattfinden. Im aktuellen Stadium muss dieser Schritt manuell ausgeführt werden. Im Rahmen zukünftiger Arbeiten kann jedoch ein Deployment System eingesetzt werden, um das Deployment zu automatisieren. Schließlich kann der Workflow im sechsten Schritt mithilfe einer Workflow Engine ausgeführt werden.

Abgesehen von der Implementierung des Quantum Skripts können alle Schritte des Generierungsprozesses automatisiert werden.

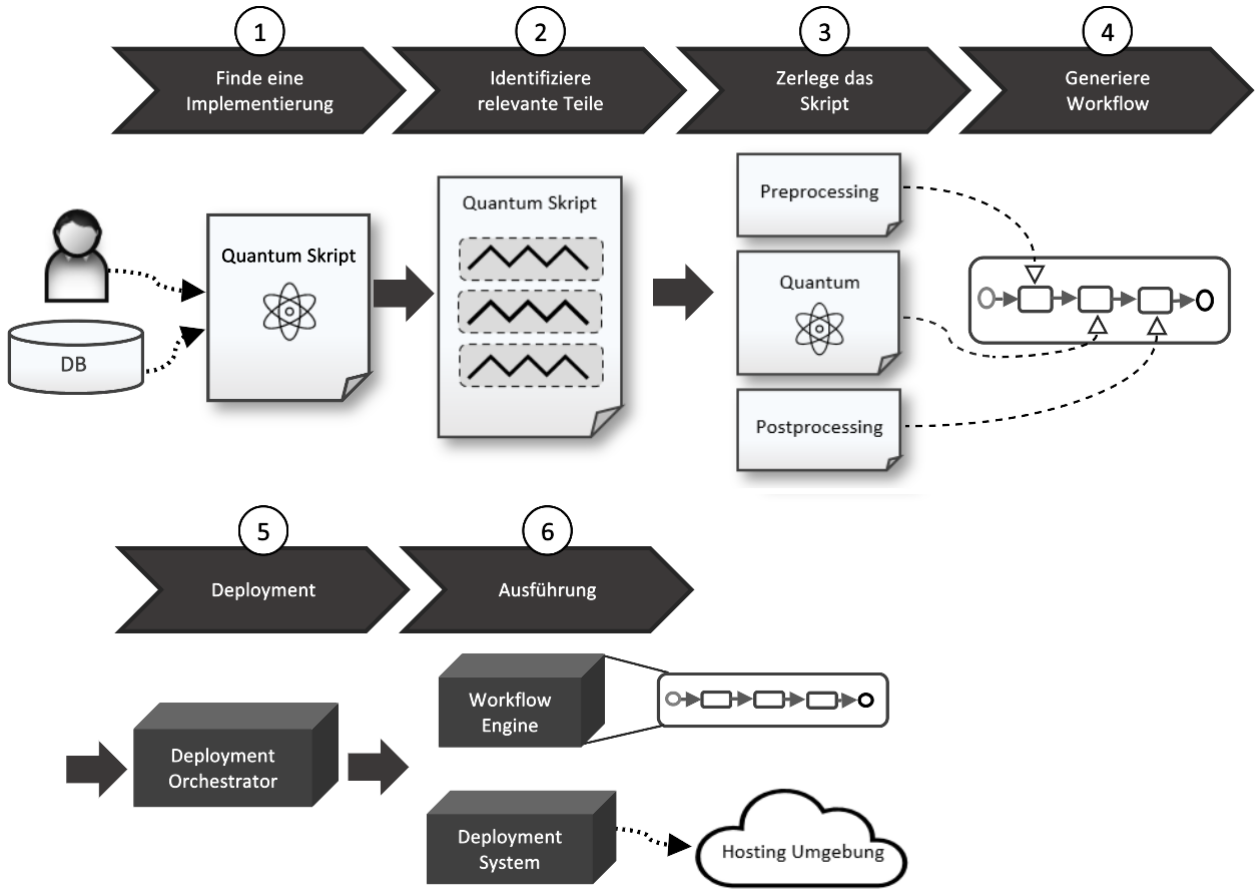

Abbildung 4.1: Übersicht über den automatisierten Generierungsprozess

4.2 Architektur

Das in dieser Arbeit entwickelte Konzept zur automatisierten Generierung von Quantum Workflows umfasst mehrere Schritte und ist auf mehrere Komponenten aufgeteilt. Insbesondere war es Teil der Arbeit ein Plugin-basiertes Framework zu entwickeln. Daher wurden die erarbeiteten Komponenten über ein Plugin in Camunda beziehungsweise das QuantME Framework integriert. Das Zusammenspiel der verschiedenen Komponenten wird in Abbildung 4.2 veranschaulicht. Die Abbildung zeigt dabei sowohl die mit Nummern gekennzeichneten Verarbeitungsschritte als auch die wichtigsten Artefakte.

Der Prozess beginnt mit der Eingabe eines Quantum Skripts durch einen Benutzer. Die Eingabe erfolgt über die grafische Benutzeroberfläche von Camunda beziehungsweise über die Plugin-Erweiterung des QuantME Frameworks. Innerhalb der QuantME Erweiterung befindet sich das hier entwickelte *ScriptSplitterPlugin*. Dieses übernimmt im Wesentlichen zwei Aufgaben. Zum einen vermittelt es zwischen Benutzeroberfläche und der eigentlichen *Script Splitter* Komponente. Für diese Vermittlung ist die Subkomponente *Input* verantwortlich. Zum

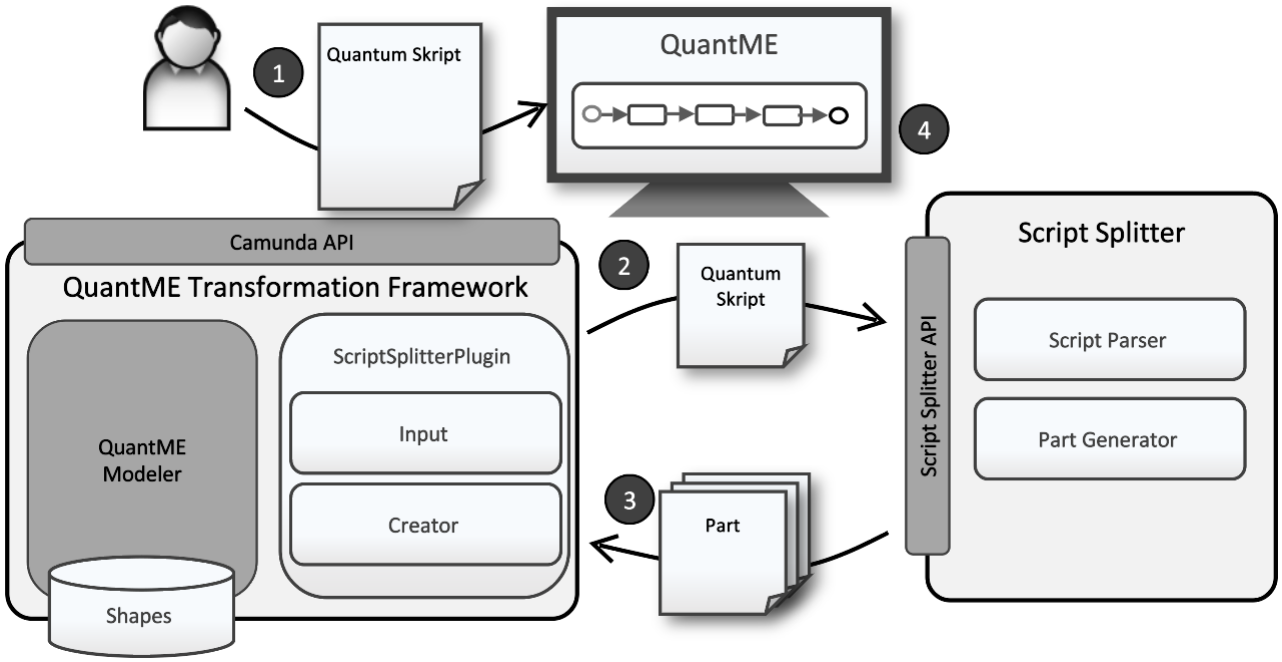

Abbildung 4.2: Abstrakte Architektur des Generierungsframeworks

anderen ist das Plugin im Rahmen der Subkomponente *Creator* dafür zuständig, Workflow Elemente zu generieren. Die Eingabe wird im zweiten Schritt zunächst an den *Script Splitter* weitergeleitet. Mit Hilfe des in dieser Arbeit entwickelten Algorithmus kann das übergebene Quantum Skript hier vom *Script Parser* analysiert werden. Als Ergebnis der Analyse wird das zunächst zusammenhängende Quantum Skript in drei Stücke zerlegt. Der *Part Generator* überführt die einzelnen Stücke des Skripts anschließend in eigenständige Teile. Diese drei Teile werden, im dritten Schritt, wiederum an das Plugin zurückgeschickt. Jeder Teil des Quantum Scripts muss nun zu einem Task verarbeitet werden. Der *Creator* kann zu diesem Zweck auf die Modellierungsfähigkeiten des QuantME Frameworks zurückgreifen. So ist es insbesondere möglich, passende Workflow Elemente zu generieren und diese zusammen zu modellieren. Im vierten Schritt wird schließlich das Ergebnis des Generierungsprozesses in der Benutzeroberfläche dargestellt.

