



Hochschule für Angewandte Wissenschaften Hamburg  
*Hamburg University of Applied Sciences*

# Bachelorarbeit

Leonhard Dahl

**Simuquant: Ein Simulator für Quantenschaltkreise**

*Fakultät Technik und Informatik  
Studiendepartment Informatik*

*Faculty of Engineering and Computer Science  
Department of Computer Science*

Leonhard Dahl

**Simuquant: Ein Simulator für Quantenschaltkreise**

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik  
am Department Informatik  
der Fakultät Technik und Informatik  
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Thomas Canzler  
Zweitgutachter: Prof. Dr. rer. nat. Stephan Pareigis

Eingereicht am: 21. August 2012

**Leonhard Dahl**

**Thema der Arbeit**

Simuquant: Ein Simulator für Quantenschaltkreise

**Stichworte**

Quantencomputer, Quantenschaltkreis, Simulator, Simulation, Zustandsvektor, Parallelisierung, Scala, Java, Swing, GUI.

**Kurzzusammenfassung**

Aktuelle Implementierungen von Quantencomputern befinden sich in einem experimentellen Stadium und sind nicht allgemein zugänglich. Daher stellen Software-Simulationen im Allgemeinen die einzige Möglichkeit dar, Quantenalgorithmen experimentell zu untersuchen. Diese Bachelorarbeit beschäftigt sich mit der Konzeption und Realisierung eines Simulators für Quantenschaltkreise mit einer graphischen Benutzeroberfläche. Neben einem ausführlichen Grundlagenteil, welcher auch Nichtexperten den Einstieg in die Arbeit ermöglichen soll, wird der Entwicklungsprozess des Simulators, realisiert in Scala, präsentiert.

**Leonhard Dahl**

**Title of the paper**

Simuquant: A Quantum Circuit Simulator

**Keywords**

Quantum computer, quantum circuit, simulator, simulation, state vector, parallel computing, Scala, Java, Swing, GUI.

**Abstract**

Current implementations of quantum computers are in an experimental stage, and are generally not accessible to the public. Therefore, software simulations are in general the only way to explore quantum algorithms experimentally. This bachelor thesis deals with the design and implementation of a simulator for quantum circuits with a graphical user interface. In addition to an extensive introduction to the basics of quantum computation, which allows for non-experts to follow the topic, this thesis presents the development process of the simulator, implemented in Scala.

# Inhaltsverzeichnis

<b>I. Einleitung</b>	<b>1</b>
<b>1. Einführung</b>	<b>2</b>
1.1. Motivation . . . . .	2
1.2. Ziel der Arbeit . . . . .	2
1.3. Abgrenzung . . . . .	3
1.4. Aufbau der Arbeit . . . . .	3
<b>II. Hauptteil</b>	<b>4</b>
<b>2. Grundlagen der Quanteninformatik</b>	<b>5</b>
2.1. Quantenzustände . . . . .	5
2.1.1. Hinweise zur Notation . . . . .	6
2.1.2. Quantenbits . . . . .	6
2.1.3. Superposition . . . . .	8
2.1.4. Verschränkung . . . . .	8
2.1.5. Dekohärenz . . . . .	9
2.2. Berechnungen mit Quantenzuständen . . . . .	10
2.2.1. Quantengatter . . . . .	11
2.2.2. Zusammengesetzte Operatoren . . . . .	17
2.2.3. Pseudo-klassische Operatoren . . . . .	17
2.2.4. Messung . . . . .	17
2.2.5. Das No-Cloning Theorem . . . . .	19
2.3. Quantenalgorithmen . . . . .	19
2.3.1. Deutsch-Algorithmus . . . . .	20
2.3.2. Shor-Algorithmus . . . . .	21
2.3.3. Grover-Algorithmus . . . . .	24
2.3.4. Quantenteleportation . . . . .	25
<b>3. Ähnliche Arbeiten</b>	<b>27</b>
3.1. jQuantum . . . . .	27
3.2. Zeno . . . . .	28
3.3. QCAD 2.0 . . . . .	29

<b>4. Analyse</b>	<b>30</b>
4.1. Simulationsmethode . . . . .	30
4.1.1. Naiver Ansatz . . . . .	31
4.1.2. Optimierte Methode . . . . .	31
4.2. Anforderungen . . . . .	33
4.2.1. Funktionale Anforderungen . . . . .	33
4.2.2. Nicht-funktionale Anforderungen . . . . .	36
4.3. Konzeptuelles Datenmodell . . . . .	37
4.4. Anwendungsfälle . . . . .	40
4.4.1. Standard Functionality . . . . .	40
4.4.2. Edit Circuit . . . . .	41
4.4.3. Stepwise Simulation . . . . .	41
4.5. Konzeption der GUI . . . . .	43
4.5.1. Struktur . . . . .	43
4.5.2. Editor für Quantenschaltkreise . . . . .	44
4.5.3. Zustandsanzeige und Messergebnisse . . . . .	44
<b>5. Entwurf</b>	<b>45</b>
5.1. Technologie . . . . .	45
5.1.1. Scala . . . . .	45
5.1.2. Swing . . . . .	47
5.2. Architektur . . . . .	47
5.2.1. Externe Komponenten . . . . .	48
5.2.2. Core . . . . .	49
5.2.3. Simulator . . . . .	54
5.2.4. GUI . . . . .	57
5.3. Simulationsalgorithmen . . . . .	64
5.3.1. Basiszustände und Indexmanipulation . . . . .	64
5.3.2. Transformation von Teilregistern . . . . .	65
5.3.3. Auswertung von Oracle-Funktionen . . . . .	68
5.3.4. Messung . . . . .	69
5.3.5. QFT . . . . .	70
5.3.6. Grover Diffusionsoperator . . . . .	71
<b>6. Realisierung und Test</b>	<b>72</b>
6.1. Umfang der Implementierung . . . . .	72
6.2. Test . . . . .	72
6.3. Performanceanalyse . . . . .	73
<b>7. Beispielhafte Anwendung und Verifikation</b>	<b>76</b>
7.1. Deutsch-Algorithmus . . . . .	76
7.2. Shor-Algorithmus . . . . .	79
7.3. Grover-Algorithmus . . . . .	82

7.4. Quantenteleportation . . . . .	85
<b>III. Schluss</b>	<b>89</b>
<b>8. Fazit</b>	<b>90</b>
8.1. Zusammenfassung . . . . .	90
8.2. Ausblick . . . . .	91
<b>IV. Anhang</b>	<b>92</b>
<b>Literaturverzeichnis</b>	<b>93</b>
<b>Abkürzungsverzeichnis</b>	<b>96</b>
<b>Symbolverzeichnis</b>	<b>97</b>

# Tabellenverzeichnis

5.1.	Basisoperatoren der FIEE . . . . .	54
5.2.	Basisoperatoren der CEE . . . . .	54
7.1.	Verifikation: Deutsch-Algorithmus, Schritt 3: Folgezustände . . . . .	77
7.2.	Verifikation: Deutsch-Algorithmus, Schritt 3: Folgezustände . . . . .	77
7.3.	Verifikation: Deutsch-Algorithmus, Schritt 4: Folgezustände . . . . .	78
7.4.	Verifikation: Deutsch-Algorithmus, Schritt 3: Messergebnis . . . . .	78
7.5.	Verifikation: Shor-Algorithmus, Schritt 1 - 4: Folgezustände . . . . .	80
7.6.	Verifikation: Shor-Algorithmus, Schritt 5: Vier mögliche Messergebnisse . . . . .	81
7.7.	Verifikation: Grover-Algorithmus, Schritt 1 - 4: Folgezustände . . . . .	83
7.8.	Verifikation: Grover-Algorithmus, Folgezustände nach den Iterationen . . . . .	84
7.9.	Verifikation: Quantenteleportation, Folgezustände nach Schritt 2 und 4 . . . . .	86
7.10.	Verifikation: Quantenteleportation, Schritt 7: Folgezustände . . . . .	87
7.11.	Verifikation: Quantenteleportation, Schritt 9: Endzustände . . . . .	88

# Abbildungsverzeichnis

2.1.	Schematische Darstellung von Quantenschaltkreisen . . . . .	11
2.2.	Hadamard-Gatter . . . . .	12
2.3.	Pauli-Gatter . . . . .	13
2.4.	Phasengatter . . . . .	13
2.5.	Swap-Gatter . . . . .	14
2.6.	Gesteuertes Gatter . . . . .	14
2.7.	CNOT-Gatter . . . . .	15
2.8.	Toffoli-Gatter . . . . .	16
2.9.	Fredkin-Gatter . . . . .	16
2.10.	Zusammengesetzte Operatoren . . . . .	17
2.11.	Quantenschaltkreis des Deutsch Algorithmus . . . . .	20
2.12.	Quantenschaltkreis der Quanten Fouriertransformation . . . . .	23
2.13.	Quantenschaltkreis zur Bestimmung der Ordnung . . . . .	24
2.14.	Quantenschaltkreis des Grover-Algorithmus . . . . .	25
2.15.	Quantenschaltkreis der Quantenteleportation . . . . .	26
4.1.	Transformation der unteren Bits . . . . .	32
4.2.	UML2 Klassendiagramm: Konzeptuelles Datenmodell . . . . .	38
4.3.	UML2 Anwendungsfalldiagramm . . . . .	42
4.4.	GUI-Mockup: Hauptfenster . . . . .	43
4.5.	Darstellung komplexer Zahlen in der GUI . . . . .	44
5.1.	UML2 Komponentendiagramm . . . . .	47
5.2.	UML2 Paketdiagramm: Überblick . . . . .	48
5.3.	UML2 Paketdiagramm: Core . . . . .	49
5.4.	UML2 Klassendiagramm: Matrix . . . . .	50
5.5.	UML2 Kommunikationsdiagramm: Simulator Protokoll . . . . .	55
5.6.	UML2 Klassendiagramm: QuantumOperator . . . . .	56
5.7.	UML2 Klassendiagramm: Architektur der GUI . . . . .	58
5.8.	UML2 Klassendiagramm: MVC und Command-Pattern . . . . .	59
5.9.	UML2 Klassendiagramm: Technisches Datenmodell . . . . .	61
5.10.	UML2 Klassendiagramm: Szenengraph . . . . .	62
6.1.	Performanceanalyse: Messreihe 1 . . . . .	73
6.2.	Performanceanalyse: Messreihe 2 . . . . .	74

6.3. Performanceanalyse: Messreihe 3 . . . . .	75
7.1. Verifikation: Deutsch-Algorithmus, Schaltkreis . . . . .	76
7.2. Verifikation: Shor-Algorithmus, Schaltkreis . . . . .	79
7.3. Verifikation: Grover-Algorithmus, Schaltkreis . . . . .	82
7.4. Verifikation: Quantenteleportation, Schaltkreis . . . . .	85

# Listings

5.1.	Beispiel: Parallel for in Scala . . . . .	46
5.2.	Beispiel: Überladen von Operatoren . . . . .	46
5.3.	Beispiel: @Properties . . . . .	63
5.4.	Pseudocode: Transformation von Teilregistern . . . . .	66
5.5.	Pseudocode: Gesteuerte Transformation von Teilregistern . . . . .	67
5.6.	Pseudocode: Auswertung von Oracle-Funktionen . . . . .	68
5.7.	Pseudocode: Auswahl eines zufälligen Basiszustandes . . . . .	69
5.8.	Pseudocode: Messung von Teilregistern . . . . .	70

**Teil I.**

**Einleitung**

# 1. Einführung

## 1.1. Motivation

Da bisher keine günstige, skalierbare physikalische Realisierung für Quantencomputer existiert, die Theorie aber ein sehr komplexes Maß angenommen hat, ist die Simulation von Quantencomputern mittels klassischer Computer das einzig zugängliche Mittel zur experimentellen Verifikation und Veranschaulichung der Funktionsweise von Quantencomputern.

Die Gesetze der Quantenmechanik sind für gewöhnlich kontraintuitiv, d.h. sie widersprechen der Art und Weise, wie wir gewohnt sind zu denken. Um ein Verständnis der Theorie zu erlangen, beispielsweise im Rahmen einer Lehrveranstaltung, kann es daher sehr hilfreich sein, sich die Formalismen mit einem Simulator anschaulich zu machen.

Neben Quantenprogrammiersprachen und Kommandozeilen-basierten Simulatoren, gibt es den Ansatz, die Simulation von Quantenschaltkreisen auf Basis einer graphischen Benutzerschnittstelle zu realisieren. Die Einbettung in eine GUI kann zum einen durch bessere Visualisierungs- und einfachere Bearbeitungsmöglichkeiten die Verständlichkeit und Präsentierbarkeit verbessern, als auch einen Kontrast zum sonst rein theoretischen Stoff darstellen. Script-basierte Simulatoren hingegen sind besser geeignet um komplexere Probleme durch generisch formale Notation zu beschreiben.

Von den bereits existierenden GUI-basierten Simulatoren konnte keiner den Autor völlig überzeugen, alle weisen bezüglich Funktionsumfang und/oder Bearbeitungsmöglichkeiten Schwächen auf.

## 1.2. Ziel der Arbeit

Im Rahmen dieser Bachelorarbeit soll ein GUI-basierter Simulator für Quantenschaltkreise à la "What you see is what you get" entwickelt werden, mit dem Fokus auf der Einsetzbarkeit

als Lernmittel auf einem Allzweck-Computer. Der Anwender soll sich möglichst wenig mit Eigenheiten der Software auseinandersetzen müssen, und Schaltkreise idealerweise live, z.B. im Kontext einer Präsentation, konstruieren und simulieren können.

Ein weiterer interessanter Aspekt ist die Effizienz des Simulators. Wie in 4.1 im Detail behandelt, ergibt sich für die Simulation von Quantenalgorithmen auf einem klassischen Rechner im allgemeinen eine super-polynomielle Laufzeit (siehe Abschnitt 4.1). Da sich, wenn man von einer Erweiterung des Mooreschen Gesetzes ausgeht, die Anzahl der Kerne pro Mikroprozessor zukünftig in regelmäßigen Zeitintervallen verdoppeln wird (HBK06), scheint es sinnvoll die Simulationsalgorithmen nach Möglichkeit parallelisierbar zu gestalten.

Die Anwendung soll ohne große Umstände einsetzbar sein, Plattformunabhängigkeit wird als selbstverständlich erachtet. Um den Entwicklungsaufwand bezüglich Portabilität und Parallelisierung möglichst gering zu halten, wird Scala als Entwicklungsplattform verwendet.

### 1.3. Abgrenzung

Einige Ansätze zur parallelisierten Simulation von Quantenschaltkreisen haben das Ziel, auf Cluster-Systemen bzw. Supercomputern besonders große Quantenregister simulierbar zu machen. In Abgrenzung dazu soll der hier entwickelte Simulator die parallele Architektur aktueller und kommender Allzweck-Computer ausnutzen. Dabei steht weiterhin nicht im Vordergrund besonders große Quantenregister zu simulieren, eine gewisse Problemgröße (ca. 20 Qubit) wird zunächst als ausreichend erachtet um ein großes Spektrum an Quantenalgorithmen exemplarisch simulierbar zu machen.

### 1.4. Aufbau der Arbeit

Der Aufbau dieser Arbeit gestaltet sich folgendermaßen. Zunächst werden in Kapitel 2 die relevanten Grundlagen der Quanteninformatik erläutert. Kapitel 3 gibt einen Überblick über bereits existierende, vergleichbare Simulatoren. In Kapitel 4, 5 und 6 wird der Entwicklungsprozess in den Phasen Analyse, Entwurf/Design und Realisierung/Test, sowie die Ergebnisse einer Performanceanalyse dargestellt. Kapitel 7 beinhaltet die Verifikation und Präsentation der Funktionalität des Simulators am Beispiel von vier bekannten Quantenalgorithmen, dem Algorithmus von Deutsch, dem Shor-Algorithmus, dem Grover-Algorithmus sowie der Quantenteleportation.

**Teil II.**

**Hauptteil**

Wer behauptet die  
Quantenmechanik sei  
einleuchtend, hat sie nicht  
wirklich verstanden.

---

(Niels Bohr)

## 2. Grundlagen der Quanteninformatik

Dieses Kapitel gibt einen Überblick über die nötigen Grundlagen der Quanteninformatik. Um die abstrakte Arbeitsweise von Quantencomputern zu verstehen, ist es nicht notwendig einen Abschluss in Physik zu haben, da die Theorie weitestgehend unabhängig von der physikalischen Realisierung betrachtet werden kann. Nichtsdestotrotz ist dies in seiner Gesamtheit gewiss kein einfaches Thema, welches im Rahmen dieser Arbeit nur angeschnitten werden kann. Die folgende theoretische Betrachtung richtet sich primär nach [NC00](#), [Hom08](#) und [Bru03](#)<sup>1</sup>.

### 2.1. Quantenzustände

Der Titel "Quantenmechanik" leitet sich aus der Beobachtung ab, dass einige physikalische Systeme quantisierbare Eigenschaften aufweisen. Bezüglich einer solchen Eigenschaft wird einem quantenmechanischen System eine finite Anzahl von *Basiszuständen* zugeordnet. Photonen besitzen beispielsweise die Eigenschaft der Polarisation mit nur zwei Basiszuständen: "horizontal polarisiert" und "vertikal polarisiert". In den folgenden Abschnitten werden die grundlegenden Eigenschaften von einfachen quantenmechanischen Systemen, sogenannten Quantenbits, oder auch Qubits, und deren Bedeutung für die Informationsverarbeitung erläutert.

---

<sup>1</sup>Referenzen auf bestimmte Werke, außerhalb des Kontextes von Zitierungen, werden im Folgenden ohne Klammern geschrieben.

### 2.1.1. Hinweise zur Notation

Quantenzustände werden üblicherweise in der *Bra-Ket-Notation*, auch *Dirac-Notation*<sup>2</sup> notiert. Diese verwendet spitze Klammern um Vektoren zu kennzeichnen, angelehnt an die allgemeine Schreibweise für das Skalarprodukt  $\langle \cdot, \cdot \rangle$ . Ein *Ket* stellt einen Spaltenvektor in einem komplexen Hilbertraum  $C^n$  dar:

$$|a\rangle = \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix}, \quad (2.1)$$

das zugehörige *Bra* ist definiert als das hermitesch Konjugierte des Kets:

$$\langle a| = (|a\rangle^*)^T = [a_1^* \quad \dots \quad a_n^*]. \quad (2.2)$$

Daher lässt sich das Skalarprodukt zweier Vektoren in dieser Form als Matrixmultiplikation eines Bras mit einem Ket ausdrücken:

$$\langle a|b\rangle = \langle a, b\rangle = a \cdot b = [a_1^* \quad \dots \quad a_n^*] \cdot \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = \sum_i a_i^* b_i. \quad (2.3)$$

### 2.1.2. Quantenbits

Wie ein klassisches Bit hat das Quantenbit zwei Basiszustände,  $|0\rangle$  und  $|1\rangle$ . Im Gegensatz zum klassischen Bit kann sich das Qubit zu bestimmten Anteilen gleichzeitig in beiden Basiszuständen befinden. Diese Überlagerung von Zuständen wird *Superposition* genannt. Physikalisch kann ein Qubit beispielsweise durch die Polarisation eines Photons, mit den Basiszuständen  $|\leftrightarrow\rangle = |0\rangle$  "horizontal polarisiert" und  $|\updownarrow\rangle = |1\rangle$  "vertikal polarisiert", repräsentiert werden (Bru03)<sup>3</sup>. Dies ist aber nur eine von vielen Möglichkeiten, jedes Zweizustandssystem eignet sich als Qubit (NC00).

Der Zustand eines Qubits ist praktisch nur über eine Messung zugänglich, diese bezeichnet den Übergang der Information in die klassische Welt, also die, in welcher wir leben und gewohnt sind zu denken. Im Fall des Photons kann eine Messung beispielsweise die Passage eines Polarisationsfilters mit anschließender Detektion, "ist das Photon durch den Filter gekommen?",

---

<sup>2</sup>Nach dem britischen Physiker Paul Adrien Maurice Dirac.

<sup>3</sup>Zitierungen mit lokalem Bezug werden im Folgenden in runden Klammern (...) gekennzeichnet.

bedeuten (vgl. Bru03). Dabei kann immer nur einer der beiden klassischen Basiszustände ermittelt werden und der exakte quantenmechanische Zustand des Systems wird irreversibel verändert. Das Ergebnis der Messung hängt von den Wahrscheinlichkeitsamplituden  $\alpha$  und  $\beta$  des Qubits ab; der vollständige quantenmechanische Zustand wird formal beschrieben als

$$\alpha|0\rangle + \beta|1\rangle \quad \alpha, \beta \in \mathbb{C}. \quad (2.4)$$

Die reelle Messwahrscheinlichkeit für  $|0\rangle$  beträgt  $|\alpha|^2$ , die für  $|1\rangle$  beträgt  $|\beta|^2$ . Die Gesamtwahrscheinlichkeit muss immer 1 bleiben, es gilt also

$$|\alpha|^2 + |\beta|^2 = 1. \quad (2.5)$$

Das Qubit lässt sich auch als zweidimensionaler Vektor der Länge 1 darstellen,

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \alpha \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \quad (2.6)$$

diese Form wird *Zustandsvektor* genannt. [vgl. Hom08]<sup>4</sup>

Systeme mit mehreren Qubits werden *Quantenregister* genannt. Allgemein ist der Zustand eines zusammengesetzten Systems definiert über das Tensorprodukt der Teilsysteme. Fasst man zwei Qubits zu einem System zusammen, ergibt sich der Gesamtzustand über das Tensorprodukt der Zustandsvektoren:

$$|xy\rangle = |x\rangle \otimes |y\rangle = \begin{bmatrix} \alpha_x \\ \beta_x \end{bmatrix} \otimes \begin{bmatrix} \alpha_y \\ \beta_y \end{bmatrix} = \begin{bmatrix} \alpha_x \alpha_y \\ \alpha_x \beta_y \\ \beta_x \alpha_y \\ \beta_x \beta_y \end{bmatrix}. \quad (2.7)$$

Dieser Ausdruck lässt sich wiederum als Linearkombination der Basiszustände beschreiben:

$$\alpha_x \alpha_y |00\rangle + \alpha_x \beta_y |01\rangle + \beta_x \alpha_y |10\rangle + \beta_x \beta_y |11\rangle. \quad (2.8)$$

---

<sup>4</sup>Zitierungen mit Bezug auf einen gesamten Absatz werden im Folgenden jeweils nach dem Punkt, in eckigen Klammern [...] gekennzeichnet.

Die Anzahl der Basiszustände  $b_n$  für ein  $n$ -Bit Quantenregister beträgt  $2^n$ . Da das Tensorprodukt für Vektoren dem Ausmultiplizieren der einzelnen Elemente entspricht, wird der  $\otimes$  Operator in zusammengesetzten Quantenzuständen häufig weggelassen:

$$|x\rangle \otimes |y\rangle = |x\rangle|y\rangle = |xy\rangle = |x, y\rangle. \quad (2.9)$$

Basiszustände können alternativ auch dezimal notiert werden, z.B.  $|1000\rangle = |8\rangle$ . [vgl. [Hom08](#)]

### 2.1.3. Superposition

Die Tatsache, dass sich Quantensysteme gleichzeitig in mehreren Zuständen, also in Superposition befinden können, eröffnet für die Informationsverarbeitung eine interessante Perspektive: Es ist möglich Funktionen auf einem Quantenregister quasi für alle möglichen Eingabewerte in einem Rechenschritt auszuwerten. Diese Eigenschaft wird *Quantenparallelismus* genannt und stellt den Grund für die potentiell höhere Rechenleistung von Quantencomputern dar. Ganz so einfach ist es aber leider doch nicht, denn die  $2^n$  Ergebnisse der Funktion sind nicht zugänglich. Über die Messung erhält man nur ein einziges, dabei entscheidet der Zufall welches. Um aus der Superposition einen echten Nutzen zu ziehen, bedarf es einer geschickten Transformation der Problemstellung. Der historisch erste bekannte Quantenalgorithmus, welcher die Superposition auszunutzen vermag, ist der *Algorithmus von Deutsch* (siehe [2.3.1](#)); mit diesem Algorithmus lässt sich eine globale Eigenschaft einer Funktion mit nur einer Auswertung (eben für alle Eingabewerte gleichzeitig) derselben feststellen. [vgl. [NC00](#)]

### 2.1.4. Verschränkung

Zusammengesetzte quantenmechanische Systeme haben eine weitere interessante Eigenschaft, sie können verschränkt sein: Für bestimmte überlagerte Zustände ergibt sich eine nicht lokale Abhängigkeit der beteiligten Elemente. Es kann also sein, dass nach der Messung eines Elektrons am Ort A etwas über ein anderes, vormalig mit ersterem verschränktes Elektron an einem zweiten Ort B (möglicherweise das andere Ende des Universums) bekannt wird. Die Information ist instantan gegeben, es handelt sich dabei nicht um einen Informationsaustausch sondern um die Eigenschaft eines zusammengesetzten Systems dessen Teile sich an zwei räumlich entfernten Orten befinden. Seit Einstein, Podolski und Rosen dieses Phänomen 1935 erstmals aufzeigten ([EPR35](#)), war es Gegenstand kontroverser Diskussionen, verständlicherweise, denn es widerspricht unserer gewohnten Wahrnehmung und Denkweise. Formal lässt es sich aber

einfach herleiten: Zwei Photonen befinden sich in der Superposition  $\alpha|\uparrow\rangle|\uparrow\rangle + \beta|\leftrightarrow\rangle|\leftrightarrow\rangle$ . Misst man eines der beiden Photonen so steht bereits fest, dass das andere den selben Zustand haben muss.

Allgemein wird ein Zustand als verschränkt bezeichnet, wenn er sich nicht in ein Tensorprodukt der einzelnen Bits faktorisieren lässt. Beispielsweise lässt sich der unverschränkte Zustand  $|\psi\rangle = \alpha|00\rangle + \beta|01\rangle$  folgendermaßen umformen:

$$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle = |0\rangle \otimes (\alpha|0\rangle + \beta|1\rangle). \quad (2.10)$$

Der Zustand

$$|\Omega\rangle = \alpha|00\rangle + \beta|11\rangle \quad (2.11)$$

hingegen ist verschränkt, da er sich nicht weiter faktorisieren lässt. Ein Zustand wird als maximal verschränkt bezeichnet, wenn die Messwahrscheinlichkeiten gleich verteilt sind:  $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$  ist maximal verschränkt, dagegen ist die Verschränkung von  $\frac{1}{2}|00\rangle + \frac{\sqrt{3}}{2}|11\rangle$  schwächer. [Hom08]

Inzwischen gilt die Verschränkung als experimentell bewiesen, es existieren sogar bereits kommerzielle Anwendungen, beispielsweise werden Geräte zur sicheren Kommunikation mittels *Quantenkryptographie* angeboten <sup>5</sup>, eine Technologie die auf Verschränkung basiert (siehe 2.3.4).

### 2.1.5. Dekohärenz

Die Gesetze der Quantenmechanik gelten zunächst für sehr kleine "Teilchen", Photonen, Elektronen, Atome sowie kleine Moleküle (Fullerene, siehe HHB<sup>+</sup>04), diese verhalten sich als isoliertes physikalisches System wellenartig. Dort wo diese "Wellen" in Interaktion mit ihrer Umwelt treten, wird aber, paradoxerweise, ein Teilchencharakter beobachtet. Die Quantentheorie kann dieses Verhalten nicht vollständig aufklären, bezieht es allerdings in das Gesamtkonzept mit ein: In dem Moment wo ein quantenmechanisches "Teilchen" mit der Außenwelt in Interaktion tritt, also ein Informationsaustausch stattfindet bzw. *auch nur die Möglichkeit dazu besteht*, kollabiert die Wellenfunktion in einen ihrer klassischen Basiszustände, welcher Ausgangspunkt einer erneuten kohärenten Entwicklung ist. Ein solcher Informationsaustausch kann beispielsweise schon durch die Wechselwirkung mit einem (geradezu allgegenwärtigen) Wärmephoton möglich sein (vgl. HHB<sup>+</sup>04). Somit wird deutlich, dass es

---

<sup>5</sup>ID Quantique: <http://www.idquantique.com/>

sich bei quantenmechanischen Zuständen um äußerst fragile Artefakte handelt; eine kohärente Entwicklung im Rahmen eines Quantencomputers ist bisher selbst bei extrem niedrigen Temperaturen und unter minutiöser Abschottung von der Umgebung nur von kurzer Dauer.

Dekohärenz ist ein, nach fast 100 Jahren Forschung auf dem Gebiet der Quantenmechanik immer noch nicht vollständig verstandenes Phänomen und stellt das größte Hindernis bei der praktischen Realisierung von Quantencomputern dar.