4.3 Script Splitting Algorithmus

Wie in den vorangestellten Kapiteln bereits geschildert wurde, werden Quantenalgorithmen häufig in einem einzelnen Skript implementiert. Typischerweise enthält dieses Skript nun mehrere Teile. Jeder Teil erfüllt dabei jeweils eine Aufgabe des gesamten Quantenalgorithmus. Namentlich sind diese Aufgaben *Preprocessing*, *Quantenberechnung* und *Postprocessing*. Die einzelnen Teile lassen sich dann später als Workflow Elemente orchestrieren. Dieser Beobachtung folgend, ist es das Ziel des Script Splitting Algorithmus, die einzelnen Teile zu identifizieren

und diese zu trennen um somit eigenständige Skripts zu erzeugen, welche von einem Workflow verwendet werden können.

Um Quellcode zuverlässig analysieren zu können, wurden in Abschnitt 2.2 Syntax Bäume eingeführt. Der hier geschilderte Algorithmus greift auf dieses Konzept als interne Datenstruktur zurück. Ferner unterliegt die folgende Beschreibung des Algorithmus den Annahmen aus Abschnitt 3.2.

Der Pseudocode in Algorithmus 4.1 spiegelt den entwickelten Algorithmus wider. Die Eingabe des Algorithmus ist der Quellcode des Quantum Skripts. Aus diesem Quellcode wird zunächst ein Syntaxbaum konstruiert. In Abschnitt 2.2 wurde bereits beschrieben, wie ein solcher Baum erzeugt werden kann. Auf Grundlage des Syntaxbaums arbeitet der Algorithmus ab jetzt nur noch mit dessen Knoten. Es werden nun als Erstes alle Knoten herausgefiltert, welche eine Bibliothek mit Bezug zum Quantum-Computing einbinden. Genauer gesagt also Knoten, die einen speziellen Import darstellen. Die Suche nach diesen Knoten ist der Lesbarkeit halber in die Hilfsfunktion Algorithmus 4.2 ausgelagert. Hierbei werden alle Import Knoten betrachtet und mit der vordefinierten Menge *QUANTUMSET* verglichen. In *QUANTUMSET* sind typische Bibliotheken des Quantum-Computing, wie z.B. qiskit von IMB CITE, definiert. Der Algorithmus toleriert somit auch eine Veränderung beziehungsweise Erweiterung hin zu verschiedenen Bibliotheken. Im Rahmen des Prototyps dieser Arbeit beschränkt sich die Menge der Bibliotheken jedoch allein auf Qiskit. Die gefilterten Import-Knoten können nun als Indikator für Quantenberechnungen dienen.

Um den Teil der Knoten zu identifizieren, welcher Quantenberechnungen beinhaltet, wird jeder Knoten des Syntaxbaums nun auf eine Referenz zu den Import Knoten getestet. Falls eine solche Referenz vorliegt, wird der aktuelle Knoten dem Teil *Quantenberechnung* zugeordnet. Beispielsweise würde so ... Zusätzlich wird nun der Index des Knotens gespeichert. Der Index bezeichnet hier die Position des betreffenden Statements im Quellcode. Über die gespeicherten Indizes wird Beginn und Ende des Quanten-Teils festgehalten. Am Ende dieses ersten Schritts stehen somit alle Knoten fest, welche tatsächlich für einen Aufruf der Funktionalität einer Quantenbibliothek verantwortlich sind.

Im zweiten Schritt werden die restlichen Knoten betrachtet. Für diese muss nun entschieden werden, ob sie zum *Preprocessing* oder zum *Postprocessing* gehören. Zu diesem Zweck betrachtet der Algorithmus die relative Position jedes verbleibenden Knotens im Verhältnis zum Beginn beziehungsweise Ende des zuvor identifizierten Quanten-Teils. Aufgrund der in Annahme A1 beschriebenen Struktur des Skript-basierten Quellcodes müssen diejenigen Schritte der Implementierung, welche zuerst im Quellcode stehen, auch zuerst ausgeführt werden. Dementsprechend gehört ein Statement genau dann zum *Preprocessing*, wenn dessen Index kleiner ist als der Beginn des Quanten-Teils. Analog gehört ein Statement genau dann zum *Postprocessing*, wenn dessen Index größer ist als das Ende des Quanten-Teils. Der Algorithmus iteriert daher über die verbliebenen Knoten und weist diese entsprechend den genannten Regeln zu.

Als Ergebnis liefert der Algorithmus schließlich eine Aufteilung der Knoten des Syntaxbaums in die drei gewünschten Teile. An dieser Stelle sei angemerkt, dass eine Menge von Knoten in

der konkreten Implementierung wieder in eine valide Programmstruktur überführt werden können.

Nachdem der Algorithmus eine Aufteilung des Quellcodes berechnet hat, müssen die einzelnen Teile daher noch einmal genauer betrachtet werden. Im einzelnen müssen Schleifen und Abhängigkeiten im Sinne gemeinsam genutzter Variablen speziell behandelt werden. Die Details der Implementierung werden in Abschnitt 5.1 noch einmal vertieft.

Für eine Schleife ist hier in erster Linie zu beurteilen, wie sich die enthaltenen Statements mit den Grenzen der einzelnen Teile überschneiden. Insbesondere müssen Fälle behandelt werden, in denen die gefundene Zerlegung des Quellcodes gleichzeitig einen Schnitt durch eine Schleife bedeuten würde. Bei der Behandlung von Schleifen können in diesem Kontext drei relevante Fälle auftreten. Im einfachsten Fall kann eine Schleife komplett innerhalb eines Teils liegen. Hier ist zunächst keine weitere Anpassung nötig. Ein weiterer Fall ergibt sich, wenn eine Schleife mehrere Teile schneidet beziehungsweise in einem Teil beginnt und im nächsten Teil endet. Dieser Fall kann dann auftreten, wenn einige Statements innerhalb der Schleife eine Quantenbibliothek aufrufen und andere nicht. Um die Funktionalität des Quellcodes aufrechtzuerhalten, müssen nun die Grenzen der Teile verschoben werden. Beginn und Ende des Teils *Quantenberechnung* werden daher soweit ausgedehnt dass die gesamte Schleife umschlossen wird. Im dritten und letzten Fall ist die Schleife so gestaltet, dass sie einen Teil komplett umschließt. Genauer enthält die Schleife hierbei exakt diejenigen Statements, aus denen der Algorithmus einen Teil gebildet hat. Eine Anpassung der Grenzen der Teile ist hier nicht notwendig. Die Schleife eignet sich jedoch gut, um später direkt im Workflow modelliert zu werden.

Zu guter Letzt muss definiert werden, wie mit gemeinsam genutzten Variablen verfahren wird. Ein einfaches Beispiel hierfür ist die Zuweisung einer Variable X während des *Preprocessing* und deren Verwendung innerhalb der *Quantenberechnung*. Nach der Aufteilung des Quantum Skripts wäre die Beispiel-Variable sowie der in ihr gespeicherte Wert nur innerhalb des *Preprocessing* bekannt. Es muss also eine Übergabe zwischen den Teilen stattfinden beziehungsweise ein Datenfluss definiert werden. Um dies zu realisieren, wird für jeden Teil eine Menge von benötigten und bereitgestellten Variablen eingeführt. Im Beispiel würde die Variable X folglich vom *Preprocessing* bereitgestellt und andererseits von der *Quantenberechnung* benötigt werden. Der Datenfluss beziehungsweise die Parameter-Übergabe kann dann über die Schnittmenge dieser Variablen-Mengen definiert werden. Ein Teil erhält somit alle benötigten Variablen von seinem Vorgänger.

Algorithmus 4.1 Script Splitting Algorithmus

```
function SPLIT(sourceCode)
    tree ← CONSTRUCTSYNTAXTREE(sourceCode)
    importNodes ← FINDQUANTUMIMPORTNODES(tree)
    for all node ∈ tree do
        if HASREFERENCE(node, importNodes) then
            qcNodes ← qcNodes ∪ node
            if firstVisit then
                firstQcNode ← INDEXOF(node)
                lastQcNode ← INDEXOF(node)
            else
                lastQcNode ← INDEXOF(node)
            end if
        end if
    end for
    for all remainingNode ∈ tree \ qcNodes do
        if INDEXOF(remainingNode) < firstQcNode then
            preNodes ← preNodes ∪ remainingNode
        else if INDEXOF(remainingNode) > lastQcNode then
            postNodes ← postNodes ∪ remainingNode
        end if
    end for
    return preNodes, qcNodes, postNodes
end function
```

Algorithmus 4.2 Hilfsfunktion

```
function FINDQUANTUMIMPORTNODES(tree)
    allImports ← FINDIMPORTNODES(tree)
    quantumImports ← ∅
    for all import ∈ allImports do
        if import ∈ QUANTUMSET then
            quantumImports ← quantumImports ∪ import
        end if
    end for
    return quantumImports
end function
```

4.4 Generierung von Workflow-Elementen

Das Ergebnis des Algorithmus aus Abschnitt 4.3 ist eine Aufteilung eines gegebenen Quantum Scripts in *Preprocessing*, *Quantenberechnung* und *Postprocessing*. Zunächst sind dies jedoch nur Stücke eines Skript-basierten Quellcodes. Es bedarf nun also einer Transformation dieser Teile hin zu einem Workflow.