### 2.2. Berechnungen mit Quantenzuständen

Die Zeitentwicklung eines kohärenten Quantensystems lässt sich durch *unitäre* Transformationen beschreiben (NC00). Eine unitäre Transformation ist längen- und winkelerhaltend, sie beeinflusst somit die Gesamtwahrscheinlichkeit des Systems nicht, welche, wie in 2.1.2 beschrieben, immer 1 bleiben muss. Dies gilt allgemein für Drehungen oder Spiegelungen. Um nachzuweisen dass ein gegebener Operator unitär ist, genügt es zu zeigen dass für die zugehörige Matrix  $A$  folgende Gleichung erfüllt wird:

$$A^\dagger A = I. \quad (2.12)$$

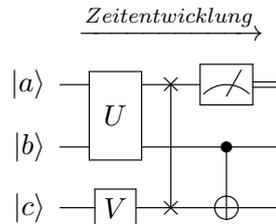
Die adjungierte  $A^\dagger = (A^T)^*$  einer unitären Matrix ist gleichzeitig deren inverse Matrix ([Hom08]). Somit sind alle unitären Transformationen auch *reversibel*:

$$\begin{aligned} A|\psi\rangle &= |\psi'\rangle \\ A^\dagger|\psi'\rangle &= |\psi\rangle. \end{aligned} \quad (2.13)$$

Aus der Reversibilität folgen einige charakteristische Eigenschaften von *Quantenschaltkreisen*. Jeder Quantenschaltkreis, sowie jeder darin enthaltene Operator besitzt ebenso viele Eingabe- wie Ausgabe-Qubits. Die in klassischen Schaltkreisen üblichen Operationen FANIN (Zusammenführung von zwei oder mehr Bits) und FANOUT (Kopieren eines Bits) sind nicht möglich, da diese nicht reversibel sind, einzig die Messung wird als nicht reversibler Operator akzeptiert (NC00, S. 23).

Die gebräuchliche schematische Darstellungsform für Quantenschaltkreise (siehe Abbildung 2.1) zeigt Leitungen für Quantenbits bzw.  $\sim$ Register als horizontale Linien, unterbrochen von vertikalen Blöcken, welche die Anwendung von Operatoren spezifizieren. Klassische Leitungen (z.B. nach einer Messung) werden durch doppelte Linien gekennzeichnet. Für gesteuerte *Gatter*

(siehe die folgenden Abschnitte) werden die Steuerbits jeweils mit einem Punkt markiert und durch eine vertikale Linie verbunden. CNOT und verwandte Gatter werden mit einem  $\oplus$ -Symbol auf dem Zielbit dargestellt, die Zielbits des SWAP-Gatters werden mit einem  $\times$ -Symbol markiert. [siehe Hom08, BBC<sup>+</sup>95, NC00]



**Abbildung 2.1.:** Schematische Darstellung von Quantenschaltkreisen

In den folgenden Abschnitten werden zunächst die grundlegenden Quantengatter, sowie *pseudo-klassische* Operatoren (Öme12a) betrachtet. Darauf folgt eine Erläuterung der Messung, und wie sich diese *nicht reversible* Operation in das Gesamtbild der Quantenschaltkreise und  $\sim$ -Algorithmen einfügt. Des Weiteren wird gezeigt, dass Quantenzustände allgemeinen nicht kopiert werden können, diese Tatsache ist Gegenstand des *No-Cloning-Theorems*.

### 2.2.1. Quantengatter

*Quantengatter* stellen, in Analogie zu den Gattern klassischer Schaltkreise, die Logikbausteine von *Quantenschaltkreisen* dar (vgl. Hom08). Ein  $n$ -Bit Quantengatter entspricht einer unitären Transformation und kann als komplexe Matrix  $U \in \mathbb{C}^{2^n \times 2^n}$  dargestellt werden.

Prinzipiell lässt sich jedes klassische Logikgatter auch mit Quantengattern realisieren. Reversible Logikgatter können direkt durch Permutationsmatrizen  $\{0, 1\}^{2^n \times 2^n}$  beschrieben werden, das dem NOT entsprechende Quantengatter hat beispielsweise folgende Matrixdarstellung:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad (2.14)$$

die Anwendung von  $X$  auf ein Qubit  $|\psi\rangle$  vertauscht die Amplituden für  $|0\rangle$  und  $|1\rangle$ :

$$X|\psi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}. \quad (2.15)$$

Da allerdings die meisten klassischen Gatter nicht reversibel sind, müssen gewisse Umwege gemacht werden. Eine Menge an Quantengattern wird als universell bezeichnet wenn auf deren Basis alle logischen Operationen realisierbar sind. Es ist beweisbar, dass das CNOT in Verbindung mit beliebigen ein-Bit Gattern eine solche Menge bildet, aber auch das *Toffoli*-Gatter, oder das *Fredkin*-Gatter an sich sind universell. [vgl. NC00, S. 18, 156, 188 ff]

Quantengatter haben nicht zwingend ein klassisches Pendant. Es existieren weitaus mehr Quantengatter als klassische Logikgatter, welche über reine zweiwertige Logikoperationen hinaus Superposition erzeugen und manipulieren, sowie die *Phase*<sup>6</sup> eines Quantenzustandes verändern können. [vgl. NC00]

### 2.2.1.1. Einfache Gatter

#### Hadamard-Gatter

Mittels des *Hadamard*-Gatters<sup>7</sup> lässt sich ein klassischer Zustand in eine gleichmäßige Superposition überführen:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Da  $H = H^\dagger$ , und somit  $H^2 = I$ , stellt eine zweite Anwendung von  $H$  auf selbige Superposition den ursprünglichen klassischen Zustand wieder her. Gleichmäßige Superpositionen sind ein essentieller Bestandteil vieler Quantenalgorithmien, deshalb ist  $H$ , sowie dessen Erweiterung für  $n$  Bits  $W_n = H^{\otimes n}$  (Walsh-Hadamard Transformation), eines der wichtigsten Quantengatter. Abbildung 2.2 zeigt die schematische Anwendung des Hadamard Gatters.

$$\alpha|0\rangle + \beta|1\rangle \longrightarrow \boxed{H} \longrightarrow \alpha \frac{|0\rangle+|1\rangle}{\sqrt{2}} + \beta \frac{|0\rangle-|1\rangle}{\sqrt{2}}$$

Abbildung 2.2.: Hadamard-Gatter

---

<sup>6</sup>Interessant sind *relative Phasenfaktoren*, d.h. Phasenunterschiede zwischen den Elementen des Zustandsvektors. Diese können, abhängig von der Basis (siehe Abschnitt 2.2.4), das Ergebnis einer Messung beeinflussen. Zustände die sich nur um einen *global Phasenfaktor*, welchen alle Elemente des Zustandsvektors gemeinsam haben, unterscheiden, können als physikalisch equivalent betrachtet, und durch Messung nicht unterschieden werden. [NC00]

<sup>7</sup>Nach französischen Mathematiker Jacques Hadamard.

### Pauli-Gatter

Abbildung 2.3 zeigt schematisch die Auswirkung der *Pauli-Gatter*<sup>8</sup> X, Y, Z, definiert durch die *Pauli Matrizen*  $\sigma_1, \sigma_2, \sigma_3$  (siehe NC00, S. 65), dabei entspricht das X-Gatter wie bereits erwähnt der Negation:

$$X = \sigma_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad Y = \sigma_2 = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad Z = \sigma_3 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

$$\alpha|0\rangle + \beta|1\rangle \longrightarrow \boxed{X} \longrightarrow \beta|0\rangle + \alpha|1\rangle$$

$$\alpha|0\rangle + \beta|1\rangle \longrightarrow \boxed{Y} \longrightarrow -i\beta|0\rangle + i\alpha|1\rangle$$

$$\alpha|0\rangle + \beta|1\rangle \longrightarrow \boxed{Z} \longrightarrow \alpha|0\rangle - \beta|1\rangle$$

**Abbildung 2.3.:** Pauli-Gatter

### Phasengatter

Das *Phasengatter*  $R_\theta$  versieht ein Qubit mit einem relativen Phasenfaktor  $e^{\theta i}, \theta \in \mathbb{R}$ . Oftmals wird auch  $R_n, n \in \mathbb{N}^+$  mit  $\theta = e^{\frac{2\pi i}{n}}$  verwendet.  $R_\theta$  und  $R_n$  entsprechen den folgenden Matrizen:

$$R_\theta = \begin{bmatrix} 1 & 0 \\ 0 & e^{\theta i} \end{bmatrix} \quad R_n = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{2\pi i}{n}} \end{bmatrix}.$$

Abbildung 2.4 zeigt die schematische Anwendung des Phasengatters auf ein Qubit.

$$\alpha|0\rangle + \beta|1\rangle \longrightarrow \boxed{R_\theta} \longrightarrow \alpha|0\rangle + e^{\theta i}\beta|1\rangle$$

$$\alpha|0\rangle + \beta|1\rangle \longrightarrow \boxed{R_n} \longrightarrow \alpha|0\rangle + e^{\frac{2\pi i}{n}}\beta|1\rangle$$

**Abbildung 2.4.:** Phasengatter

---

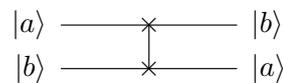
<sup>8</sup>Nach dem österreichischen Physiker Wolfgang Pauli.

### Swap-Gatter

Das *Swap*-Gatter führt zu einer Vertauschung zweier Qubits und ist definiert durch folgende Matrix:

$$\text{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Abbildung 2.5 zeigt die schematische Anwendung des SWAP-Gatters.



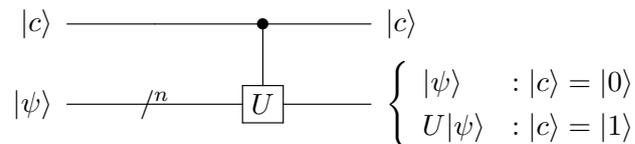
**Abbildung 2.5.:** Swap-Gatter

#### 2.2.1.2. Gesteuerte Gatter

Prinzipiell können alle Quantengatter in Abhängigkeit vom Zustand eines oder mehrerer Qubits angewendet werden. Die entstehenden *gesteuerten Gatter* haben allgemein die Form:

$$C(U) = \begin{bmatrix} I_{2^n} & 0 \\ 0 & U \end{bmatrix}, U \in \mathbb{C}^{2^n \times 2^n}.$$

Dabei kann angenommen werden das  $U$  bereits selbst gesteuert ist, auf diese Weise lassen sich mehrfach gesteuerte Gatter bilden. Abbildung 2.6 zeigt die schematische Anwendung eines unbestimmten gesteuerten  $n$ -Bit Gatters  $U$ .



**Abbildung 2.6.:** Gesteuertes Gatter

### CNOT-Gatter

Das CNOT-Gatter bewirkt eine gesteuerte Negation, d.h. angewandt auf ein 2-Qubit Register  $|ab\rangle$  wird  $|b\rangle$  überall dort negiert, wo  $|a\rangle = 1$ , es werden also die Amplituden für  $|10\rangle$  und  $|11\rangle$  vertauscht. Als Matrix lässt sich das CNOT folgendermaßen darstellen:

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Abbildung 2.7 zeigt die schematische Anwendung des CNOT-Gatters.

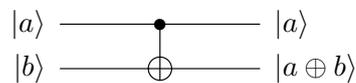


Abbildung 2.7.: CNOT-Gatter

### Toffoli-Gatter

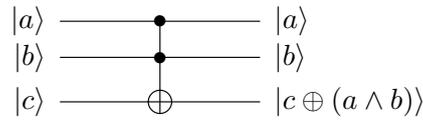
Das Toffoli-Gatter<sup>9</sup>, oder auch CCNOT, ist ein universelles drei-Bit Quantengatter, und wird definiert durch folgende Matrix:

$$\text{TOFFOLI} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Durch die Anwendung des Toffoli-Gatters auf ein Quantenregister  $|abc\rangle$  wird das Zielbit  $|c\rangle$  in Abhängigkeit von zwei Kontrollbits  $|a\rangle$  und  $|b\rangle$  negiert. Abbildung 2.8 zeigt die schematische Anwendung des Toffoli-Gatters.

---

<sup>9</sup>Nach Tommaso Toffoli.



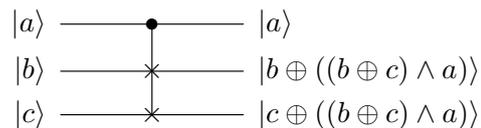
**Abbildung 2.8.:** Toffoli-Gatter

### Fredkin-Gatter

Das *Fredkin*-Gatter <sup>10</sup>, im Prinzip ein gesteuertes SWAP-Gatter, ist ein universelles drei-Bit Quantengatter, und ist definiert durch folgende Matrix:

$$\text{FREDKIN} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Durch Anwendung des Fredkin-Gatter auf ein drei-Bit Quantenregister  $|abc\rangle$  werden die beiden Zielbits  $|bc\rangle$  abhängig von dem Steuerbit  $|a\rangle$  vertauscht. Abbildung 2.9 zeigt die schematische Anwendung des Fredkin-Gatters.



**Abbildung 2.9.:** Fredkin-Gatter

---

<sup>10</sup>Nach Edward Fredkin.

### 2.2.2. Zusammengesetzte Operatoren

Es ist möglich einzelne Operatoren zu einem größeren Operator zusammenzufassen. Dies geschieht, ebenso wie bei Quantenzuständen, über das Tensorprodukt, siehe Abbildung 2.10.

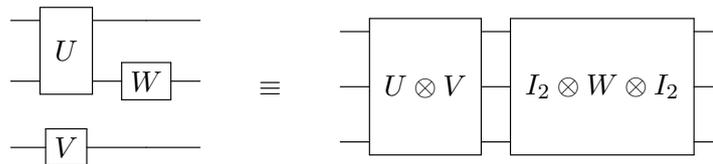


Abbildung 2.10.: Zusammengesetzte Operatoren (vgl. Hom08)

### 2.2.3. Pseudo-klassische Operatoren

Oftmals ist es hilfreich Teile eines Quantenalgorithmus mittels klassischer digitaler Funktionen  $f : \{0, 1\}^n \rightarrow \{0, 1\}^n$  zu substituieren, wenn die tatsächliche Realisierung als Quantenschaltkreis nicht im Mittelpunkt steht. Da der Inhalt verborgen ist, werden diese Operatoren als *Oracle* oder *Black-Box* bezeichnet, Ömer spricht auch von *pseudo-klassischen Operatoren* (Öme12a). Unter der Prämisse, dass  $f$  bijektiv, somit reversibel ist, kann auf Basis einer universellen Menge an Quantengattern immer ein Quantenschaltkreis gefunden werden, welcher  $f$  implementiert. Nicht bijektive Funktionen werden häufig durch eine zusätzliche  $\oplus$ -Verknüpfung des Funktionswertes mit dem  $y$ -Register reversibel gemacht:

$$|x\rangle|y\rangle \xrightarrow{U_f} |x\rangle|f(x) \oplus y\rangle. \quad (2.16)$$

### 2.2.4. Messung

Wie bereits erwähnt muss ein Quantensystem gemessen werden um etwas über dessen Zustand in Erfahrung zu bringen. Dieser Informationsaustausch führt unweigerlich zur Dekohärenz, und somit zur irreversiblen Zerstörung des Quantenzustandes. Die Messung ist ein statistischer, somit streng genommen *nicht deterministischer*, im Allgemeinen nicht reversibler Prozess. Ein *statistischer Determinismus* ergibt sich allerdings aus der Interpretation der Wahrscheinlichkeitsamplituden. Somit lässt sich das Ergebnis einer einzelnen Messung zwar nicht voraussagen, allerdings könnte man <sup>11</sup> durch wiederholte Messung an  $n$  äquivalenten Zuständen, Aussagen

---

<sup>11</sup>Vorausgesetzt es stehen  $n$  gleichwertige Quantenzustände zur Verfügung, was praktisch sehr schwierig, näherungsweise machbar, bis unmöglich ist (siehe 2.2.5).

bezüglich der Wahrscheinlichkeitsamplituden näherungsweise deterministisch verifizieren. [Can11]

Neben der Messung eines gesamten Quantenregisters sind auch Teilmessungen, sowie Messungen in beliebigen *Basen* möglich. Eine weit verbreitete Form der Messung im Kontext von Quantencomputern ist die *Projektionsmessung* (NC00, S. 87): Die Wahrscheinlichkeitsamplituden  $p_i$  werden bezüglich einer *orthonormalen* Basis  $B = \{b_1, \dots, b_n\}$  aus dem Zustand  $|\psi\rangle$  heraus projiziert:

$$p_i = \langle b_i | \psi \rangle. \quad (2.17)$$

$B$  ist orthonormal wenn  $|b_i| = 1$  und  $\langle b_i | b_j \rangle = 0$  für  $i \neq j$ , also alle Basisvektoren die Länge 1 aufweisen und untereinander orthogonal sind, und kann durch eine Rotation der Standardbasis erhalten werden. Eine Teilmessung entspricht essentiell der Projektion in einen  $k$ -dimensionalen Unterraum von  $B$ :  $E = e_1, \dots, e_k$ , und führt zu einer Reduktion der Superposition, jedoch nicht unbedingt zum vollständigen Kollaps, nur das Teilsystem von Interesse wird in einen seiner Basiszustände überführt. Durch die Messung wird der Index  $i$  eines Basiszustandes bezüglich der Messbasis ermittelt, der Folgezustand des Quantensystems ergibt sich als

$$|\psi'\rangle = \sum_{i=1}^k \frac{p_i |e_i\rangle}{\sqrt{|p_i|^2}}, \quad p_i = \langle e_i | \psi \rangle. \quad (2.18)$$

Beispielsweise führt die Messung des ersten Bits  $|x\rangle$  eines Zwei-Bit Quantenregisters  $|xy\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$  in der Standardbasis zu zwei möglichen Resultaten: Mit der Wahrscheinlichkeit  $|\alpha_{00}|^2 + |\alpha_{01}|^2$  befindet sich  $|xy\rangle$  nach der Messung im Zustand

$$\frac{\alpha_{00}|00\rangle + \alpha_{01}|01\rangle}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}}, \quad (2.19)$$

mit der Wahrscheinlichkeit  $|\alpha_{10}|^2 + |\alpha_{11}|^2$  im Zustand

$$\frac{\alpha_{10}|10\rangle + \alpha_{11}|11\rangle}{\sqrt{|\alpha_{10}|^2 + |\alpha_{11}|^2}}. \quad (2.20)$$

In diesem Beispiel wurde der Verständlichkeit halber die Standardbasis  $\{|0\rangle, |1\rangle\}$  verwendet, die Projektionen entsprechen somit den Amplituden des Zustandsvektors. Alternativ könnte der Zustand bezüglich  $\{|+\rangle, |-\rangle\}$  mit  $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$  und  $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$ , der sogenannten *Hadamard Basis*, aber auch in jeder anderen Orthonormalbasis betrachtet werden. Nielsen und Chuang weisen darauf hin, dass Messungen in beliebigen Basen zwar machbar, aber nicht unbedingt leicht zu realisieren sind. [vgl. Hom08, NC00]

### 2.2.5. Das No-Cloning Theorem

Ist es möglich einen Quantenzustand in seiner Ganzheit zu kopieren? Ausgangspunkt der Überlegung sei ein zwei Bit Quantenregister  $|\psi\rangle \otimes |s\rangle$ . Ziel ist es, den Zustand von  $|\psi\rangle$  auf das Zielbit  $|s\rangle$  zu übertragen:

$$|\psi\rangle \otimes |s\rangle \rightarrow |\psi\rangle \otimes |\psi\rangle. \quad (2.21)$$

Das *No-Cloning Theorem* besagt, dass keine unitäre Transformation existiert welche dies bewerkstelligen könnte, ausgenommen es handelt sich bei  $|\psi\rangle$  um einen orthogonalen, also um einen klassischen Zustand (vgl. NC00, S. 532). Um dieses Problem besser zu verstehen eignet sich ein Experiment mit dem CNOT Gatter:  $|x\rangle|y\rangle \rightarrow |x\rangle|y \oplus x\rangle$ . Es scheint so, als würde dieses Gatter den gewünschten Effekt erzielen:

$$\alpha|00\rangle + \beta|10\rangle \xrightarrow{\text{CNOT}} \alpha|00\rangle + \beta|11\rangle. \quad (2.22)$$

Allerdings zeigt sich, dass dies nur für klassische Zustände funktioniert, denn

$$\alpha|00\rangle + \beta|11\rangle = \alpha^2|00\rangle + \alpha\beta|01\rangle + \alpha\beta|10\rangle + \beta^2|11\rangle \quad (2.23)$$

lässt sich nur für  $\alpha\beta = 0$  lösen. Im Allgemeinen führt die Anwendung des CNOT zu einem verschränkten Zustand. [vgl. Hom08, S. 82]

## 2.3. Quantenalgorithmen

Nachdem der 1992 von David Deutsch und Richard Jozsa vorgeschlagene *Deutsch-Jozsa-Algorithmus* (2.3.1) die prinzipielle Nutzbarkeit des Quantenparallelismus herausgestellt hat, wurden in den darauffolgenden Jahren die beiden bekanntesten Quantenalgorithmen entwickelt, der Faktorisierungsalgorithmus von Shor (2.3.2) und der Suchalgorithmus von Grover (2.3.3). Insbesondere der Shor-Algorithmus erregte großes Aufsehen, da er große Zahlen exponentiell schneller faktorisieren kann als jeder bekannte klassische Algorithmus, und stellt somit eine potentielle Gefahr für das RSA-Kryptosystem dar.

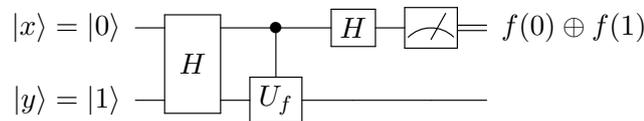
Als verteilter Algorithmus ist des Weiteren die Quantenteleportation (2.3.4) von besonderem Interesse. Auf dem selben Prinzip beruht die bereits kommerzialisierte Quantenkryptographie, eine Technologie zur sicheren Verteilung von kryptographischen Schlüsseln über möglicherweise unsichere Kommunikationskanäle.

### 2.3.1. Deutsch-Algorithmus

Der Algorithmus von Deutsch, nach dem Physiker David Deutsch, löst mit Hilfe des Superpositionsprinzips von Quantenzuständen das so genannte *Problem von Deutsch*. Die Problemstellung wird häufig anhand eines klassischen Szenarios erklärt: Eine *faire* Münze zeigt auf der einen Seite Kopf, auf der anderen Zahl. Um zu prüfen ob dies wirklich der Fall ist, und es sich nicht um eine *unfaire* Münze, deren Seiten beide Kopf bzw. Zahl zeigen, handelt, müssen nach den Regeln der klassischen Physik/Logik beide Seiten betrachtet werden. – Mit Hilfe der Quantenmechanik lässt sich die Beschaffenheit der "Münze" mit nur einer Betrachtung bestimmen. Um das Problem formal zu beschreiben, wird nun die Münze mit einer zweiwertigen Funktion  $f : \{0, 1\} \rightarrow \{0, 1\}$  substituiert. Es existieren genau vier solche Funktionen:

$$\begin{array}{l}
 a) \quad f(x) = 0 \quad : \quad 0 \rightarrow 0, \quad 1 \rightarrow 0 \\
 b) \quad f(x) = 1 \quad : \quad 0 \rightarrow 1, \quad 1 \rightarrow 1 \\
 c) \quad f(x) = x \quad : \quad 0 \rightarrow 0, \quad 1 \rightarrow 1 \\
 d) \quad f(x) = x \oplus 1 \quad : \quad 0 \rightarrow 1, \quad 1 \rightarrow 0
 \end{array}
 \left. \begin{array}{l} \\ \\ \\ \end{array} \right\} \begin{array}{l} \textit{konstant} : f(0) = f(1) \\ \\ \\ \textit{balanciert} : f(0) \neq f(1) \end{array} \quad (2.24)$$

Da  $f$  nicht in allen Fällen reversibel ist, nämlich nicht im konstanten Fall, wird das *Oracle*  $f$  in der Form  $U_f : |x\rangle|y\rangle \rightarrow |x\rangle|y \oplus f(x)\rangle$  angewendet. Dies garantiert Reversibilität, und somit die Realisierbarkeit mittels unitärer Transformationen (siehe 2.2.3). [vgl. Hom08]



**Abbildung 2.11.:** Quantenschaltkreis des Deutsch Algorithmus

Der Algorithmus, Abbildung 2.11 zeigt den entsprechenden Quantenschaltkreis, verwendet zwei Qubits  $|x\rangle|y\rangle$ , welche zunächst mit dem Startzustand

$$|\psi_0\rangle = |0\rangle|1\rangle \quad (2.25)$$

präpariert werden:  $|x\rangle|y\rangle \leftarrow |0\rangle|1\rangle$ . Als nächstes werden beide Qubits durch Hadamard-Transformation in Superposition versetzt:  $|x\rangle|y\rangle \leftarrow H|x\rangle H|y\rangle$ , dadurch geht das System in den Zustand  $\psi_1$  über:

$$|\psi_1\rangle = \left[ \frac{|0\rangle + |1\rangle}{\sqrt{2}} \right] \left[ \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right] \quad (2.26)$$

Nun folgt die Anwendung des Oracles:  $|x\rangle|y\rangle \leftarrow |x\rangle|y \oplus f(x)\rangle$ , der Folgezustand des Systems ist

$$|\psi_2\rangle = \begin{cases} \pm \left[ \frac{|0\rangle+|1\rangle}{\sqrt{2}} \right] \left[ \frac{|0\rangle-|1\rangle}{\sqrt{2}} \right] & , \text{ falls } f \text{ konstant} \\ \pm \left[ \frac{|0\rangle-|1\rangle}{\sqrt{2}} \right] \left[ \frac{|0\rangle-|1\rangle}{\sqrt{2}} \right] & , \text{ falls } f \text{ balanciert.} \end{cases} \quad (2.27)$$

Durch Umformung kann gezeigt werden, dass sich die Funktionswerte von  $f$  in den Amplituden des Zustandes widerspiegeln (Hom08):

$$\begin{aligned} |\psi_2\rangle &= \frac{1}{2} (|0\rangle|0 \oplus f(0)\rangle - |0\rangle|1 \oplus f(0)\rangle + |1\rangle|0 \oplus f(1)\rangle - |1\rangle|1 \oplus f(1)\rangle) \\ &= \frac{1}{2} (|0\rangle (|f(0)\rangle - |1 \oplus f(0)\rangle) + |1\rangle (|f(1)\rangle - |1 \oplus f(1)\rangle)) \\ &= \frac{1}{2} \left( (-1)^{f(0)}|0\rangle (|0\rangle - |1\rangle) + (-1)^{f(1)}|1\rangle (|0\rangle - |1\rangle) \right) \\ &= \frac{1}{2} \left( (-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle \right) (|0\rangle - |1\rangle). \end{aligned} \quad (2.28)$$

Eine weitere Anwendung der Hadamard-Transformation auf  $|x\rangle$ :  $|x\rangle|y\rangle \leftarrow H(|x\rangle)|y\rangle$ , bringt das System in den Zustand

$$|\psi_3\rangle = \begin{cases} \pm|0\rangle \left[ \frac{|0\rangle-|1\rangle}{\sqrt{2}} \right] & , \text{ falls } f \text{ konstant} \\ \pm|1\rangle \left[ \frac{|0\rangle-|1\rangle}{\sqrt{2}} \right] & , \text{ falls } f \text{ balanciert,} \end{cases} \quad (2.29)$$

woraufhin  $|x\rangle$  gemessen wird. Ist das Ergebnis  $|0\rangle$ , so ist  $f$  konstant, ist es  $|1\rangle$  so ist  $f$  balanciert. [NC00]

### 2.3.2. Shor-Algorithmus

Der Algorithmus von Shor stellt ein Faktorisierungsverfahren dar, und ermöglicht es, echte Teiler einer zusammengesetzten Zahl effizient, d.h. in polynomialer Zeit zu berechnen. Aktuell ist kein klassisches Verfahren bekannt, welches dieses Problem ähnlich effizient lösen könnte. Die Hypothese, dass ein solches Verfahren nicht existiert, stellt die Grundlage populärer asymmetrischer Kryptosysteme dar. Beispielsweise basiert die Sicherheit des RSA-Kryptosystems darauf, dass ein Rückschluss auf den *geheimen* Schlüssel unter Kenntnis des *öffentlichen* Schlüssels das Auffinden der Primfaktoren zweier großer Zahlen erfordert. Auf einem Quantencomputer mit ausreichend Qubits würde dieses Problem durch den Shor-Algorithmus in den lösbaren Bereich gerückt werden. [vgl. Hom08]

### Faktorisierungsproblem

Das Faktorisierungsproblem lässt sich auf das Bestimmen der *Periode*, auch *Ordnung* einer Funktion zurückführen ([Wik12c](#)). Angenommen es soll ein echter Teiler der natürlichen Zahl  $N$  gefunden werden, welche sich aus dem Produkt zweier Primzahlen  $n_1$  und  $n_2$  ergibt. Um  $N$  zu faktorisieren, wählt man eine zu  $N$  teilerfremde Zahl

$$1 < a < N, \quad ggT(a, N) = 1, \quad (2.30)$$

und bestimmt die Periode  $r$  der Funktion

$$a^x \bmod N, \quad a^{x+r} \bmod N = a^x \bmod N. \quad (2.31)$$

Vorausgesetzt  $r$  ist gerade und  $a^{r/2} \not\equiv -1 \pmod N$ , kann nun mit Hilfe des *Euklidischen Algorithmus* möglicherweise ein nicht trivialer Teiler von  $N$  bestimmt werden, die zwei Kandidaten sind  $ggT(a^{r/2} - 1, N)$  und  $ggT(a^{r/2} + 1, N)$  <sup>12</sup>. Wird  $a$  zufällig gewählt, und entspricht  $N$  nicht der Potenz einer Primzahl  $N \neq p^k$ , so führt diese Methode im ersten Durchlauf mit einer Wahrscheinlichkeit  $> \frac{1}{2}$ , nach  $k$  Wiederholungen mit einer Wahrscheinlichkeit  $> (\frac{1}{2})^k$  (vgl. [Hom08](#)) zu einem brauchbaren Ergebnis. [vgl. [Öme12c](#)]

### Algorithmus

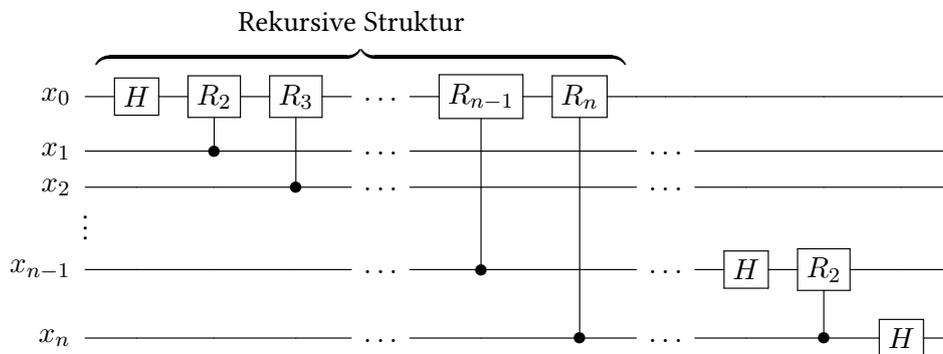
1. Wenn  $N$  gerade, dann ist das **Ergebnis** 2.
2. Wähle eine Zufallszahl  $a \in [2, 3, \dots, N - 1]$ .
3. Wenn  $ggT(a, N) > 1$ , dann ist das **Ergebnis**  $a$ .
4. Finde die Periode  $r$  der Funktion  $a^x \bmod N$ .
5. Wenn  $r$  ungerade oder  $a^{r/2} \equiv -1 \pmod N$ , **Kein Ergebnis**.
6. Berechne  $k_1 = ggT(a^{r/2} - 1, N)$ . Wenn  $1 < k_1 < N$ , dann ist  $k_1$  ein **Ergebnis**.  
Berechne  $k_2 = ggT(a^{r/2} + 1, N)$ . Wenn  $1 < k_2 < N$ , dann ist  $k_2$  ein **Ergebnis**.
7. Ansonsten **Kein Ergebnis**.