Ein minimaler Workflow muss im Kontext dieser Arbeit für jeden Teil des Quantum Skripts einen Task beinhalten und diese mit einem passenden Kontrollfluss verbinden. Im BPMN Standard stehen dazu verschiedene Arten von Tasks zur Verfügung [OPM11]. Für den hier beschriebenen Anwendungsfall erscheinen Skript-Tasks und Service-Tasks vielversprechend. Beide Arten bieten die Möglichkeit, beliebige beziehungsweise vom Benutzer spezifizierte Skripts auszuführen. Insofern wären beide Task Arten geeignet um einen Teil des Quantum Skripts abzubilden. Mithilfe eines Skript-Tasks könnte ein (Teil-)Skript direkt in das Modell integriert werden. Bei genauerer Betrachtung fällt jedoch auf, dass ein Skript-Task nur diejenigen Bibliotheken beziehungsweise Umgebungen verwenden kann, welche in der im Hintergrund agierenden Engine vorinstalliert sind. Demnach müsste die Engine also auch spezifische Bibliotheken wie qiskit vollständig unterstützen. Alternativ müssten die vom Quanten Skript abgeleiteten Abhängigkeiten zu externen Bibliotheken durch zusätzlichen Aufwand überwunden werden. Auf der anderen Seite greift ein Service-Task auf die Funktionalität einer externen Implementierung zu. Es müssen daher keine zusätzlichen Abhängigkeiten aufgebaut werden. Ferner können die einzelnen Teile des Quantum Skripts als eigenständige Anwendungen realisiert werden. In der Orchestrierung können diese dann über spezifische Topics mit den passenden Tasks verbunden werden. Aus dieser Überlegung leitet sich die hier gewählte Umsetzung mithilfe von Service-Tasks ab.

Die abgeleiteten Tasks müssen im nächsten Schritt mithilfe eines Kontrollflusses verbunden werden. Dadurch wird insbesondere definiert, in welcher Reihenfolge die Tasks ausgeführt werden. Die Ausführungsreihenfolge der Tasks im Workflow entspricht der Position der Teile im Quantum Skript. Folglich wird als Erstes das *Preprocessing*, dann die *Quanteberechnung* und schließlich das *Postprocessing* ausgeführt. Durch diese Reihenfolge wird zeitgleich der Datenfluss impliziert. Beispielsweise kann im *Preprocessing* eine Vorbereitung der Eingabedaten stattfinden. Die Ausführung des Quantenschaltkreises in der *Quanteberechnung* ist dann signifikant von diesen Daten abhängig. Aus diesem Grund werden bei der Aufteilung des Quantum Skripts für jeden Teil zusätzlich Parameter- und Rückgabe-Listen erstellt. Sobald die Berechnung abgeschlossen ist, kann die Rückgabe eines Teils somit an den betroffenen Task zurückgeschickt werden. Im Workflow wird die Rückgabe anschließend in Form von Variablen an den nächsten Task weitergeleitet.

Ein Quantenalgorithmus beinhaltet typischerweise nicht nur einfach ausgeführte Schritte, sondern auch Schleifen. Um die Vorteile eines Workflows ausnutzen zu können, sollten die Iterationen der Schleife über den Workflow gesteuert werden. In einem Workflow können Schleifen unter Verwendung von Gateways modelliert werden. Beispielsweise kann eine Schleife um einen Task modelliert werden, sodass dieser mehrfach ausgeführt wird. Das Workflow

Modell muss daher um passende Gateways ergänzt werden. Somit entstehen nun also mehrere mögliche Pfade. Es ist deshalb notwendig, die Bedingung für das Betreten eines Pfads korrekt zu spezifizieren. Genauer soll diese Bedingung dabei so definiert werden, dass sie die Schleife des Quantum Scripts ersetzt. Damit wird die Schleife vom Script in den Workflow überführt. Die Bedingung für eine solche Schleife liefert auch hier der Script Splitting Algorithmus.

Über die obige Beschreibung gelangt man schließlich zu einer konzeptuellen Vorlage für die Struktur der generierten Workflow Elemente. Das Modell kann dann automatisiert angepasst werden, sodass es zum Ergebnis des Script Splitting Algorithmus passt. In der folgenden Abbildung 4.3 ist die entstandene Vorlage dargestellt.

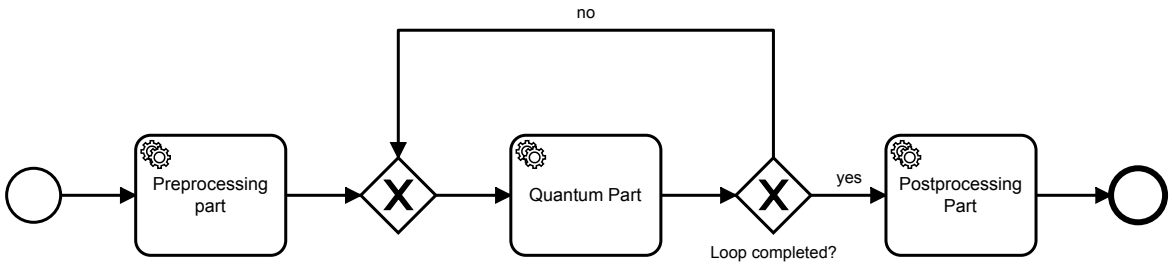

Abbildung 4.3: Vorlage für die generierten Workflow Elemente

Der abgebildete Workflow beginnt wie üblich auf der linken Seite mit einem Start-Event. Entsprechend der hier geschilderten Herleitung folgen anschließend die drei External Service Tasks und das obligatorische End-Event. Im Zentrum steht hierbei die *Quantenberechnung*, welche von einer Schleife umfasst wird. Die Schleife wird mithilfe von zwei Exklusiv-Gateways modelliert. Das hintere Gateway erzeugt hierbei eine Gabelung zwischen *Quantenberechnung* und *Postprocessing*. Das vordere Gateway führt die Pfade schließlich wieder zusammen und es entsteht die gewünschte Schleife.

5 Implementierung

In diesem Kapitel soll gezeigt werden, wie das in Kapitel 4 beschriebene Konzept prototypisch umgesetzt werden kann. Hierzu wird zunächst die Implementierung des entwickelten Algorithmus im Detail präsentiert. Als zweite Komponente wird anschließend das Plugin beschrieben, welches den Script Splitting Algorithmus mit dem Generierungsprozess und dem QuantME Framework verbindet. Zum Abschluss wird gezeigt, wie Code-Fragmente und Workflow Elemente mithilfe von Polling-Agents miteinander interagieren können.

5.1 Implementierung des Script Splitting Algorithmus

Die Implementierung des hier vorgestellten Algorithmus erforderte einigen Aufwand. Im Wesentlichen orientiert sie sich an dem in Kapitel 4 vorgestellten Konzept. Der entstandene Code wurde schließlich auf GitHub veröffentlicht [Ehr].

Am Anfang der Entwicklungsarbeit stand die Auswahl einer geeigneten Programmiersprache. Mit der gewählten Sprache musste es also möglich sein, Quellcode zu analysieren und nach Bedarf zu manipulieren. Da sich die Eingabe des Verfahrens zunächst auf Python-Skripts beschränkte, lag es nahe Python als Sprache für die Implementierung zu wählen. Prinzipiell wären zwar auch andere prominente Programmiersprachen wie Java möglich gewesen, jedoch überzeugte Python mit seiner übersichtlichen Code-Struktur. Hinzukommt, dass Python sehr viele Open-Source-Bibliotheken zur Verfügung stellt. Es kann somit in vielen Fällen auf bewährte Werkzeuge zurückgegriffen werden.

Der entwickelte Algorithmus muss auf eine interne Darstellung des zu analysierenden Quellcodes zurückgreifen können. Wie in Kapitel 2 erläutert, wird hierzu ein Syntaxbaum verwendet. Um diesen zu erstellen wird die Bibliothek RedBaron benutzt [Psy]. Genauer wird der Quellcode als Parameter verwendet, um ein RedBaron-Objekt zu erzeugen. Der Baum wird somit von einem Objekt repräsentiert. Das Objekt stellt nun einige Operationen bereit, mit denen zum Beispiel die Knoten des Baums durchsucht oder manipuliert werden können. Zur Veranschaulichung befindet sich in Listing 5.1 ein entsprechendes Code-Beispiel. Hier wird zunächst das RedBaron-Objekt erzeugt und anschließend eine Operation aufgerufen. Die gezeigte Operation *find_all* durchsucht den Syntaxbaum nach der spezifizierten Klasse von Knoten. Im Beispiel liefert die Operation folglich alle Knoten der Klasse *Import*. Also alle Knoten, die zu einem Import-Statement gehören.

Listing 5.1 Beispielhafte Verwendung der RedBaron Bibliothek

```
1 from redbaron import RedBaron
2
3 red = RedBaron(code_file) # code_file is the file which contains the source-code
4
5 imports = red.find_all("ImportNode")
```

Mithilfe des Syntaxbaums können nun die einzelnen Knoten und damit die einzelnen Statements des Quantum Skripts genauer untersucht werden. Die in Abschnitt 4.3 beschriebene Idee des Algorithmus ist es, Knoten mit Referenzen auf eine Bibliothek für Quantum Computing zu finden. In einem ersten Schritt müssen deshalb die importierten Bibliotheken betrachtet werden. Um eine Quanten-Berechnung durchzuführen, wird üblicherweise ein Aufruf einer solchen Bibliothek stattfinden. Wie in Listing 5.1 gezeigt wurde, können die Import-Statements einfach gefunden werden. Da jedoch offensichtlich nicht alle importierten Bibliotheken für Quanten-Berechnungen genutzt werden, liefert dies zunächst eine Menge von potenziellen Indikatoren. Im Rahmen dieser Arbeit ist die Bibliothek *qiskit* hingegen ein echter Indikator. Um weitere Bibliotheken zu berücksichtigen, kann die in der Implementierung vordefinierte Liste *quantum_set* entsprechend erweitert werden. Die Menge aller gefundenen Imports wird also zwischengespeichert und weiterverarbeitet. Es werden zusätzlich alle Imports nach ihrer Top-Level Bibliothek sortiert. So können auch Aufrufe an untergeordnete Pakete, wie zum Beispiel *qiskit.providers.aer*, erkannt werden. Zusätzlich werden durch diese Gruppierung auch Referenzen auf kompliziertere Imports behandelt. Ferner ist es so möglich, eine Referenz auf die Bibliothek zu erkennen, obwohl deren Name nicht direkt im betroffenen Statement vorkommt. Die Gruppierung löst damit das Problem, alle möglichen Sub-Bibliotheken statisch festzulegen. Vielmehr wird auf Grundlage der Top-Level Bibliothek dynamisch entschieden, welche Referenzen relevante Indikatoren darstellen. Intern werden die Indikatoren letzten Endes in einer Liste verwaltet. Im Rahmen einer Erweiterung könnten diese über eine Benutzereingabe von einem Experten modifiziert werden.

Um festzustellen, an welcher Stelle eine Bibliothek referenziert wird, müssen die referenzierenden Knoten identifiziert werden. Es müssen daher insbesondere diejenigen Knoten beziehungsweise Statements betrachtet werden, welche Aufrufe ausführen. Diese Knoten bezeichnet man als *CallNode*. Innerhalb eines RedBaron-Syntaxbaums werden diese typischerweise in einem *AtomtrailersNode* gespeichert. Analog zum Beispiel aus Listing 5.1 kann auch diese Klasse von Knoten heraus gefiltert werden. Jeder so gefundene Knoten kann jetzt bezüglich der von ihm referenzierten Bibliothek analysiert werden. Hierbei wird überprüft, ob in einem Aufruf einer der Indikatoren vorkommt. Dies geschieht durch einen String-Vergleich des Aufrufs mit den gruppierten Bibliotheken. Nachdem nun alle *CallNodes* gemäß ihrer Reihenfolge im Quellcode analysiert wurden, steht auch implizit fest, welches der erste beziehungsweise letzte Knoten der *Quantenberechnung* ist. Dies bildet das Fundament für die Zerlegung des Quantum Skripts in die drei gewünschten Teile.

Damit die Semantik des ursprünglichen Quellcodes erhalten bleibt, müssen als Nächstes die eventuell vorhandenen Schleifen behandelt werden. Wie alle Knoten bilden auch Schleifen eine

Klasse von Knoten im Syntaxbaum. Folglich können diese gefunden und modifiziert werden. Für die Erhaltung der Semantik sind die Schnittpunkte der Schleifen mit den Grenzen der Teile von essenzieller Bedeutung. Ferner muss verhindert werden, dass eine Schleife beim Zerlegen des Quellcodes zerrissen wird. Gemäß den in Abschnitt 4.3 beschriebenen Regeln werden demnach Beginn und Ende der *Quantenberechnung* verschoben. Technisch wird dies bewerkstelligt, indem der erste beziehungsweise letzte Knoten der *Quantenberechnung* mit den Knoten der Schleife verglichen wird. So kann zum Beispiel der Knoten, welcher die Schleife einleitet, zum neuen Beginn des Quanten-Teils werden. Die Behandlung verschachtelter Schleifen kann hierbei rekursiv erfolgen. Somit verbleibt eine innere Schleife schließlich innerhalb des Teils, während die äußere Schleife speziell behandelt wird. Damit später eine geeignete Orchestrierung in einem Workflow möglich ist, wird zusätzlich die Schleifen-Bedingung der äußeren Schleife abgefragt und gespeichert. Außerdem muss zeitgleich dafür gesorgt werden, dass die Kontrolle über die Iterationen einer Schleife an den Workflow abgegeben werden kann. Demzufolge soll die Schleifenbedingung durch einen Parameter definiert werden können. Zunächst wird dazu die Schleifen-Variable in einen Parameter verschoben. Zusätzlich wird die Abbruchbedingung der Schleife so angepasst, dass die Schleife nur genau ein mal betreten wird. Somit ist es möglich den Wert der Schleifen-Variable von außen zu spezifizieren, ohne die Struktur des Quellcodes zu verändern.

Die Aufteilung des Quellcodes bedeutet letztlich auch, dass Daten und Variablen von einem Teil in den Nächsten übertragen werden müssen. Es ist daher notwendig jeden Teil in eine Form zu bringen, die eine solche Übergabe ermöglicht. Die hier vorgestellte Implementierung bettet daher alle Teile in eine eigene Python-Funktion ein. Die Ausführung eines Teils kann somit durch einen Aufruf der entsprechenden Funktion realisiert werden. Darüber hinaus können dynamische Parameter- beziehungsweise Rückgabe-Listen generiert werden. Um diese Listen zu erstellen, werden alle Zuweisungen betrachtet, welche innerhalb eines Teils stattfinden. In der internen Datenstruktur ist eine Zuweisung als *AssignmentNode* zu finden. Die Zuweisung enthält hierbei immer zwei Teile. Diese sind zum einen das Ziel der Zuweisung und andererseits der Wert. Für den Datenfluss ist insbesondere das Ziel einer Zuweisung relevant. Die Variable können daher identifiziert werden, indem die Ziele der *AssignmentNodes* abgefragt werden. Wenn ein Knoten eines Teils eine Zuweisung verkörpert, so wird folglich das Ziel dieser Zuweisung in die Rückgabe-Liste aufgenommen. Duplikate werden in dieser Liste explizit vermieden. Die Parameter-Liste eines Teils wird der Einfachheit halber mit der Rückgabe-Liste des vorhergehenden Teils gleichgesetzt. Es sind daher immer alle Variablen des Vorgängers verfügbar.

Um das Quantum Skript schließlich zu zerlegen, werden für die einzelnen Teile neue RedBaron-Objekte erzeugt und mit den passenden Knoten gefüllt. In Listing 5.2 ist die Erstellung eines solchen Teils exemplarisch abgebildet. Dort wird als Erstes ein RedBaron-Objekt mit der zunächst leeren Funktion *quantum* konstruiert (Zeile 2). In der Schleife werden die zuvor identifizierten Knoten in die neue Instanz kopiert (Zeile 5-7). Anschließend werden die bereitgestellten Variablen berechnet und in das Return-Statement der Funktion aufgenommen (Zeile 8-11). Da der Syntaxbaum an dieser Stelle keine Liste erwartet, müssen die Variablen kurzfristig zu einer String-Kette transformiert werden (Zeile 11). Daraufhin werden die benö-

tigten Variablen zusammen mit der Schleifenbedingung in die Parameter-Liste eingefügt (Zeile 13-16). Zum Abschluss werden die importierten Bibliotheken an den Anfang des neuen Teils kopiert (Zeile 18-19). Der generierte Teil beinhaltet nun alle nötigen Elemente, um selbstständig ausgeführt zu werden.

Listing 5.2 Code-Ausschnitt für die Erstellung des Teils *Quantenberechnung*

```
1     # define part as redBaron instance
2     qc_part_code = RedBaron("def quantum(): return 0")
3
4     qc_part_code[0].value = ""
5     for i in range(first_qc_index, last_qc_index + 1):
6        # +1 is mandatory to include the last node due to indexing reasons
7        qc_part_code[0].value.append(str(red[i].copy()))
8     post_req = __get_prov_vars(qc_part_code)
9     # add return statement with matching arguments
10    qc_part_code[0].value.append("return ")
11    qc_part_code[0].value[-1].value = str(post_req)
12    # add arguments to function, especially the loop-condition
13    quantum_req = __get_prov_vars(pre_part_code)
14    quantum_req.append(condition[0])
15    for arg in quantum_req:
16       qc_part_code[0].arguments.append(arg)
17    # add imports
18    for im in import_nodes:
19       qc_part_code.insert(0, im.copy())
```

Um Aufrufe entgegen zu nehmen enthält die Implementierung des Algorithmus zusätzlich eine Schnittstelle in Form einer Flask-App [Gri18]. Über die Schnittstelle können somit beispielsweise HTTP-POST-Anfragen empfangen werden. Mithilfe einer solchen Anfrage können insbesondere die Eingabe-Daten übermittelt werden. Im hier beschriebenen Fall wird die Schnittstelle von der Komponente innerhalb des Camunda-Plugins entsprechend verwendet, um den Algorithmus aufzurufen.

5.2 Implementierung des Plugins

Die in dieser Arbeit entwickelte Softwarelösung soll es ermöglichen, Workflows zu generieren. Die in Abschnitt 5.1 beschriebene Implementierung ermöglicht es, gegebenen Quellcode zu analysieren und diesen zu zerlegen. Die resultierenden Teile müssen nun in einen Workflow überführt werden. Hierzu wurde ein Plugin-basiertes Framework entwickelt, welches sich in Camunda beziehungsweise die Erweiterung QuantME eingliedert. Im entsprechenden GitHub Projekt ist der entstandene Code in der Branch *qsplitter* zu finden. Die einzelnen Teile des entwickelten Plugins werden im folgenden Abschnitt der Reihe nach vorgestellt.

Um den Generierungsprozess starten zu können, muss es eine Schnittstelle für den Benutzer geben. QuantME stellt bereits eine grafische Benutzeroberfläche für die Modellierung von Workflows zur Verfügung. Der erste Teil des Plugins erweitert daher die Toolbar dieser Benutzeroberfläche. In Abbildung 5.1 befindet sich ein Screenshot der erweiterten Benutzeroberfläche. Es ist die standardmäßige Oberfläche von QuantME zu sehen, wobei sich am rechten Ende der Toolbar ein neuer Button befindet. In der Grafik ist dieser leicht hervorgehoben. Mit einem Klick auf den Button kann der Script Splitting Algorithmus aufgerufen werden. Als Bestätigung erscheint zunächst die Info-Box, welche in der Abbildung mittig zu sehen ist. Um die Eingabe zu empfangen, kann an dieser Stelle eine Benutzereingabe implementiert werden. Der Prototyp führt hingegen direkt einen Aufruf mit vordefinierten Daten aus.