---

<sup>12</sup>Warum das so ist, soll hier nicht weiter beleuchtet werden, stattdessen wird auf die Literatur verwiesen, z.B. [[Öme12c](#)] [[NC00](#), S. 226 ff, 625 ff] [[Hom08](#), S. 202].

### Quanten Fouriertransformation

Kernproblem des Algorithmus ist das auffinden der Periode  $r$ , welches mit Hilfe der *Fouriertransformation* gelöst werden kann. Dies ist der Punkt an dem der Quantencomputer den entscheidenden Vorteil hat: Die Fouriertransformation kann auf einem Quantencomputer exponentiell effizienter berechnet werden als auf einem klassischen Rechner, die *QFT* (Quanten Fouriertransformation) benötigt zur Transformation von  $n$  Qubits nur  $O(n^2)$  Rechenschritte, während die klassische *FFT* (Fast Fourier Transformation) im Vergleich dazu  $O(n2^n)$  Rechenschritte benötigen würde. Abbildung 2.12 zeigt den Quantenalgorithmus der QFT. [vgl. NC00]



**Abbildung 2.12.:** Quantenschaltkreis der Quanten Fouriertransformation. Nicht gezeigt werden Swap-Gatter zur Bitumkehr (wie bei einigen FFT-Algorithmen) am Ende des Schaltkreises. [vgl. NC00]

### Bestimmung der Ordnung

Um die Ordnung  $r < N$  der Funktion  $a^x \bmod N$  zu bestimmen, werden zunächst zwei Quantenregister der Länge  $t$  und  $L$  initialisiert:

$$|x\rangle|y\rangle \leftarrow |0\rangle|0\rangle. \tag{2.32}$$

$L$  muss ausreichend groß sein um  $N$  binär zu kodieren:  $L \equiv \lceil \log(N) \rceil$ . Bei der Wahl von  $t$  wird in NC00 ein Richtwert von  $t = O(L + \lceil \log(1 - \epsilon) \rceil)$  angegeben <sup>13</sup> um eine ausreichende

<sup>13</sup>Im experimentellen Versuch ist die Periode zumeist schon bekannt, weshalb es möglich ist, wesentlich kleinere Werte für  $t$  zu verwenden.

Genauigkeit (relativ zu  $\epsilon$ ) zu erreichen (siehe [NC00](#), S. 236, 227). Das erste Register  $|x\rangle$  wird mittels der Hadamard-Transformation in eine gleichmäßige Superposition gebracht

$$|x\rangle|y\rangle \leftarrow H(|x\rangle)|y\rangle, \quad (2.33)$$

woraufhin  $|y\rangle$  durch Anwendung des *Oracles*

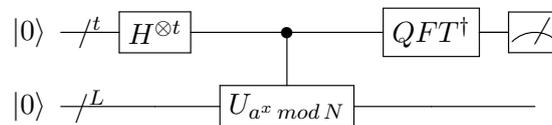
$$|x\rangle|y\rangle \leftarrow |x\rangle|y \oplus (a^x \bmod N)\rangle \Leftrightarrow |x\rangle|a^x \bmod N\rangle \quad (2.34)$$

in eine Superposition über alle  $a^x \bmod N$  versetzt wird. Um die Periode für eine Messung zugänglich zu machen wird die QFT auf  $|x\rangle$  angewendet

$$|x\rangle|y\rangle \leftarrow QFT(|x\rangle)|y\rangle. \quad (2.35)$$

Zum Schluss wird  $|x\rangle$  gemessen. Das Ergebnis der Messung lässt sich beschreiben als  $y \approx \lambda \frac{2^t}{r}$  mit  $\lambda \in \mathbb{N}^0$ . Es existiert ein effizienter klassischer Algorithmus um daraus die Periode  $r$  zu gewinnen (Kettenbruchzerlegung, siehe [NC00](#), S. 230, 635). [[Öme12c](#), vgl.]

Abbildung 2.13 zeigt den eben beschriebenen Algorithmus als Quantenschaltkreis.



**Abbildung 2.13.:** Quantenschaltkreis zur Bestimmung der Ordnung. [vgl. [NC00](#)]

### 2.3.3. Grover-Algorithmus

Lov Grover entdeckte 1996, dass Quantencomputer effizienter in unstrukturierten Daten suchen können als klassische Computer. Die Problemstellung kann man sich folgendermaßen vorstellen: Finde im Telefonbuch die Adresse einer Person, von der ausschließlich die Telefonnummer bekannt ist ([Gro96](#)). Die Einträge in Telefonbüchern sind bekanntlicher Weise nach Namen alphabetisch sortiert, so dass nur eine Möglichkeit bleibt: Alle Einträge von vorne bis Hinten (die Reihenfolge spielt eigentlich keine Rolle) durchgehen bis die Telefonnummer gefunden ist. Ebenso würde ein klassischer Computer verfahren. Die Komplexität dieses Algorithmus ist  $O(N)$ , mit  $N$  der Anzahl zu durchsuchender Datensätze. Grovers Quan-

tenalgorithmus findet das gesuchte Element mit hoher Wahrscheinlichkeit in  $O(\sqrt{N})$  Zeit, ist somit quadratisch schneller als der klassische Algorithmus. *Mit hoher Wahrscheinlichkeit*, denn es handelt sich um einen probabilistischen Algorithmus: Ausgehend von einer gleichmäßigen Superposition wird die Amplitude des gesuchten Elements  $k \in \mathbb{N}$ , sprich des  $k$ ten Basiszustandes successiv erhöht. Ein einzelner Schritt, die Grover-Iteration, sieht folgendermaßen aus:

1. negiere die Amplitude von  $|k\rangle$
2. verstärke die Amplitude von  $|k\rangle$  durch Spiegelung der Amplituden am Mittelwert

Diese Prozedur wird so lange wiederholt bis die Messwahrscheinlichkeit für  $k$  ein Optimum erreicht, dann wird gemessen. Die Anzahl der benötigten Iterationen  $m$  für eine maximale Messwahrscheinlichkeit von  $p_k > \frac{N-1}{N}$  beträgt  $m \approx \frac{\pi\sqrt{N}}{4}$  (Öme12b). Setzt man die Grover-Iteration darüber hinaus weiter fort, so würde die Amplitude für  $|k\rangle$  wieder abgeschwächt. Abbildung 2.14 zeigt den Grover-Algorithmus in schematischer Darstellung.

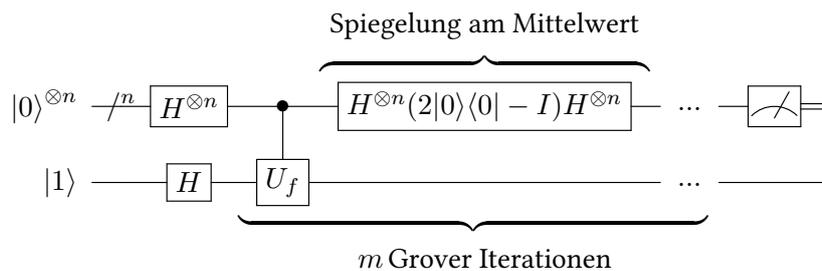


Abbildung 2.14.: Quantenschaltkreis des Grover-Algorithmus [Wik12a]

### 2.3.4. Quantenteleportation

Auf dem Phänomen der Verschränkung beruhend, ermöglicht die Quantenteleportation den Transport eines Quantenzustandes über beliebige Distanzen: An beiden Endpunkten, *Alice* und *Bob* genannt, steht je die Hälfte eines EPR-Paars  $|ab\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$  zur Verfügung, Alice besitzt  $|a\rangle$ , Bob  $|b\rangle$ . Die Messung des Zustandes auf einer Seite korreliert zu 100% mit der Messung auf der anderen Seite. Ziel der Prozedur ist es, einen unbekanntem Zustand  $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$  von Alices Seite zu Bob zu *teleportieren*. Dazu verschränkt Alice  $|a\rangle$  mit dem Qubit  $|x\rangle = |\psi\rangle$  und nimmt eine Messung in der Hadamard-Basis an beiden Qubits vor;

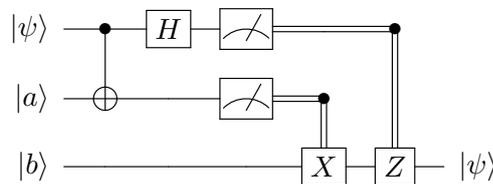
dies hat direkte (nicht lokale) Auswirkungen auf  $|b\rangle$ , da  $|a\rangle|b\rangle$  in einen der folgenden vier *Bell-Zustände*<sup>14</sup> übergeht:

$$\begin{aligned}
 \Omega^+ &= \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \\
 \Omega^- &= \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle) \\
 \Psi^+ &= \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) \\
 \Psi^- &= \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)
 \end{aligned}
 \tag{2.36}$$

Ohne weitere Informationen kann Bob damit nichts anfangen, daher teilt Alice Bob auf klassischem Weg das Ergebnis ihrer Messung an  $|x\rangle|a\rangle$  mit, woraufhin Bob  $|b\rangle$  in die richtige Basis drehen kann:

$$|\psi\rangle = \begin{cases} |b\rangle & , \text{ falls } x = 0 \wedge a = 0 \\ X(|b\rangle) & , \text{ falls } x = 0 \wedge a = 1 \\ Z(|b\rangle) & , \text{ falls } x = 1 \wedge a = 0 \\ Z(X(|b\rangle)) & , \text{ falls } x = 1 \wedge a = 1 \end{cases}
 \tag{2.37}$$

Nach der entsprechenden Transformation ist Bob im Besitz des Quantenzustandes  $|\psi\rangle$ , auf Alice Seite jedoch wurde  $|x\rangle$  durch die Messung irreversibel verändert. Durch die Quantenteleportation wird somit keine Kopie der zu übertragenden Information angefertigt, dies ist nach dem *No-Cloning-Theorem* unmöglich, siehe Abschnitt 2.2.5. Weiterhin ist es mit der Quantenteleportation nicht möglich, Nachrichten mit Überlichtgeschwindigkeit zu übertragen, da ein tatsächlicher Informationsaustausch ist nur dann möglich ist, wenn zusätzliche Informationen über einen klassischen Kanal übertragen werden. Abbildung 2.15 zeigt den Algorithmus der Quantenteleportation in schematischer Darstellung. [vgl. Hom08]



**Abbildung 2.15.:** Quantenschaltkreis der Quantenteleportation. Es wird angenommen dass  $|a\rangle$  und  $|b\rangle$  bereits verschränkt sind.

---

<sup>14</sup>Benannt nach dem Physiker John Bell

## 3. Ähnliche Arbeiten

Es existieren bereits diverse Software-Simulatoren für Quantencomputer. Diese unterscheiden sich stark in Zielsetzung, technischer Umsetzung, sowie Komplexität/Flexibilität. Im Kontext dieser Arbeit sind insbesondere GUI-basierte Simulatoren von Interesse, welche den Fokus auf die Simulation von (beliebigen) Quantenschaltkreisen legen. In diesem Kapitel werden drei mit der hier angestrebten Lösung vergleichbare Simulatoren vorgestellt. Es werden insbesondere zwei Kriterien untersucht, zum einen die verfügbaren Operatoren, wovon sich die allgemeine Flexibilität bei der Konstruktion von Schaltkreisen ableiten lässt, zum anderen werden die Features der Benutzeroberfläche charakterisiert.

### 3.1. jQuantum

Eine portable GUI Anwendung auf Java/Swing Basis, entwickelt von Andreas de Vries (FH Südwestfalen University of Applied Sciences) und 2004 veröffentlicht <sup>1</sup>. *jQuantum* ermöglicht die Konstruktion und schrittweise Simulation von beliebigen Quantenschaltkreisen. Der Autor von *jQuantum* gibt an, dass bis zu 15 Qubits simuliert werden können, in der Praxis zeigt sich allerdings, dass mit aktueller Hardware auch größere Register in angemessener Zeit simulierbar sind. Eine Eigenheit von *jQuantum* ist die Aufteilung des Quantenzustandes in zwei *tatsächlich* getrennte Register (der Gesamtzustand wird nicht mittels Tensorprodukt gebildet), *X-Register* und *Y-Register*. Vermutlich wurde diese Aufteilung gemacht, um bestimmte Operatoren zu vereinfachen: Die QFT und die Grover-Iteration können nur auf jeweils ein ganzes Register angewendet werden, das Oracle bekommt jeweils ein ganzes Register als X- und Y-Argument.

#### Operatoren

H, CNOT, X, Y, Z, S,  $S^\dagger$ , T,  $\sqrt{X}$ , TOFFOLI, QFT,  $QFT^{-1}$ , Oracle (definiert durch einen algebraischen Ausdruck), Rotation operator (beliebige gesteuerte Ein-Bit-Rotation um die X-,

---

<sup>1</sup>Frei verfügbar unter: <http://jqantum.sourceforge.net/index.html>

### 3. Ähnliche Arbeiten

---

Y- oder Z-Achse), *Grover Operator* sowie *Measurement* (Messung einzelner Bits oder eines gesamten Registers).

#### Benutzeroberfläche

Die Bearbeitungswerkzeuge in der GUI sind rudimentär, Schaltkreise können nur sequentiell konstruiert werden. Möchte man an der Struktur etwas ändern, bleibt nur die Möglichkeit den Schaltkreis bis zum gewünschten Schritt zu löschen und die Operatoren sukzessiv neu anzulegen. Quantenzustände werden als Zustandsvektoren repräsentiert, die Anzeige erfolgt in Form einer Liste der komplexen Amplituden (bei großen Zuständen wird die Liste zu einer Matrix umgebrochen), kodiert als farbige Fläche (*Farbe*  $\hat{=}$  *Winkel*, *Helligkeit*  $\hat{=}$  *Magnitude*).

### 3.2. Zeno

Ebenfalls auf Java/Swing basierend ermöglicht *Zeno*<sup>2</sup> die Konstruktion sowie schrittweise Simulation von beliebigen Schaltkreisen mit sowohl *reinen*, als auch *gemischten* Zuständen (*Dichtematrix*<sup>3</sup>). *Zeno* wurde 2004 ursprünglich von Gustavo Eulalio M. Cabral an der *Universidade Federal de Campina Grande* (Brasilien), im Zusammenhang einer Masterarbeit entwickelt. 2006 wurde eine überarbeitete Version veröffentlicht.

#### Operatoren

*Zeno* erlaubt es, ein-, zwei- und drei-Bit Operatoren als unitäre Matrizen, als auch parametrische Rotationsgatter und Messoperatoren frei zu definieren. Des Weiteren können die Gatter mit beliebig vielen Steuerbits versehen werden. Vordefiniert sind folgende Operatoren: X, Y, Z, H, S, T, FREDKIN, SWAP, *Rotation* (X-, Y-, Z-Rotation unter Angabe der Winkel), *Measure* (Einzelbit-Messung in der Standardbasis), *Partial trace*.

#### Benutzeroberfläche

Quantenschaltkreise werden in *Zeno* tabellarisch organisiert, jede Zeile bezeichnet ein Qubit und pro Spalte kann ein Operator eingefügt werden. Operatoren können intuitiv verschob-

---

<sup>2</sup>Frei verfügbar unter: [http://dsc.ufcg.edu.br/~iquanta/zeno/index\\_en.html](http://dsc.ufcg.edu.br/~iquanta/zeno/index_en.html)

<sup>3</sup>Siehe NC00, Kap. 2.4

ben/konfiguriert und wieder entfernt werden, bei allen Gattern können durch Rechtsklick Steuerbits hinzugefügt werden. Es ist, wie bereits erwähnt, möglich eigene Operatoren anzulegen. Matrizen werden als Tabelle komplexer Zahlen, mit jeweils separaten Eingabefeldern für den Real- und Imaginärteil numerisch spezifiziert. Simulationsergebnisse können tabellarisch/textuell, oder als Histogramm visualisiert werden.

### 3.3. QCAD 2.0

Ein Windows basierter QC-Simulator, programmiert in C++. Ermöglicht die Konstruktion sowie Simulation (schrittweise Simulation ist nicht verfügbar) beliebiger Quantenschaltkreise. QCAD wurde von Hiroshi Watanabe, Masaru Suzuki und Junnosuke Yamazaki an der Universität von Tokyo sowie der Nagoya Universität entwickelt und 2001 erstmals veröffentlicht<sup>4</sup>.

#### Verfügbare Operatoren

H, NOT, CNOT, TOFFOLI, X, Y, Z, SWAP, (steuerbare) *Z-Rotation* sowie *Measurement* (Einzelbit-Messung, nur am Ende des Schaltkreises).

#### Benutzeroberfläche

Operatoren können im Schaltkreis an beliebiger Stelle eingefügt, konfiguriert und wieder gelöscht werden (Verschieben ist nicht möglich). Ähnlich wie bei *Zeno* wird ein tabellarisches Layout verwendet, allerdings können mehrere Operatoren in einem Schritt (einer Spalte) zusammengefasst werden. Quantenzustände werden als Zustandsvektoren repräsentiert, zur Anzeige stehen drei verschiedene Ansichten zur Verfügung:

- Standard View (Liste komplexer Amplituden als Text und Vektor-Grafik)
- HSV-View (3D-Barchart:  $Farbe \hat{=} Winkel$ ,  $Höhe \hat{=} Magnitude$ )
- 2D HSV View ( $Farbe \hat{=} Winkel$ ,  $Helligkeit \hat{=} Magnitude$ )

Messergebnisse können in einer separaten Ansicht angezeigt werden.

---

<sup>4</sup>Frei verfügbar unter: <http://qcad.sourceforge.jp/>

## 4. Analyse

Im folgenden Kapitel werden die Ergebnisse der Analyse-Phase dargestellt. Zunächst wird die gewählte Simulationsmethode (4.1) beschrieben. Daraufhin folgt eine Aufstellung der Anforderungen (4.2) an die zu entwickelnde Software. Weiterhin wird das konzeptionelle Datenmodell präsentiert (4.3), und es werden die grundlegenden Anwendungsfälle (4.4) aufgezeigt. Am Ende des Kapitels wird das Konzept der graphischen Oberfläche (4.5) vorgestellt.

Während einer Machbarkeitsstudie wurde ein initialer Prototyp in Java entwickelt, anhand dessen die Realisierbarkeit der fundamentalen Funktionen des Simulator evaluiert wurde. Ausgehend von den daraus gezogenen Erkenntnissen, wurden einige grundlegende Entscheidungen für den Entwicklungsprozess getroffen, welche im weiteren Verlauf der Entwicklung teils iterativ überarbeitet worden sind.

### 4.1. Simulationsmethode

Aus der bisher nicht widerlegten Vermutung, dass Quantencomputer gegenüber klassischen Computern bei bestimmten Problemstellungen exponentiell effizienter arbeiten, lässt sich bereits ableiten, dass die Simulation der entsprechenden Quantenalgorithmen mittels klassischer Computer nur ineffizient möglich ist. Fände man einen klassischen Algorithmus welcher Quantenalgorithmen effizient simuliert, so bräuchte man keinen Quantencomputer mehr zu bauen.

Der einfachste Ansatz zur Simulation von Quantenschaltkreisen, im Folgenden als *naiver Ansatz* bezeichnet, leitet sich direkt aus der formalen Beschreibung von Quantenzuständen und Operatoren als komplexe Vektoren und Matrizen ab. Da diese Methode sehr ineffizient ist, wurden verschiedene optimierte Techniken entwickelt, wovon einige für bestimmte Problemstellungen deutlich bessere Eigenschaften aufweisen [Vid03](#) [Via07](#), während andere die Komplexität allgemein verringern [OD98](#) [AMS07](#). Bemerkenswert ist der Ansatz von [Via07](#), welcher bei der Simulation des Grover-Algorithmus nahezu die Laufzeitcharakteristiken eines

echten Quantencomputers aufweist und damit die Nützlichkeit von Quantencomputern für derartige Problemstellungen generell in Frage stellt (vgl. VMH05).

Es ist abzusehen, dass neue Prozessorgenerationen in Zukunft mehr und mehr auf Parallele Architekturen setzen werden, während die Taktung der einzelnen Cores vermutlich unverändert bleibt (vgl. HBK06). Daher wird der Fokus in dieser Arbeit auf die Parallelisierbarkeit der Simulation gelegt. Ausgehend vom naiven Ansatz (4.1.1) wird in Abschnitt 4.1.2 die gewählte Optimierungsstrategie beschrieben.

### 4.1.1. Naiver Ansatz

Ein  $n$ -Bit Quantensystem  $|\psi\rangle$  wird als Zustandsvektor  $\vec{v} \in \mathbb{C}^{2^n}$  dargestellt. Unitäre Operatoren werden durch Multiplikation mit entsprechenden Matrizen  $U \in \mathbb{C}^{2^n \times 2^n}$  simuliert, welche entsprechend durch Tensor-Verknüpfung kleiner Matrizen konstruiert werden können (siehe 2.2.2). Die Messung entspricht der Projektion auf eine Orthonormalbasis  $B = \{\vec{b}_1, \vec{b}_2, \dots, \vec{b}_n\}$ , sowie der statistischen Selektion eines Basiszustandes (ist  $B$  die Standardbasis kann auf die Berechnung der Skalarprodukte mit den Basisvektoren  $\vec{b}_i$  verzichtet werden).

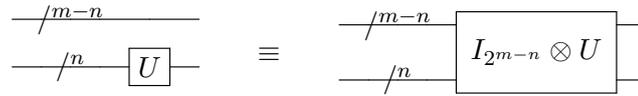
Prinzipiell haben klassische Simulationsalgorithmen welche diesen Ansatz verfolgen exponentielle Komplexität. Benutzt man explizit  $2^n \times 2^n$  große Matrizen als Operatoren, so wird schon für einen 16-Bit Operator eine Matrix mit  $65536 \times 65536$  Elementen benötigt, bei 16 Byte =  $2 \cdot \text{sizeof}(\text{double})$  pro Element ergibt sich ein Speicherbedarf von 64 Gigabyte. Die Laufzeit der Multiplikation entspricht  $O((2^n)^2)$ . Es wird klar, dass diese Vorgehensweise absolut nicht praktikabel ist, da schon für die Simulation kleiner Quantenregister astronomische Ressourcen verlangt werden.

### 4.1.2. Optimierte Methode

Untersucht man die Struktur der Operatoren welche durch Tensorverknüpfung (siehe 2.2.2) entstehen, so zeigt sich, dass die resultierenden Matrizen eine stark redundante Blockstruktur aufweisen<sup>1</sup>. Besonders interessant ist der Fall wenn ein  $n$ -Bit Operator  $U$  auf die unteren Bits eines  $m$ -Bit ( $m > n$ ) Quantenregisters angewendet werden soll, siehe Abbildung 4.1.

---

<sup>1</sup>Dies ist auch der Ansatz von [Via07].



**Abbildung 4.1.:** Transformation der unteren Bits

Da der Operator kleiner ist als das Register, muss dieser durch Tensorverknüpfung mit der entsprechenden Einheitsmatrix  $I_{2^{m-n}}$  auf die benötigte Dimension gebracht werden. Dabei entsteht generell eine periodische blockdiagonale Matrix:

$$I_{2^{m-n}} \otimes U = \begin{bmatrix} U_1 & 0 & \dots & 0 \\ 0 & U_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & U_{(m-n)} \end{bmatrix} \quad (4.1)$$

Da die Blöcke  $U_i$  identisch sind, muss explizit keine  $2^m \times 2^m$  Matrix produziert werden, es genügt  $U$  im Speicher zu halten und bei der Multiplikation auf die entsprechenden Segmente des Zustandsvektors anzuwenden. Vorausgesetzt  $U$  ist klein, beispielsweise  $2 \times 2$ , reduziert sich die Laufzeit-Komplexität der Matrix-Multiplikation auf  $O(2^n)$ , ein quadratischer Vorteil gegenüber dem naiven Algorithmus, während der Speicherbedarf für den Operator *exponentiell* schrumpft. Weiterhin lässt sich die Matrix-Multiplikation ausgezeichnet parallelisieren. Zwar bleibt die Laufzeit super-polynomial, aber die maximale Problemgröße wird proportional zur Anzahl der Recheneinheit (Cores<sup>2</sup>) deutlich erhöht. [vgl. AMS07]

Selbstverständlich kann man nicht erwarten, dass alle Operatoren in dieser Form angewendet werden. Der Aufwand, die Bits eines Quantenregisters so umzusortieren, dass die *Standard-Form* (AMS07) wie in Abbildung 4.1 erreicht wird, ist allerdings vergleichsweise gering, da dies einfach durch Permutation der Indizes während der Matrix-Multiplikation erledigt werden kann. [vgl. AMS07]

Gesteuerte Gatter können, entgegen der Tatsache, dass deren Matrizen mit steigender Anzahl der Steuerbits exponentiell anwachsen, proportional zur Anzahl der Steuerbits effizienter

<sup>2</sup>Optimierung für Cluster-Systeme, wie in AMS07 beschrieben, ist nicht Ziel dieser Arbeit.

angewendet werden. Die Matrix eines gesteuerten Gatters  $C(U)$  weist folgende Topologie auf:

$$C(U) = \begin{bmatrix} I & \\ & U \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \\ & u_{1,1} & u_{1,2} & \dots & u_{1,2^n} \\ & u_{2,1} & u_{2,2} & \dots & u_{2,2^n} \\ & \vdots & \vdots & \ddots & \vdots \\ & u_{2^n,1} & u_{2^n,2} & \dots & u_{2^n,2^n} \end{bmatrix}. \quad (4.2)$$

Der obere Block stellt die Einheitsmatrix dar, hat somit bei der Multiplikation keinen Effekt, und kann effektiv ignoriert werden. Dies ermöglicht es, beliebig gesteuerte Gatter parametrisiert zu simulieren, ohne die entsprechende Matrix explizit zu erzeugen. Es wird sogar eine, zur Anzahl der Steuerbits proportionale Steigerung der Effizienz erreicht.