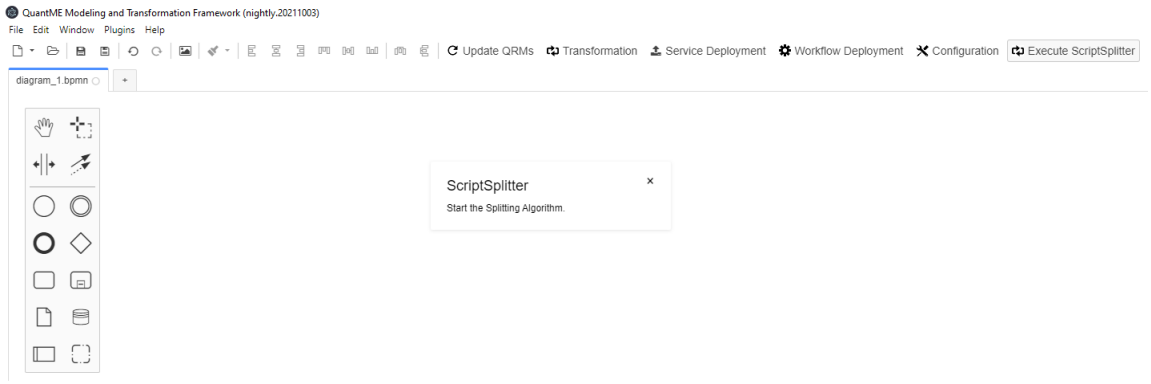

Abbildung 5.1: Screenshot der grafischen Benutzeroberfläche

Das Plugin übernimmt neben der grafischen Darstellung auch einen Teil der Logik des Verfahrens. Zwar liefert die Implementierung aus Abschnitt 5.1 bereits alle notwendigen Fragmente, jedoch müssen noch passende Workflow Elemente generiert und orchestriert werden.

Wie bereits in Abschnitt 4.4 angedeutet wurde, stützt sich die Generierung der einzelnen Elemente zunächst auf eine Vorlage. Um diese Vorlage umzusetzen, wurde ein Hilfsskript entwickelt. Ein Ausschnitt dieses Skripts kann in Listing 5.3 betrachtet werden.

Zu sehen ist ein Teil der Funktion *createTemplate*. Die Funktion nutzt mit Hilfe des übergebenen *modeler*-Objekts die API von Camunda, um mehrere BPMN Elemente zu erstellen. Zunächst werden dazu einige Hilfsvariablen eingeführt, um einfacher mit der API interagieren zu können. Anschließend werden zufällige Topic-Namen generiert. Im weiteren Verlauf werden Workflow und Skript Teile mithilfe dieser Topics kommunizieren. Als Nächstes zeigt der Code-Ausschnitt die Modellierung der neuen Workflow Elemente. Insbesondere kann hier mit Hilfe von *modeling.createShape(...)* eine neues Workflow Element erzeugt werden. In Listing 5.3 wird exemplarisch ein Service Task für den *Preprocessing* Teil generiert. Der neue Task wird anschließend über das zugehörige Business-Objekt entsprechend angepasst. Analog dazu wird

anschließend ein Gateway erstellt. Im letzten Abschnitt werden die generierten Elemente miteinander verbunden. Dies geschieht mit Hilfe der in BPMN üblichen Konnektoren. Es werden also neue Konnektoren erstellt und gleichzeitig passende Start- und Endpunkte definiert. So können mit dem Aufruf aus Zeile 29 die beiden neu generierten Elemente verbunden werden.

Listing 5.3 Code-Ausschnitt für die Erzeugung von Workflow Elementen

```
1  export function createTemplate(modeler) {
2    // initialize modeling helpers
3    let modeling = modeler.get('modeling');
4    let elementRegistry = modeler.get('elementRegistry');
5    let elementFactory = modeler.get('bpmnFactory');
6    // get root element of the current diagram and the root process
7    const definitions = modeler.getDefinitions();
8    const rootElement = getRootProcess(definitions);
9    var process = elementRegistry.get(rootElement.id);
10
11   // get the random topic names
12   var topics = getTopics();
13   // get the (default) start event
14   var startEvent = elementRegistry.get('StartEvent_1');
15
16   var preprocessingTask = modeling.createShape(
17   { type: 'bpmn:ServiceTask' }, { x: 100, y: 100 }, process, {});
18   var preprocessingTaskBo = elementRegistry.get(preprocessingTask.id).businessObject;
19   preprocessingTaskBo.name = 'Preprocessing Part';
20   preprocessingTaskBo.type = 'external';
21   preprocessingTaskBo.topic = topics[1];
22
23   var splittingGateway = modeling.createShape({ type: 'bpmn:ExclusiveGateway' }, { x:
         50, y: 50 }, process, {});
24   var splittingGatewayBo = elementRegistry.get(splittingGateway.id).businessObject;
25   splittingGatewayBo.name = 'Quantum-Loop';
26
27    // connect components
28   modeling.connect(startEvent, preprocessingTask, { type: 'bpmn:SequenceFlow' });
29   modeling.connect(preprocessingTask, splittingGateway, { type: 'bpmn:SequenceFlow' });
30   ...
31   }
```

Der generierte Workflow muss im nächsten Schritt angepasst werden, um das Ergebnis des Script Splitting Algorithmus widerspiegeln zu können. Der Algorithmus liefert schließlich nicht nur eine Zerlegung des übergebenen Quellcodes, sondern auch Informationen über dessen ursprüngliche Struktur. Diese zusätzlichen Informationen werden genutzt, um die generierten Workflow Elemente passend zu orchestrieren. Hierbei sind im Wesentlichen zwei Schritte durchzuführen. Zunächst muss die vorbereitete Schleife eine passende Bedingung erhalten. Wie in Abschnitt 5.1 beschrieben, speichert der Algorithmus die dafür vorgesehene Bedingung während seiner Analyse ab. Folglich kann die Bedingung über einen Parameter übergeben und

in das Modell eingefügt werden. Die Anpassung geschieht dann, analog zur Generierung, über ein Business-Objekt. In diesem Fall wird das Business-Objekt desjenigen Gateways modifiziert, welcher zwischen *Quantenberechnung* und *Postprocessing* steht. Die Schleife im Workflow ersetzt somit die Schleife innerhalb des ursprünglichen Quellcodes. Der hier implementierte Mechanismus überträgt die Bedingung aus dem Quellcode direkt in das Modell. Um an dieser Stelle einen validen BPMN-Ausdruck zu erhalten, muss diese automatisch eingetragene Bedingung noch manuell angepasst werden. Abschließend muss dafür Sorge getragen werden, dass die generierten Tasks mit den einzelnen Teilen des Quantum Skripts verbunden werden. Die Tasks sind hierbei jeweils in Form eines External Task implementiert. Demzufolge können sie über eine Topic mit einer externen Anwendung in Verbindung gesetzt werden. Im Zuge der Generierung der Workflow Elemente werden deshalb auch Topic-Namen erzeugt. Jeder Task wird somit bereits bei der Generierung mit einer eindeutigen Topic ausgestattet. Auf der anderen Seite werden die einzelnen Teile des Quantum Skripts über Funktionsaufrufe angesprochen. Diese Aufrufe übernimmt für jeden Teil ein designierter Polling-Agent. Die Implementierung eines solchen Polling-Agents wird in Abschnitt 5.3 genauer erläutert. Somit kann dieser ebenfalls über eine Topic kommunizieren. Um die Komponenten nun final zu verbinden, muss jedem Polling-Agent der Topic-Name des für ihn vorgesehenen Tasks mitgeteilt werden. Das Plugin legt somit für jeden Agent die passende Topic fest. Anschließend können diese manuell gestartet werden. Die einzelnen Teile des Quantum Skripts stehen nun bereit um über den jeweiligen Polling-Agent aufgerufen zu werden.

5.3 Implementierung der Polling-Agents

In den vorangestellten Abschnitten wurde mehrfach auf den Aufruf der Teile über sogenannte Polling-Agents verwiesen. Die in dieser Arbeit gewählte Implementierung eines solchen Agents soll nun näher betrachtet werden.

Als Polling-Agent wird hier die Implementierung einer Komponente bezeichnet, welche dafür zuständig ist, kontinuierlich nach einem Arbeitsauftrag zu fragen. Diese Komponente implementiert somit das Polling Consumer Pattern [HW04]. Sobald der Polling-Agent bereit ist, stellt er also eine Anfrage an die Camunda-Engine. Die Anfrage beinhaltet dabei insbesondere einen speziellen Topic-Namen. Schließlich soll jeder Agent genau dem für ihn zugeschnittenen Task zugeordnet werden. Wenn der Workflow nun also an einem bestimmten Task ankommt, legt dieser unter der entsprechenden Topic einen Arbeitsauftrag an. Der passende Polling-Agent wird diesen dann entgegennehmen. Sobald dies geschehen ist, kann er einen Teil des zerlegten Quantum Skripts aufrufen. Die für den Aufruf notwendigen Daten erhält der Agent über die Variablen des mit der Topic verbundenen External-Tasks. Nachdem der Funktionsaufruf abgeschlossen ist, liegt dem Polling-Agent schließlich auch der entsprechende Rückgabewert vor. Als Ergebnis kann dieser Rückgabewert nun wiederum an die Camunda-Engine geschickt werden.

Die Implementierung des Polling Consumer Patterns stellt in diesem Zusammenhang nicht die einzig mögliche Lösung dar. So wäre es beispielsweise auch möglich gewesen, die einzelnen Teile des Quantum Skripts explizit aufzurufen. Also etwa eine Art von REST-Schnittstelle zu verwenden. In diesem Fall könnte jeder Teil, an der entsprechenden Stelle, direkt von der Camunda-Engine aufgerufen werden. Es stellt sich jedoch heraus, dass die Camunda-Engine diese Art von direkten Aufrufen nicht ohne weiteres anbietet. Insbesondere erfordert dies also zusätzlichen Implementierungsaufwand, wohingegen External-Tasks von Camunda bereits unterstützt werden. Aus diesem Grund wird in dieser Arbeit auf den bewährten Lösungsansatz des Polling Consumers zurückgegriffen.

In Listing 5.4 ist exemplarisch die Implementierung des Polling-Agents für die *Quantenberechnung* angegeben. Zu sehen ist hierbei zunächst die Definition der *poll* Operation. In dieser Operation wird gemäß dem zuvor geschilderten Schema eine Anfrage an die Camunda-Engine gestellt und die erhaltene Antwort anschließend verarbeitet. In Zeile 23 findet dementsprechend der Aufruf der *Quantenberechnung* statt. Das Ergebnis der Berechnung wird schließlich an die Camunda-Engine und damit zum passenden Task zurückgeschickt.

Listing 5.4 Code-Ausschnitt eines Polling-Agents

```
    import sys
    import threading
    import requests
    import quantumPart

    def poll():

       body = {
          "workerId": "QuantumPollingAgent",
          "maxTasks": 1,
          "topics":
             [{"topicName": topic,
               "lockDuration": 100000000
               }]
       }

       response = requests.post(pollingEndpoint + '/fetchAndLock', json=body)

    if response.status_code == 200:
       for externalTask in response.json():
          variables = externalTask.get('variables')
          values = []
          for var in variables.values():
             values.append(var["value"])

          # call the wrapper function of the pre part
          result = quantumPart.quantum(*values)

          # provided variables
          prov_variables = dict()
          for var in result:
             prov_variables[var] = {"value": var}

          body = {
             "workerId": "QuantumPollingAgent",
             "variables": prov_variables
          }

          response = requests.post(pollingEndpoint + '/' + externalTask.get('id') +
              '/complete', json=body)

    threading.Timer(20, poll).start()

pollingEndpoint = sys.argv[1]
topic = sys.argv[2]
poll()
```

5.4 Anwendungsbeispiel

In diesem Anwendungsbeispiel wird aus einer gegebenen Implementierung eines Quantenalgorithmus mithilfe des hier entwickelten Verfahrens ein Workflow generiert. Dabei wird der gegebene Quellcode zunächst mithilfe des Script Splitting Algorithmus in mehrere Teile zerlegt. Anschließend werden die generierten Teile automatisiert in einem Workflow orchestriert.

Eingabe

Als Eingabe erwartet das Verfahren die Skript-basierte Implementierung eines Quantenalgorithmus. Die hierfür exemplarisch gewählte Quellcode kann in Listing 5.5 betrachtet werden. Es handelt sich hier um eine Konstruktion eines kleinen Quantenalgorithmus und dessen Ausführung. Der hier gewählte Beispiel-Quellcode stammt zum Teil aus dem Qiskit-Textbook [ACB+20].

Die Eingabe ist nun also ein Quantum Skript, welches mehrere Abschnitte beinhaltet. Zunächst werden einige Bibliotheken importiert (Zeile 2 ff.). Anschließend werden die Daten für die Berechnung vorbereitet. Es ist zu sehen, wie die Anzahl der benötigten Bits beziehungsweise *qubits* festgelegt (Zeile 12 ff.). Als nächstes werden Schaltkreise für die Berechnung und die Messung erstellt. Hierbei wird ein Schaltkreis für die Kalibrierung erstellt (Zeile 25 ff.). Es wird ein Backend für die Ausführung beziehungsweise für die Simulation ausgewählt (Zeile 28). Der erste Schaltkreis kann nun bereits ausgeführt werden (Zeiel 29). Aus dem so berechneten Zwischenergebnis wird anschließend eine Matrix für die Kalibrierung abgeleitet (Zeile 32). In den folgenden Schritten wird der Quantenschaltkreis initialisiert und mit einigen Gates versehen (Zeile 34 ff). Schließlich wird auch der Quantenschaltkreis an das Backend geschickt und somit ausgeführt (Zeile 43). Nachdem die Ausführung abgeschlossen ist, werden die Ergebnisse abgerufen (Zeile 44). Das Ergebnis kann jetzt weiter interpretiert oder auch weiter verarbeitet werden. In diesem Beispiel findet zum Abschluss eine Ausgabe des Ergebnis statt (Zeile 49).

Die konstruierte Eingabe enthält verschiedene typische Elemente einer Implementierung eines Quantenalgorithmus. Insbesondere sind auch mehrere Schleifen vorhanden. Die Schleifen können in einer praktischen Anwendung beispielsweise genutzt werden, um eine Vielzahl von Ausführungen des Quantenalgorithmus zu erreichen. In diesem Beispiel wird die gesamte Berechnung mehrfach wiederholt.

Ausgabe

Im folgenden wird die aus dem Beispiel entstandene Ausgabe beziehungsweise das Ergebnis des Script Splitting Algorithm präsentiert.

Listing 5.5 Code-Ausschnitt eines beispielhaften Quantum Skripts

```
# Import Qiskit classes
import qiskit
from qiskit import QuantumRegister, QuantumCircuit, ClassicalRegister
from qiskit.providers.aer import noise # import AER noise model

# Measurement error mitigation functions
from qiskit.ignis.mitigation.measurement import (complete_meas_cal,
                                     CompleteMeasFitter,
                                     MeasurementFilter)

# Calculate parameters
number_of_independent_results = 5
qubits = 5
classical_bits = 3

for i in range(number_of_independent_results):
   # Generate a noise model for the qubits
   noise_model = noise.NoiseModel()
   for qi in range(5):
      read_err = noise.errors.readout_error.ReadoutError([[0.75, 0.25], [0.1, 0.9]])
      noise_model.add_readout_error(read_err, [qi])

   # Generate the measurement calibration circuits
   # for running measurement error mitigation
   qr = QuantumRegister(qubits)
   meas_cals, state_labels = complete_meas_cal(qubit_list=[2, 3, 4], qr=qr)
   # Execute the calibration circuits
   backend = qiskit.Aer.get_backend('qasm_simulator')
   job = qiskit.execute(meas_cals, backend=backend, shots=1000,
       noise_model=noise_model)
   cal_results = job.result()
   # Make a calibration matrix
   meas_fitter = CompleteMeasFitter(cal_results, state_labels)
   # Make a 3Q GHZ state
   cr = ClassicalRegister(qubits)
   ghz = QuantumCircuit(qr, cr)
   ghz.h(qr[2])
   ghz.cx(qr[2], qr[3])
   ghz.cx(qr[3], qr[4])
   ghz.measure(qr[2], cr[0])
   ghz.measure(qr[3], cr[1])
   ghz.measure(qr[4], cr[2])
   # Execute the GHZ circuit (with the same noise model)
   job = qiskit.execute(ghz, backend=backend, shots=1000, noise_model=noise_model)
   results = job.result()

# Results without mitigation
counts = results.get_counts()
# interpret the results or do some further processing
# ...
print("Results: ", counts)
```

Der Algorithmus hat das gegebene Quantum Skript in drei Teile aufgespalten. In Listing 5.6 ist der entsprechenden Code-Ausschnitt des ersten Teils zu sehen. Der erste Teil (*Preprocessing*) beinhaltet hierbei denjenigen Abschnitt, welcher noch aus klassischen Berechnungen und Vorbereitungen besteht. In diesem Beispiel ist dies also die Definition der Anzahl der benötigten *Qubits*. Wie in Kapitel 5 beschrieben, wird der extrahierte Quellcode von einer passenden Funktion eingeschlossen. Insbesondere wird hierbei eine Rückgabe generiert, welche die Variablen für die nachfolgenden Teile bereitstellt. In Listing 5.6 ist diese im abschließenden Return-Statement zu sehen (Zeile 29).

Listing 5.6 Code-Ausschnitt des resultierenden Preprocessing Teils

```
1  import ...
2
3  def pre():
4
5
6      # Calculate parameters
7      number_of_independent_results = 5
8      qubits = 5
9      classical_bits = 3
10
11     return [number_of_independent_results, qubits, classical_bits]
```

Der zweite Teil (*Quantenberechnung*) verkörpert die Aufrufe einer Bibliothek des Quantum-Computing. Der passende Code-Ausschnitt befindet sich in Listing 5.2. Die Struktur aller Teile ist im wesentlichen sehr ähnlich. Im unterschied zum ersten Teil enthält die umschließende Funktion *quantum* hier jedoch zusätzlich eine Parameter-Liste. Über diese Parameter hat die *Quantenberechnung* in diesem Beispiel Zugriff auf die zuvor definierte Anzahl der Bits. Bei genauerer Betrachtung des abgebildeten Quellcodes wird ein Unterschied zur ursprünglichen Eingabe deutlich. So wurde die äußere Schleife beziehungsweise deren Bedingung modifiziert (Zeile 5). Durch diese Modifikation wird die Schleife nur noch einmal durchlaufen. Um trotzdem Zugriff auf die in der Schleifenbedingung verwendete Variable zu haben, muss diese nun von außen zugeführt werden. Der Algorithmus hat daher einen zusätzlichen Parameter *i* für die enthaltene Schleifenbedingung generiert.