## 4.2. Anforderungen

In den folgen Abschnitten werden die Anforderungen an die zu entwickelnde Software aufgestellt. Es wird zwischen funktionalen und nicht funktionalen Anforderungen unterschieden.

### 4.2.1. Funktionale Anforderungen

Funktionale Anforderungen lassen sich in konkrete Software-Funktionalität übertragen und sind im Allgemeinen eindeutig formulierbar.

#### 4.2.1.1. Korrektheit

Die Korrektheit der Berechnungen ist essentiell wichtig für die Nutzbarkeit des Simulators. Dabei gilt es zwei verschiedene Aspekte zu beachten, zum Einen muss eine Berechnung den von ihr erwarteten Algorithmus korrekt ausführen, zum Anderen sollte die numerische Qualität des Ergebnisses nicht unterhalb einer gewissen Fehlertoleranz liegen.

Um in der Softwareentwicklung die Korrektheit eines Algorithmus zu verifizieren werden Unit-Tests und Code-Reviews angesetzt. Es ist außerdem sinnvoll das Problem in eine möglichst gut verständliche, abstrakte Struktur zu zerlegen um Fehlern vorzubeugen.

Die numerische Stabilität einer Berechnung hängt sowohl von dem zugrundeliegenden numerischen Datentyp, als auch von dem verwendeten Algorithmus ab. In der Regel wird für wissenschaftliche Anwendungen der 64-Bit IEEE Floating-Point-Typ verwendet; auf den heute üblichen Rechnerarchitekturen (sowie in der JVM) ist dies der präziseste native Typ. Reicht die Präzision nicht aus, bleibt nur die Möglichkeit auf software-basierte Langzahlarithmetik (arbitrary-precision arithmetic) zurückzugreifen, was mit erheblichen Performance-Einbußen einhergehen kann.

**[REQ F 1]** Um die Präzision des Simulators offen zu lassen, soll eine Abstraktionsebene für die benötigten numerische Typen, Real und Complex, eingeführt werden, so dass die konkrete Implementierung austauschbar bleibt.

##### 4.2.1.2. Funktionsumfang

**[REQ F 2.1]** Die Software soll (zumindest) die (schrittweise) Simulation der in Abschnitt 2.3 vorgestellten Algorithmen ermöglichen. Um dies zu erreichen sollen folgende Operationen auf Quantenregistern simuliert werden können:

1. beliebige ein- bis drei Bit *Quantengatter*
2. Auswertung von beliebigen *Oracle*-Funtionen, definitert als algebraischer Ausdruck
3. Messung in der Standardbasis, optional auch in anderen Basen
4. Die *Quanten Fourier Transformation* (QFT)
5. Der *Grover Diffusionsoperator*

QFT und Grover Diffusionsoperator können auch manuell mit den Grundgattern konstruiert werden, allerdings ist es komfortabel diese wie einfache Operatoren behandeln zu können.

**[REQ F 2.2]** Des weiteren soll es möglich sein, einzelne Operatoren zu "Unterprogrammen" zu gruppieren, so dass unter anderem Teile von Schaltkreisen wiederverwendbar und wiederholbar sein können (Vorteilhaft beispielsweise beim Grover-Algorithmus).

**[REQ F 2.3]** Die Software soll eine graphische Benutzeroberfläche mit einem WYSIWYG-Editor für Quantenschaltkreise, sowie Anzeigen für den Zustand der Simulation zur Verfügung stellen (siehe auch 4.5).

**[REQ F 2.4]** Die Bearbeitungsmöglichkeiten des Editors sollen gewissen Ansprüchen genügen, d.h. konkret wird folgende Funktionalität erwartet:

1. *Einfügen* von Elementen
2. *Verschieben* von Elementen
3. *Duplizieren* von Elementen
4. *Konfiguration* von Elementen (Eigenschaften änderbar)
5. *Löschen* von Elementen
6. *Undo/Redo*-Funktionalität

**[REQ F 2.5]** Für die schrittweise Simulation sollen die Funktionen

1. *Schritt vor*
2. *Schritt zurück*
3. *Zurücksetzen*
4. *Gesamtdurchlauf*

zur Verfügung stehen.

**[REQ F 2.6]** Simulationsergebnisse sollen so dargestellt werden, dass die Wahrscheinlichkeitsamplituden den jeweiligen Basiszuständen visuell zugeordnet werden können. Standardmäßig soll dazu eine tabellarische Anzeige verwendet werden. Der Übersichtlichkeit halber sollen Null-Elemente des Zustandsvektors ausgeblendet werden können.

##### 4.2.1.3. Plattformunabhängigkeit

**[REQ F 3]** Die Software soll im Sinne von "Compile once, run everywhere" plattformunabhängig sein.

### 4.2.1.4. Erweiterbarkeit

Ein weiterer wichtiger Aspekt der Softwareentwicklung ist die Erweiterbarkeit. Es soll ohne großen Aufwand möglich sein neue Features hinzuzufügen bzw. bestehende zu ändern oder zu erweitern. Konkret interessant ist in diesem Fall die Erweiterbarkeit bezüglich der zur Verfügung stehenden Operatoren.

**[REQ F 4.1]** Es soll möglich sein im Frontend neue Gates anhand einer komplexen Matrix zu definieren. Die Elemente der Matrix sollen als algebraische Ausdrücke akzeptiert und ausgewertet werden, dies ist angenehm für den Benutzer und ermöglicht zusätzlich eine symbolische Anzeige der Matrix, beispielsweise mittels einem TeX/LaTeX Renderer.

**[REQ F 4.2]** Operatoren deren Implementierung als Matrix nicht machbar bzw. zu ineffizient ist, sollen als Plugin nachgerüstet werden können. Ein Beispiel wäre der *Groover-Diffusionsoperator*  $2|\psi\rangle\langle\psi| - I$ , mit  $|\psi\rangle$  einer gleichmäßigen Superposition (siehe [NC00](#), S. 251): die resultierende Matrix ist in der Anzahl der Bits exponentiell groß, voll besetzt, und lässt sich nicht als Tensorprodukt kleinerer Matrizen darstellen <sup>3</sup>.

### 4.2.2. Nicht-funktionale Anforderungen

Nicht-funktionale Anforderungen definieren eher subjektives Verhalten der Software, und sind meistens weniger eindeutig als funktionale Anforderungen. Um nicht-funktionale Anforderungen am Endprodukt überprüfbar zu machen, müssen bestimmte Rahmenbedingungen festgelegt werden, von denen sich konkrete Metriken ableiten lassen.

#### 4.2.2.1. Verständlichkeit

Da der Simulator einem Benutzer ohne große Vorkenntnisse in der Quanteninformatik zugänglich sein soll, muss der Gegenstand der Simulation möglichst verständlich dargestellt werden. **[REQ NF 1]** Die graphische Darstellung soll daher der in der Fachliteratur allgemein gebräuchlichen entsprechen (siehe [\[BBC<sup>+</sup>95\]](#), [\[NC00\]](#)), so dass schnell ein direkter Bezug hergestellt werden kann.

---

<sup>3</sup>Es hat sich gezeigt, dass der Diffusionsoperator aus einer Kombination grundlegender Gatter realisiert werden kann, allerdings ist es benutzerfreundlich, für diese Transformation einen *Macro-Block* anzulegen. Gleiches gilt für die *QFT*.

### 4.2.2.2. Effizienz

Wie in Kapitel 4.1 beschrieben, stellt die Simulation von Quantenschaltkreisen allgemein ein super-polynomiales Problem dar. [REQ NF 2] Als Anforderung für die Nützlichkeit des Simulators wird definiert, dass die Anwendung aller Basisoperatoren (Transformation durch Gate, Auswertung einer Blackbox-Funktion und Messung) auf einem 20-Bit Quantenregister in weniger als einer Sekunde ausgeführt werden kann. Für den Speicherbedarf wird keine Vorgabe gemacht. Als Referenzarchitektur wird ein Thinkpad w520 mit Intel® Core™ i7-2760QM Quadcore (2.4 - 3.5 GHz) Prozessor und 16 GB RAM angegeben.

## 4.3. Konzeptuelles Datenmodell

Im konzeptuellen (fachlichen) Datenmodell werden die Entitäten, deren Attribute sowie Beziehungen untereinander erfasst. Die Betrachtung soll zunächst rein fachlicher Natur sein, technische Details werden in der Entwurfsphase ausgearbeitet.

In diesem Fall gilt es eine Darstellungsform für Quantenschaltkreise zu finden, welche sowohl graphische Anzeige und Bearbeitung, als auch die schrittweise Simulation ermöglicht. Abbildung 4.2 zeigt das konzeptuelle Datenmodell als UML2 Klassendiagramm.

### Circuit

Ein Circuit (Quantenschaltkreis) besteht aus einer Sequenz von Operatoren, unterteilt in Simulationsschritte, auf einer Menge von Quantenbits (einem Quantenregister). Die Quantenbits können zu Anzeigezwecken in logische Subregister unterteilt werden, dabei muss allerdings beachtet werden, dass diese untereinander verschränkt sein können; der Zustand einzelner logischer Register kann also nicht separat betrachtet werden, es zählt immer der Gesamtzustand des Systems welcher sich über das Tensorprodukt der Teilsysteme herleiten lässt.

### Register

(Quantenregister) Stellt die logische Gruppierung einer Menge an Bits dar. Zu Anzeigezwecken können Register mit einem Label (name) versehen werden.

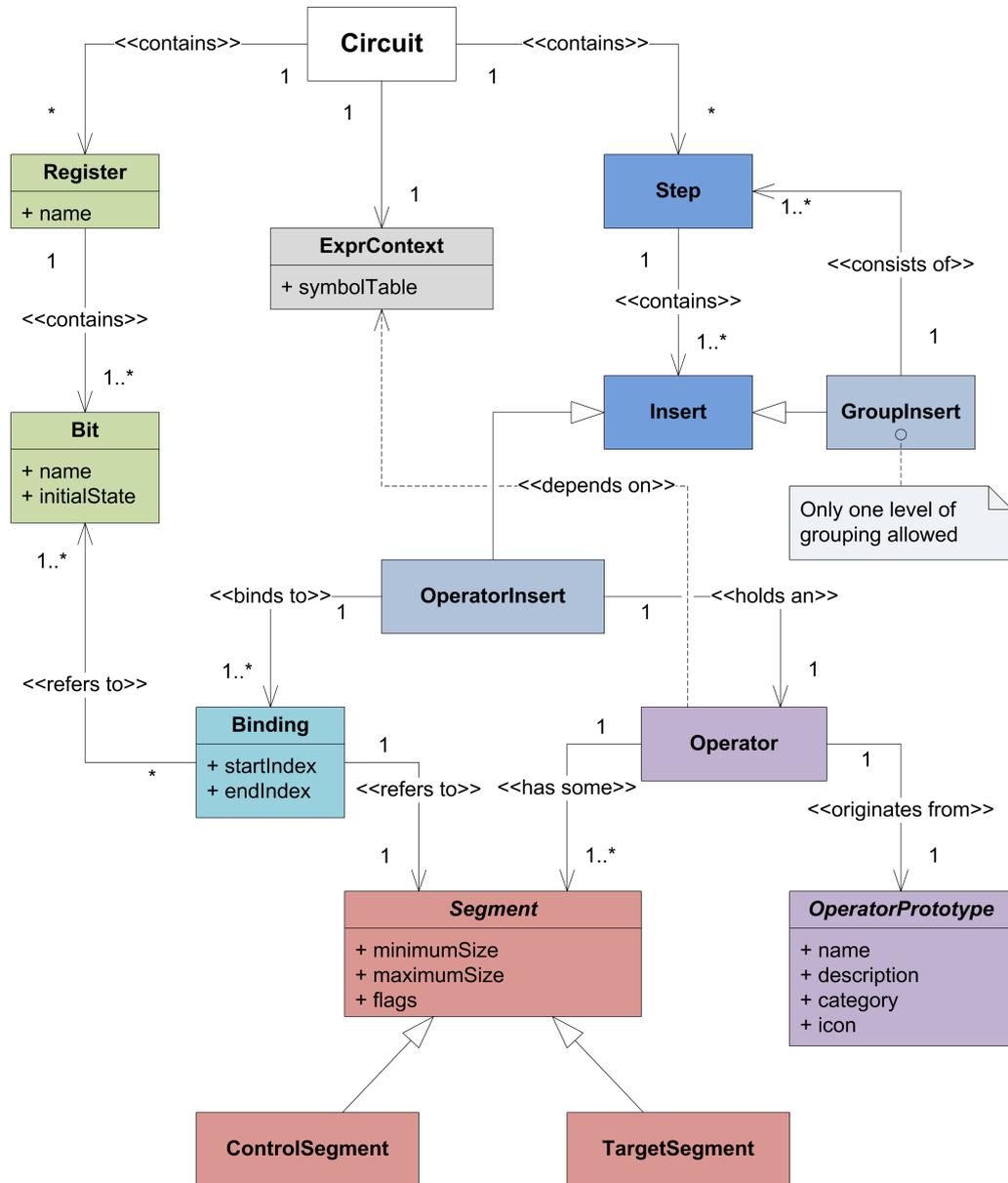


Abbildung 4.2.: UML2 Klassendiagramm: Konzeptuelles Datenmodell

### **Bit**

(Quantenbit) Repräsentiert die kleinste Informationseinheit des Quantenschaltkreises. Kennt seinen initialen Zustand ( $|0\rangle$  oder  $|1\rangle$ ) und kann zu Anzeigezwecken mit einem Label (name) versehen werden.

### **Step**

Fasst ein oder mehrere Inserts zu einem *Simulationsschritt* zusammen. Aus Gründen der Übersichtlichkeit müssen die Inserts bezüglich ihrer Bindungen disjunkt sein.

### **Insert**

Stellt ein "Halterung" für beliebige Operatoren dar. Jedes Insert beansprucht ein Intervall von Bits.

### **GroupInsert**

Definiert eine Gruppierung von Operatoren bzw. Simulationsschritten. Es ist nur eine Ebene vorgesehen, weiter verschachtelte Gruppen würden Schwierigkeiten bei der Anzeige, sowie der Bearbeitungsfunktionalität mit sich bringen.

### **OperatorInsert**

Bindet einen Operator an eine Menge von Bits. Pro Segment des Operators existiert eine entsprechende Bindung.

### **Operator**

Stellt einen beliebigen Operator, basierend auf einem `OperatorPrototype`, mit einer Menge an Segmenten dar.

### **OperatorPrototype**

Definiert eine Klasse von Operatoren und fungiert als Factory.

### **Segment**

Abstraktion zur Gruppierung von Ein-/Ausgabebits eines Operators. Es existieren verschiedene Typen von Segmenten. Zunächst lässt sich zwischen Kontroll- und Zielsegmenten unterscheiden. Des Weiteren sollen bestimmte Segmente einem fixierten Intervall entsprechen, andere sollen variabel sein. Beispielsweise soll ein *Oracle*-Operator einer beliebigen Anzahl an  $X$ - und  $Y$ -Bits zugewiesen werden können. Einige Segmente sollen anders angezeigt werden als andere, z.B. das CNOT-Gatter wird üblicherweise mit einem  $\oplus$ -Symbol auf dem Zielbit, ohne Rahmen, angezeigt, dazu werden einen `Segment Flags` zugeordnet. Bisher ist `NoBox` die einzig notwendige Anzeigeoption, sie bewirkt, dass das Segment ohne Rahmen angezeigt wird.

### **Binding**

Referenziert ein Intervall von Bits über deren globalen Index (lokaler Index würde den Index innerhalb des Registers bezeichnen).

### **ExprContext**

Erfasst globale (bzgl. des Schaltkreises/Dokuments) Definitionen zur Nutzung in algebraischen Ausdrücken.

## **4.4. Anwendungsfälle**

Die folgenden Anwendungsfälle, siehe Abbildung 4.3, betreffen die Interaktionsmöglichkeiten des Benutzers mit der Software (über die GUI). Da es sich insgesamt um sehr einfache, für GUI-Anwendungen typische Anwendungsfälle handelt, und des Weiteren keine restriktiven Vorgaben für die Umsetzung gemacht werden, wird auf eine tabellarische Spezifikation verzichtet. Ziel ist somit weniger die formal exakte Modellierung als viel mehr die übersichtliche Darstellung der Funktionalität aus Perspektive des Benutzers.

### **4.4.1. Standard Functionality**

Unter diesem abstrakten Anwendungsfall wird die übliche Standardfunktionalität moderner GUI-Anwendungen zusammengefasst: Speichern und Laden von Dokumenten sowie Undo/Redo Funktionalität.

### 4.4.2. Edit Circuit

Dem Benutzer sollen diverse Bearbeitungsmöglichkeiten zur Verfügung stehen. So sollen Elemente des Schaltkreises eingefügt, bearbeitet, und gelöscht werden können. Das Einfügen von Elementen kann mit zusätzlichen Dialogen realisiert werden, im Allgemeinen soll die Bearbeitung aber möglichst direkt (Drag & Drop, Tastatureingabe) erfolgen. Für alle abgeleiteten Anwendungsfälle gilt der Ablauf:

1. Benutzer wählt Bearbeitungskommando
2. Anwendung berechnet modifizierten Schaltkreis
3. Anwendung setzt Undo-Schritt
4. Anwendung setzt *modified-Flag* des Dokuments
5. Anwendung zeigt modifizierten Schaltkreis an

### 4.4.3. Stepwise Simulation

Die Simulation soll sowohl schrittweise vorwärts und rückwärts (nur möglich wenn alle Operatoren des letzten Schrittes reversibel sind), sowie im Ganzen möglich sein. Für alle abgeleiteten Anwendungsfälle gilt der Ablauf:

1. Benutzer wählt Simulationskommando
2. Anwendung berechnet Folgezustand und evtl. Messergebnisse
3. Anwendung zeigt Folgezustand und evtl. Messergebnisse an

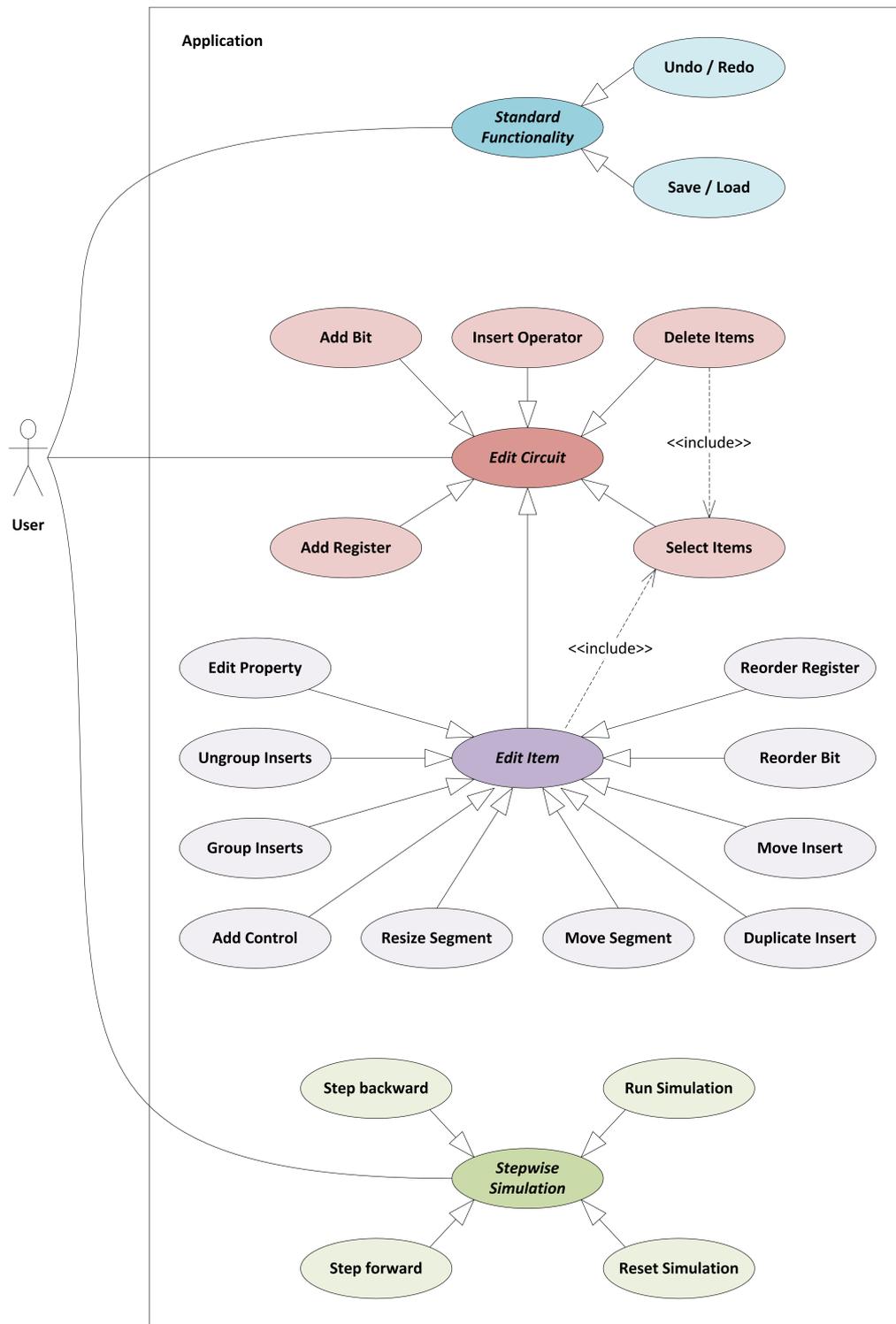


Abbildung 4.3.: UML2 Anwendungsfalldiagramm

## 4.5. Konzeption der GUI

In den folgenden Abschnitten wird das Grobkonzept der graphischen Benutzeroberfläche vorgestellt.

### 4.5.1. Struktur

Die GUI soll dem Benutzer die Möglichkeit geben, Quantenschaltkreise graphisch zu konstruieren, um diese dann schrittweise simulieren zu können. Die Resultate der Simulationsschritte, also Folgezustände und Messergebnisse, sollen direkt, und möglichst übersichtlich dargestellt werden. Beides, der Editor sowie die Simulationsergebnisse, sollen dem Benutzer in einem Fenster pro Dokument (siehe Abbildung 4.4) präsentiert werden.

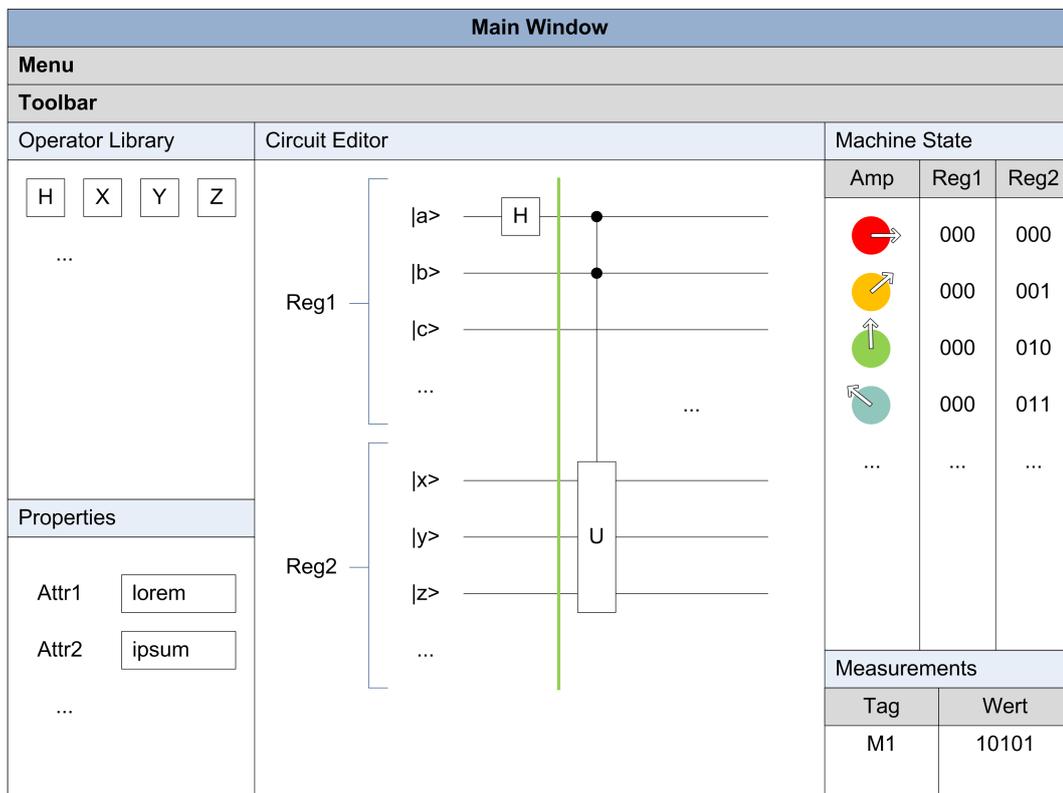


Abbildung 4.4.: GUI-Mockup: Hauptfenster

#### 4.5.2. Editor für Quantenschaltkreise

Der Editor-Part untergliedert sich in drei Teile:

**Circuit Editor** Stellt den Schaltkreis graphisch dar. Die Elemente können mit der Maus ausgewählt und bearbeitet/positioniert werden.

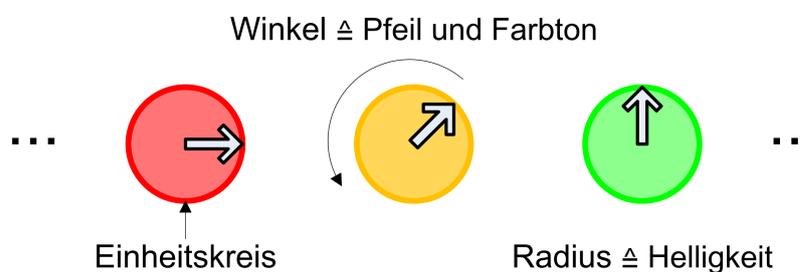
**Operator Library** Zeigt die verfügbaren Operatoren in einer Liste an, und ermöglicht das Hinzufügen von Elementen zum Schaltkreis mittels Drag & Drop.

**Properties** Zeigt die Eigenschaften der aktuell ausgewählten Elemente des Schaltkreises an, und ermöglicht es, diese direkt zu verändern.

#### 4.5.3. Zustandsanzeige und Messergebnisse

Für die Betrachtung der Simulationsergebnisse stehen zwei Ansichten zur Verfügung:

**Machine State** Zeigt den jeweils aktuellen Zustandsvektor des Gesamtsystems an. Standardmäßig soll diese Ansicht tabellarisch die komplexen Amplituden (siehe Abbildung 4.5) in Verbindung mit den Basiszuständen der einzelnen Register darstellen, allerdings sind auch alternative Darstellungsformen wünschenswert, werden jedoch hier nicht näher spezifiziert.



**Abbildung 4.5.:** Darstellung komplexer Zahlen in der GUI

**Measurements** Da die Ergebnisse von Teilmessungen oftmals nicht direkt vom Zustandsvektor abgelesen werden können, existiert eine separate Ansicht, welche jeweils die Messergebnisse des letzten Schrittes in numerischer Form darstellt. Um einzelne Messungen identifizierbar zu machen, könne diese Mit einem *Tag* versehen werden, so dass immer eine Zuordnung zwischen Messoperator und Messergebnis gemacht werden kann.

## 5. Entwurf

In diesem Kapitel werden die technischen Designentscheidungen zur Realisierung des Simulators dargestellt. Zunächst wird die Wahl der Technologie (5.1) begründet, darauf folgt eine detaillierte Beschreibung der Architektur (5.2) und der wichtigsten Datenstrukturen. Als Letztes werden die verwendeten Kernalgorithmen (5.3) zur Simulation von Quantenschaltkreisen vorgestellt.

### 5.1. Technologie

Grundsätzlich soll Java als Laufzeitumgebung genutzt werden. Gründe dafür sind zum einen die Plattformunabhängigkeit sowie die hohe Zuverlässigkeit und Flexibilität. Alternativ wurde C++ erwogen. Argumente für C++ (bzw. Sprachen die zu nativem Maschinencode compilieren) wären die im Allgemeinen besseren Performance-Eigenschaften und Optimierungsmöglichkeiten für rechenintensive Aufgaben. Dabei ist aber zu beachten, dass nativer Maschinencode immer nur für eine Architektur besonders gut optimiert werden kann. Java Bytecode ist in dieser Hinsicht offen für plattformspezifische Optimierungen. Die virtuelle Maschine kennt die Zielplattform in vielen Fällen besser als der Programmierer, und kann bei der Erzeugung des nativen Codes (JIT-Compilation) *zur Laufzeit* entscheiden, welche Bereiche des Programms wie stark optimiert werden [vgl. Wik12b].

#### 5.1.1. Scala

Als Programmiersprache findet Scala (Version 2.9.2) Verwendung. Scala ist eine relativ neue, *objektfunktionale* Programmiersprache, welche zu Java-Bytecode compiliert wird und mit Java nahezu 100% interoperabel ist. Unter anderem bietet Scala gegenüber Java

- **funktionale Sprachfeatures** (Funktionen höherer Ordnung, immutable Objekte)
- **Pattern matching**

- ein **überlegenes Typsystem** (u.a. Mixin Komposition, Typeinferenz, Typen höherer Ordnung, verbesserte generische Typen)
- **implizite** Typkonversionen (und Parameter etc.)
- **XML-Literale**
- eine sehr **kompakte Syntax**.

*In Scala schreibt man weniger Code und weniger Code bedeutet weniger Fehler!*

[Ess11]

Des Weiteren enthält die *Scala Standard Library* einige interessante API's:

- **funktionale Kollektionen**
- seit Version 2.9 auch **parallele Kollektionen** (siehe [PBRO11](#))
- eine Implementierung des **Aktorenmodells** (inspiriert von *Erlang*)
- **kombinatorische Parser** (Parser Combinators)
- eine leichtgewichtige Wrapper-API für **Swing**
- u.v.m.

Insbesondere Scalas Ausdrucksstärke bezüglich funktionaler und paralleler Programmierung war ausschlaggebend für die Wahl als Entwicklungsumgebung. Listing 5.1 zeigt beispielhaft den minimalen Mehraufwand zur Parallelisierung einer einfachen Kollektion (Range) in Scala.

```
1 (0 until n) foreach { i => ... } // Sequential Execution
2 (0 until n).par foreach { i => ... } // Parallel Execution
```

**Listing 5.1:** Beispiel: Parallel for in Scala

Aber auch die Möglichkeit in Scala Operatoren zu überladen ist ein deutlicher Vorteil gegenüber Java im Kontext von wissenschaftlich mathematischen Anwendungen. Listing 5.2 soll dies verdeutlichen.

```
1 // (Assume a, b and c are objects, not primitive values)
2 a.add(b.mul(c)); // Java style
3 a + b * c // Scala style
```

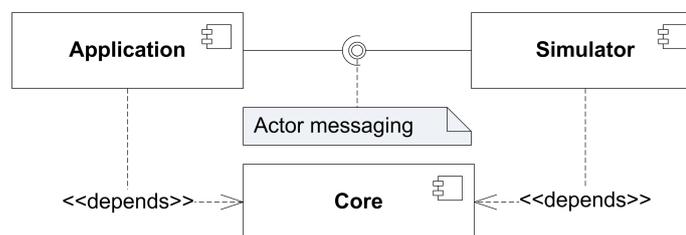
**Listing 5.2:** Beispiel: Überladen von Operatoren

### 5.1.2. Swing

Für die Realisierung der GUI wird Swing (`scala.swing`) verwendet. Als Alternativen wären SWT (Eclipse Foundation) und Qt-Jambi (Nokia) zu nennen; beide ermöglichen im Gegensatz zu Swing die Nutzung der nativen Widgets (wodurch der Look des Betriebssystems exakt adaptiert wird), benötigen aber zusätzliche native Komponenten. Swing ist 100% Java (alle nativen Komponenten sind im JRE enthalten) und damit maximal portabel.

## 5.2. Architektur

Zunächst wird die Software in zwei grundlegende Komponenten unterteilt, das Simulator-"Backend" (im folgenden als *Simulator* bezeichnet) sowie die GUI-Anwendung (im folgenden als *Anwendung* bzw. *Application* oder *GUI* bezeichnet). Der Simulator stellt ein Interface in Form eines `scala.actors.Actor` mit einem einfachen Nachrichtenprotokoll zur Verfügung. Die GUI-Anwendung definiert ein abstraktes Datenmodell zur Modellierung der Quantenschaltkreise und des Simulationsvorganges, sowie eine graphische Oberfläche auf Basis von Swing (`scala.swing`), nach Maßgabe der Konzeption in 4.5.



**Abbildung 5.1.:** UML2 Komponentendiagramm

Abbildung 5.1 zeigt anhand eines Komponentendiagramms die grundsätzliche Einteilung in Komponenten, sowie deren Schnittstellen und Abhängigkeiten untereinander. Einige gemeinsam genutzte Module werden in der Komponente *Core* zusammengefasst. So finden sich in *Core* die grundlegenden mathematischen Datentypen (Komplexe Zahlen, Matrizen und Vektoren), darauf aufbauend eine kompakte API zur Simulation von Quantenschaltkreisen, sowie eine Bibliothek zur Auswertung von algebraischen Ausdrücken. Abbildung 5.2 gibt einen Überblick über die Paketstruktur auf oberster Ebene.

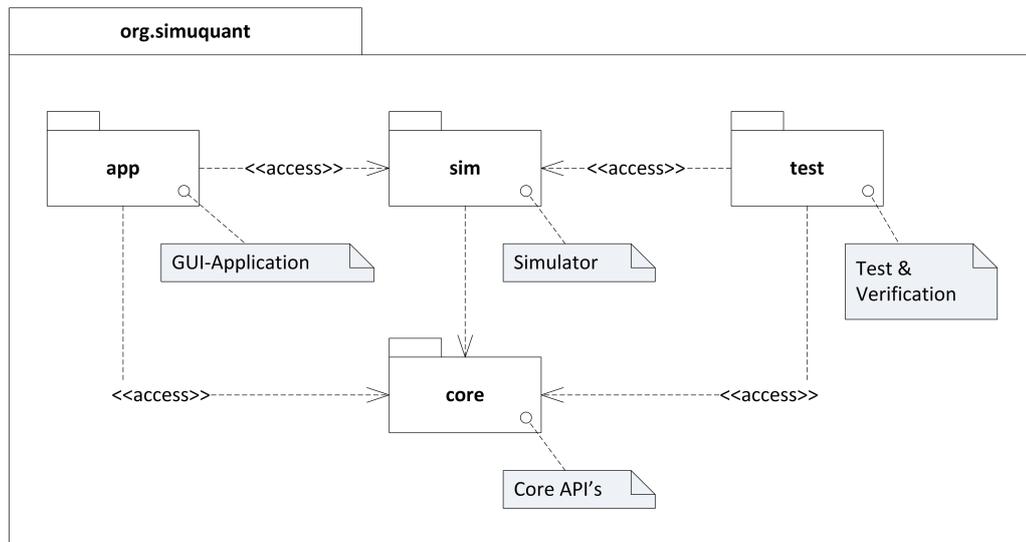


Abbildung 5.2.: UML2 Paketdiagramm: Überblick

### 5.2.1. Externe Komponenten

#### **log4j + logula**

Als Logging-Komponente wird *logula* in Verbindung mit *log4j* verwendet, *logula* wrapt *log4j* in eine Scala-freundliche API. Beide Pakete stehen unter der GPL 3 kompatiblen Apache Lizenz 2.0.

#### **jlatexmath**

*jlatexmath* ermöglicht es LaTeX Formeln in Swing Anwendungen anzuzeigen, nutzbar unter GPL 2+.

#### **InfoNode Docking Windows**

*InfoNode Docking Windows* ist ein Docking Framework auf Basis von Swing, nutzbar unter GPL 2.

### 5.2.2. Core

Die Komponente *Core* enthält die Basisbibliotheken, *math*, *quantum* und *expr*; das Paketdiagramm 5.3 gibt einen Überblick über die innere Struktur.

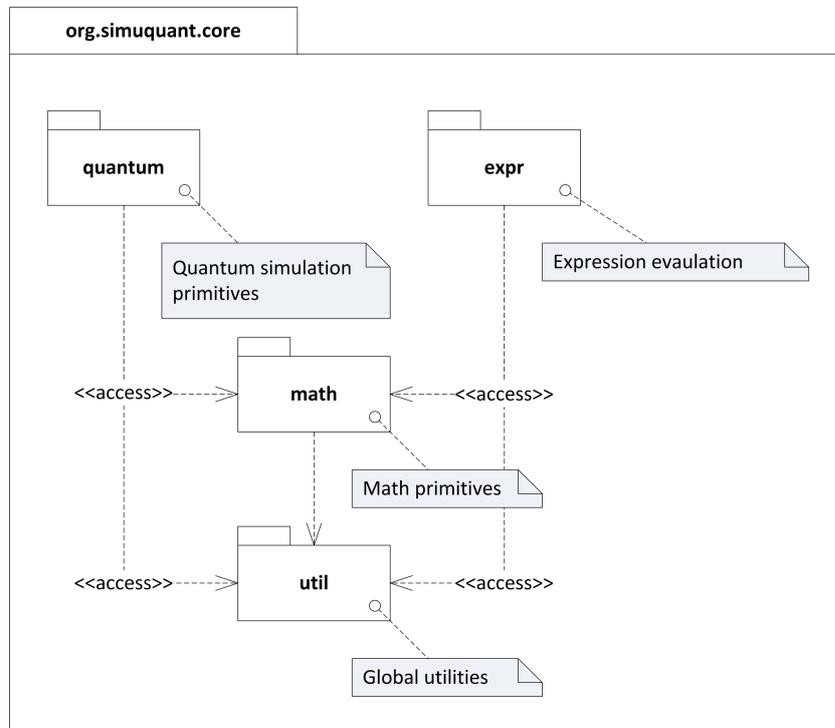


Abbildung 5.3.: UML2 Paketdiagramm: Core

Alle in *Core* enthaltenen Module sind *immutable*, d.h. mutierende Operationen erzeugen immer ein neues Objekt. Dadurch entsteht zwar ein gewisser Overhead, für die angestrebten Problemgrößen stellt das aber kein Problem dar. Im folgenden werden die mathematische Basistypen (5.2.2.1), der Simulationskernel (5.2.2.2) und die *Expression Evaluation Engine* (5.2.2.3) vorgestellt.

#### 5.2.2.1. Mathematische Basistypen

Um entsprechend der Anforderung [REQ F 1] den numerischen Basistyp austauschbar zu gestalten, und da das Java/Scala Standard API keine komplexen Zahlen definiert, werden im

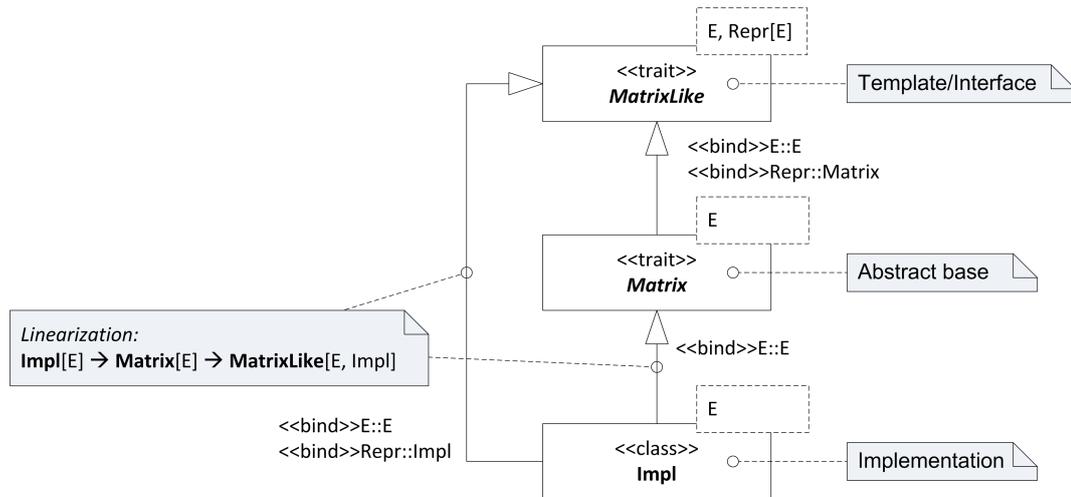


Abbildung 5.4.: UML2 Klassendiagramm: Matrix

Paket `math` die beiden Datentypen `Real` und `Complex` definiert. Des Weiteren enthält `math` die generischen Datentypen `Matrix` und `Vector`.

### Real und Complex

Um die gesamte Software nicht auf `Double` (oder einen anderen numerischen Typ) festzulegen, dient `Real` als eine Abstraktionsebene, welche es ermöglicht den grundlegenden numerischen Typ jederzeit auszutauschen. Es hat sich allerdings gezeigt, dass `Double` im Prinzip ausreichend präzise ist, weshalb `Double` die Standardimplementierung von `Real` ist. Auf `Real` aufbauend stellt `Complex` eine komplexe Zahl dar.

### Matrix und Vector

Die generischen Typen `Matrix` und `Vector` dienen der Repräsentation von Matrizen und Vektoren beliebiger Elemente sowie beliebiger Dimension. Das Interface ermöglicht jeweils für dünnbesetzte Matrizen bzw. Vektoren optimierte Implementierungen. Die Umsetzung soll im Stil der *Scala Collections API* möglichst allgemein gehalten werden, das Klassendiagramm 5.4 zeigt die typische Vererbungshierarchie am Beispiel von `Matrix`. Auffällig ist der Typparameter `Repr[E]` (*Repräsentation*), dieser dient dazu den Rückgabotyp der Operationen, welche als Templates im Trait `MatrixLike` definiert werden, zu bestimmen: Operationen auf `Impl` sollen auch `Impl` zurückgeben, nicht den abstrakten Typ. Die Interfaces von `Matrix` und `Vector` stellen einen zentralen, transparenten Mechanismus zur Parallelisierung beliebiger

Operationen zur Verfügung, in Form einer generischen Factory-Methode `realize`:

```
realize[T] : (m : Int × n : Int × f : (Int, Int) ⇒ T) ⇒ Matrix[T]
realize[T] : (n : Int × f : Int ⇒ T) ⇒ Vector[T]
```

`realize` realisiert eine beliebige Funktion `f`, welche Tupeln von Zeilen- und Spaltenindizes `(Int, Int)` (`Vector` benötigt nur Zeilenindizes) je ein Element zuordnet, als `Matrix` der Dimension `m × n` bzw. Vektor der Dimension `n`. Um Sparse-Matrix-Implementierungen zu unterstützen, können Nullelemente auf `null` abgebildet werden. Selbstverständlich wird angenommen dass `f` *pure*, d.h. seiteneffektfrei ist. Somit kann die spezifische Implementierung die Auswertung von `f` bedenkenlos parallelisieren, beispielsweise analog zu Listing 5.1 mit Scalas *Parallel Collection Framework*.

### 5.2.2.2. Simulationskernel

Dieses Paket enthält die grundlegende API zur Simulation von Quantenschaltkreisen, bestehend aus den folgenden drei Datentypen.

#### QuantumState

Repräsentiert einen Quantenzustand (Quantenregister) als Zustandsvektor (`Vector[Complex]`) und definiert drei grundlegende Operationen (die Algorithmen werden in Abschnitt 5.3 detailliert beschrieben):

**transform** Transformiert den gesamten Quantenzustand oder ein Teilregister mit einem Gate (`QuantumGate`).

**evaluate** Wertet eine beliebige Oracle-Funktion  $|x\rangle|y\rangle \xrightarrow{U_f} |x\rangle|f(x) \oplus y\rangle$  für beliebige  $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$  und zwei nicht überlappende Teilregister  $|x\rangle$  und  $|y\rangle$  aus.

**measure** Führt eine Messung des gesamten Quantenzustandes oder eines Teilregisters durch und liefert sowohl den Folgezustand als auch das binäre Ergebnis als `Tuple[Int, QuantumState]`.

Teilregister werden mittels des Typs `Qureg` als eine geordnete Menge der Bit-Indizes spezifiziert. Die Indizes müssen dabei nicht aufeinanderfolgend sein und können auch das gesamte Register *umsortieren*. Man kann also sagen, dass ein Teilregister eines  $n$ -Qubit Quantenzustan-

des in diesem Kontext eine beliebige Anordnung (durch Indizierung) von  $m \in [1, n]$  der  $n$  Qubits bezeichnet.

Da `QuantumState` auf der bereits generisch parallelisierten Datenstruktur `Vector` (siehe 5.2.2.1) beruht, muss an dieser Stelle kein weiterer Aufwand betrieben werden, um die Operationen zu parallelisieren.

### QuantumGate

Repräsentiert prinzipiell eine unitäre Matrix (`Matrix[Complex]`) der Dimension  $2^n$ , also ein  $n$ -Bit Quantengatter und definiert Operationen zur Erzeugung, sowie zur Extraktion gesteuerter Gates. Des Weiteren stellt `QuantumGate` die Standard-Gatter als Literale zur Verfügung.

### Qureg

Stellt, zum Zweck der Adressierung von Teilregistern, eine geordnete Menge von Bit-Indizes  $\{i_1, i_2, \dots, i_n\}$  dar, und enthält einige Bit-Level Operationen (diese werden in Abschnitt 5.3.1 im Detail beschreiben).

#### 5.2.2.3. Auswertung algebraischer Ausdrücke

Um während der Laufzeit des Simulators benutzerdefinierte Funktionen auswerten zu können, wird ein Parser sowie ein Interpreter benötigt. Zum einen sollen *Oracle-Funktionen* ausgewertet werden, zum anderen soll der Benutzer komplexwertige Parameter, oder auch ganze Matrizen für Operatoren definieren können. Für diese beiden Ansätze ergeben sich verschiedene Anforderungen an Laufzeiteigenschaften sowie Syntax.

Ein *Oracle* wird definiert durch eine Funktion  $f : \{0, 1\}^n \mapsto \{0, 1\}^m$ , also `Int`  $\mapsto$  `Int`, mit entsprechender Syntax für arithmetische, logische und bitweise Operatoren und Funktionen. Da das *Oracle* während der Simulation (online) für jeden Basiszustand ausgewertet werden muss, sollte der Interpreter so effizient wie möglich sein.

Komplexwertige Ausdrücke als Parameter für Operatoren werden offline, d.h. nach der Eingabe nur ein Mal ausgewertet, daher ist die Effizienz des Interpreters weniger wichtig. Die Syntax beschränkt sich auf arithmetische Operatoren und Funktionen.

Eine Möglichkeit wäre eine fusionierte *Engine*, welche sowohl Integer- als komplexwertige Ausdrücke übersetzen und interpretieren kann. Allerdings ergibt sich durch die Typisierung

## 5. Entwurf

---

der Ausdrücke eine kombinatorische Erweiterung welche die Komplexität unnötig potenziert. Die Nutzung externer Bibliotheken<sup>1 2 3</sup> scheint nicht sinnvoll zu sein, da diese in der Regel sehr allgemein gehalten sind und somit einerseits syntaktisch/semantisch auf die spezielle Aufgabe zugeschnitten werden müssten und sehr wahrscheinlich die Performance-Anforderungen nicht halten könnten. Daher werden zwei getrennte, aber sehr einfache *Engines* entwickelt (es hat sich gezeigt das *Parser*, *Codegenerator* und *Interpreter* in Scala einfach und elegant realisiert werden können), *fiee* (Fast Integer Expression Engine) und *cee* (Complex Expression Engine).

### fiee

Die *Fast Integer Expression Engine* ermöglicht das Compilieren integerwertiger Ausdrücke zu Programmen auf Basis eines Bytecodes, sowie die Ausführung der Programme mittels eines Interpreters. Der Fokus liegt auf der Effizienz des Interpreters. Aufgrund der einfachen Typisierung ist es möglich Instruktionen sowie Operanden in einem einzigen `Array[Long]` zu speichern, und mittels *Tables witch*<sup>4</sup> schnell abzuarbeiten. Alle Basisoperatoren, wie in Tabelle 5.1 aufgelistet, folgen den Präzedenz-Regeln von Java, und werden als Instruktionen direkt in den Bytecode integriert. Weitere Funktionen können mittels einer Symboltabelle definiert werden, welche zur Compilezeit gebunden wird.

---

Unäre Operatoren					
-	NEG	Negation	~	NOT	Logisches Not

---

Binäre Operatoren					
*	MUL	Multiplikation		OR	Logisches Oder
/	DIV	Division	^	XOR	Logisches exklusives Oder
%	MOD	Modulo (Divisionsrest)	<	LT	Kleiner als
+	ADD	Addition	<=	LTE	Kleiner oder gleich
-	SUB	Subtraktion	>	GT	Größer als
<<	SHL	Bitweise Linksschieben	>=	GTE	Größer oder gleich
>>	SHR	Bitweise Rechtsschieben	==	EQ	Gleich
&	AND	Logisches Und	!=	NEQ	Ungleich

---

Funktionen und Symbole		
pow	$(b \times e) \mapsto b^e$	Ganzzahlige Potenz
mod	$(a \times b) \mapsto a \bmod b$	Divisionsrest

---

<sup>1</sup> See (Scala Expression Engine) <http://scee.sourceforge.net/>

<sup>2</sup> JEval (<http://jeval.sourceforge.net/>)

<sup>3</sup> MVEL (<http://mvel.codehaus.org/>)

<sup>4</sup> `switch`-Statement auf Basis einer Sprungtabelle

## 5. Entwurf

---

`mexp`  $(b \times e \times m) \mapsto b^e \bmod m$  Modulare Exponentialfunktion

---

**Tabelle 5.1.:** Basisoperatoren der FIEE

### cee

Die *Complex Expression Engine* ermöglicht das Parsen von komplexwertigen Ausdrücken zu einer Baumstruktur, sowie die rekursive Auswertung durch Traversierung eben jener Baumstruktur. Der Fokus liegt auf der Einfachheit (der Implementierung), sowie der Flexibilität. Tabelle 5.2 zeigt die Basisoperatoren, sowie die standardmäßig definierten Funktionen. Weitere Funktionen können mittels einer Symboltabelle definiert werden.

---

<i>Unäre Operatoren</i>					
-	NEG	Negation			
<i>Binäre Operatoren</i>					
*	MUL	Multiplikation	+	ADD	Addition
/	DIV	Division	-	SUB	Subtraktion
%	MOD	Modulo (Divisionsrest)			
<i>Funktionen und Symbole</i>					
pi		Kreiszahl PI	sqrt	$\mathbb{C} \mapsto \mathbb{R}$	Reelle Quadratwurzel
e		Eulersche Zahl	sin	$\mathbb{C} \mapsto \mathbb{C}$	Komplexe Sinusfunktion
re	$\mathbb{C} \mapsto \mathbb{R}$	Realteil	cos	$\mathbb{C} \mapsto \mathbb{C}$	Komplexe Kosinusfunktion
im	$\mathbb{C} \mapsto \mathbb{R}$	Imaginärteil	tan	$\mathbb{C} \mapsto \mathbb{C}$	Komplexer Tangens
abs	$\mathbb{C} \mapsto \mathbb{R}$	Absolutwert	exp	$\mathbb{C} \mapsto \mathbb{C}$	Komplexe Exponentialfunktion
arg	$\mathbb{C} \mapsto \mathbb{R}$	Argument	log	$\mathbb{C} \mapsto \mathbb{C}$	Komplexer Logarithmus
norm	$\mathbb{C} \mapsto \mathbb{R}$	Norm	pow	$\mathbb{C} \times \mathbb{C} \mapsto \mathbb{C}$	Komplexe Potenz

---

**Tabelle 5.2.:** Basisoperatoren der CEE

### 5.2.3. Simulator

Diese Komponente definiert einen leichtgewichtigen Service auf Basis des Aktorenmodells. Über ein allgemein gehaltenes, erweiterbares Nachrichtenprotokoll (siehe Abschnitt 5.2.3.1) können andere Komponenten Operationen auf einem Quantenzustand (`QuantumState`) ausführen und erhalten Informationen über Folgezustände, Messergebnisse sowie Fehler.

### 5.2.3.1. Nachrichtenprotokoll

Das Nachrichtenprotokoll des Simulators wurde mit dem Ziel entworfen, verschiedenartige Frontends zu ermöglichen. Wie für Aktoren in Scala typisch, werden die Nachrichten als immutable case Klassen definiert. Abbildung 5.5 zeigt das vollständige Nachrichtenprotokoll zwischen einem Client und dem Simulator.

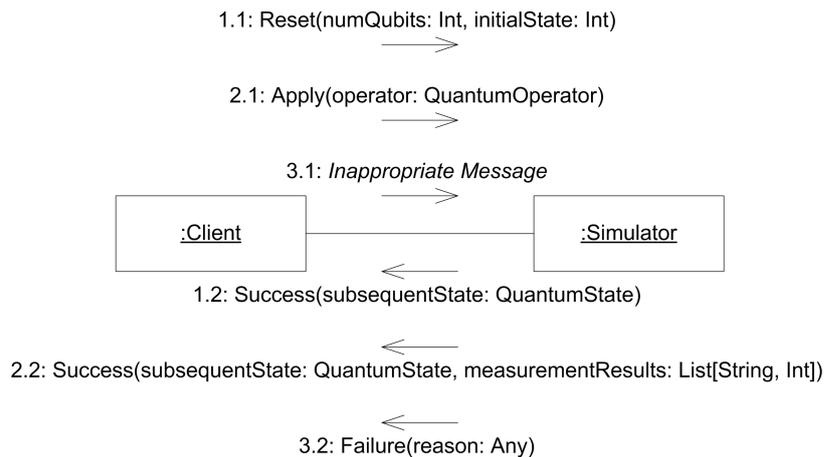


Abbildung 5.5.: UML2 Kommunikationsdiagramm: Simulator Protokoll

Nach dem Start befindet sich der Simulator in einem nicht initialisierten Zustand und muss mittels der `Reset` Nachricht initialisiert werden (`Reset` kann aber auch später jederzeit verwendet werden um den Simulator zurückzusetzen). Danach kann der interne Zustand durch senden der `Apply` Nachricht mit beliebigen Operatoren manipuliert werden. Die Antwort des Simulators (`Success`) enthält immer den Folgezustand und möglicherweise eine Liste von Messergebnissen. Auf fehlerhafte Anfragen reagiert der Simulator mit der `Failure` Nachricht.

### 5.2.3.2. QuantumOperator

Operatoren werden durch das Trait `QuantumOperator` bzw. dessen direkte Ableitungen `UnitaryOperator`, `MeasurementOperator` und `CompositeOperator` repräsentiert. Um einen gewissen Grad an Erweiterbarkeit zu erreichen definiert jeder Operator seine Operation `QuantumState`  $\mapsto$  `QuantumState` selbst, und garantiert korrekte Ausführung. `MeasurementOperator` gibt bei der Anwendung zusätzlich zum Folgezustand Informationen bezüglich des Messergebnisses (`String`, `Int`), d.h. `Tag`  $\rightarrow$  `Messwert` zurück (Messungen können mit einem Tag versehen werden, um für den Benutzer nachvollziehbar angezeigt werden zu können).

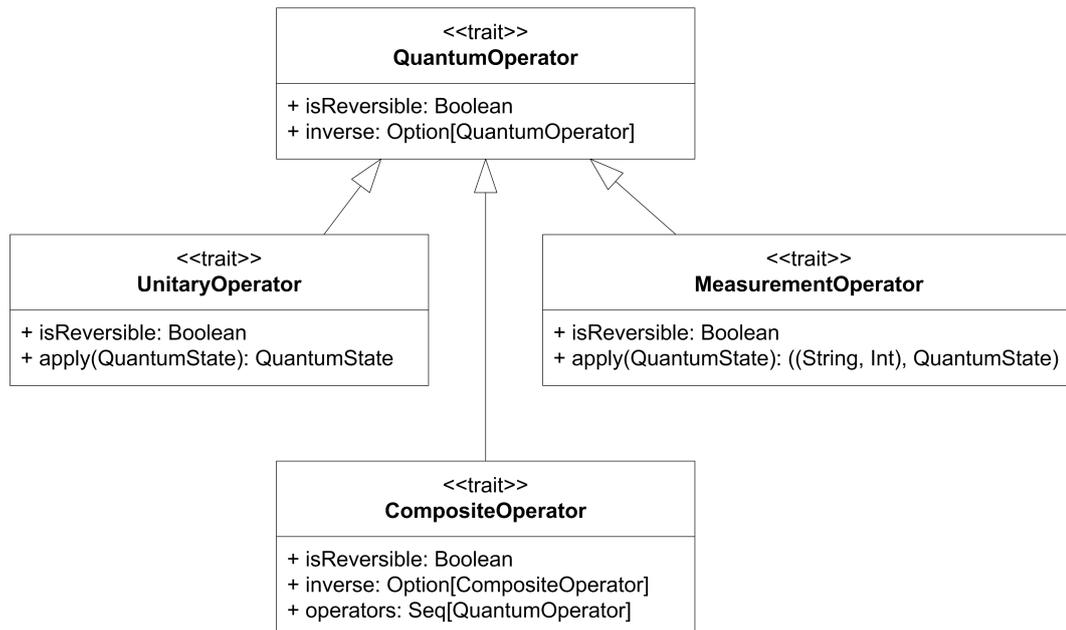


Abbildung 5.6.: UML2 Klassendiagramm: QuantumOperator

### Standard Operatoren

`Gate(reg: Qureg)(gate: QuantumGate, kronPow: Int = 1)`

Transformation eines spezifizierten Teilregisters mittels `QuantumGate`. Der Parameter `kronPow` ermöglicht es Gates so anzuwenden, als würden sie  $n$  Mal mit sich selbst tensorverknüpft. Dabei ist zu beachten, dass `reg` entsprechend viele Bits adressieren muss, also `size(gate) * kronPow`.

`ControlledGate(ctrl: Qureg)(target: Qureg)(gate: QuantumGate, kronPow: Int = 1)`

Gesteuerte Transformation eines spezifizierten Teilregisters mittels `QuantumGate`.

`Oracle(x: Qureg)(y: Qureg)(f: Long => Long)`

Oracle Auswertung auf spezifizierten X- und Y-Teilregistern mit einer gegebenen Funktion.

`StandardMeasurement(reg: Qureg)(tag: String = "")`

Messung eines spezifizierten Teilregisters in der Standardbasis.

`QFT(reg: Qureg) & InverseQFT(reg: Qureg)`

Shortcut Operatoren für den Algorithmus der *Quanten Fourier Transformation* auf einem spezifizierten Teilregister.

`GrooverDiffusion(reg: Qureg)`

Anwendung des Groover Diffusionsoperators (Spiegelung am Mittelwert) auf einem spezifizierten Teilregister.

### 5.2.4. GUI

In den folgenden Abschnitten werden die wichtigsten Entwurfsaspekte der GUI-Anwendung gezeigt und erläutert.

#### 5.2.4.1. Architektur

Abbildung 5.7 zeigt den Aufbau der GUI im Überblick.

#### **Dokumentenverwaltung**

Für die Verwaltung von Dokumenten wurde ein *Single Document Interface* gewählt. Die Anwendung (`Trait DocumentManager`) verwaltet ein oder mehrere offene Dokumente (`Document`) welche jeweils einen Schaltkreis (Das Datenmodell: `Circuit`) sowie ein oder mehrere Fenster (`DocumentFrame`) verwalten. Sämtliche Standardfunktionalität, das Laden und Speichern von Dokumenten, Undo/Redo-Funktionalität sowie die Fensterverwaltung wird komplett vom den drei Modulen `DocumentManager`, `Document` und `DocumentFrame` übernommen.

#### **MVC und Command-Pattern**

Um Datenmodell und Präsentation (View) zu trennen wird das *Model View Controller* Pattern verwendet. Essentiell wichtig ist dabei die Unabhängigkeit des Modells (*Domain-Daten und ~Logik*) vom View, es soll höchstens eine implizite Verbindung vom Modell zum View bestehen (Observer Pattern). Der Controller dient der Kapselung von Präsentationslogik bzw. Verarbeitung von Benutzereingaben. [vgl. GHJ95]

Wie in Abbildung 5.7 zu erkennen ist, existiert nur ein Controller (`SimulationController`). Dieser nimmt Bearbeitungsevents aus der View-Hierarchie, dessen Wurzel `DocumentFrame`

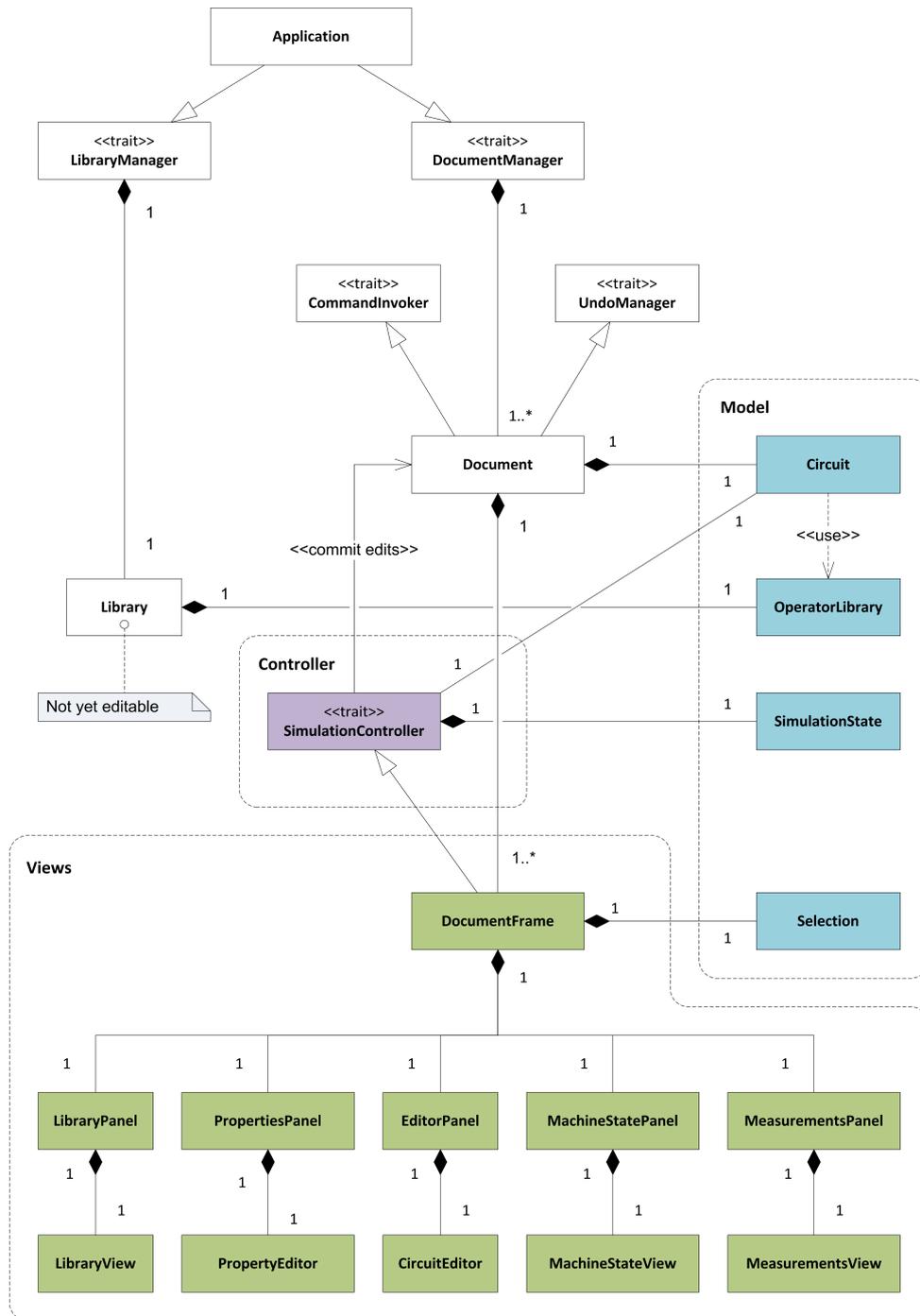
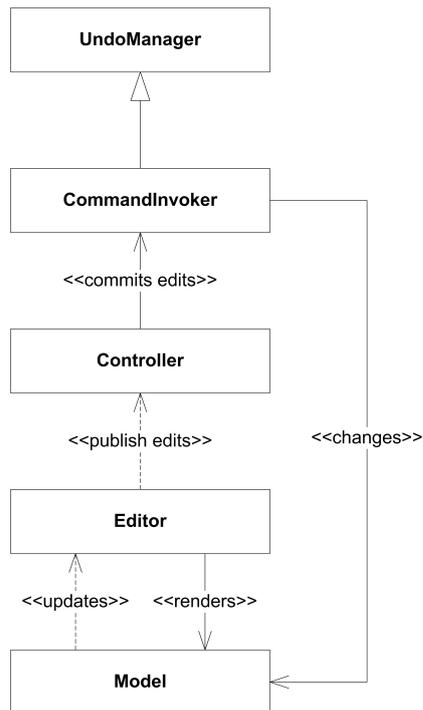


Abbildung 5.7.: UML2 Klassendiagramm: Architektur der GUI

darstellt, entgegen, bildet entsprechende `Command`-Objekte und leitet diese an den `CommandInvoker` (implementiert von `Document`) weiter. Das `Command`-Pattern [GHJ95] wird benutzt um den Undo/Redo-Mechanismus zu realisieren: Der `Controller` selbst verändert das Modell nicht, dafür ist der `CommandInvoker` zuständig, durch Aufruf eines `Commands`. Dieses wird auf den Undo/Redo-Stack gelegt, um später wieder rückgängig gemacht werden zu können. Abbildung 5.8 zeigt das Zusammenspiel von MVC und `Command`-Pattern.



**Abbildung 5.8.:** UML2 Klassendiagramm: MVC und `Command`-Pattern

Das Modell dient hier ausschließlich der Strukturgebung, Anzeige, Bearbeitung und Persistenz eines abstrakten Modells von Quantenschaltkreisen. Sämtliche Simulationslogik wird an die Simulator-Komponente delegiert.

### Event-Modell

Zur Vereinheitlichung sämtlicher Event-Mechanismen, inklusive des Observer Patterns, wird das Publish-subscribe bzw. Publish-react Framework von `scala.swing` verwendet. Da allerdings `scala.swing` nur einen Teil von `java.swing` abdeckt, müssen leider an einigen Stellen die Java typischen `EventListener` verwendet werden.

### 5.2.4.2. Technisches Datenmodell

Es wurde zunächst versucht das Datenmodell funktional, d.h. immutable umzusetzen, dabei hat sich schnell gezeigt dass dies unpraktikabel ist: Modifikationen an den einzelnen Entitäten würden nicht lokale Änderungen in der Hierarchie mit sich bringen <sup>5</sup>, was bei wachsender Komplexität des Modells zu Problemen führt. Daher wird eine mutable Datenstruktur verwendet, welche alle Änderungen hierarchisch mittels Publish-subscribe Mechanismus signalisiert. Abbildung 5.9 zeigt die wichtigsten Module des technischen Datenmodells in einem UML2 Klassendiagramm. Dabei ist zu beachten, dass Getter und Setter in Scala dem *Uniform Access Principle* folgen, sich also beim Aufruf syntaktisch nicht von direktem Zugriff auf Variablen in Java unterscheiden. In dem Diagramm wurden mutable Felder der Entitäten so notiert, als wären es `public` Attribute, dahinter verbergen sich allerdings reguläre Getter und Setter, welche ggf. überschrieben werden um Invarianten zu prüfen und die Observer im Falle von Änderungen zu benachrichtigen. Einige dieser Felder werden mittels Annotation als *Eigenschaften* (`@Property`) gekennzeichnet, dies ermöglicht einen generischen Eigenschaften-Editor im Frontend (siehe 5.2.4.5).

Zur Persistenz des Modells wird ein XML-Format verwendet. Aufgrund von Scalas integrierter Syntax für XML-Literale ist die Erzeugung und Interpretation von XML recht komfortabel realisierbar. Daher wurde auf die Einbindung einer Serialisierungs-Bibliothek verzichtet.

---

<sup>5</sup>Beispiel: Möchte man neue Funktionalität für eine Entität auf der dritten Ebene einbringen, welche das Objekt modifiziert, so muss auf oberster Ebene ebenfalls ein Interface zu diesen Modifikatoren geschaffen werden.

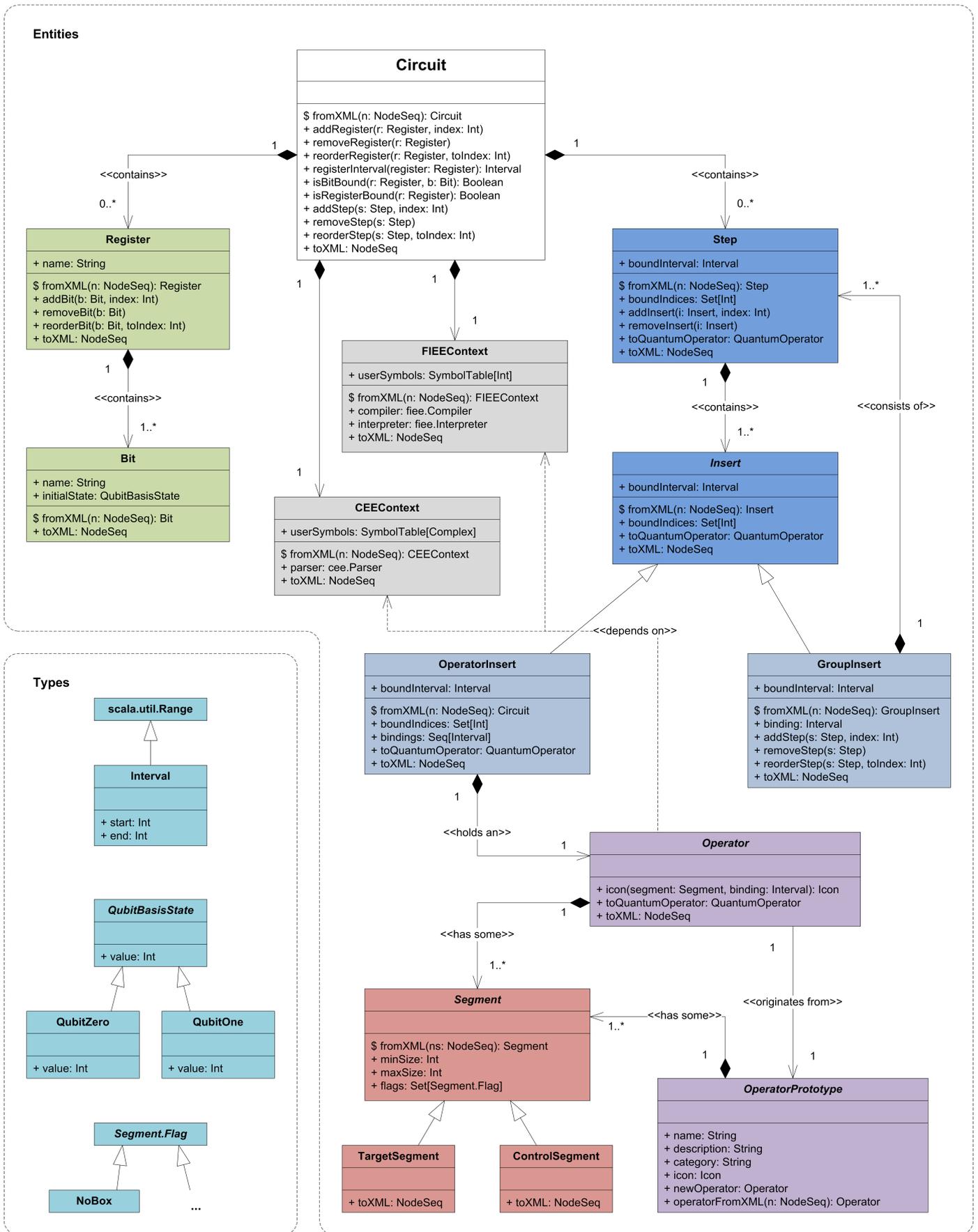


Abbildung 5.9.: UML2 Klassendiagramm: Technisches Datenmodell

### 5.2.4.3. Editor für Quantenschaltkreise

Zentrum der graphischen Oberfläche ist der Editor für Quantenschaltkreise. Um den Anforderungen an Darstellung und Bearbeitungsfunktionalität zu entsprechen, hat sich die Konstruktion eines Szenengraphen mit interaktiven Elementen als äußerst nützlich erwiesen. Der erste Versuch, den Szenengraphen mit `scala.swing.Component` als Knoten zu implementieren, wurde verworfen, da `scala.swing.Component` für diesen Zweck zu wenig Flexibilität bietet <sup>6</sup>. Stattdessen wird eine eigene Struktur verwendet, wie in Abbildung 5.10 visualisiert.

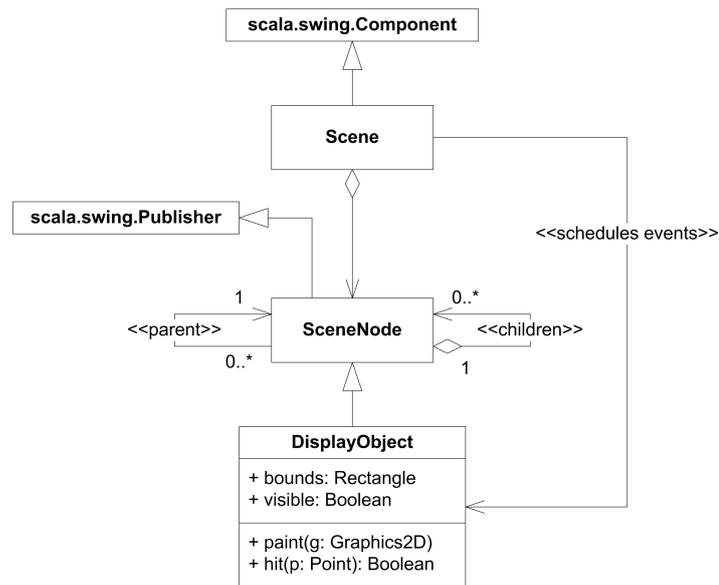


Abbildung 5.10.: UML2 Klassendiagramm: Szenengraph

### 5.2.4.4. Operator-Bibliothek

Ursprünglich war geplant, die Operator-Bibliothek editierbar zu gestalten. Dies wurde allerdings aus Zeitgründen zurückgestellt. Unter *editierbar* ist zu verstehen, dass der Benutzer Gatter als Matrix definiert, und diese in der Bibliothek persistieren kann. Ohne diese Funktionalität können benutzerdefinierte Gatter trotzdem benutzt werden, lediglich die Speicherung als *Templates* ist nicht möglich. Nach dem aktuellen Stand enthält die Bibliothek eine Anzahl vordefinierter Gatter (H, X, Y, Z, S, T,  $R_\theta$ , CNOT, SWAP, TOFFOLI und FREDKIN), frei definierbare ein-, zwei- und drei-Bit Gatter, sowie Messoperator, Oracle, QFT & QFT<sup>†</sup> und den Grover-Diffusionsoperator *D*.

<sup>6</sup>Swing Widgets sind strikt rechteckig (was für die Elemente des Szenengraphen nicht unbedingt gelten soll), die Organisation von Widgets in Ebenen (Z-Order) ist absolut ungeeignet und die Behandlung von Eingabeevents folgt einem recht strikten Schema, welches für die Verwendung in standard GUI-Elementen zugeschnitten ist.

#### 5.2.4.5. Eigenschaften-Editor

Die editierbaren Eigenschaften einzelner Elemente des Modells werden mittels Introspektion in einem generischen Editor zugänglich gemacht. Dazu ist es notwendig einige Metadaten mit den Objekten des Modells zu verknüpfen. Folgende Felder sind vorgesehen:

1. Titel der Eigenschaftenseite
2. Liste der Eigenschaften mit jeweils:
  - a) Titel der Eigenschaft
  - b) Bezeichner für Getter und Setter (zur Introspektion)

Zur Realisierung werden die Annotationen `@Properties` und `@Property` verwendet, Listing 5.3 zeigt exemplarisch wie diese zu verwenden sind.

```
1 @Properties(  
2   title = "Properties of Object X",  
3   properties = Array(  
4     new Property(label = "Field A", name = "fieldA"),  
5     new Property(label = "Field B", name = "fieldB")  
6   )  
7 )  
8 class X {  
9   ...  
10  def fieldA: A = ...  
11  def fieldA_ = (value: A): Unit = ...  
12  def fieldB: B = ...  
13  def fieldB_ = (value: B): Unit = ...  
14  ...  
15 }
```

Listing 5.3: Beispiel: `@Properties`

Die Typen der einzelnen Eigenschaften werden zur Laufzeit durch Introspektion festgestellt (z.B. `String`, `Int`, etc.), woraufhin ein entsprechender Editor angezeigt werden kann. Im *Setter* sollten immer zuerst die Invarianten geprüft werden, daraufhin ggf. der Wert des Feldes geändert und ein `PropertyChange`-Event veröffentlicht werden. Der Editor kann dann auf das `PropertyChange`-Event reagieren, und den angezeigten Wert des Feldes, welcher möglicherweise mit Constraints behaftet ist, aktualisieren.

#### 5.2.4.6. Visualisierung der Simulationsergebnisse

Für die Darstellung der Zustandsvektoren wird ein `JTable` (`scala.Table`) verwendet, mit einer Spalte für die Anzeige der Amplitude (graphische Darstellung wie in Abschnitt 4.5 beschrieben) und je einer weiteren Spalte pro Register, in welcher der entsprechende Basiszustand binär oder dezimal angezeigt wird. Um die Daten des Zustandsvektors nicht zu duplizieren, wird ein eigenes `TableModel` verwendet; dieses greift direkt auf die Elemente des Zustandsvektors zu. Anzeigeoptionen können über ein Menü eingestellt werden.

Neben den Zustandsvektoren, werden die Messergebnisse des letzten Simulationsschritts ebenfalls in einem `JTable`, mit den Spalten 'Tag' und 'Value' dargestellt.

### 5.3. Simulationsalgorithmen

In den folgenden Abschnitten werden die Kernalgorithmen zur Simulation von Quantenschaltkreisen vorgestellt.

#### 5.3.1. Basiszustände und Indexmanipulation

Die Basiszustände eines Zustandsvektors (in der Standardbasis) entsprechen den Zeilenindizes in der Vektorrepräsentation. Beispielsweise entspricht die Binärzahl 0101 dem Basiszustand  $|0101\rangle$ , dessen Amplitude im Zustandsvektor in der fünften Zeile zu finden ist. Um die in Abschnitt 4.1 beschriebene Simulationmethode zu realisieren, werden einige Funktionen zur Manipulation dieser Indizes, auf Basis von spezifizierten Teilregistern (`Qureg`) benötigt. Zunächst sollen die Elemente der Zustandsvektoren während der Multiplikation umsortiert werden. Dies kann erreicht werden, indem die Bits der binären Zeilenindizes entsprechend permutiert werden. Weiterhin ist es notwendig die Indices von Teilregistern aus den globalen Indizes der Basiszustände zu extrahieren, sowie zu kombinieren, beispielsweise um die X- und Y-Werte für ein Oracle zu bestimmen, und anhand der Funktionswertes den Ausgabeindex zu berechnen (siehe 5.3.3).

`Qureg` definiert die folgenden Funktionen um die Indizes der Basiszustände auf Bit-Ebene zu manipulieren:

`mask` : `r : Qureg`  $\mapsto$  `Int`

Erzeugt eine Bitmaske mit den durch `r` indizierten Bits gleich 1.

Beispiel: `mask(Qureg(3, 1)) = 1 0 1 0 b`

- `extract` :  $r : \text{Qureg} \times i : \text{Int} \mapsto \text{Int}$   
Extrahiert die durch  $r$  indizierten Bits der Reihenfolge nach aus dem Integer  $i$ .  
Beispiel: `extract(Qureg(3, 1), 1 0 0 1 b) = 0 0 0 1 b`
- `insert` :  $r : \text{Qureg} \times i : \text{Int} \mapsto \text{Int}$   
*Injiziert* die ersten  $n$  Bits des Integers  $i$  der Reihenfolge nach an die durch  $r$  spezifizierten Indices.  
Beispiel: `insert(Qureg(3, 1), 0 0 0 1 b) = 1 0 0 0 b`
- `shuffle` :  $r : \text{Qureg} \times i : \text{Int} \mapsto \text{Int}$   
Permutiert die Bits des Integers  $i$ , so dass die  $n$  durch  $r$  indizierten Bits der Reihenfolge nach in den ersten  $n$  Bits des Rückgabewertes zu finden sind, während die restlichen Bits verdrängt werden.  
Beispiel: `shuffle(Qureg(3, 1), 1 0 0 1 b) = 0 1 0 1 b`
- `unshuffle` :  $r : \text{Qureg} \times i : \text{Int} \mapsto \text{Int}$   
Permutiert die Bits des Integers  $i$ , so dass die ersten  $n$  Bits an den durch  $r$  spezifizierten Indizes des Rückgabewertes zu finden sind, während die restlichen Bits verdrängt werden.  
Beispiel: `unshuffle(Qureg(3, 1), 0 1 0 1 b) = 1 0 0 1 b`
- `reverse` :  $r : \text{Qureg} \times i : \text{Int} \mapsto \text{Int}$   
Permutiert die Bits des Integers  $i$ , so dass die  $n$  durch  $r$  indizierten Bits in umgekehrter Reihenfolge im Rückgabewert auftauchen.  
Beispiel: `reverse(Qureg(3, 1), 1 0 0 1 b) = 0 0 1 1 b`

### 5.3.2. Transformation von Teilregistern

Die Transformation von Quantenzuständen (`QuantumState`) mit Quantengattern (`QuantumGate`) wird wie in Abschnitt 4.1.2 beschrieben realisiert. Ausgangspunkt ist ein relativ unkomplizierter, optimierter Algorithmus zur Multiplikation einer blockdiagonalen Matrix, deren Blöcke alle identisch sind, mit einem Vektor. Um bestimmte Teilregister (`Qureg`) zu adressieren, wird der Zustandsvektor permutiert. Listing 5.4 beschreibt den Algorithmus in Pseudocode.

```

1 Function
2   transform(s, g, r) ↦ s'
3
4 Parameters
5   s ∈  $\mathbb{C}^{2^n}$                                 ▷ Original State vector
6   g ∈  $\mathbb{C}^{2^m \times 2^m}$                           ▷ Gate matrix,  $m \leq n$ 
7   reg ∈ Qureg                                    ▷ Subregister specification
8
9 Result
10  s' ∈  $\mathbb{C}^{2^n}$                                 ▷ Transformed State vector
11
12 Algorithm
13  for i ← [0, 2n) do
14    k ← shuffle(reg, i)                          ▷ Permute row index
15    mbase ← k mod 2m                            ▷ Base row on the matrix
16    vbase ← k - mbase                            ▷ Base row on the vector
17    sum ← 0                                         ▷ Accumulator
18    for j ← [0, 2m) do
19      vrow ← unshuffle(reg, vbase + j)          ▷ Final row index on the vector
20      a ← gmbase,j                            ▷ Matrix element
21      b ← svrow                                ▷ Vector element
22      sum ← sum + a · b                        ▷ Accumulate dot product
23    end for
24    s'i ← sum
25  end for

```

Listing 5.4: Pseudocode: Transformation von Teilregistern

Die äußere Schleife (Ansatzpunkt für die Parallelisierung) iteriert über alle  $2^n$  Element des Eingavektors  $s$ , während in der inneren Schleife über jeweils einen Teil des Vektors, und eine entsprechende Zeile der Matrix  $g$  mit je  $2^m$  Elementen das Skalarprodukt gebildet wird. Es ist an dieser Stelle weiterhin möglich das die Implementierungen von Vector und Matrix für dünnbesetzte Vektoren und Matrizen optimiert sind, Zeile 22 muss (darf) also nur ausgeführt werden wenn  $a$  und  $b$  nicht null sind, dies wurde der Einfachheit halber im Pseudocode weggelassen.

In Zeile 14 wird zunächst der Laufindex  $i$  gemäß der Methode aus Abschnitt 4.1.2 permutiert. Auf Basis dieses Indizes wird ein Block des Vektors, sowie eine Zeile in der Matrix gewählt. Um korrekt über die Elemente des Blocks zu iterieren, werden die Zeilenindizes  $vbase + j$  in Zeile 19 jeweils wieder in die richtige Reihenfolge "zurück permutiert".

## 5. Entwurf

Der Algorithmus kann einfach angepasst werden, um gesteuerte Gatter effizienter zu realisieren, wie in 4.1 beschrieben. Listing 5.5 zeigt den modifizierten Algorithmus.

```

1 Function
2   transform( $s, g, c, reg$ )  $\mapsto s'$ 
3
4 Parameters
5    $s \in \mathbb{C}^{2^n}$  ▷ Original State vector
6    $g \in \mathbb{C}^{2^m \times 2^m}$  ▷ Gate matrix,  $m \leq n$ 
7    $c \in \mathbb{N}$  ▷ Number of controll bits
8    $reg \in \text{Qureg}$  ▷ Subregister specification
9
10 Result
11    $s' \in \mathbb{C}^{2^n}$  ▷ Transformed State vector
12
13 Algorithm
14   for  $i \leftarrow [0, 2^n)$  do
15      $k \leftarrow \text{shuffle}(reg, i)$  ▷ Permute row index
16      $mbase \leftarrow k \bmod (2^m)^c$  ▷ Base row on the controlled matrix
17     if  $mbase < 2^{m+c} - 2^m$  then ▷ Case 1: controlled/unaffected element
18        $s'_i \leftarrow s_i$ 
19     else ▷ Case 2: partial dot product on non-zero block
20        $msub \leftarrow mbase - 2^c$  ▷ Row in the gate matrix
21        $vbase \leftarrow k - msub$  ▷ Base index on the state vector
22        $sum \leftarrow 0$  ▷ Accumulator
23       for  $j \leftarrow [0, 2^m)$  do
24          $vrow \leftarrow \text{unshuffle}(reg, vbase + j)$  ▷ Final row index on the vector
25          $a \leftarrow g_{msub,j}$  ▷ Matrix element
26          $b \leftarrow s_{vrow}$  ▷ Vector element
27          $sum \leftarrow sum + a \cdot b$  ▷ Accumulate dot product
28       end for
29        $s'_i \leftarrow sum$ 
30     end if
31   end for

```

**Listing 5.5:** Pseudocode: Gesteuerte Transformation von Teilregistern

### 5.3.3. Auswertung von Oracle-Funktionen

Dieser Algorithmus permutiert einen Zustandsvektor anhand der Transformation  $|x\rangle|y\rangle \xrightarrow{U_f} |x\rangle|f(x) \oplus y\rangle$ , wie in Abschnitt 2.2.3 beschrieben. Listing 5.6 zeigt den Algorithmus in Pseudocode. Für jeden Zeilenindex werden die binären Indizes  $x$  und  $y$  der X- und Y-Teilregister mittels `extract` extrahiert, woraufhin der Ausgabeindex  $i'$  nach der Auswertung von  $f(x)$  durch bitweise Logikoperationen<sup>7</sup> gebildet werden kann. Die äußere Schleife kann parallelisiert werden.

```

1 Function
2   evaluate( $s, f, xreg, yreg$ )  $\mapsto s'$ 
3
4 Parameters
5    $s \in \mathbb{C}^{2^n}$                                  $\triangleright$  Original state vector
6    $f \in \mathbb{N} \mapsto \mathbb{N}$                          $\triangleright$  Oracle function
7    $xreg \in \text{Qureg}$                               $\triangleright$  X-register specification
8    $yreg \in \text{Qureg}$                               $\triangleright$  Y-register specification
9
10 Result
11    $s' \in \mathbb{C}^{2^n}$                              $\triangleright$  Transformed State vector
12
13 Algorithm
14   for  $i \leftarrow [0, 2^n)$  do
15      $x \leftarrow \text{extract}(xreg, i)$               $\triangleright$  Get x bit pattern
16      $y \leftarrow \text{extract}(yreg, i)$               $\triangleright$  Get y bit pattern
17      $y' \leftarrow f(x) \otimes y$                   $\triangleright$  Apply oracle, xor with y-bits
18      $i' \leftarrow \text{bitor}(\text{bitand}(i, \text{invert}(\text{mask}(yreg))), \text{insert}(yreg, y'))$   $\triangleright$  Output index
19      $s'_i \leftarrow s_{i'}$ 
20   end

```

**Listing 5.6:** Pseudocode: Auswertung von Oracle-Funktionen

<sup>7</sup>`bitand(a, b)`: Bitweise AND Operation  
`bitor(a, b)`: Bitweise OR Operation  
`invert(x)`: Einerkomplement

### 5.3.4. Messung

Die Messung wird in zwei Schritten realisiert: Zunächst wird, unter Beachtung der Wahrscheinlichkeitsverteilung, ein zufälliger Basiszustand  $i$  gewählt, siehe Listing 5.7. Im zweiten Schritt wird mittels `extract` der Basiszustand des Teilregisters, welches gemessen werden soll, extrahiert - dies ist das binäre Messergebnis - sowie darauf basieren der Folgezustand berechnet, siehe Listing 5.8.

#### Teil 1

Erzeuge eine Zufallszahl  $x$  zwischen 0 und 1, ziehe dann so lange die reellen Wahrscheinlichkeitsamplituden  $p_i$  der einzelnen Basiszustände ab, bis das Vorzeichen von  $x$  wechselt. Der letzte Wert von  $i$  ist ein, der Wahrscheinlichkeitsverteilung von  $s$  entsprechender, zufällig gewählter Basiszustand, bezüglich der Standardbasis.

```

1 Function
2   sample( $s$ )  $\mapsto$   $b$ 
3
4 Parameters
5    $s \in \mathbb{C}^{2^n}$  ▷ State vector
6
7 Result
8    $b \in \mathbb{N}$  ▷ Index of sampled basis state
9
10 Algorithm
11    $x \leftarrow \text{random}([0, 1])$  ▷ Accumulator
12   for  $i \leftarrow [0, 2^n)$  do
13      $w_i \leftarrow s(i)$  ▷ Complex amplitude
14      $p_i \leftarrow w_i \cdot w_i^*$  ▷ Real probability
15      $x \leftarrow x - p_i$  ▷ Subtract from accumulator
16     if  $x < 0$  then
17        $b \leftarrow i$  ▷ This is it
18     return
19   end
20 end
21 error ▷ State vector not properly normalized

```

**Listing 5.7:** Pseudocode: Auswahl eines zufälligen Basiszustandes

**Teil 2**

Der in Teil 1 gewählte Basiszustand bezieht sich zunächst auf den Gesamtzustand des Systems. Um eine Teilmessung durchzuführen werden nun die entsprechenden Bits extrahiert, dazu wird `extract` verwendet. Da in der Standardbasis gemessen wird, kann die Projektion in den Unterraum eines Teilregisters durch Auslöschung der hierzu orthogonalen Amplituden, sowie Normierung des reduzierten Zustandsvektors realisiert werden.

```

1 Function
2  measure(s, reg) ↦ (b, s')
3
4 Parameters
5  s ∈ ℂ2n                                ▷ State vector
6  reg ∈ Qureg                                ▷ Subregister specification
7
8 Result
9  b ∈ ℕ                                       ▷ Index of measured basis state
10 s' ∈ ℂ2n                                   ▷ Subsequent state vector
11
12 Algorithm
13 b ← sample(s)
14 for i ← [0, 2n) do
15   if bitand(i, mask(reg)) = bitand(b, mask(reg)) then
16     s'_i ← s_i                                ▷ Include non-orthogonal basis state
17   else
18     s'_i ← 0                                    ▷ Mute orthogonal basis state
19   end if
20 end for
21 s' ← s' / √⟨s', s'⟩                            ▷ Normalize

```

**Listing 5.8:** Pseudocode: Messung von Teilregistern

Da die tatsächliche Berechnung von  $2^n$  Skalarprodukten des Zustandsvektors mit einer beliebigen Orthonormalbasis zu ineffizient ist, wird hier nur die Messung in der Standardbasis realisiert. Jedoch kann eine Messung in beliebiger Basis trotzdem durchgeführt werden, nämlich indem der Zustand vor der Messung in die gewünschte Basis transformiert, und nach der Messung wieder zurücktransformiert wird.

**5.3.5. QFT**

Die QFT, sowie die inverse QFT, werden durch programmiertechnische Anwendung grundlegender Gatter, wie in Abbildung 2.12 realisiert, und als "Macro-Operatoren" definiert.

### 5.3.6. Grover Diffusionsoperator

Der Grover Diffusions Operator zur Spiegelung eines Quantenregisters an dessen Mittelwert,

$$D = \frac{2}{n} \sum_{i,j} |i\rangle\langle j| - I = H^n(|0\rangle\langle 0| - I)H^n, \quad (5.1)$$

kann mittels einer Kombination aus Hadamard-, Not- und einem gesteuerten Phasengatter realisiert werden:

$$-D = H^n(X^n C^n(Z) X^n) H^n, \quad Z \equiv R_\pi, \quad (5.2)$$

wobei  $C^n(Z)$  ein  $n$ -fach gesteuertes  $Z$ -Gatter darstellt ( $-D \equiv D$  in Bezug auf die globale Phase) ([Öme12b](#)). Der Übersichtlichkeit halber wird  $D$  als "Macro-Operator" definiert.

## 6. Realisierung und Test

Dieses Kapitel behandelt einige Aspekte der Implementierung. Zunächst wird ein kurzer Überblick über den aktuellen Implementierungsstand gegeben (6.1). Daraufhin folgt eine Erläuterung der durchgeführten Tests (6.2), sowie die Ergebnisse einer Performanceanalyse (6.3).

### 6.1. Umfang der Implementierung

Es konnten weitestgehend alle formulierten Anforderungen implementiert werden. Da allerdings von Anfang mit eingeplant wurde, dass, aufgrund der Komplexität der Software und des begrenzten Zeitrahmens, einige Anforderungen möglicherweise nicht mehr umgesetzt werden können, wurde die Implementierung einiger sekundär priorisierter Features aufgeschoben: [REQ F 2.2], die Möglichkeit Unterprogramme zu bilden, und [REQ F 4.2], ein Plugin-Mechanismus für die Operator-Bibliothek. Weiterhin wäre es für den Benutzer praktisch, wenn selbstdefinierte Gatter als Templates in der Bibliothek gespeichert werden könnten (nicht explizit in den Anforderungen definiert). Diese Features werden als offene Punkte in den Ausblick aufgenommen (Abschnitt 8.2).

Sourcecode und Binaries befinden auf der CD im Anhang, sind aber auch auf <http://sourceforge.net/projects/simuquant/> unter GPL 3+ Lizenz verfügbar.

### 6.2. Test

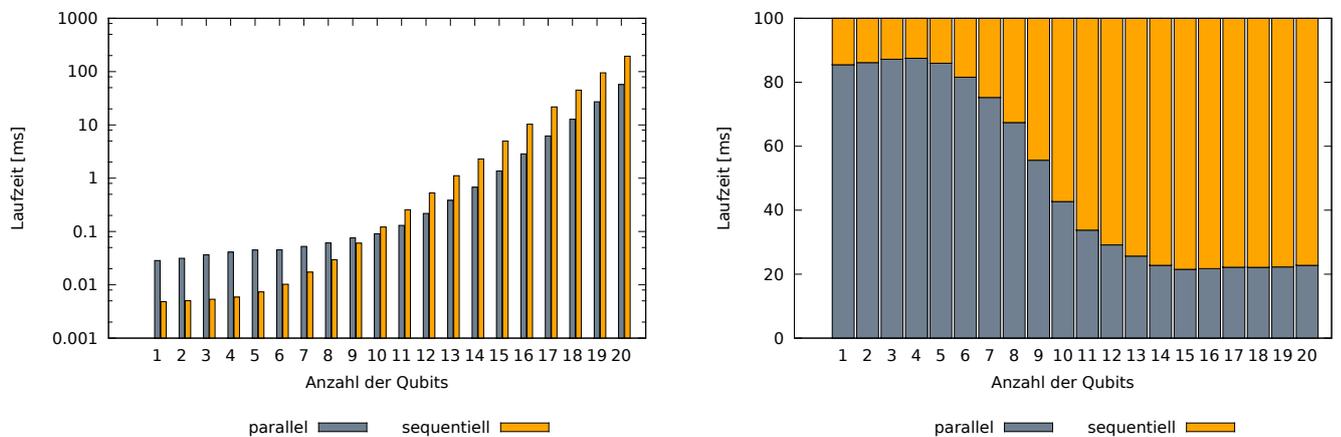
Um die Korrektheit der Implementierung zu überprüfen werden die Module der Komponenten *Core* und *Simulator*, Modul-Tests unterzogen, dazu wird `scalatest` in Verbindung mit `scalacheck` als Framework verwendet. In vielen Fällen werden wiederholte Tests mit zufällig erzeugten Eingabedaten gemacht um eine möglichst breit gestreute Menge allgemeiner Fälle zu verifizieren. Die Eingabedaten werden hierbei auf Basis einer Zufallssequenz so gewählt, dass immer voraussagbar ist, welches Ergebnis erwartet wird. Neben den randomisierten Tests werden bestimmte Fälle zusätzlich manuell getestet. Die Benutzeroberfläche hat sich erfahrungsgemäß als ausreichend zuverlässig erwiesen, daher wurden keine aufwändigen GUI-Tests gemacht.

### 6.3. Performanceanalyse

Neben funktionalen Tests wurden einige Zeitmessungen durchgeführt, um die Wirksamkeit des Optimierungsansatzes zu überprüfen. Nachfolgend werden drei verschiedene Messreihen vorgestellt. Es wird die Java 1.6 Laufzeitumgebung, sowie die *Java HotSpot™ VM (build 20.7-b02, mixed mode)* (die VM wird jeweils vorher mit den entsprechenden Funktionen warmgelaufen), auf einem Thinkpad w520, mit Intel® Core™ i7-2760QM Quadcore Prozessor (2.4 - 3.5 GHz) verwendet. Die Heap-Begrenzung wird ausreichend hoch angesetzt (2 GB), der Verlauf der Speicherauslastung wird nicht ausgewertet. Jeder Test wird an verschiedenen Registergrößen, bis 20 Qubit, durchgeführt, die Zustandsvektoren werden dabei mit randomisierten Amplituden (voll besetzt) erzeugt. Gemessen wird nur die jeweilige Transformation, es wird jeweils für 100 Iterationen der Mittelwert berechnet. Die selben Testfälle, ebenfalls mit randomisierten Daten, finden sich in den Modul-Tests wieder, so dass die Korrektheit der Berechnungen sichergestellt ist.

#### 1. Messreihe

Der erste Test soll herausstellen, inwieweit die Transformation vollbesetzter Zustandsvektoren mit einem einfachen Gatter (H) auf Mehrkern-Systemen skaliert. Dazu wird die Parallelisierung für einen Durchlauf ausgeschaltet, der Algorithmus bleibt der selbe. Abbildung 6.1 zeigt die Ergebnisse der ersten Messreihe in zwei verschiedenen Skalierungsmodi.



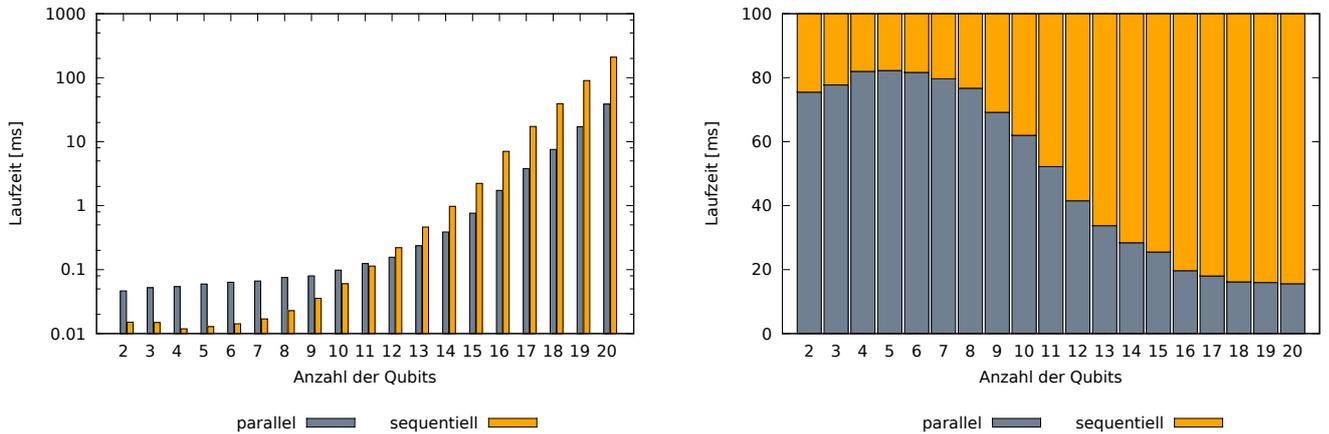
(a) Laufzeitvergleich: Parallele und sequentielle Anwendung des H-Gatters, *logarithmische* Skalierung

(b) Laufzeitvergleich: Parallele und sequentielle Anwendung des H-Gatters, *prozentuale* Skalierung

**Abbildung 6.1.:** Performanceanalyse: Messreihe 1

## 2. Messreihe

Im zweiten Test wird die Implementierung von *Oracle*-Funktionen, analog zum ersten Test, auf ihre Parallelisierbarkeit hin überprüft. Abbildung 6.2 zeigt die Ergebnisse der zweiten Messreihe in zwei verschiedenen Skalierungsmodi.



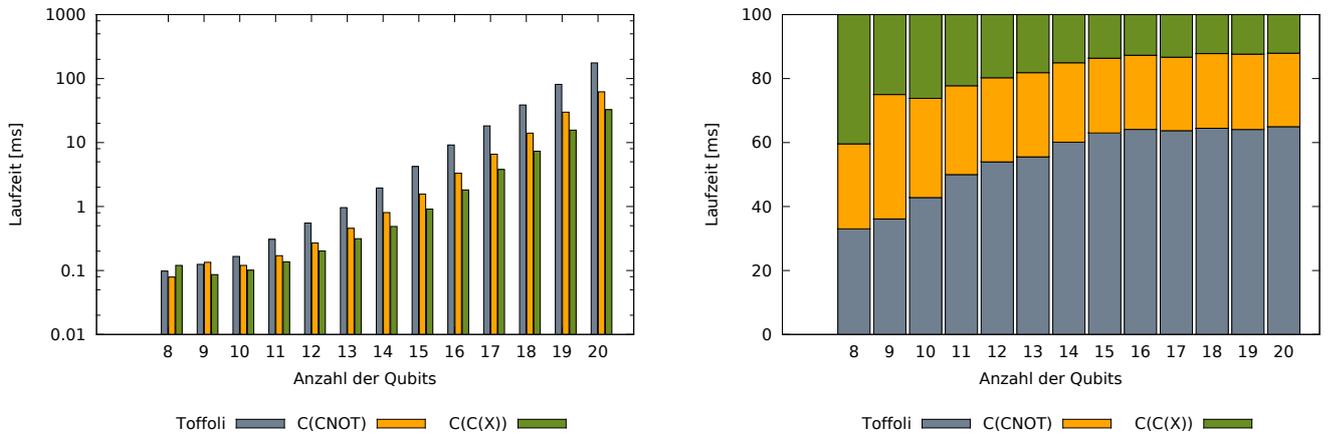
(a) Laufzeitvergleich: Parallele und sequentielle Anwendung eines Oracles, *logarithmische* Skalierung

(b) Laufzeitvergleich: Parallele und sequentielle Anwendung eines Oracles, *prozentuale* Skalierung

Abbildung 6.2.: Performanceanalyse: Messreihe 2

## 3. Messreihe

Der dritte Test unterscheidet sich von den ersten beiden insofern, als dass hierbei nicht parallele gegen sequentielle Implementierung gestellt wird. Diese Messreihe soll zeigen, dass die Implementierung gesteuerter Gatter mit dem in 5.3 vorgestellten Algorithmus effizienter ist, als die einfache Transformation mit einer entsprechenden Matrix. Abbildung 6.3 zeigt die Ergebnisse des dritten Tests in zwei verschiedenen Skalierungsmodi.



(a) Laufzeitvergleich: Toffoli, C(CNOT) und C(C(X)), logarithmische Skalierung

(b) Laufzeitvergleich: Toffoli, C(CNOT) und C(C(X)), prozentuale Skalierung

Abbildung 6.3.: Performanceanalyse: Messreihe 3

### Bewertung

Die erste Messreihe zeigt deutlich, dass die parallelisierte Implementierung um einen konstanten Faktor, proportional zur Anzahl der Prozessorkerne, effizienter ist als eine sequentielle (siehe Abschnitt 5.3.2). Bei größeren Registern, über 10 Qubits, nähert sich der Laufzeitunterschied einem Faktor von 4 an, während sich im Bereich unter 10 Qubit der Overhead für die Parallelisierung deutlich abzeichnet. Die zweite Messreihe zeigt bei der Auswertung von *Oracle*-Funktionen einen asymptotisch noch größeren Laufzeitunterschied, dies liegt vermutlich daran, dass dieser Algorithmus (siehe Abschnitt 5.3.3) die 8 *Hyperthreads* des Systems besser auslasten kann. Bei der dritten Messreihe zeigen sich, wie vermutet, in der Anzahl der Steuerbits exponentielle Laufzeitunterschiede, da die Größe der Gatter-Matrizen sich mit jedem "virtuellen" Steuerbit halbiert (siehe Abschnitt 4.1.2 und 5.3.2).

## 7. Beispielhafte Anwendung und Verifikation

Um die Funktionalität der Software zu demonstrieren, sowie zu verifizieren, werden in den folgenden Abschnitten die Quantenalgorithmen aus Abschnitt 2.3 beispielhaft mit dem Simulator abgearbeitet, während die Ergebnisse der einzelnen Simulationsschritte mit den postulierten verglichen werden. Es werden Screenshots von den Anzeigen der GUI, insbesondere des *Machine State* Panels aufgeführt.

### 7.1. Deutsch-Algorithmus

In diesem Abschnitt wird der Deutsch-Algorithmus, in Bezug auf Abschnitt 2.3.1, mit dem Simulator Schritt für Schritt nachvollzogen. Abbildung 7.1 zeigt den Schaltkreis im Editor.

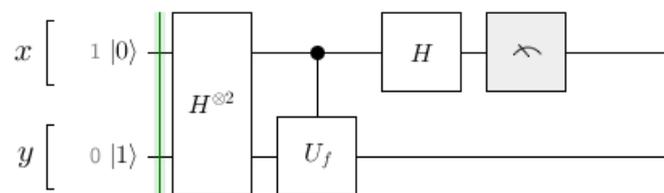


Abbildung 7.1.: Verifikation: Deutsch-Algorithmus, Schaltkreis

**Schritt 1 und 2**

Zunächst werden die zwei Register  $|x\rangle|y\rangle$  mit  $|0\rangle|1\rangle$  initialisiert. Im zweiten Schritt wird das Hadamard Gatter benutzt um  $|x\rangle|y\rangle$  in eine Superposition zu versetzen. Tabelle 7.1 stellt jeweils die Anzeige der Folgezustände  $|\psi_0\rangle$  (Gleichung 2.25) und  $|\psi_1\rangle$  (Gleichung 2.26) dar.

Schritt 1			Schritt 2		
$ \psi_0\rangle$			$ \psi_1\rangle$		
Amplitude	x	y	Amplitude	x	y
	0	0		0	0
	0	1		0	1
	1	0		1	0
	1	1		1	1

Tabelle 7.1.: Verifikation: Deutsch-Algorithmus, Schritt 3: Folgezustände

**Schritt 3**

Der dritte Schritt beinhaltet die Anwendung des Oracles, ab hier gibt es jeweils vier mögliche Resultate, abhängig von der Wahl der Funktion  $f$ . Tabelle 7.2 zeigt den Folgezustand in Zusammenhang mit der gewählten Funktion, dieser entspricht  $|\psi_2\rangle$  wie in Gleichung 2.27.

Fall a	Fall b	Fall c	Fall d
$f(x) = 0$	$f(x) = 1$	$f(x) = x$	$f(x) = x \otimes 1$
Amplitude	Amplitude	Amplitude	Amplitude
x	x	x	x
y	y	y	y
0	0	0	0
0	0	0	0
0	0	0	0
1	0	1	1
1	0	0	0
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

Tabelle 7.2.: Verifikation: Deutsch-Algorithmus, Schritt 3: Folgezustände

**Schritt 4**

Nach erneuter Anwendung der Hadamard-Transformation auf  $|x\rangle$  befindet sich das Register im Zustand  $|\psi_3\rangle$  aus Gleichung 2.29. Die Zustandsanzeige bestätigt dies, siehe Tabelle 7.3.

Fall a			Fall b			Fall c			Fall d		
$f(x) = 0$			$f(x) = 1$			$f(x) = x$			$f(x) = x \otimes 1$		
Amplitude	x	y	Amplitude	x	y	Amplitude	x	y	Amplitude	x	y
	0	0		0	0		0	0		0	0
	0	1		0	1		0	1		0	1
	1	0		1	0		1	0		1	0
	1	1		1	1		1	1		1	1

**Tabelle 7.3.:** Verifikation: Deutsch-Algorithmus, Schritt 4: Folgezustände

**Schritt 5**

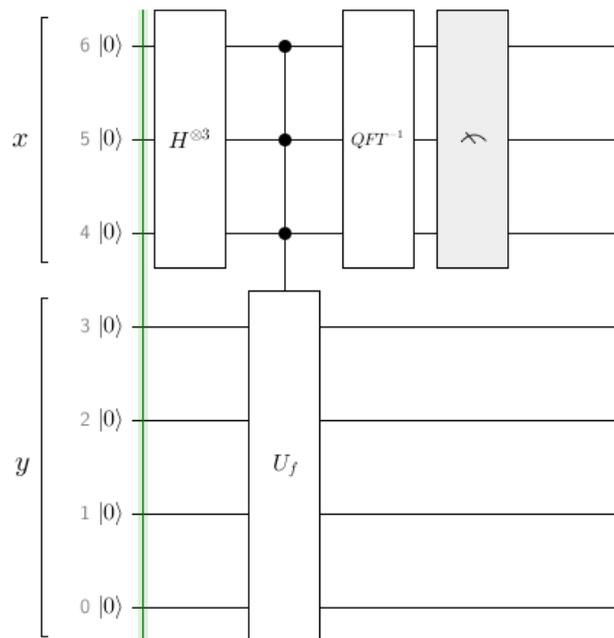
Die Messung von  $|x\rangle$  am Ende des Schaltkreises verändert den Zustandsvektor nicht, da die gesamte Messwahrscheinlichkeit auf einem der Basiszustände liegt. Das Messergebnis gibt allerdings Aufschluss über die Beschaffenheit der Funktion  $f$ , und lässt sich auch beschreiben als  $f(0) \otimes f(1)$ . Tabelle 7.4 zeigt die Ansicht der Messergebnisse, zusammen mit den jeweiligen Funktionen.

Fall a und b			Fall c und d		
$f(x) = 0, \quad f(x) = 1$			$f(x) = x, \quad f(x) = x \otimes 1$		
Tag	Value (Dec)	Value (Bin)	Tag	Value (Dec)	Value (Bin)
M	0	0	M	1	1

**Tabelle 7.4.:** Verifikation: Deutsch-Algorithmus, Schritt 3: Messergebnis

## 7.2. Shor-Algorithmus

Im Folgenden wird der Shor-Algorithmus, in Bezug auf Abschnitt 2.3.2, analog zu dem 2000 bei IBM durchgeführten NMR-Experiment [VSB<sup>+</sup>01](#) simuliert. Ziel ist es, die Zahl  $N = 15$  in die beiden Primfaktoren  $n_1 = 3$  und  $n_2 = 5$  zu zerlegen. Es wird ein  $a$  zwischen 1 und 15 gewählt, wobei  $a$  teilerfremd zu  $N$  sein muss; daher gibt es nur zwei Möglichkeiten:  $a = 7$  oder  $a = 11$ . Für diese Simulation soll  $a = 7$  verwendet werden. Als nächstes gilt es die Periode  $r$  der Funktion  $a^x \bmod N$  zu bestimmen, dazu wird der Quantenschaltkreis wie in Abbildung 7.2 verwendet.



**Abbildung 7.2.:** Verifikation: Shor-Algorithmus, Schaltkreis. Die Funktion  $f$  ist festgelegt als  $\text{mod}(\text{pow}(7, x), 15)$ .

Die Folgezustände der einzelnen Schritte, beginnend mit Schritt 1, der Initialisierung, werden in Tabelle 7.5 gezeigt. Nach Schritt 4 sind bereits die möglichen Messergebnisse für  $|x\rangle$  zu erkennen, dies sind  $|0\rangle$ ,  $|2\rangle$ ,  $|4\rangle$  und  $|6\rangle$ .

7. Beispielhafte Anwendung und Verifikation

Schritt 1 Initialisierung			Schritt 2 $H( x\rangle)$			Schritt 3 Oracle			Schritt 4 QFT		
Amplitude	x	y	Amplitude	x	y	Amplitude	x	y	Amplitude	x	y
	0	0		0	0		0	1		0	1
				1	0		1	7		0	4
				2	0		2	4		0	7
				3	0		3	13		0	13
				4	0		4	1		2	1
				5	0		5	7		2	4
				6	0		6	4		2	7
				7	0		7	13		2	13
										4	1
										4	4
										4	7
										4	13
										6	1
										6	4
										6	7
										6	13

Tabelle 7.5.: Verifikation: Shor-Algorithmus, Schritt 1 - 4: Folgezustände

Fall a			Fall b			Fall c			Fall d		
$ x\rangle =  0\rangle$			$ x\rangle =  2\rangle$			$ x\rangle =  4\rangle$			$ x\rangle =  6\rangle$		
Amplitude	x	y	Amplitude	x	y	Amplitude	x	y	Amplitude	x	y
	0	1		2	1		4	1		6	1
	0	4		2	4		4	4		6	4
	0	7		2	7		4	7		6	7
	0	13		2	13		4	13		6	13

**Tabelle 7.6.:** Verifikation: Shor-Algorithmus, Schritt 5: Vier mögliche Messergebnisse

Tabelle 7.6 zeigt die vier möglichen Folgezustände nach der Messung von  $|x\rangle$ . Das Messergebnis lässt sich als  $\lambda \frac{2^t}{r}$ , mit  $\lambda \in \{0, 1, 2, 3\}$  und  $t = 3$  formulieren:

$$\text{Fall a : } 0 \cdot 8/4 = 0$$

$$\text{Fall b : } 1 \cdot 8/4 = 2$$

$$\text{Fall c : } 2 \cdot 8/4 = 4$$

$$\text{Fall d : } 3 \cdot 8/4 = 6.$$

Wenn  $\lambda$  und  $r$  teilerfremd sind, dies ist für  $\lambda = 1$  und  $\lambda = 3$  der Fall, kann mit der Kettenbruchzerlegung die Periode  $r = 4$  identifiziert werden. War die Bestimmung der Periode erfolgreich, ist es möglich mit dem euklidischen Algorithmus zwei echte Teiler von  $N = 15$  zu berechnen:

$$n_1 = ggT(7^{4/2} - 1, 15) = 3$$

$$n_2 = ggT(7^{4/2} + 1, 15) = 5.$$

### Overflow bei der Auswertung des Oracles

Bei der Auswertung der Funktion des *Oracles* in der Form  $\text{mod}(\text{pow}(a, x), N)$  kann es zu einem *Integer-Overflow* kommen wenn  $a$  und  $t$  zu groß gewählt werden, so dass 64 Bit nicht mehr ausreichen um die Potenz  $a^x$  darzustellen. In manchen Fällen kann das Problem durch eine andere Wahl von  $a$  und/oder  $t$  vermieden werden, als generische Lösung wurde jedoch die *diskrete Exponentialfunktion* (engl. modular exponentiation, siehe NC00, S. 228) mit einer entsprechend optimierten Implementierung (wie in Sch95 beschrieben) als Standard-Funktion in der *Expression Engine* (siehe Abschnitt 5.2.2.3) definiert:

$$\text{mexp} : (\mathbf{b} \times \mathbf{e} \times \mathbf{m}) \mapsto \mathbf{b}^{\mathbf{e}} \text{ mod } \mathbf{m}.$$

### 7.3. Grover-Algorithmus

Um die Simulation des Grover-Algorithmus anschaulich zu präsentieren, wird eine Problemgröße von  $N = 8$  gewählt, gesucht wird das Element  $k = 2$ . Um den Suchraum darzustellen wird ein Quantenregister  $X$  mit drei Qubits verwendet, weiterhin benötigt das Oracle ein weiteres Qubit  $|aux\rangle$ , welches mit  $|1\rangle$  initialisiert wird. Entsprechend der Problemgröße sind  $m = (\pi\sqrt{N})/4 \approx 2,2214\dots$  Iterationen nötig um mit maximaler Wahrscheinlichkeit das korrekte Ergebnis zu erhalten; da keine halben Iterationen möglich sind, muss  $m$  auf 2 gerundet werden.

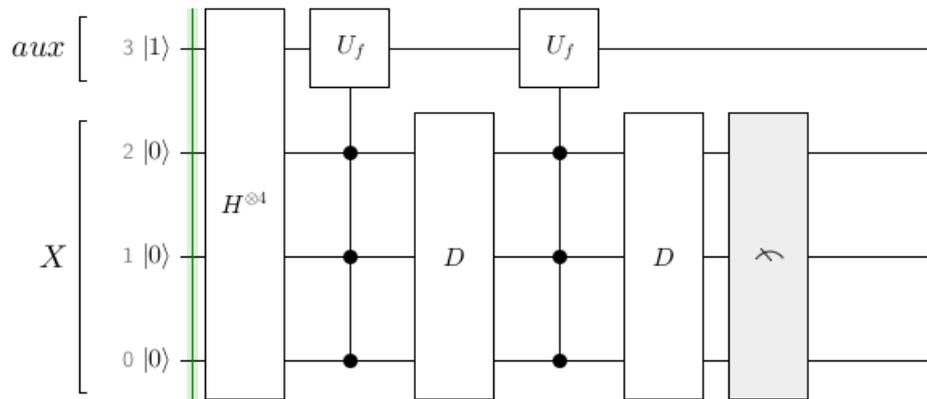


Abbildung 7.3.: Verifikation: Grover-Algorithmus, Schaltkreis

Tabelle 7.7 zeigt die Folgezustände der ersten drei Simulationsschritte, es sind die Auswirkungen des Oracles, sowie des Diffusionsschrittes zu erkennen. Zum Vergleich werden in Tabelle 7.8 die Folgezustände der beiden Grover-Iterationen nebeneinander dargestellt. Eine Messung nach Schritt 6 führt mit recht hoher Wahrscheinlichkeit zum Ergebnis 2, es sind allerdings auch Ausreißer zu vermerken, da, wie man sieht, immer noch eine geringe Wahrscheinlichkeit über die restlichen Basiszustände verteilt ist.

7. Beispielhafte Anwendung und Verifikation

Schritt 1 Initialisierung			Schritt 2 $H( aux\rangle X\rangle)$			Schritt 3 Oracle			Schritt 4 Diffusion		
Amplitude	aux	X	Amplitude	aux	X	Amplitude	aux	X	Amplitude	aux	X
○	0	0	●→	0	0	●→	0	0	●→	0	0
○	0	1	●→	0	1	●→	0	1	●→	0	1
○	0	2	●→	0	2	●←	0	2	●→	0	2
○	0	3	●→	0	3	●→	0	3	●→	0	3
○	0	4	●→	0	4	●→	0	4	●→	0	4
○	0	5	●→	0	5	●→	0	5	●→	0	5
○	0	6	●→	0	6	●→	0	6	●→	0	6
○	0	7	●→	0	7	●→	0	7	●→	0	7
●→	1	0	●←	1	0	●←	1	0	●←	1	0
○	1	1	●←	1	1	●←	1	1	●←	1	1
○	1	2	●←	1	2	●→	1	2	●←	1	2
○	1	3	●←	1	3	●←	1	3	●←	1	3
○	1	4	●←	1	4	●←	1	4	●←	1	4
○	1	5	●←	1	5	●←	1	5	●←	1	5
○	1	6	●←	1	6	●←	1	6	●←	1	6
○	1	7	●←	1	7	●←	1	7	●←	1	7

Tabelle 7.7.: Verifikation: Grover-Algorithmus, Schritt 1 - 4: Folgezustände

**Schritt 4**  
1. Iteration

**Schritt 5**  
2. Iteration

Amplitude	aux	X	Amplitude	aux	X
	0	0		0	0
	0	1		0	1
	0	2		0	2
	0	3		0	3
	0	4		0	4
	0	5		0	5
	0	6		0	6
	0	7		0	7
	1	0		1	0
	1	1		1	1
	1	2		1	2
	1	3		1	3
	1	4		1	4
	1	5		1	5
	1	6		1	6
	1	7		1	7

**Tabelle 7.8.:** Verifikation: Grover-Algorithmus, Folgezustände nach den Iterationen

## 7.4. Quantenteleportation

Der Algorithmus der Quantenteleportation, wie in Abschnitt 2.3.4 beschrieben, setzt voraus, dass  $|a\rangle$  und  $|b\rangle$  verschränkt sind. Des Weiteren ist es zum Zweck der Demonstration sinnvoll, für  $|\psi\rangle$  einen überlagerten, mit einem relativen Phasenfaktor behafteten Zustand zu verwenden, so dass der Effekt der beiden gesteuerten Rotationen  $X$  und  $Z$  auf  $|b\rangle$  sichtbar wird. Daher wird der Quantenschaltkreis wie in Abbildung 7.4 verwendet. Schritt 1 und 2 dienen der Herstellung von  $|\psi\rangle$ , Schritt 3 und 4 versetzen  $|a\rangle$  und  $|b\rangle$  und einen verschränkten Zustand, Schritt 5 bis 9 stellen die eigentliche Teleportation dar.

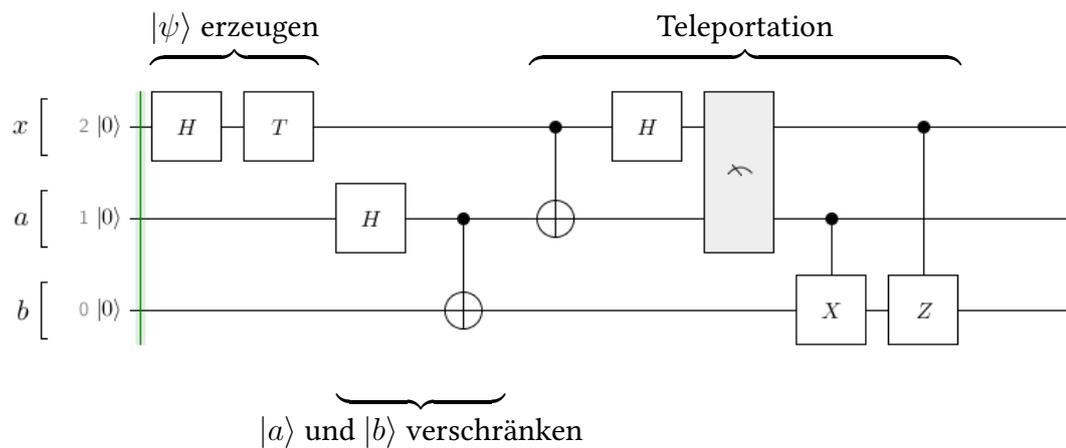


Abbildung 7.4.: Verifikation: Quantenteleportation, Schaltkreis

**Schritt 1 bis 4**

Die Schritte 1 bis 4 dienen der Vorbereitung des Quantenregisters  $|x, a, b\rangle$ . Tabelle 7.9 zeigt den Folgezustand nach Herstellung von  $|\psi\rangle$  (Schritt 1 und 2):

$$|\psi\rangle = \frac{1}{\sqrt{2}}(\alpha|0\rangle + \beta|1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle),$$

sowie den Zustand nach der Verschränkung von  $|a\rangle$  und  $|b\rangle$  (Schritt 3 und 4):

$$|x, a, b\rangle = |\psi\rangle \otimes \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle).$$

Schritt 2				Schritt 4			
Herstellung von $ \psi\rangle$				Verschränkung von $ a\rangle$ und $ b\rangle$			
Amplitude	x	a	b	Amplitude	x	a	b
	0	0	0		0	0	0
	0	0	1		0	0	1
	0	1	0		0	1	0
	0	1	1		0	1	1
	1	0	0		1	0	0
	1	0	1		1	0	1
	1	1	0		1	1	0
	1	1	1		1	1	1

**Tabelle 7.9.:** Verifikation: Quantenteleportation, Folgezustände nach Schritt 2 und 4

**Schritt 5 bis 7**

In den Schritten 5 bis 7 wird die sogenannte *Bell-Messung* an  $|x, a\rangle$  durchgeführt. Zunächst wird  $|x\rangle$  durch das CNOT-Gatter mit  $|a\rangle$  in Verschränkung gebracht, woraufhin  $|x\rangle$  in die Hadamard-Basis transformiert wird um letztendlich beide Qubits in der gedrehten Basis zu messen. Tabelle 7.10 zeigt die vier möglich Folgezustände nach der Messung:

**Fall a :**  $|x, a, b\rangle = \frac{1}{\sqrt{2}}(\alpha|000\rangle + \beta|001\rangle)$

**Fall b :**  $|x, a, b\rangle = \frac{1}{\sqrt{2}}(\alpha|011\rangle + \beta|010\rangle)$

**Fall c :**  $|x, a, b\rangle = \frac{1}{\sqrt{2}}(\alpha|100\rangle - \beta|101\rangle)$

**Fall d :**  $|x, a, b\rangle = \frac{1}{\sqrt{2}}(\alpha|111\rangle - \beta|110\rangle)$ .

Fall a				Fall b				Fall c				Fall d			
x = 0, a = 0				x = 0, a = 1				x = 1, a = 0				x = 1, a = 1			
Amplitude	x	a	b												
	0	0	0		0	0	0		0	0	0		0	0	0
	0	0	1		0	0	1		0	0	1		0	0	1
	0	1	0		0	1	0		0	1	0		0	1	0
	0	1	1		0	1	1		0	1	1		0	1	1
	1	0	0		1	0	0		1	0	0		1	0	0
	1	0	1		1	0	1		1	0	1		1	0	1
	1	1	0		1	1	0		1	1	0		1	1	0
	1	1	1		1	1	1		1	1	1		1	1	1

**Tabelle 7.10.:** Verifikation: Quantenteleportation, Schritt 7: Folgezustände

**Schritt 8 und 9**

Um den ursprünglichen Zustand  $|\psi\rangle$  im Qubit  $|b\rangle$  herzustellen, führt Bob zwei bedingte Rotationen,  $X$  und  $Z$  durch. Diese Transformationen sind im Prinzip "klassisch gesteuert", d.h. Bob erhält von Alice ein klassisches Signal mit dem Ergebnis der *Bell-Messung*, woraufhin beispielsweise ein klassischer Computer die Entscheidung darüber trifft ob  $X$ ,  $Z$  oder  $ZX$  ausgeführt wird:

**Fall a :**  $|\psi\rangle = |b\rangle$

**Fall b :**  $|\psi\rangle = X|b\rangle$

**Fall c :**  $|\psi\rangle = Z|b\rangle$

**Fall d :**  $|\psi\rangle = ZX|b\rangle$

Es ist allerdings äquivalent "quantenmechanisch gesteuerte" Gatter ( $X^{M_a}, Z^{M_x}$ ) zu verwenden [vgl. NC00, S. 28, 186].

$X$  vertauscht die Amplituden  $\alpha$  und  $\beta$ ,  $Z$  negiert  $\beta$ , anhand der Graphiken (siehe Tabelle 7.10) ist leicht zu erkennen, dass  $|b\rangle$  dadurch in den Zustand  $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$  versetzt wird, die jeweiligen Endzustände werden in Tabelle 7.11 gezeigt.

Fall a				Fall b				Fall c				Fall d			
Amplitude	x	a	b												
	0	0	0		0	0	0		0	0	0		0	0	0
	0	0	1		0	0	1		0	0	1		0	0	1
	0	1	0		0	1	0		0	1	0		0	1	0
	0	1	1		0	1	1		0	1	1		0	1	1
	1	0	0		1	0	0		1	0	0		1	0	0
	1	0	1		1	0	1		1	0	1		1	0	1
	1	1	0		1	1	0		1	1	0		1	1	0
	1	1	1		1	1	1		1	1	1		1	1	1

**Tabelle 7.11.:** Verifikation: Quantenteleportation, Schritt 9: Endzustände

**Teil III.**

**Schluss**

## 8. Fazit

In diesem Kapitel werden die Resultate dieser Arbeit resümierend zusammengefasst (8.1), und es wird in einem Ausblick (8.2) gezeigt, welche Teile offen gelassen wurden bzw. wo noch Bedarf für Erweiterungen oder Verbesserungen besteht.

### 8.1. Zusammenfassung

Möchte man sich dem komplexen Themenbereich der Quanteninformatik nähern, ist es sinnvoll, die theoretischen Überlegungen experimentell zu überprüfen. Da aktuelle Implementierungen von Quantumcomputern sehr teuer, und zudem auf wenige Qubits begrenzt sind<sup>1</sup>, stellt die Simulation mit klassischen Mitteln im Moment die einzige Möglichkeit dazu dar. Ziel dieser Arbeit war die Entwicklung eines Simulators für Quantenschaltkreise, mit dessen Hilfe beliebige Quantenalgorithmen auf Basis des Gatter-Modells graphisch konstruiert und simuliert werden können. Die Schwerpunkte bei der Zielsetzung waren zum einen die Entwicklung einer geeigneten Benutzeroberfläche, als auch die effiziente, parallelisierbare Realisierung der Simulationsalgorithmen.

Der theoretische Teil dieser Arbeit befasst sich mit den Grundlagen der Quanteninformatik. Die Thematik wurde recht ausführlich behandelt, um auch Nichtexperten einen Einstieg zu ermöglichen. Es wurde versucht, möglichst wenig physikalische Details zu erörtern, der Fokus wurde auf die informationstechnisch relevanten Phänomene und Formalismen gelegt. Am Ende des Grundlagenteils wurden vier bekannte Quantenalgorithmen vorgestellt, an denen der zu entwickelnde Simulator später verifiziert werden sollte.

Im praktischen Teil wurde die angestrebte Software entwickelt. Die Arbeit unterteilt sich dahingehend in Analyse, Entwurf, Realisierung und Test. Zunächst wurden während der Analyse die Zielsetzungen detailliert, und konzeptionell erfasst. Da die Simulation von Quantenschaltkreisen grundsätzlich in die NP-Komplexitätsklasse fällt, musste eine Methode gefunden werden, welche mit Problemgrößen von ca. 20 Qubits fertig wird. In der Entwurfsphase wurden Designentscheidungen getroffen, um die Architektur und wichtige technische Details für die Realisierung in Scala festzulegen.

Abschließend kann festgestellt werden, dass die Zielsetzung dieser Arbeit, abgesehen von einigen kleinen Details (siehe Abschnitt 8.2), vollständig erfüllt wurde. Die Funktio-

---

<sup>1</sup>Adiabatische Quantencomputer sind an dieser Stelle außer acht gelassen.

nalität des Simulators konnte, wie vorgesehen, erfolgreich verifiziert werden (siehe Kapitel 7), weiterhin konnten die Simulationsalgorithmen erfolgreich parallelisiert werden (siehe Abschnitt 6.3), so dass voraussichtlich ein Nutzen aus der parallelen Architektur kommender Prozessorgenerationen gezogen werden kann. Der Simulator wird im Beta-Stadium auf <http://sourceforge.net/projects/simuquant/> öffentlich zugänglich gemacht.

## 8.2. Ausblick

### Features

Wie aus den Entwicklungsdokumenten hervorgeht, wurde ursprünglich spezifiziert, dass Operatoren zu Gruppen zusammenfassbar sein sollten. Dadurch ließen sich wiederholbare Unterprogramme realisieren, was für bestimmte Quantenalgorithmen (beispielsweise den Grover-Algorithmus o.ä.) vorteilhaft wäre. Des Weiteren war geplant, die Operator-Bibliothek für den Benutzer editierbar (Speichern von Gatter-Templates), und durch einen Plugin-Mechanismus erweiterbar zu machen. Aus Zeitgründen musste die Implementierung dieser Features aufgeschoben werden.

### Internationalisierung

Die GUI ist momentan ausschließlich in englischer Sprache realisiert. Es wäre kein besonders großer Aufwand die einzelnen Zeichenketten zur Internationalisierung auszulagern, allerdings sollte sich die Struktur der GUI vorher ausreichend stabilisiert haben (möglicherweise werden in der Zukunft noch größere Änderungen gemacht, wie im vorhergehenden Absatz angesprochen).

### Bedienungsanleitung

Es wäre am wünschenswert gewesen, eine Bedienungsanleitung für *Simuquant* direkt mit der Software zu veröffentlichen. Aus Zeitgründen konnte diese vor Abgabe der Bachelorarbeit nicht mehr fertiggestellt werden, soll aber nach der Abgabe weiter ausgearbeitet, und zu einem späteren Zeitpunkt (online) zur Verfügung gestellt werden.

### Simulationsmethode

Während der Arbeit hat sich, in einem fortgeschrittenen Stadium, herausgestellt, dass es möglicherweise lohnenswert wäre die *QuIDD*-Datenstruktur von [Via07](#) näher zu untersuchen, da diese anscheinend in fast allen Fällen, durch Ausnutzung von Redundanzen, erheblich weniger Speicherbedarf aufweist, und in bestimmten Fällen die Laufzeiteigenschaften echter Quantencomputer annähert (Grover-Algorithmus) (vgl. [VMH05](#)).

**Teil IV.**

**Anhang**

# Literaturverzeichnis

- [AMS07] AGGOUR, Kareem S. ; MATTHEYSES, Robert M. ; SHULTZ, Joseph: Efficient quantum computing simulation through dynamic matrix restructuring and distributed evaluation. In: *Cluster Computing, IEEE International Conference on 0* (2007), S. 1–10. ISBN 978-1-4244-1387-4
- [BBC<sup>+</sup>95] BARENCO, Adriano ; BENNETT, Charles H. ; CLEVE, Richard ; DiVINCENZO, David P. ; MARGOLUS, Norman ; SHOR, Peter ; SLEATOR, Tycho ; SMOLIN, John A. ; WEINFURTER, Harald: Elementary gates for quantum computation. In: *Phys. Rev. A* 52 (1995), Nov, S. 3457–3467
- [Bru03] BRUß, D.: *Quanteninformaton*. Fischer, 2003 (Fischer-Taschenbücher). – ISBN 3-596-15563-0
- [Can11] CANZLER, T.: *Script zur Vorlesung: Quantenrechner und Quantenkryptographie*. 2011. – unveröffentlicht
- [EPR35] EINSTEIN, A. ; PODOLSKY, B. ; ROSEN, N.: Can Quantum-Mechanical Description of Physical Reality Be Considered Complete? In: *Physical Review* 47 (1935), May, S. 777–780
- [Ess11] ESSER, F.: *Scala für Umsteiger*. 1. Oldenbourg Wissenschaftsverlag, 2011. – ISBN 978-3-486-59693-9
- [GHJ95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph a.: *Design Patterns – Elements of Reusable Object-Oriented Software*. 1. Amsterdam : Addison-Wesley Longman, 1995. – ISBN 0-201-63361-2
- [Gro96] GROVER, Lov K.: A fast quantum mechanical algorithm for database search. (1996), S. 212–219. ISBN 0-89791-785-5
- [HBK06] HELD, Jim ; BAUTISTA, Jerry ; KOEHL, Sean: *From a Few Cores to Many: A Tera-scale Computing Research Overview*. 2006. – Intel whitepaper
- [HHB<sup>+</sup>04] HACKERMÜLLER, L ; HORNBERGER, K ; BREZGER, B ; ZEILINGER, Anton ; ARNDT, M: Decoherence of matter waves by thermal emission of radiation. In: *Nature* 427 (2004), Feb, Nr. quant-ph/0402146, S. 711–714. 5 p
- [Hom08] HOMEISTER, Matthias: *Quantum Computing verstehen*. 2. Auflage. Wiesbaden : Vieweg, 2008. – XII, 303 S.. – ISBN 978-3-8348-0436-5

- [NC00] NIELSEN, Michael A. ; CHUANG, Isaac L.: *Quantum Computation and Quantum Information (Cambridge Series on Information and the Natural Sciences)*. 1. Cambridge University Press, 2000 <http://www.worldcat.org/isbn/521635039>. – ISBN 0–521–63503–9
- [OD98] OBENLAND, Kevin M. ; DESPAIN, Alvin M.: A Parallel Quantum Computer Simulator. In: *High Performance Computing '98* (1998)
- [Öme12a] ÖMER, Bernhard: *Components of a Quantum Computer*. <http://tph.tuwien.ac.at/~oemer/doc/quprog/node8.html>, 2012. – Online; Zugriff: 2012-08-02
- [Öme12b] ÖMER, Bernhard: *Grover's Database Search*. <http://tph.tuwien.ac.at/~oemer/doc/quprog/node17.html>, 2012. – Online; Zugriff: 2012-07-30
- [Öme12c] ÖMER, Bernhard: *Shor's Algorithm for Quantum Factorization*. <http://tph.tuwien.ac.at/~oemer/doc/quprog/node18.html>, 2012. – Online; Zugriff: 2012-07-30
- [PBRO11] PROKOPEC, Aleksandar ; BAGWELL, Phil ; ROMPF, Tiark ; ODERSKY, Martin: A generic parallel collection framework. In: *Proceedings of the 17th international conference on Parallel processing - Volume Part II*. Berlin, Heidelberg : Springer-Verlag, 2011 (Euro-Par'11). – ISBN 978–3–642–23396–8, S. 136–147
- [Sch95] SCHNEIER, Bruce: *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. New York, NY, USA : John Wiley & Sons, Inc., 1995. – ISBN 0–471–11709–9
- [Via07] VIAMONTES, George F.: *Efficient quantum circuit simulation*. Ann Arbor, MI, USA, University of Michigan, Diss., 2007
- [Vid03] VIDAL, Guifré: Efficient Classical Simulation of Slightly Entangled Quantum Computations. In: *Phys. Rev. Lett.* 91 (2003), Oct, S. 147902+
- [VMH05] VIAMONTES, George F. ; MARKOV, Igor L. ; HAYES, John P.: Is Quantum Search Practical? In: *Computing in Science and Engineering* 7 (2005), S. 62–70. – ISSN 1521–9615
- [VSB<sup>+</sup>01] VANDERSYPEN, Lieven M. K. ; STEFFEN, Matthias ; BREYTA, Gregory ; YANNONI, Costantino S. ; SHERWOOD, Mark H. ; CHUANG, Isaac L.: Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance. In: *Nature* 414 (2001), Dezember, Nr. 6866, S. 883–887. – ISSN 0028–0836
- [Wik12a] WIKIPEDIA: *Grover's algorithm - Wikipedia*. [http://en.wikipedia.org/wiki/Grover%27s\\_algorithm](http://en.wikipedia.org/wiki/Grover%27s_algorithm), 2012. – Online; Zugriff: 2012-07-30
- [Wik12b] WIKIPEDIA: *HotSpot - Wikipedia*. <http://de.wikipedia.org/wiki/HotSpot>, 2012. – Online; Zugriff: 2012-07-11

[Wik12c] WIKIPEDIA: *Shor-Algorithmus*. <http://de.wikipedia.org/wiki/Shor-Algorithmus>, 2012. – Online; Zugriff: 2012-07-30

# Abkürzungsverzeichnis

API	Application Programming Interface
CEE	Complex Expression Engine
FFT	Fast Fourier Transformation
FIEE	Fast Integer Expression Engine
GUI	Graphical User Interface
HSV	Hue Saturation Value
JIT	Just in Time Compilation
JRE	Java Runtime Environment
JVM	Java Virtual Machine
QC	Quantum Computation
QFT	Quantum Fourier Transformation
UML	Unified Modeling Language
VM	Virtual Machine
WYSIWYG	What you see is what you get
XML	Extensible Markup Language

# Symbolverzeichnis

$\mathbb{R}$	Menge der reellen Zahlen.
$\mathbb{C}$	Menge der komplexen Zahlen.
$\mathbb{N}$	Menge der natürlichen Zahlen inkl. 0: $\{0, 1, 2, 3, \dots\}$ .
$\mathbb{N}^+$	Menge der natürlichen Zahlen ohne 0: $\{1, 2, 3, \dots\}$ .
$\mathbb{C}^n$	Komplexer, $n$ -dimensionaler Vektorraum, bzw. Menge der $n$ -dimensionalen komplexen Vektoren.
$\mathbb{C}^{m \times n}$	Menge der komplexen $m \times n$ -Matrizen.
$\otimes$	Operator-Symbol für das Tensorprodukt.
$A^{\otimes n}$	$n$ te Tensorpotenz: $A$ wird $n$ mal mit sich selbst Tensorverknüpft.
$\oplus$	Operator-Symbol für exklusives Oder bzw. Addition modulo 2.
$\langle a $	Zeilenvektor in einem komplexen Hilbertraum.
$ a\rangle$	Spaltenvektor in einem komplexen Hilbertraum.
$\langle a, b\rangle$	Skalarprodukt der Vektoren $a$ und $b$ , äquivalent zu $\langle a b\rangle$ .
$\xrightarrow{A}$	Übergang durch Transformation mit $A$ .
$\vec{v}$	Genereller Vektor.

$A^T$	Transponierte Matrix.
$A^\dagger$	Adjungierte Matrix.
$A^*$	Komplex konjugierte Matrix.
$\epsilon$	Sehr kleine Zahl, $\epsilon \in \mathbb{R}$ .

## Versicherung über Selbstständigkeit

*Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.*

Hamburg, 21. August 2012 

---

 Leonhard Dahl