Listing 5.7 Code-Ausschnitt des resultierenden Quanten Teils

```
import ...

def quantum(number_of_independent_results, qubits, classical_bits, i, ):

   for _loop_dummy in range(1):
      # Generate a noise model for the qubits
      noise_model = noise.NoiseModel()
      for qi in range(5):
         read_err = noise.errors.readout_error.ReadoutError([[0.75, 0.25], [0.1,
             0.9]])
         noise_model.add_readout_error(read_err, [qi])

      # Generate the measurement calibration circuits
      # for running measurement error mitigation
      qr = QuantumRegister(qubits)
      meas_cals, state_labels = complete_meas_cal(qubit_list=[2, 3, 4], qr=qr)

      # Execute the calibration circuits
      backend = qiskit.Aer.get_backend('qasm_simulator')
      job = qiskit.execute(meas_cals, backend=backend, shots=1000,
          noise_model=noise_model)
      cal_results = job.result()

      # Make a calibration matrix
      meas_fitter = CompleteMeasFitter(cal_results, state_labels)

      # Make a 3Q GHZ state
      cr = ClassicalRegister(classical_bits)
      ghz = QuantumCircuit(qr, cr)
      ghz.h(qr[2])
      ghz.cx(qr[2], qr[3])
      ghz.cx(qr[3], qr[4])
      ghz.measure(qr[2], cr[0])
      ghz.measure(qr[3], cr[1])
      ghz.measure(qr[4], cr[2])

      # Execute the GHZ circuit (with the same noise model)
      job = qiskit.execute(ghz, backend=backend, shots=1000, noise_model=noise_model)
      results = job.result()


   return [noise_model, read_err, qr, meas_cals, state_labels, backend, job,
       cal_results, meas_fitter, cr, ghz, results, number_of_independent_results,
       qubits, classical_bits]
```

Der dritte Teil spiegelt das *Postprocessing* wider. Der entsprechende Code-Ausschnitt ist in Listing 5.8 angegeben. Dieser Teil beinhaltet ebenfalls eine Parameter- und eine Rückgabe-Liste. Es ist festzustellen, dass die Parameter-Liste aus den Rückgabe-Listen der vorangestellten Teile

abgeleitet wurde. Innerhalb des Teils findet die Aufbereitung der Ergebnisse statt. In diesem Beispiel wird hierbei das Ergebnis der Berechnungen ausgegeben. Über das abschließende Return-Statement wird das Ergebnis außerdem als Rückgabewert der Funktion zurückgegeben.

Listing 5.8 Code-Ausschnitt des resultierenden Postprocessing Teils

```
1  import ...
2
3  def post(noise_model, read_err, qr, meas_cals, state_labels, backend, job,
       cal_results, meas_fitter, cr, ghz, results, number_of_independent_results,
       qubits, classical_bits, ):
4
5      counts = results.get_counts()
6      # solve the linear system
7      # ...
8      print("Results: ", counts)
9      return [counts]
```

Der Script Splitting Algorithmus konnte das gegebene Beispiel demnach in drei Teile aufspalten. Die Transformation zu einem Workflow erfolgt nun in einem zweiten Schritt. Nachdem der Algorithmus erfolgreich ausgeführt wurde, wird eine Antwort an die aufrufende Komponente des Camunda-Plugins geschickt. Wie in Kapitel 5 beschrieben, wird dann zunächst eine Workflow-Vorlage generiert. Mithilfe der Antwort des Algorithmus wird diese anschließend angepasst werden. Insbesondere wird hierbei die Bedingung für die Schleife entsprechend gesetzt.

Zur Veranschaulichung befindet sich in Abbildung 5.2 ein passender Screenshot. In der Mitte des Abbildung ist der generierte Workflow zu sehen. Die einzelnen Teile werden durch jeweils einen Service-Task repräsentiert.

Die zuvor besprochene Schleife ist in Form des Kontrollflusses zwischen den beiden Gateways abgebildet. Auf der rechten Seite der Abbildung ist das *Properties Panel* von Camunda eingeblendet. Hier können einzelne Elemente des Workflows genauer inspiziert werden. In der Abbildung ist hierbei bereits die Schleife beziehungsweise der entsprechende Konnektor ausgewählt. Die Schleifenbedingung befindet sich daher im Feld *Expression*. Es ist zu erkennen, dass der Ausdruck $i < (number\_of\_independent\_results)$ der ursprünglichen Schleifenbedingung im Quellcode entspricht.

In Kapitel 5 wurde beschrieben, wie die einzelnen Teile des Quantum Skripts unter Verwendung von Topics mit den entsprechenden Service-Tasks verbunden werden können. Hierzu steht für jeden Teil ein Polling-Agent bereit. Die Polling-Agents müssen jedoch noch gestartet werden. Diese Aufgabe erfolgt nach aktuellem Stand der Entwicklung manuell. Jeder Polling-Agent bekommt dabei den Endpoint der aktiven Camunda-Engine und die passende Topic als Parameter übergeben.

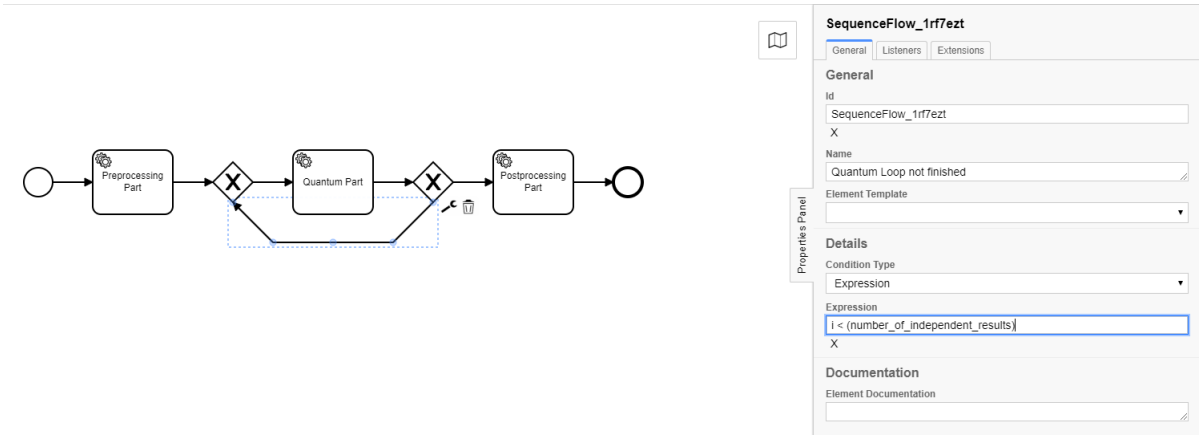

Abbildung 5.2: Screenshot des generierten Workflows in Camunda

6 Related Work

Das Thema Quantum Workflows ist Gegenstand aktueller Forschung. So gibt es Arbeiten rund um den Lebenszyklus von Quantum Software [WBLV21] [WBL+20]. Ferner wurde von Weder et al. eine Technik vorgeschlagen, um Quantum Computing in Workflows zu integrieren [WBLW20]. Mithilfe des MODULO Frameworks wird die Modellierung von Quantum Workflows erleichtert [WBL21]. Insbesondere wird hierbei eine integrierte Toolchain vorgeschlagen, um Modelle von Quantum Workflow in native Modelle zu transformieren. So wird die Portabilität des Workflows sichergestellt und gleichzeitig dessen Deployment unterstützt [WBL21]. Diese Technik zeigt, wie es möglich ist, Elemente des Quantum Computing innerhalb von Workflows abzubilden. Jedoch liegt der Fokus dieser Arbeit auf der Erweiterung der Modellierungsmöglichkeiten und nicht auf einer entsprechenden automatisierten Generierung entsprechender Elemente eines Workflows. Yang et al. präsentieren in ihrer Arbeit ein Service-orientiertes Framework für die Durchführung von quantenmechanischen Simulationen von Materialeigenschaften [YBD+10]. Diese Arbeit spezialisiert sich auf die Simulation von Materialien und insbesondere auf die Simulation von Modellen im atomaren Maßstab. Das Service-orientierten Frameworks unterstützt dabei das Management aller zu einer Simulation notwendigen Teilaufgaben [YBD+10]. Für die automatisierte Koordination der Teilaufgaben wird hier wiederum ein Workflow eingesetzt. Um den Workflow letztlich zu konstruieren, wird eine grafische Benutzeroberfläche zur Verfügung gestellt. Die Erstellung der beteiligten Workflow Elemente geschieht somit manuell. PennyLane ist ein Open-source Framework, welches entwickelt wurde, um die Optimierung von Quantenalgorithmen und *hybrid quantum-classical*--Algorithmen zu erleichtern [BIS+18]. Das verwendete Modell beschreibt einen hybriden Algorithmus mithilfe von Quanten- und Klassischen-Knoten. Die Knoten sind dabei für die jeweiligen Abschnitte der Berechnung eines Optimierungsproblems zuständig [BIS+18]. Diese Arbeit beschäftigt sich also mit der Charakterisierung bestimmter Teile eines Optimierungsverfahrens. Das beschriebene Modell wird hierbei genutzt um die Abschnitte einer Berechnung abzubilden und nicht um diese innerhalb einer gegebenen Implementierung automatisiert zu erkennen.

Die Generierung von Workflows wurde von Chun et al. untersucht [CAA02]. Diese Untersuchung stützt sich auf spezifisches Wissen einer Domäne. Außerdem liegt der Schwerpunkt hierbei auf der Generierung klassischer Workflows. In der Arbeit von Baranowski et al. wird eine Analyse von (Ruby-)Skripts beschrieben [BBBM12]. Diese soll es ermöglichen, Abhängigkeiten zwischen Methoden zu detektieren und schließlich eine Repräsentation in Form eines Workflows zu finden. Nichtsdestotrotz bleibt hierbei die Lücke zwischen klassischen Algorithmen und Berechnungen auf einem Quantencomputer bestehen.

7 Zusammenfassung und Ausblick

Eine bereits existierende Implementierung eines Quantenalgorithmus in einem Workflow umzusetzen erfordert großen Zeitaufwand weitreichende Kenntnisse aus verschiedenen Domänen. Das Ziel dieser Arbeit war es daher, ein Verfahren zur automatisierten Generierung von Quantum Workflows zu entwickeln. Hierzu wurde ein Algorithmus konzipiert, welcher es unter bestimmten Annahmen erlaubt, gegebene Implementierungen von Quantenalgorithmen zu analysieren und die für das Quantum-Computing relevanten Teile zu identifizieren. Ferner kann eine gegebene Implementierung somit in mehrere Teile zerlegt werden. Es wurde außerdem eine Lösung aufgezeigt, um die einzelnen Teilstücke des Quantenalgorithmus anschließend in einem Workflow abzubilden. Im zweiten Teil der Arbeit wurde das entwickelte Verfahren durch eine eigene Implementierung umgesetzt. Anschließend wurde die Implementierung als Plugin in das QuantME Framework eingegliedert. Zum Abschluss der Arbeit wurde anhand eines Anwendungsbeispiels exemplarisch gezeigt, wie das entwickelte Verfahren angewandt werden kann. Es ist nun also möglich, aus einem gegebenen Quantenalgorithmus automatisiert einen Workflow zu generieren.

Ausblick

Das vorgeschlagene Konzept zur Analyse von implementierten Quantenalgorithmen ist durch mehrere Annahmen eingeschränkt. Um das Verfahren weiterzuentwickeln, können diese Annahmen abgeschwächt werden. Im Zuge zukünftiger Forschung kann somit auch der hier implementierte Algorithmus erweitert werden. So kann beispielsweise die Menge der unterstützen Quanten-Bibliotheken erweitert werden. Ferner wäre es möglich den Overhead beim Datenfluss zwischen den Workflow Elementen zu verringern. Hierzu kann die Auswahl der zwischen den Elementen übergebenen Variablen verfeinert werden. Zusätzlich kann die Implementierung des Algorithmus ergänzt werden, um eine Interaktion mit dem Benutzer zu erlauben. Es wäre dann möglich, einem Experten die Chance zu geben, spezifische Anforderung im Hinblick auf die verwendeten Quantenbibliotheken einzubringen. Eine weitere mögliche Erweiterung ist es, die Eingabe mehrerer Skripts zu unterstützen. So können mehrere Quantenalgorithmen kombiniert und entsprechend komplexere Workflows generiert werden. Der Generierungsprozess muss hierzu angepasst werden, um alle entstandenen Workflow Elemente passend miteinander zu verbinden.

Literaturverzeichnis

[ACB+20] A. Asfaw, A. Corcoles, L. Bello, Y. Ben-Haim, M. Bozzo-Rey, S. Bravyi, N. Bronn, L. Capelluto, A. C. Vazquez, J. Ceroni, R. Chen, A. Frisch, J. Gambetta, S. Garion, L. Gil, S. D. L. P. Gonzalez, F. Harkins, T. Imamichi, H. Kang, A. h. Karamlou, R. Loredo, D. McKay, A. Mezzacapo, Z. Minev, R. Movassagh, G. Nannicini, P. Nation, A. Phan, M. Pistoia, A. Rattew, J. Schaefer, J. Shabani, J. Smolin, J. Stenger, K. Temme, M. Tod, S. Wood, J. Wootton. *Learn Quantum Computation Using Qiskit*. 2020. URL: http://community.qiskit.org/textbook (zitiert auf S. 19, 42).

[AK15] G. Aagesen, J. Krogstie. „BPMN 2.0 for modeling business processes“. In: *Handbook on Business Process Management 1*. Springer, 2015, S. 219–250 (zitiert auf S. 21).

[BBBM12] M. Baranowski, A. Belloum, M. Bubak, M. Malawski. „Constructing workflows from script applications“. In: *Scientific Programming* 20.4 (2012), S. 359–377 (zitiert auf S. 49).

[BBD+09] H. J. Briegel, D. E. Browne, W. Dür, R. Raussendorf, M. Van den Nest. „Measurement-based quantum computation“. In: *Nature Physics* 5.1 (2009), S. 19–26 (zitiert auf S. 18).

[BIS+18] V. Bergholm, J. Izaac, M. Schuld, C. Gogolin, M. S. Alam, S. Ahmed, J. M. Arrazola, C. Blank, A. Delgado, S. Jahangiri et al. „Pennylane: Automatic differentiation of hybrid quantum-classical computations“. In: *arXiv preprint arXiv:1811.04968* (2018) (zitiert auf S. 49).

[BLF+20] J. Barzen, F. Leymann, M. Falkenthal, D. Vietz, B. Weder, K. Wild. „Relevance of near-term quantum computing in the cloud: a humanities perspective“. In: *International Conference on Cloud Computing and Services Science*. Springer. 2020, S. 25–58 (zitiert auf S. 15, 23).

[BYM+98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier. „Clone detection using abstract syntax trees“. In: *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE. 1998, S. 368–377 (zitiert auf S. 19).

[CAA02] S. A. Chun, V. Atluri, N. R. Adam. „Domain knowledge-based automatic workflow generation“. In: *International Conference on Database and Expert Systems Applications*. Springer. 2002, S. 81–93 (zitiert auf S. 49).

[DJ92] D. Deutsch, R. Jozsa. „Rapid solution of problems by quantum computation“. In: *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 439.1907 (1992), S. 553–558 (zitiert auf S. 17, 18).

[Ehr] T.-J. Ehret. *ScriptSplitter*. https://github.com/UST-QuAntiL/qscript-splitter (zitiert auf S. 33).

[Ell99] C. A. Ellis. „Workflow technology“. In: *Computer Supported Cooperative Work, Trends in Software Series* 7 (1999), S. 29–54 (zitiert auf S. 21, 22).

[Gri18] M. Grinberg. *Flask web development: developing web applications with python*. Ö'Reilly Media, Inc.", 2018 (zitiert auf S. 36).

[HW04] G. Hohpe, B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004 (zitiert auf S. 39).

[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), S. 044007 (zitiert auf S. 15).

[LB21] F. Leymann, J. Barzen. „Hybrid Quantum Applications Need Two Orchestrations in Superposition: A Software Architecture Perspective“. In: *arXiv preprint arXiv:2103.04320* (2021) (zitiert auf S. 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: *arXiv preprint arXiv:2003.06256* (2020) (zitiert auf S. 15, 17).

[LR99] F. Leymann, D. Roller. *Production workflow: concepts and techniques*. Prentice Hall PTR, 1999 (zitiert auf S. 21, 22).

[MM04] N. Milanovic, M. Malek. „Current solutions for web service composition“. In: *IEEE Internet Computing* 8.6 (2004), S. 51–59 (zitiert auf S. 21).

[NM+19] E. National Academies of Sciences, Medicine et al. *Quantum computing: progress and prospects*. National Academies Press, 2019 (zitiert auf S. 15).

[OPM11] O. Omg, R. Parida, S. Mahapatra. „Business process model and notation (bpmn) version 2.0“. In: *Object Management Group* 1.4 (2011) (zitiert auf S. 21, 22, 31).

[Pre18] J. Preskill. „Quantum computing in the NISQ era and beyond“. In: *Quantum* 2 (2018), S. 79 (zitiert auf S. 15, 17, 18).

[Psy] B. "Psycojoker". *RedBaron*. https://github.com/PyCQA/redbaron (zitiert auf S. 20, 33).

[RP11] E. G. Rieffel, W. H. Polak. *Quantum computing: A gentle introduction*. MIT Press, 2011 (zitiert auf S. 17).

[WBL+20] B. Weder, J. Barzen, F. Leymann, M. Salm, D. Vietz. „The quantum software lifecycle“. In: *Proceedings of the 1st ACM SIGSOFT International Workshop on Architectures and Paradigms for Engineering Quantum Software*. 2020, S. 2–9 (zitiert auf S. 15, 18, 19, 49).

[WBL21] B. Weder, J. Barzen, F. Leymann. „MODULO: Modeling, Transformation, and Deployment of Quantum Workflows“. In: *Proceedings of the 24th International Enterprise Distributed Object Computing Workshop (EDOCW 2021)*. EDOCW. 2021 (zitiert auf S. 49).

[WBLV21] B. Weder, J. Barzen, F. Leymann, D. Vietz. „Quantum Software Development Lifecycle“. In: *arXiv preprint arXiv:2106.09323* (2021) (zitiert auf S. 49).

[WBLW20] B. Weder, U. Breitenbücher, F. Leymann, K. Wild. „Integrating quantum computing into workflow modeling and execution“. In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE. 2020, S. 279–291 (zitiert auf S. 15, 22, 23, 49).

[YBD+10] X. Yang, R. P. Bruin, M. T. Dove, A. Walkingshaw, T. V. Mortimer-Jones, R. Sinclair, D. J. Wilson, V. Milman, T. Donovan. „A service-oriented framework for running quantum mechanical simulations of material properties in a grid environment“. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 40.4 (2010), S. 485–490 (zitiert auf S. 49).

[ZWZ+19] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, X. Liu. „A novel neural source code representation based on abstract syntax tree“. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, S. 783–794 (zitiert auf S. 20).

Alle URLs wurden zuletzt am 08. 10. 2021 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift