Fachbereich Informatik und Mathematik
Institut für Informatik

Masterarbeit

Implementierung von Algorithmen zum schnellen Lösen von Quantifizierten Booleschen Formeln in Java

Süleyman Omari

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß

2015

Erklärung gemäß Master-Ordnung 2008 §24 Abs. 12
Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Frankfurt am Main, 08.09.2015

Süleyman Omari

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mir während des Studiums und bei der Masterarbeit zur Seite standen. Besonders möchte ich mich bei Herrn Dr.Sabel und Herrn Prof. Dr.Schmidt-Schauß für Ihre Hilfe und Ihr Engagement bedanken. An dieser Stelle möchte ich gerne Johann Wolfgang von Goethe zitieren: **Das schönste Glück des denkenden Menschen ist, das Erforschliche erforscht zu haben und das Unerforschliche zu verehren.**

Inhaltsverzeichnis

1 Einleitung **5**

2 Grundlagen **7**

2.1 Aussagenlogik 7
2.1.1 Syntax und zentrale Begriffe der Aussagenlogik 8
2.1.2 Semantik der Aussagenlogik 10
2.1.3 Eigenschaften von aussagenlogischen Formeln 10
2.1.4 Normalformen 11
2.2 Einordnung der Komplexität 13
2.2.1 Komplexitätsklassen 13
2.2.2 Lage der Komplexitätsklassen 14
2.2.3 Erfüllbarkeitsproblem SAT und QSAT 15
2.3 Quantifizierte Aussagenlogik 16
2.3.1 Syntax der Quantifizierten Aussagenlogik 17
2.3.2 Semantik der Quantifizierten Aussagenlogik 17
2.3.3 Freie / gebundene Variablen 18
2.3.4 Pränex-Normalform PNF 18
2.3.5 Pränex-Konjunktive-Normalform PKNF 19
2.3.6 P-Space Vollständigkeit von QSAT 19
2.3.7 Polynomialzeit-Hierarchie 19
2.4 SAT- und QBF-Solver und das DPLL-Verfahren 21
2.4.1 DPLL-Algorithmus 24
2.4.1.1 Erweiterung des DPLL-Algorithmus zur Berechnung von Modellen 25
2.4.1.2 Eigenschaften des DPLL-Algorithmus 25
2.4.2 Praxis und Entwicklung von SAT-Solvern 26
2.4.2.1 SAT-Solver SAT4J 27
2.4.3 QBF-Solver und Stand der Technik 27

3 Lösungsverfahren für QBFs **30**

3.1 Verfahren 1: QTrial 31
3.1.1 Der Algorithmus QTrial 32
3.1.2 Vor- und Nachteile 33
3.1.3 Korrektheit 33
3.2 Verfahren 2: QBFtoSAT 34
3.2.1 NNF-Transformierung 35
3.2.2 Quantoreneliminierung 36
3.2.3 Übersetzung in aussagenlogische Klauseln 37

3.2.4 Vor- und Nachteile 38
3.2.5 Korrektheit 39
3.3 Verfahren 3: QDPLL 41
3.3.1 Vorverarbeitung 41
3.3.1.1 PNF-Transformierung 42
3.3.1.2 PKNF-Transformierung 42
3.3.2 Klassifizierung von Klauseln / Lemma zur Klassifizierung 43
3.3.3 Algorithmus QDPLL 44
3.3.4 Vor- und Nachteile 46
3.3.5 Korrektheit 47
3.4 Optimierung der Verfahren 48
3.4.1 Erkennen und Behandeln von PK-Unit-Klauseln 48
3.4.2 Schnelle CNF-Methode 49

4 Implementierung und experimentelle Analyse 50
4.1 Programmiersprache Java und verwendete Bibliotheken 50
4.1.1 Entwicklung von Java 51
4.1.2 Verwendung von Java 52
4.1.3 Java und seine Eigenschaften 52
4.1.4 Wichtige Konzepte 53
4.1.5 Verwendete Bibliotheken 55
4.2 Klassenhierarchie und Formeldarstellung 57
4.2.1 Interne Formeldarstellung in Java 60
4.3 Implementierung der Inferenzverfahren 61
4.3.1 Implementierung QTrial-Verfahren 61
4.3.2 Implementierung QBFtoSAT-Verfahren 64
4.3.3 Implementierung QDPLL-Verfahren 68
4.4 Ein- und Ausgabeschnittstellen 73
4.4.1 Textuelle Formelausgabe 73
4.4.2 Einlesen von Formeln 76
4.4.3 Graphische Benutzeroberfläche 77
4.5 Experimentelle Ergebnisse 79
4.5.1 Einfache Testfälle 79
4.5.2 Laufzeitmessung 80
4.5.2.1 Testbedingungen 80
4.5.2.2 Messergebnisse 82

5 Zusammenfassung und Ausblick 90
5.1 Zusammenfassung 90
5.2 Fazit und Ausblick 91

Literaturverzeichnis 98

Kapitel 1

Einleitung

Das Erfüllbarkeitsproblem der Aussagenlogik, auch genannt SAT, untersucht die Fragestellung, ob eine beliebige aussagenlogische Formel erfüllbar ist. Für etliche Formeln ist die Bestimmung der Erfüllbarkeit äußerst komplex. So komplex, dass SAT sogar ein $\mathcal{NP}$-vollständiges Problem ist. Solche Probleme können mit nicht deterministischen Turing-Maschinen in polynomieller Zeit entschieden werden (siehe 2.2.5). Nichtsdestotrotz zeigt sich in der Praxis, dass moderne SAT-Solver, Programme die das SAT-Problem lösen, oft Erfüllbarkeit schnell entscheiden können. Mittlerweile verfügen diese über eine Vielzahl an Heuristiken und sind sogar dazu in der Lage Probleme mit mehreren Millionen Variablen in bereits wenigen Sekunden zu lösen. Dabei erstreckt sich ihr Anwendungsgebiet über verschiedenste Bereiche. Ob in der Verifikation von Hard- und Software, beim Planen und Konfigurieren, in der Bioinformatik oder Kryptoanalyse. Fast überall werden Solver verwendet wie in 2.4.2 beschrieben. Von Vorteil ist, dass sich Probleme bzw. Sprachen der Komplexitätsklasse $\mathcal{NP}$ meistens unproblematisch als SAT-Problem formulieren lassen. Jedoch stößt die SAT-Kodierung, bei zunehmender Komplexität der zu beschreibenden Probleme, an ihre Grenzen und versagt oftmals.

Um Probleme außerhalb von $\mathcal{NP}$ kompakt beschreiben zu können und andere Komplexitätsklassen der Polynomialzeit-Hierarchie abzustecken, bietet sich die Möglichkeit an, Probleme in $QSAT$ zu überführen. $QSAT$ ist das Entscheidungsproblem der Quantifizierten-Aussagenlogik und untersucht die Fragestellung, ob eine beliebige quantifizierte boolesche Formel, kurzum QBF genannt, gültig ist. Die Quantifizierte-Aussagenlogik stellt eine Erweiterung zur klassischen Aussagenlogik dar, um die Möglichkeit aussagenlogische Variablen existenziell oder universell zu quantifizieren (siehe 2.3). Somit sind in $QSAT$ kodierte Probleme zwar in ihrer Darstellungsform bedeutend kompakter als entsprechende aussagenlogische Formeln, bergen aber den Nachteil, das sie deutlich schwieriger zu lösen sind. Bei $QSAT$ spricht man daher auch von einem sogenanntes $\mathcal{PSPACE}$-vollständigen Problem. Solche Probleme sind mit polynomiellem Speicherplatz mit einer deterministischen Turing-Maschine entscheidbar 2.2.7.

In den letzten Jahren wurden enorme Fortschritte bei der Suche nach $QSAT$-Algorithmen errungen. Trotzdem wird in der Praxis auf SAT-Solver gesetzt.

Deren dominierende Rolle rührt einerseits daher, dass sie schneller sind, da die meisten *QSAT*-Algorithmen lediglich Erweiterungen von *SAT*-Algorithmen darstellen und andererseits daher, dass die *SAT*-Community größer ist und somit die Weiterentwicklung von *SAT*-Solvern stärker vorangetrieben wird. Zwangsläufig muss jedoch die Entwicklung von *QSAT*-Algorithmen verstärkt werden, da zukünftige Probleme immer komplexer werden und zusätzlich die Menge an Daten exorbitant wächst [Hen14]. Prognosen laut [Jü13] besagen, dass die Menge an Daten die im Jahr 2020 erstellt, vervielfältigt und verbraucht werden, bei über 40 Zettabytes ($40 * 10^{21}$ Bytes) liegen wird. Diese Zukunftsaussichten sollten genug Motivation schüren, mehr Zeit in die Entwicklung von *QSAT*-Algorithmen zu investieren.

Daher ist ein Ziel dieser Arbeit bestehende *QSAT*-Verfahren allgemein verständlich darzustellen, zu erläutern und zu analysieren. Hierfür werden drei unterschiedliche Verfahren zum Lösen von QBFs vorgestellt. Das erste Verfahren namens *QTrial* wird in Abschnitt 3.1.1 erläutert. Es basiert auf einem rekursiven Algorithmus, der durch naives Ausprobieren versucht die Gültigkeit einer Formel zu überprüfen. Ein durchaus intelligenterer Ansatz wird im zweiten Verfahren namens *QBFtoSAT* verfolgt. Hierbei werden QBFs in äquivalente aussagenlogische Formeln transformiert und deren Gültigkeit mit einem SAT-Solver überprüft (siehe Abschnitt 3.2). Das letzte Verfahren wird in Abschnitt 3.3 beschrieben. Es heißt *QDPLL* und basiert auf dem **D**avis-**P**utnam-**L**oveland-**L**ogemann Verfahren, einem backtracking-basierten Suchalgorithmus, der die Erfüllbarkeit einer aussagenlogischen Formel in konjunktiver Normalform entscheidet. Ein weiteres angestrebtes Ziel besteht in der Implementierung der drei Verfahren in der modernen Programmiersprache Java. Hierbei liegt der Fokus in der objektorientierten Darstellung von Formeln. Ein drittes Ziel, dass in dieser Arbeit verfolgt wird, besteht darin, die verschiedenen Verfahren auf ihre Laufzeit zu testen und experimentell miteinander zu vergleichen. Das letzte Ziel dient didaktischen Zwecken, d.h. es soll beim Umgang und dem Lösen von QBFs helfen. Daher soll eine graphische Benutzeroberfläche (kurzum GUI) entwickelt werden, die es erlaubt QBFs einzugeben und durch Knopfdruck mit den drei implementieren Algorithmen automatisch auf Gültigkeit zu überprüfen.

Die Arbeit gliedert sich in fünf Kapitel. Im Anschluss an dieses Kapitel werden in Kapitel 2 die wesentlichen Grundlagen, die für das Verständnis der Algorithmen notwendig sind, erläutert. Daher umfasst Kapitel 2 die Grundlagen der Aussagenlogik, Quantifizierten-Aussagenlogik und natürlich wichtige Kenntnisse bezüglich dem SAT-Solving. In Kapitel 3 werden die Lösungsverfahren der einzelnen *QSAT*-Algorithmen beschrieben. Insgesamt werden drei unterschiedliche Verfahren vorgestellt. Im darauffolgendem Kapitel 4 wird die Programmiersprache Java vorgestellt. Hierbei werden nicht nur Entwicklung und Eigenschaften der Sprache dargelegt, sondern ebenfalls für die Implementierung notwendigen Datenstrukturen vorgestellt. Anschließend wird auf die Implementierung der Algorithmen eingegangen und die experimentellen Ergebnisse, die mit diesen Programmen erreicht wurden, dargelegt. Diese umfassen einfache Testfälle und ausführlichere Laufzeitmessungen. Kapitel 5 bietet eine Zusammenfassung, ein Fazit und einen Ausblick in die Zukunft.

Kapitel 2

Grundlagen

Dieses Kapitel beinhaltet sämtliche Grundlagen, die für das Verständnis der Lösungsverfahren elementar sind. Zu Beginn werden in 2.1 die Grundlagen der Aussagenlogik erörtert und betont, das dieses logische System die Grundlage der Quantifizierten-Aussagenlogik bildet. Anschließend werden in 2.2 relevante theoretische Komplexitätsklassen definiert und die Einordnung des $QSAT$-Problems in die Klasse $\mathcal{PSPACE}$ vorgenommen. Im Anschluss darauf wird in 2.3 die Quantifizierte-Aussagenlogik vorgestellt. Dabei steht insbesondere dessen Erfüllbarkeitsproblem $QSAT$ im Fokus. Im nächsten Abschnitt 2.4 werden Bestandteile, Anwendungsgebiete und Beispiele für Solver gegeben. Hierbei wird speziell auf den *DPLL*-Algorithmus eingegangen, der Grundlage der meisten Solver ist. Zum Schluss wird in 2.4.3 ein aktueller Stand der Technik gewährt, worin andere Vorgehensweisen zur Lösung von $QSAT$ dargelegt werden.

2.1 Aussagenlogik

Hauptbestandteil der Aussagenlogik sind Aussagen. Aussagen entsprechen Behauptungen, die entweder wahr oder falsch seine können (siehe [Got03]). Durch die Verwendung von logischen Verknüpfungen, den sogenannten Junktoren, lassen sich atomare Aussagen zu komplexen Aussagen verknüpfen. Formeln können auf unterschiedliche Arten verknüpft werden. Daher existieren mehrere mehrstellige Junktoren, von denen hier folgende fünf relevant sind.
Relevante Junktoren: $\neg$, $\wedge$, $\vee$, $\implies$, $\iff$

Wie bereits erwähnt unterscheidet man zwischen atomaren und komplexen Aussagen [Kre06]. Atomare Aussagen sind Aussagen, die keinen Junktor enthalten wie beispielsweise die Aussage: „ *Asien ist ein Kontinent* “. Zusammengesetzte Aussagen hingegen, sind Aussagen, die mindestens einen Junktor enthalten, wie die Aussage: „ *Asien ist ein Kontinent und Europa ist ein Kontinent*“. Aussagen lassen sich mithilfe aussagenlogischer Variablen wesentlich kompakter darstellen.

Beispiel 2.1.1.
Seien A und B aussagenlogische Variablen
A = *„Asien ist ein Kontinent“* B = *„Europa ist ein Kontinent“*

Die Aussagenlogik hat einfach verständliche Inferenzmethoden und dient als Basis für viele andere Logiken. Laut [Sch79] stellt sowohl die Quantifizierte-Aussagenlogik als auch die Prädikatenlogik eine Erweiterung dieser Logik dar.

2.1.1 Syntax und zentrale Begriffe der Aussagenlogik

Die Syntax der Aussagenlogik ist einfach und besitzt eine kleine Anzahl an Formationsregeln.

Definition 2.1.2 (Syntax der Aussagenlogik)**.**

Eine beliebige aussagenlogische Formel wird induktiv aus atomaren Formeln erzeugt. Die Aussagenlogik wird durch folgende Grammatik bestimmt (siehe [uDDS13]):

$$A ::= X \mid (A \wedge A) \mid (A \vee A) \mid (\neg A) \mid (A \implies A) \mid (A \iff A) \mid 0 \mid 1$$

Die Bezeichnung hierbei ist folgende:

$A \wedge B$ *die Konjunktion von A und B*
$A \vee B$ *die Disjunktion von A und B*
$\neg A$ *die Negation von A*
$A \implies B$ *die Implikation von A und B*
$A \iff B$ *die Biimplikation von A und B*
0 *steht für eine falsche Aussage*
1 *steht für eine wahre Aussage*

Definition 2.1.3 (Atom)**.**
Eine aussagenlogische Variable, 0 *oder* 1 *bezeichnet man als Atom. [Bec06]*

Definition 2.1.4 (Literal)**.**

Ein Literal ist ein Atom oder die Negation eines Atoms. Im ersten Fall spricht man von einem positiven Literal, im zweiten Fall von einem negativem Literal [Das05].

Beispiel 2.1.5.

1. X *ist ein Atom.*
2. X, $\neg X$ *und* 0 *sind Literale.*
3. $\neg X(\neg X)$ *ist weder Atom noch Literal.*

Die Klammerung ist für aussagenlogische Formeln wichtig, da Junktoren verschiedene Prioritäten besitzen und somit ein beliebiger Ausdruck nicht geklammert semantisch etwas ganz anderes bedeuten kann als beabsichtigt [Bre15].

Beispiel 2.1.6. *Gegeben sind folgende aussagenlogische Formeln:*

1. $((1 \vee 0) \wedge 0) \mapsto 0$
2. $1 \vee 0 \wedge 0 \mapsto 1$

Junktoren haben verschiedene Prioritäten wie beispielsweise in der Mathematik Punkt vor Strich Vorrang hat, hat die Konjunktion Vorrang vor allen anderen Junktoren (siehe[Kre06]). Daher ist die Klammerung von aussagenlogischen Formeln äußerst wichtig, da die Formeln sonst semantisch nicht korrekt sein könnten. Das Beispiel 2.1.6 verdeutlicht die Relevanz der Klammerung. Die erste geklammerte Formel hat den Wahrheitswert 0 und die zweite nicht geklammerte Formel den Wahrheitswert 1. Die beiden Formeln unterscheiden sich lediglich in der Klammerung. Dies hat jedoch unterschiedliche Ergebnisse der beiden aussagenlogischen Formeln zur Folge

2.1.2 Semantik der Aussagenlogik

Definition 2.1.7 (Semantik der Aussagenlogik)**.**

Eine Interpretation I ist eine Funktion $I : \{aussagenlogische\ \ Variable\} \longrightarrow \{0,1\}$*. Die Fortsetzung von I auf Formel F ordnet F einen Wahrheitswert zu wobei gilt [uDDS13]:*

$$I(F \wedge G) := \begin{cases} \text{1, wenn I(F)=1 und I(G)=1} \\ \text{0, sonst} \end{cases}$$

$$I(F \vee G) := \begin{cases} \text{1, wenn I(F)=1 oder I(G)=1} \\ \text{0, sonst} \end{cases}$$

$$I(\neg F) := \begin{cases} \text{1, wenn I(F)=0} \\ \text{0, sonst} \end{cases}$$

$$I(F \implies G) := \begin{cases} \text{1, wenn I(F)=0 oder I(F)=1=I(G)} \\ \text{0, sonst} \end{cases}$$

$$I(F \iff G) := \begin{cases} \text{1, wenn I(F)=I(G)} \\ \text{0, sonst} \end{cases}$$

$$I(0) := 0$$

$$I(1) := 1$$

Eine Interpretation I ist genau dann ein Modell für die Aussage F wenn $I(F) = 1$ *gilt. Dies kann als* $I \models F$ *notiert werden. Zwei Formeln F und G heißen äquivalent, gdw. für alle Interpretationen I die Gleichung* $I(F) = I(G)$ *gilt.*

2.1.3 Eigenschaften von aussagenlogischen Formeln

Folgende Fragen kann man sich zu einer gegebenen aussagenlogischen Formel F stellen :

Ist die Formel F allgemeingültig ?
Ist die Formel F unerfüllbar ?
Ist die Formel F erfüllbar ?
Ist die Formel F falsifizierbar ?

Welche Eigenschaft die aussagenlogische Formel F aufweist hängt von ihrer Interpretation ab.

Definition 2.1.8.

Sei F eine Formel [uDDS13]:

1. *F ist allgemeingültig auch Tautologie genannt, gdw. für alle Interpretationen I gilt $I \models F$.*
2. *F ist unerfüllbar auch Widerspruch genannt, gdw. für alle Interpretationen I gilt $I(F) = 0$.*
3. *F ist erfüllbar auch konsistent genannt, gdw. eine Interpretationen I existiert mit $I \models F$.*
4. *F ist falsifizierbar, gdw. eine Interpretationen I existiert mit $I(F) = 0$.*

Beispiel 2.1.9.

Die Formel $A \vee \neg A$ ist allgemeingültig und erfüllbar. Jedoch ist die Formel weder widersprüchlich noch falsifizierbar.

Die Formel $\neg\ (A \wedge \neg A)$ ist unerfüllbar und falsifizierbar. Jedoch ist die Formel weder allgemeingültig noch erfüllbar.

Die Formel $A \vee B$ ist erfüllbar und falsifizierbar. Jedoch ist die Formel weder allgemeingültig noch widersprüchlich.

2.1.4 Normalformen

Aussagenlogische Formeln können in Normalformen überführt werden. Normalformen sind Formelarten, die sich nach einem bestimmten Muster bilden lassen (siehe[Bec06]). Ein Vorteil solcher Darstellungsformen liegt darin, dass bestimmte Formeleigenschaften leichter zu erkennen sind. Falls beispielsweise F in *KNF* vorliegt, kann die Eigenschaft der Tautologie deutlich schneller überprüft werden. Zudem hat man durch die Verwendung von Normalformen eine einheitliche Darstellung und Ergebnisse können besser miteinander verglichen werden. Daher dienen Normalformen als Eingabe für etliche Algorithmen wie beispielsweise als Eingabe für den *DPLL*-Algorithmus [Let13].

Definition 2.1.10.

Konjunktive Normalform
Eine aussagenlogische Formel F ist in konjunktiver Normalform(KNF), falls sie eine Konjunktion von Disjunktionen von Literalen ist (siehe [Bec06]). Die Formel F muss in folgender Form vorliegen, wobei $L_{i,j}$ Literale sind:

$$F = (L_{1,1} \vee \ldots \vee L_{1,n_1}) \wedge \ldots \wedge (L_{m,1} \vee \ldots \vee L_{m,n_m})$$

Wenn eine Formel in KNF vorliegt, dann lässt sich in Zeit $O(n * log(n))$[1] die KNF auf Tautologieeigenschaft testen [uDDS13].

[1] n ist die syntaktische Größe der KNF bzw. DNF

Disjunktive Normalform:
Eine aussagenlogische Formel F ist in disjunktiver Normalform(DNF), falls sie eine Disjunktion von Konjunktionen von Literalen ist (laut [Das05]).
Die Formel F muss in folgender Form vorliegen, wobei $L_{i,j}$ Literale sind:

$$F = (L_{1,1} \wedge \ldots \wedge L_{1,n_1}) \vee \ldots \vee (L_{m,1} \wedge \ldots \wedge L_{m,n_m})$$

Wenn eine Formel in DNF vorliegt, dann kann man in Zeit $O(n * log(n))$ die DNF auf nicht Erfüllbarkeit testen [uDDS13].

Beispiel 2.1.11.
Formel in DNF: $F = (A \vee \neg B \vee \neg C) \wedge (A \vee B \vee C) \wedge (\neg A \vee \neg B \vee \neg C)$.

Formel in KNF: $G = (\neg A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge B \wedge C) \vee (A \wedge \neg B \wedge \neg C)$..

Für jede beliebige aussagenlogische Formel F existiert eine äquivalente Formel in *KNF* und *DNF*. Der Algorithmus zur Erzeugung einer äquivalenten *KNF* kann aus [Kre06] entnommen werden. Die Laufzeitkomplexität der Transformation ist exponentiell abhängig von der Größe der Eingabe. Es gibt allerdings Algorithmen, die eine *KNF* in polynomieller Laufzeit generieren unter Erhaltung der Erfüllbarkeit jedoch nicht der Äquivalenz (siehe z.B:[SW12]).

Eine Formel in *KNF* kann auch in Mengenschreibweise dargestellt werden. Diese wird Klauselnormalform bzw. auch Klauselmenge genannt.

$$F = \underbrace{(L_{1,1} \vee \ldots \vee L_{1,n_1})}_{Klausel} \wedge \ldots \wedge \underbrace{(L_{m,1} \vee \ldots \vee L_{m,n_m})}_{Klausel}$$

Mengenschreibweise:
$$\{\{L_{1,1} \ldots L_{1,n}\}, \ldots, \{L_{m,1} \ldots L_{m,n_m}\}\}$$

Definition 2.1.12 (Klausel)**.**
Eine Formel ist eine Klausel, wenn Sie aus einer Disjunktion von Literalen besteht $L_1 \vee L_2 \vee \ldots \vee L_n$.

Definition 2.1.13 (Spezielle Klauseln)**.**

1. ***Leere Klausel:***
   Eine Klausel ohne Literale wird als leere Klausel bezeichnet. Sie ist äquivalent zum Widerspruch. Wenn eine Klauselmenge die leere Klausel enthält, dann ist die Formel widersprüchlich.

2. ***Einsklausel:***
Klauseln die genau ein Literal enthalten, werden als Einsklausel bzw. Unit-Klausel bezeichnet.

Definition 2.1.14 (Isoliertes Literal)**.**
Ein Literal L_i *ist ein isoliertes Literal in der Klauselmenge* C*, gdw.* $\overline{L_i} \ni C$ *vorkommt siehe [uDDS13], wobei* $\overline{L_i} := \begin{cases} \neg\ X_i, \text{ wenn } L_i = X_i. \\ X_i, \text{ wenn } L_i = \neg\ X_i. \end{cases}$

Beispiel 2.1.15.

In der Klauselmenge $K = \{\{\neg A, B, \neg C\}, \{\neg A, \neg C, \neg B\}, \{\neg A, B\}\}$ *sind* $\{\neg A, \neg\ C\}$ *isolierte Literale, da Sie nur in negierter Form vorkommen. Das Literal* B *ist nicht isoliert, da es sowohl in positiver als auch in negativer Form in der Klauselmenge vorliegt.*

2.2 Einordnung der Komplexität

Um die Komplexität der kommenden Probleme besser einordnen zu können werden in diesem Abschnitt zunächst einige relevante Komplexitätsklassen definiert. Häufig spricht man in diesem Kontext auch von sogenannten Entscheidungsproblemen sprich einfachen Ja-Nein Varianten von algorithmischen Problemen (siehe [Sch13]). Im Anschluss darauf wird die Lage unterschiedlicher Komplexitätsklassen zueinander betrachtet und auf das offene Problem der Informatik $\mathcal{P}$ vs $\mathcal{NP}$ eingegangen. Zum Schluss wird die Komplexität der Erfüllbarkeitsprobleme *SAT* und *QSAT* analysiert.

2.2.1 Komplexitätsklassen

Die Berechenbarkeitskomplexität von Problemen bzw. Sprachen kann in sogenannte Komplexitätsklassen eingeteilt werden. Einige relevante Komplexitätsklassen sind die folgenden: $\mathcal{P}$, $\mathcal{NP}$, $Co\mathcal{NP}$ und $\mathcal{PSPACE}$ (siehe [Ren14]). In [Wie98] wird Wissen über Turing-Maschinen vermittelt. Was Turing-Maschinen sind und wie sie arbeiten ist Voraussetzung und wird hier nicht näher erläutert.

Definition 2.2.1 (**Komplexitätsklasse** $\mathcal{P}$)**.** *(siehe [BA02])*
Die Komplexitätsklasse $\mathcal{P}$ *beinhaltet genau die Menge aller Entscheidungsprobleme, die von deterministischen Turingmaschinen in Polynomialzeit gelöst werden können.*

Definition 2.2.2 (**Komplexitätsklasse** $\mathcal{NP}$)**.** *(siehe [Sch13])*
Die Komplexitätsklasse $\mathcal{NP}$ *beinhaltet genau die Menge aller Entscheidungsprobleme, die von nicht deterministischen Turingmaschinen in Polynomialzeit gelöst werden können.*

Definition 2.2.3 (Komplementärsprache)**.** *(siehe [Bre10])*
Ist L eine Sprache über dem Alphabet Σ, dann enthält die Komplementärsprache ¬L alle Wörter über Σ, die nicht in L liegen.

Komplexitätsklasse $Co\mathcal{NP}$
Ein Entscheidungsproblem befindet sich in der Komplexitätsklasse $Co\mathcal{NP}$, wenn das komplementäre Entscheidungsproblem in $\mathcal{NP}$ liegt.

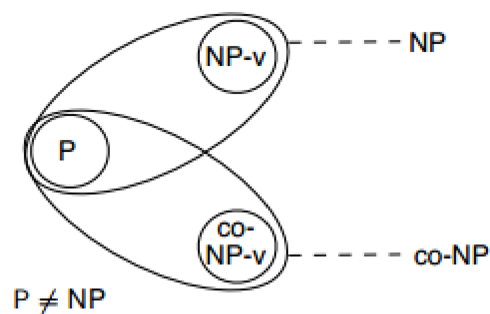
Beispiel 2.2.4. *Das SAT-Problem liegt in der Klasse $\mathcal{NP}$ und sein komplementäres Problem, die nicht Erfüllbarkeit von aussagenlogischen Formeln, liegt in der Klasse $Co\mathcal{NP}$.*

Definition 2.2.5 ($\mathcal{NP}$-Vollständigkeit)**.**
Ein Entscheidungsproblem Z heißt $\mathcal{NP}$-vollständig genau dann, wenn $Z \in \mathcal{NP}$ und Z ein $\mathcal{NP}$-hartes Problem ist. Ein Problem heißt $\mathcal{NP}$-hart genau dann, wenn sich alle Probleme in $\mathcal{NP}$ auf das Problem reduzieren lassen.

2.2.2 Lage der Komplexitätsklassen

Im Allgemeinen gilt die Annahme, dass die Komplexitätsklasse $\mathcal{P} \neq \mathcal{NP}$ ist. Die Frage $\mathcal{P}$=$\mathcal{NP}$ zählt zu einer der größten offenen Probleme der Informatik. Für die Lösung dieses Problems wurde sogar ein Preisgeld von über 1 Millionen US- Dollar ausgesetzt. Falls die Annahme $\mathcal{P} \neq \mathcal{NP}$ gelten würde, wären die Auswirkungen nicht so fatal. Falls jedoch die Annahme $\mathcal{P} = \mathcal{NP}$ gilt, könnten laut [Gro09] etliche schwierige Probleme der Informatik effizient gelöst werden. Abbildung 2.2.1 zeigt die Lage der Komplexitätsklassen zueinander. Alle Probleme der Komplexitätsklasse $\mathcal{P}$ sind ebenfalls Teil der Komplexitätsklasse $\mathcal{NP}$ und $Co\mathcal{NP}$ (siehe [oAI09]).

Abbildung 2.2.1:



2.2.3 Erfüllbarkeitsproblem SAT und QSAT

Das Erfüllbarkeitsproblem der Aussagenlogik auch genannt SAT, behandelt die Fragestellung, ob eine beliebige aussagenlogische Formel F erfüllbar ist. Auf den ersten Blick scheint es simpel zu sein diese Frage zu beantworten. Beispielsweise könnte man für eine Formel eine Wahrheitstafel generieren und die vorhandenen aussagenlogischen Variablen mit allen möglichen Kombinationen von Wahrheitswerten belegen, bis eine Zeile wahr ist. Auf diesem Weg die Erfüllbarkeit zu entscheiden, ist aber nur dann effizient, wenn die Anzahl der Variablen in F klein und die Struktur der Formel nicht zu komplex ist. Im Umkehrschluss, wenn die Anzahl der Variablen groß und die Struktur der Formel komplex ist, ist es ein äußerst schwieriges Problem und solch eine Zeile zu finden würde zu viel Zeit in Anspruch nehmen (siehe [Oma14]).

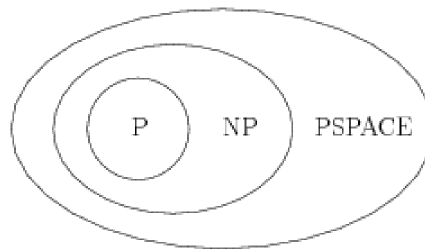
Beispiel 2.2.6.
Sei F eine aussagenlogische Formel mit n=500 Variablen. Die Untersuchung einer Zeile in der Wahrheitstafel betrage 10^{-5} *Sekunden. Bei n=500 Variablen hat die Wahrheitstafel für die Formel F genau* 2^{500} *Zeilen. Die Untersuchung von allen Zeilen der Wahrheitstafel dauert dann genau* $2^{500} * 10^{-5}$ *Sekunden. Dies entspricht einer Zeitspanne von mehreren Milliarden Jahren.*

Die Schwierigkeit von SAT wurde von Informatik-Professor Stephen A.Cook in seinem berühmten Satz von Cook nachgewiesen [Vö10]. Im Jahr 1971 zeigte er nicht nur, dass $SAT \in \mathcal{NP}$ ist, vielmehr bewies er sogar, dass es sich hierbei um ein $\mathcal{NP}$-vollständiges Problem handelt und somit zu einer der schwierigsten Probleme innerhalb der Klasse $\mathcal{NP}$ zählt. Im Satz von Cook zeigte er, dass jedes Problem der Klasse $\mathcal{NP}$ in polynomieller Zeit auf SAT zurückgeführt werden kann [Dew13]. Diese Erkenntnis ist eine von zwei notwendigen Bedingungen, um $\mathcal{NP}$-Vollständigkeit nachzuweisen siehe Abschnitt 2.2.5. Auf Basis von Cooks Arbeit konnte für viele weitere Probleme $\mathcal{NP}$-Vollständigkeit gezeigt werden. Beispielsweise gelang es Richard Karp im Jahr 1972 die $\mathcal{NP}$-Vollständigkeit für 21 weitere Probleme darzulegen (siehe [Mil13]).

Die Komplexität der Probleme ist trotz enormer Fortschritte zu gewaltig. Daher ist man laut [Sch13] bestrebt, nicht viel Zeit bei der Suche nach effizienten Algorithmen zu verschwenden, sondern nach Problemaufweichungen zu suchen, die effiziente Algorithmen erlauben. Ähnlich wie bei SAT, handelt es sich bei $QSAT$ um ein Erfüllbarkeitsproblem. $QSAT$ ist das Erfüllbarkeitsproblem der Quantifizierten-Aussagenlogik und beschäftigt sich mit der Fragestellung, ob eine beliebige quantifizierte boolesche Formel, kurz um QBF genannt, gültig ist. Die Gültigkeit einer QBF zu bestimmen ist allerdings wesentlich schwieriger als das Lösen einer vergleichbaren aussagenlogischen Formel. Zwar existieren einfache Algorithmen, mit denen sich die Gültigkeit einer QBF bestimmen lässt, jedoch haben diese Verfahren den Nachteil, dass sie viel Speicherplatz benötigen (siehe [Bos98]). Beispielsweise benötigt ein einfacher rekursiver Algorithmus für eine QBF mit n Quantoren $O(2^n)$ Berechnungsschritte und quadratisch viel Speicherplatz in der Länge der Formel. Das Problem liegt also laut [KT12] in $\mathcal{PSPACE}$ (siehe Abbildung 2.2.3). Auf die Quantifizierte-Aussagenlogik und die Komplexitätsklasse $\mathcal{PSPACE}$ wird im nächsten Abschnitt eingegangen.

Definition 2.2.7 (Komplexitätsklasse $\mathcal{PSPACE}$).
Die Komplexitätsklasse $\mathcal{PSPACE}$ bezeichnet die Probleme, die mit polynomiellem Speicherplatz, mit einem deterministischen Algorithmus gelöst werden können.

Abbildung 2.2.3



Ein genauerer Einblick in $\mathcal{PSPACE}$ wird in Abschnitt 2.3.7 gewährt.

2.3 Quantifizierte Aussagenlogik

Die Quantifizierte-Aussagenlogik stellt eine Erweiterung der klassischen Aussagenlogik dar, mit dem Ziel, die Gültigkeit von QBFs festzustellen (siehe [Hen14]). Die Erweiterung birgt die Möglichkeit über aussagenlogischen Variablen zu quantifizieren, sprich aussagenlogische Variablen existenziell bzw. universell zu quantifizieren. Die hier beschriebene Erweiterung ist eine konservative Erweiterung, d.h. das bestehende Theoreme erhalten bleiben (siehe [Bos98]). Wie bereits erwähnt ist das dazugehörige Entscheidungsproblem QSAT ein sogenanntes $\mathcal{PSPACE}$-vollständiges Problem. Solche Probleme können mit polynomiellem Speicherplatz, mit einem deterministischen Algorithmus gelöst werden (siehe 2.2.7). Im Grunde genommen stellt die Quantifizierte-Aussagenlogik ein natürliches Paradigma zur Charakterisierung der Komplexitätsklasse $\mathcal{PSPACE}$ dar. Innerhalb von $\mathcal{PSPACE}$ befinden sich wesentlich komplexere Probleme als in $\mathcal{NP}$ (siehe [NBP15]). Eine effiziente Implementierung für ein Problem dieser Komplexitätsklasse zu entwickeln, würde laut [Sch13] effiziente Entscheidungsverfahren für alle $\mathcal{PSPACE}$-vollständigen Probleme mit sich führen. Dies ist jedoch nicht zu erwarten, da bereits die Lösung von $\mathcal{NP}$-vollständigen Problemen implizieren würde, dass Computer die Fähigkeit hätten zu Raten.

Die Idee, weshalb überhaupt Probleme als QBF Instanz dargestellt werden sollen, wenn deren Lösung deutlich schwieriger ist, ist an dieser Stelle durchaus fraglich. Die Besonderheit und ein signifikanter Vorteil von QBFs gegenüber gewöhnlichen aussagenlogischen Formeln liegt in ihrer oftmals kompakteren und natürlichen Formulierung vieler Probleme aus den Bereichen: Planen, Abduktion, nichtmonotones Schließen oder Model Checking [GM09]. Der Bezug zur Aussagenlogik ist zudem sehr eng und kann folgendermaßen beschrieben werden. Eine aussagenlogische Formel F, mit den Variablen $\{x_1, \ldots, x_n\}$, ist

genau dann erfüllbar, wenn eine QBF $F' = \exists x_1, \dots \exists x_n$ gültig ist bzw. F ist genau dann allgemeingültig, wenn eine QBF $\forall x_1, \dots \forall x_n$ gültig ist. Im Folgenden werden sowohl offene Formeln d.h. Formeln die freie Variablen(FV) enthalten, als auch Formeln ohne freie Variablen(GV), sogenannte geschlossene Formeln, betrachtet [uMJU].

2.3.1 Syntax der Quantifizierten Aussagenlogik

Die Syntax legt fest, welche syntaktischen Gebilde als Formeln akzeptiert werden. QBFs haben folgende Syntax [uMJU]:

$$\begin{aligned} Q := \;& 0|1 \\ & |P \ (Boolesche\ Variable) \\ & |(Q_1 \wedge Q_2) \mid (Q_1 \vee Q_2) \mid (Q_1 \implies Q_2) \mid (Q_1 \iff Q_2) \\ & |(\neg Q) \\ & |\forall P.Q \mid \exists P.Q \end{aligned}$$

Weitere Junktoren lassen sich durch die Junktoren $\wedge, \vee$ und $\neg$ definieren und sind somit redundant.

2.3.2 Semantik der Quantifizierten Aussagenlogik

Die Semantik legt fest, wann eine Formel als wahr bzw. falsch angesehen wird. Eine Interpretation I ist, genau wie in der Aussagenlogik definiert, eine Abbildung von Variablen auf Wahrheitswerte. Ihre Fortsetzung auf Formeln ist analog definiert, wobei die beiden folgenden Fälle neu zu betrachten sind:

$$\begin{aligned} I(\forall x.F) = \ & True\ gdw. \ I[\tfrac{False}{x}](F) = True \ \textbf{und} \\ & I[\tfrac{True}{x}](F) = True \\ I(\exists x.F) = \ & True\ gdw. \ I[\tfrac{False}{x}](F) = True \ \textbf{oder} \\ & I[\tfrac{True}{x}](F) = True \end{aligned}$$

Die Notation $F[\frac{True}{x}]$ bedeutet, dass jedes freie Vorkommen von x in F mit $True$ belegt wird. Die Notation $F[\frac{False}{x}]$ hingegen bedeutet, dass jedes freie Vorkommen von x in F mit $False$ belegt wird. Die Notationen sind vor allem für die Lösungsverfahren in Kapitel 3 äußerst relevant. Zu beachten ist, das eine sehr ähnliche Notation existiert, die jedoch eine komplett andere Bedeutung aufweist. Die Notation $F[\neg X/X]$ bedeutet nämlich, dass parallel alle freien Vorkommen von $\neg x$ durch x und alle freien Vorkommen von x durch $\neg x$ ersetzt werden sollen.

Beispiel 2.3.1 (Übersetzung eines Planproblems als QBF)**.**
In [Rin99] wird ein bekanntes Planungsproblem aus der Blockwelt in eine QBF

überführt. Die komplette Darstellung dieser Transformation ist zu umfangreich und kann in [Rin99] nachgelesen werden. Ein Beipiel das ein bedingtes Planungsproblem in eine QBF überführt befindet sich in [Idt04]. In diesem Beispiel soll ein Roboter einen Gegenstand, der sich entweder in Auto A oder Auto B befindet, ins Haus tragen.

2.3.3 Freie / gebundene Variablen

Eine Variable kann in einer Formel entweder frei oder gebunden vorkommen. Sie liegt frei vor, wenn sie an mindestens einer Stelle innerhalb der Formel nicht quantifiziert vorkommt. Falls die Variable innerhalb der gesamten Formel quantifiziert vorliegt, ist sie gebunden. Formeln die freie Variablen enthalten, werden als offen bezeichnet und Formeln ohne freie Variablen als geschlossen. Für eine Formel F lassen sich die Mengen der freien Variablen $FV(F)$ und der gebundenen Variablen $GV(F)$ durch die folgenden Fälle berechnen:

$FV(F_1 \wedge F_2) = FV(F_1) \cup FV(F_2)$
$FV(F_1 \vee F_2) = FV(F_1) \cup FV(F_2)$
$FV(F_1 \implies F_2) = FV(F_1) \cup FV(F_2)$
$FV(F_1 \iff F_2) = FV(F_1) \cup FV(F_2)$
$FV(\neg F_1) = FV(\neg F_1)$
$FV(\exists\, x.F) = FV(F) \setminus \{x\}$
$FV(\forall\, x.F) = FV(F) \setminus \{x\}$

Nur Formeln mit Quantoren können gebundene Variablen enthalten:

$GV(F_1 \wedge F_2) = GV(F_1) \cup GV(F_2)$
$GV(F_1 \vee F_2) = GV(F_1) \cup GV(F_2)$
$GV(F_1 \implies F_2) = GV(F_1) \cup GV(F_2)$
$GV(F_1 \iff F_2) = GV(F_1) \cup GV(F_2)$
$GV(\neg F_1) = GV(\neg F_1)$
$GV(\exists\, x.F) = GV(F) \cup \{x\}$
$GV(\forall\, x.F) = GV(F) \cup \{x\}$

2.3.4 Pränex-Normalform PNF

Jede beliebige QBF kann in Pränex-Normalform, kurz um *PNF* genannt, überführt werden. Eine Formel befindet sich in *PNF*, wenn sich alle Quantoren im Präfix befinden und der Rumpf eine aussagenlogische Formel bildet. Der Algorithmus zur Erzeugung einer QBF in *PNF* ist Bestandteil des *QDPLL*-Algorithmus und ist in Abschnitt 3.3.3 detailliert beschrieben (siehe [Hen14]).

Definition 2.3.2 (Pränex-Normalform)**.**
Eine QBF F liegt in Pränex-Normalform vor, wenn sie von der Form: $F = Q_1\ x_1 \ldots Q_n\ x_n.\phi$, *mit* $Q_i \in \{\exists, \forall\}$ *ist und* ϕ *quantorenfrei ist*

2.3.5 Pränex-Konjunktive-Normalform PKNF

Die Pränex-Konjunktive-Normalform kurz um *PKNF* genannt, ist eine weitere notwendige Normalform. Eine QBF F liegt in *PKNF* vor, wenn sie von folgender Form ist:

Definition 2.3.3 (Pränex-Konjunktive-Normalform).
Sei $Q_1\ x_1 \ldots, Q_n\ x_n.\phi$ eine QBF , wobei $Q_i \in \{\exists, \forall\}$ und ϕ eine KNF ist. Es besteht die Möglichkeit ϕ auch als Menge von Klauseln zusammenfassen und somit gilt: $F = Q_1\ x_1 \ldots, Q_n x_n.\{K_1, \ldots, K_n\}$.

2.3.6 P-Space Vollständigkeit von QSAT

Der formale Beweis der $\mathcal{PSPACE}$-Vollständigkeit von $QSAT$ kann aus [Uma15] entnommen werden.

2.3.7 Polynomialzeit-Hierarchie

Etliche Probleme liegen außerhalb der Klasse $\mathcal{NP}$ bzw. $Co\mathcal{NP}$ und deshalb ist eine genauere Definition der $\mathcal{PSPACE}$ Klassenkomplexität notwendig. Die Polynomialzeit-Hierarchie unterteilt die Klasse $\mathcal{PSPACE}$ in mehrere Klassen[KT12].

Definition 2.3.4 (Polynomialzeit-Hierarchie).

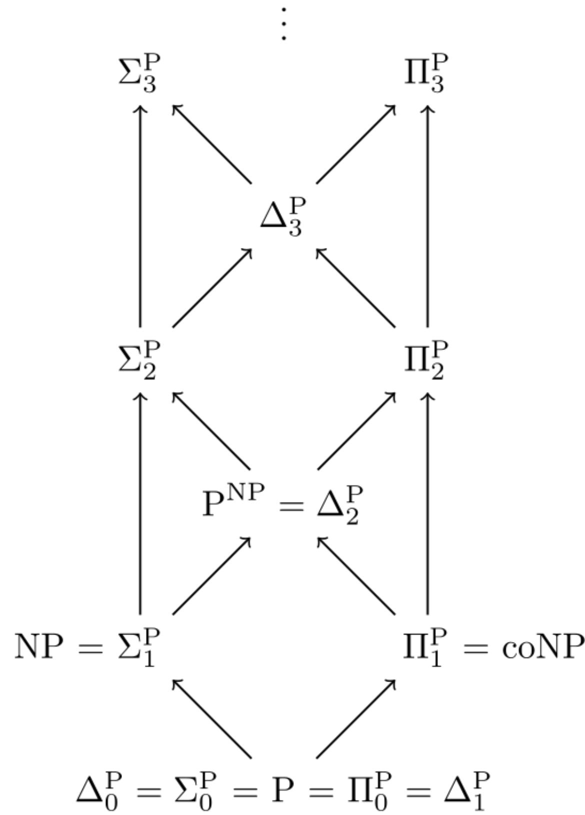
Die Struktur des Präfix einer QBF in PNF gibt viele Rückschlüsse auf dessen Komplexität. Beispielsweise nimmt bei steigender Anzahl von Quantorenwechsel die Komplexität der Formel erheblich zu [Wec13]. Jede quantorenfreie Formel sprich ohne Präfix hat den Präfix-Typ $\Sigma_0 = \Pi_0$ (Die Anzahl der $\exists$-Quantoren entspricht der Anzahl der $\forall$-Quantoren). Sei Ψ eine QBF mit dem Präfix-Typ Σ_n beziehungsweise Π_n dann ist die Formel $\exists x_1 \ldots \exists x_n \psi$ vom Präfix-Typ Σ_{n+1} und die Formel $\forall x_1 \ldots \forall x_n \psi$ vom Präfix-Typ Π_{n+1}. Beispielsweise ist die Formel $\Psi = \forall u \forall x \exists y \forall z\ \psi$ in Π_4 einzuordnen, da $\psi \in \Sigma_0$, $\forall z\ \psi \in \Pi_2$, $\exists y \forall z\ \psi \in \Sigma_2$ und $\forall x \exists y \forall z\ \psi \in \Pi_3$ [GRR+10].

Zwischen dem Präfix-Typ von QBFs und der Polynomialzeit-Hierarchie gibt es einen starken Bezug. Bevor auf diesen eingegangen wird, wird im folgenden die Polynomialzeit-Hierarchie durch die Symbole Δ_i, Σ_i und Π_i induktiv für $k \geq 0$ definiert [Bre10]:

$\Delta_0^P := \sum_0^P := \Pi_0^P := \mathcal{P};$
$\Sigma_{k+1}^P := NP^{\Sigma_k^P}, \Pi_{k+1}^P := co\Sigma_{k+1}^P,\ \Delta_{k+1}^P := P^{\Sigma_k^P}$
Δ_{k+1}^P (bzw. Σ_{k+1}^P) ist die Klasse aller Probleme, die in deterministisch (bzw. nicht deterministisch) polynomieller Zeit mit Hilfe eins Orakels für Probleme der Komplexitätsklasse Σ_k^P entschieden werden können. Orakel erweitern

Turingmaschinen und können Probleme der Komplexitätsklasse Σ_k^P *in konstanter Zeit* $O(1)$ *lösen. Die Klasse* $\Pi^{P_{k+1}}$ *beinhaltet sämtliche Probleme deren Komplement sich in* $\Sigma^{P_{k+1}}$ *befindet. Es gilt* $\Sigma_1^P = NP$, $\Pi_1^P = coNP$ *und* $\Delta_1^P = P$. *Für ein* $k \geq 1$ *wird* $\Delta_k^P \subseteq \Sigma_k^P \cap \Pi_k^P$ *angenommen. Der Beweis dieses Theorem kann aus [Rot08] entnommen werden.*

Abbildung 2.3.7 *Stufen der Polynomialzeit-Hierarchie*



Die QBFs beschreiben die Stufen der Polynomialzeit-Hierarchie vollständig. Die Polynomialzeit-Hierarchie ist die Vereinigung ihrer Stufen (siehe Abbildung 2.3.7):

$$PH = \bigcup_{i \geq 0} \sum_i^P$$

2.4 SAT- und QBF-Solver und das DPLL-Verfahren

SAT-Solver sind Programme, die aussagenlogische Formeln auf Erfüllbarkeit testen und bei positiver Erfüllbarkeit oftmals dazugehörige Modelle mit ausgeben. Der Großteil der *SAT*-Solver akzeptiert als Eingabe nur Formeln im Dimacs-Format [SC09].

Definition 2.4.1.
Das Dimacs-Format ist ein weltweit akzeptiertes Standardformat für aussagenlogische Formeln in KNF. Die erste Zeile der Eingabedatei beginnt mit einem Kommentar c. Die Anzahl der Variablen und Klauseln ist in der nächsten Zeile p cnf variables clauses definiert. Jede der darauffolgenden Zeilen entspricht einer Klausel. Ein positives Literal ist mit einer positiven Zahl und ein negatives Literal ist mit der entsprechenden negativen Zahl gekennzeichnet. Die letzte Zahl einer Klausel Zeile ist immer eine 0. *Die* 0 *gibt das Ende der Klausel bekannt [vtUCRF01].*

Das Beispiel 2.4.2 zeigt die Transformierung einer CNF ins Dimacs-Format und Lösung durch einen *SAT*-Solver.

Beispiel 2.4.2.
Folgende Formel $F = (x \vee y) \wedge (\neg x \vee z \vee \neg x) \wedge (\neg z \vee \neg y) \wedge z$ *in CNF soll mit einem SAT-Solver gelöst werden.*

→ *Transformierung ins Dimacs-Format:*
p cnf 3 4
1 2 0
-1 3 -2 0
-3 -2 0
3 0

RESULT: F IS SATISFIABLE / ***Model 1:***$\rightarrow x = 0; y = 1; z = 1$
Model 2: $\rightarrow x = 1; y = 0; z = 1$
Model 3: ...

Neben den *SAT*-Solvern gibt es natürlich auch *QBF*-Solver. Dies sind Programme, die versuchen die Gültigkeit von QBFs festzustellen. Daher gibt es für QBFs auch ein eigenes sehr ähnliches Format namens QDimacs [QBF15b]. Die Definition des Formats kann ist in [oMLaP01] notiert.

Beispiel 2.4.3.
Folgende Formel soll ins QDimacs überführt werden:

$F = \forall x_1 \exists x_2 \forall x_3 \exists x_4 x_5$
$(x_1 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge$
$(x_1 \vee \neg x_4 \vee x_5) \wedge (\neg x_1 \vee x_2 \vee x_5) \wedge)$
$(x_1 \vee \neg x_3 \vee x_4 \vee \neg x_5) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge$
$(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_5) \wedge (x_1 \vee \neg x_4) \wedge (x_3 \vee \neg x_1 \vee x_1)$

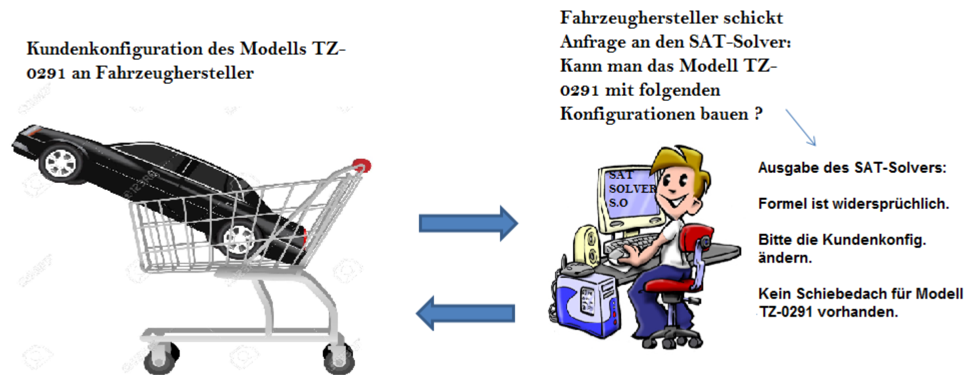
```
QDIMACS FORMAT
c Example CNF format file
c
p cnf 5 9
a 1 0
e 2 0
a 3 0
e 4 5 0
1 3 4 0
-1 3 4 0
1 -4 -5 0
-1 2 5 0
1 -3 4 -5 0
-1 3 -4 0
-1 -2 -3 -5 0
1 -4 0
3 -2 1 0
```

Mittlerweile verfügen *SAT*-Solver über eine Vielzahl an Heuristiken und sind sogar dazu in der Lage Probleme aus der Praxis mit mehreren Millionen Variablen in bereits wenigen Sekunden zu lösen (siehe [Kot07]). Im Kern basiert das Grundgerüst nahezu aller *SAT*-Solver auf dem *DPLL*-Algorithmus. Der *DPLL*-Algorithmus wurde in den sechziger Jahren von den Wissenschaftlern Martin Davis, Hilary Putnam, George Logemann und Donald W. Loveland entwickelt (siehe [Lan12]). Ihr eigentliches Ziel bestand darin, wie in [Gö04] beschrieben, mathematische Theoreme automatisch beweisen zu können. Im Prinzip beruht das Verfahren auf Fallunterscheidung und Propagieren und Ausnutzen von Informationen. Häufig wird das *DPLL*-Verfahren mit dem ursprünglichen Davis-Putnam-Verfahren verwechselt oder gar gleich gesetzt. Beim *DPLL*-Verfahren handelt es sich um eine Erweiterung mit dem Ziel, den eventuell vorhandenen exponentiellen Speicherbedarf, der beim Davis-Putnam Verfahren entstehen kann, zu vermeiden (siehe [uMJU]). Dies wird erreicht indem beim *DPLL* anstelle einer Variableneliminierung, eine Fallunterscheidung bezüglich des ausgewählten Literals durchgeführt.

Beispiel 2.4.4 (Praxisbeispiel SAT-Solver)**.** *Ein Kunde will einen Neuwagen kaufen. Er nutzt die Möglichkeit über eine Webseite die einen Konfigurator beinhaltet sein individuelles Fahrzeug zu kreieren. Seine Wahl fällt auf einen schwarzen Porsche TZ-0291. Die Serienausstattung reicht Ihm nicht aus. Daher trifft er ein paar Änderungen. Ein zusätzlich eingebautes Schiebedach und braunes Leder im Innenraum sollen in seinen TZ.0291. Nun klickt er auf prüfen und seine Anfrage wird an den Server des Fahrzeugherstellers weitergeleitet und ausgewertet. Nach kurzer Zeit erscheint auf seinem Bildschirm eine Fehlermeldung: Bitte ändern Sie Ihre Konfiguration, dass Modell TZ-0291 ist nicht mit einem Schiebedach verfügbar. Doch woher kann das System erkennen, ob die Konfiguration korrekt ist oder nicht. Der Fahrzeughersteller prüft mittels einem Solver, ob die gewünschte Konfiguration einen Widerspruch erzeugt*

oder nicht d.h. die Konfiguration wir in ein SAT-Problem übersetzt. Falls der Solver keinen Widerspruch erkennt, kann das Auto mit der gewünschten Konfiguration gebaut werden. Ansonsten wird eine Fehlermeldung ausgegeben (siehe Abbildung 2.4.4*).*

Abbildung 2.4.4 *[bil]*



2.4.1 DPLL-Algorithmus

Der DPLL-Algorithmus

Algorithmus **DPLL-Algorithmus**

Eingabe: Klauselmenge C, die keine tautologischen Klauseln enthält

Funktion DPLL(C):

```
if ∅ ∈ C // C enthält die leere Klausel
   then return true; // unerfüllbar
endif
if C = ∅ // C ist die leere Menge
   then return false; // erfüllbar
endif
if C enthält 1-Klausel {L} then
   Sei C' die Klauselmenge, die aus C entsteht, indem
     (1) alle Klauseln, die L enthalten, entfernt werden und
     (2) in den verbleibenden Klausel alle Literale L̄ entfernt werden
        (wobei L̄ = ¬X, wenn L = X und L̄ = X, wenn L = ¬X)
   DPLL(C'); // rekursiver Aufruf
endif
if C enthält isoliertes Literal L then
   Sei C' die Klauselmenge, die aus C entsteht, indem alle Klauseln, die
L enthalten,
   entfernt werden.
   DPLL(C'); // rekursiver Aufruf
endif
// Nur wenn keiner der obigen Fälle zutraf:
Wähle Atom A, dass in C vorkommt;
return DPLL(C ∪ {{A}}) ∧ DPLL(C ∪ {{¬A}}) // Fallunterscheidung
```

Die ersten beiden If-Bedingungen des *DPLL*-Algorithmus sind Abbruchbedingungen.

1.Abbruchbedingung $if\ \emptyset\ \in\ C$
Falls die Klauselmenge C die leere Klausel enthält, wird die erste Abbruchbedingung ausgeführt. Das Programm hat dann als Ausgabe den Wert *true*. Somit ist die Klauselmenge C unerfüllbar und der *DPLL*-Algorithmus endet.

2.Abbruchbedingung $if\ C\ =\ \emptyset$
Falls die Klauselmenge C der leeren Menge entspricht, wird die zweite Abbruchbedingung ausgeführt. Das Programm hat dann als Ausgabe den Wert $false$. Somit ist die Klauselmenge C erfüllbar und der *DPLL*-Algorithmus endet. Die nächsten Bedingungen des *DPLL*-Algorithmus können für die Klauselmenge C nur geprüft werden, wenn C keine leeren Klauseln enthält und nicht leer ist.

3.Bedingung $if\ C$ **enthält Eins-Klauseln**$\{L\}$
Falls die Klauselmenge C Eins-Klauseln enthält, wird die dritte Bedingung ausgeführt. Die dritte Bedingung besteht aus zwei Schritten. Im ersten Schritt werden alle Klauseln entfernt, die L enthalten. Im zweiten Schritt werden alle

Klauseln entfernt, in denen $\neg L$ vorkommt. Es entsteht eine neue Klauselmenge C', auf die der *DPLL*-Algorithmus rekursiv aufgerufen wird.

4.Bedingung $if\ C$ enthält isoliertes Literal
Falls die Klauselmenge C ein isoliertes Literal L enthält, wird die vierte Bedingung ausgeführt. In diesem Schritt werden alle Klauseln entfernt, die das isolierte Literal L enthalten. Es entsteht eine neue Klauselmenge C', auf die der *DPLL*-Algorithmus rekursiv aufgerufen wird. Wenn man die Klauseln entfernt, die isolierte Literale enthalten, wird die Erfüllbarkeit der Klauselmenge nicht verändert (Beweis in [uDDS13] Satz 3.6.12.).

Wenn keiner der oben genannten Fälle eintritt, führt der *DPLL*-Algorithmus eine Fallunterscheidung durch. Dabei wählt er ein Literal L aus und erweitert die Klauselmenge C um eine Einsklausel, die das Literal L enthält. Ebenfalls wird C um eine Einsklausel, die das negierte Literal $\neg L$ enthält, erweitert. Anschließend wird der *DPLL*-Algorithmus rekursiv auf die entstehenden Klauselmengen aufgerufen. Die Ergebnisse von $DPLL(C \cup \{\{L\}\})$ und $DPLL(C \cup \{\{L\}\})$ sind verundet. Wenn mindestens einer der beiden Aufrufe $false$ ausgibt, ist die Klauselmenge C erfüllbar (siehe [Gö04]).

2.4.1.1 Erweiterung des DPLL-Algorithmus zur Berechnung von Modellen

Der *DPLL*-Algorithmus muss folgendermaßen erweitert werden, damit zu erfüllbaren aussagenlogischen Formeln Modelle ausgegeben werden (siehe [uDDS13]):

Definition 2.4.5 (Erweiterung des *DPLL*-Algorithmus)**.**

1. *Isolierte Literale werden als wahr angenommen.*
2. *Literale in* 1*-Klauseln werden ebenfalls als wahr angenommen.*
3. *Dadurch nicht belegte Variablen können für das vollständige Modell beliebig belegt werden.*

2.4.1.2 Eigenschaften des DPLL-Algorithmus

Der *DPLL*-Algorithmus hat folgende Eigenschaften:

Definition 2.4.6 (Eigenschaften des *DPLL*-Algorithmus)**.**

1. *Der DPLL-Algorithmus terminiert immer.*
2. *Der DPLL-Algorithmus ist korrekt. Wenn das Resultat einer Eingabe erfüllbar lautet, dann war die Eingabe auch erfüllbar.*
3. *Der DPLL-Algorithmus ist vollständig. Wenn die Eingabe erfüllbar ist, dann sagt der DPLL-Algorithmus auch erfüllbar.*

2.4.2 Praxis und Entwicklung von SAT-Solvern

Probleme aus der Praxis werden immer komplexer und führen zu explodierenden Darstellungen ihrer Probleminstanzen. Solvern wird immer mehr abverlangt und die Nachfrage nach leistungsfähigeren Solvern steigt (siehe [Hen14]). Zahlreiche Heuristiken, Erweiterungen und effizientere Implementierungsmethoden haben die Leistung der Solver in den letzten Jahren erheblich gesteigert. Durch diese enormen Fortschritte und Leistungssteigerungen kann nun in Betracht gezogen werden komplexere Probleme zu untersuchen und in *SAT* bzw. *QSAT* zu kodieren [Bos98]. Alles in allem erzielen derzeit in der Praxis eingesetzte *SAT*-Solver immer bessere Ergebnisse, da zu einem mehr an diesen geforscht und zum anderem die meisten *QBF*-Solver Erweiterungen von *SAT*-Solvern darstellen. Um die Entwicklung innovativer *SAT*-Solver voranzutreiben finden regelmäßig Wettkämpfe statt. Diese fördern die Motivation und haben etliche gute Solver bereits zum Vorschein gebracht. Die Resultate zwischen verschiedenen Solvern samt weiterer interessanter Informationen sind auf der Webseite *satcompetition.org* detailliert dokumentiert (siehe [vMF15]).

In industriellen Anwendungen eingesetzte Solver basieren auf verbesserten Varianten des *DPLL*-Verfahren. Sie kommen in den verschiedensten Bereichen wie Verifikation von Hard- und Software, Scheduling, Planen und Konfigurieren, Automatische Programmanalyse oder in der Kryptoanalyse zum Einsatz. Einige bekannte Solver wie SAT4J, MiniSat, Picosat, Berkmin, RSat und zChaff sind in [Lan12] dargelegt.

Abbildung 2.42 [QBF15a]

SAT Competition 2014

affiliated with the SAT 2014 conference, July 14-17 in Vienna, Austria.
and the FLoC Olympic Games

Winners per category

Sequential Application SAT Track

#	Solver version	Author(s)	#solved
1	minisat_blbd	Jingchao Chen	109
2	Riss BlackBox	Enrique Matos Alfonso and Norbert Manthey	107
3	SWDiA5BY	Chanseok Oh	106

Die Abbildung 2.4.2 zeigt einen Ausschnitt aus der *satcompetition.org* Webseite mit den Gewinnern der Kategorie sequentielle *SAT*-Programme. Ähnlich wie für *SAT*-Solver finden Wettbewerbe zur Performance Bewertung von *QBF*-Solvern statt. Auf zahlreichen Verbesserungsmöglichkeiten wie beispielsweise durch Vorverarbeitung, Random Restart, clause learning, parallelization oder intelligent Backtracking wird in dieser Arbeit nicht eingegangen.

2.4.2.1 SAT-Solver SAT4J

SAT4J ist ein Java Paket, das eine Menge an *SAT*-Solvern beinhaltet. Das *SAT4J* Paket kann beispielsweise von der *sat4j.org* Webseite heruntergeladen werden [ES04]. Um es zu verwenden, muss es in die Bibliothek des entsprechenden Programms eingebunden werden. Die Funktionsweise des *SAT4J SAT*-Solvers ist relativ simpel: Man übergibt ihm eine KNF im Dimacs-Format und erhält als Resultat den Wahrheitswert der *KNF*. Zusätzlich gibt der *SAT*-Solver entsprechende Modelle bei positiver Erfüllbarkeit aus. *SAT4J* kann nicht nur das SAT-Problem zu lösen, sondern auch weitere Probleme wie *MAXSAT*, *Pseudo-Boolean* oder *MUS*. Um eine Idee zu erhalten, zu was *SAT4J* alles in der Lage ist, sollten die case studies betrachtet werden. In der Praxis wird der Solver in vielen unterschiedlichen Bereichen eingesetzt wie beispielsweise in Forschungsgruppen, bei der Softwareentwicklung oder Vorlesungen der Künstlichen Intelligenz und Software-Verifikation [Lan12]. Ebenfalls wird *SAT4J* im *QBFtoSAT*-Verfahren verwendet. Dabei werden verschiedenen Methoden verwendet wie beispielsweise die $isSatisfiable()$ Methode, um den Wahrheitswert der Formel festzustellen. Alle notwendigen Methoden sollten in der Dokumentation nachgelesen werden.

2.4.3 QBF-Solver und Stand der Technik

QBF-Solver sind bisweilen noch nicht so intensiv erforscht und von solch großer praktischer Relevanz wie ihre Konkurrenten die *SAT*-Solver. Trotzdessen wurden in den letzten Jahren enorme Fortschritte bei der Suche nach effizienten Algorithmen gemacht (siehe [Hen14]). Mittlerweile existieren einige interessante und vielversprechende Ansätze, die eine solide Grundbasis schaffen und die Motivation zur weiteren Erforschung fördern. Der hier beschriebene Stand der Technik basiert auf Erkenntnissen, die aus unterschiedlichen wissenschaftlichen Ausarbeitungen zum Thema *QBF*-Solver gewonnen wurden. In diesen Ausarbeitungen wurden Algorithmen, die die Gültigkeit von QBFs entscheiden, untersucht und auf ihre Lautzeit getestet. Nach Evaluation sämtlicher Ausarbeitungen verstärkt sich der Eindruck, das die meisten Verfahren auf dem *DPLL*-Algorithmus aufbauen (siehe [Bos98]). Zusätzlich hat sich herauskristallisiert, dass häufig gewöhnliche Verfahren, mit denen man aussagenlogischen Formeln entscheiden kann, in leicht modifizierter Art für QBFs auftreten.

Die Algorithmen können in fünf unterschiedliche Kategorien aufgeteilt werden. Natürlich kann es vorkommen, dass weitere Kategorien bzw. Unterkategorien existieren, daher soll diese Auflistung nur einen Überblick verschaffen. Die erste und gleichzeitig auch größte Kategorie bilden Algorithmen, die auf dem *DPLL*-Verfahren basieren. Beispiele solcher Algorithmen können aus der untenstehenden Tabelle 2.4.3 entnommen werden. Die zweite Kategorie beinhaltet Algorithmen, die beliebige QBFs in äquivalente aussagenlogische Formeln transformieren und anschließend versuchen deren Gültigkeit mit einem *SAT*-Solver zu entscheiden. Die dritte Kategorie beinhaltet inkrementelle QBF-Solver d.h. Algorithmen, die versuchen Parallelen zu zukünftigen Eingaben aus vorher gewonnenen Erkenntnissen aufzubauen. Die vorletzte Kategorie

bilden probabilistische Algorithmen. Zu erwähnen ist, dass sich probabilistische Methodiken im Bereich der QBF Lösung bisher nicht wirklich bewährt haben. Nicht nur sequentielle Algorithmen sind eine Technik, um die Gültigkeit von QBFs zu prüfen. Auch parallele Algorithmen bieten eine Möglichkeit dies zu tun. Der Aspekt der Parallelität ist äußerst wichtig und wird durch Fortschritte im Bereich von Multi-core Prozessoren in der Zukunft eine immer größere Rolle spielen. Parallele Algorithmen sind Bestandteil der letzten Kategorie. Einige bekannte QBF-Solver sind die folgenden: Quaffle, Quantor, DepQBG, QuBE, Skizzo [Hen14].

Algorithmen basierend auf dem DPLL-Verfahren
Im folgenden werden einige Algorithmen die auf dem *DPLL*-Verfahren basieren vorgestellt. Die Algorithmen sind nur grob beschrieben und sollen dem Leser lediglich einen Überblick geben. Es ist nicht Gegenstand dieser Arbeit die einzelnen Algorithmen detailliert vorzustellen. Um somit die Algorithmen aus der Tabelle 2.4.3 verstehen zu können, sollte der gut strukturierte Pseudocode aus [EG11] unbedingt betrachtet werden.

Tabelle 2.4.3

Algorithmus	Laufzeit	Beschreibung des Algorithmus
Q-DLL	Öfter schlechtere Laufzeiten als seine Kontrahenten	Minimal erweiterter DPLL-Algorithmus. Damit die Gültigkeit einer Formel festgestellt werden kann muss der Suchbaum einer beliebigen Formel solange mit chronologischen Rückschritte abgearbeitet werden bis der Fall auftritt.
Evaluate	Meistens bessere Laufzeit als *Q-DLL* Algorithmus	Ähnlich wie der *Q-DLL* Algorithmus. Es erfolgen zwei Schritte. Schritt 1: Versuch die QBF auf SAT zu reduzieren. Schritt 2: Lösung der SAT-Instanz durch einen SAT-Solver.
Q-DLL-BJ	Meistens bessere Laufzeit als *Q-DLL* Algorithmus	Ähnlich wie der *Q-DLL* Algorithmus. Durch das Entfernen der chronologischen Rückschritte, die beispielsweise der *Q-DLL* macht, erhält man einen verkleinerten Suchbaum. Von diesem erhofft man sich beim Durchlaufen Laufzeitverbesserungen.
Q-DLL-LN	Schlechtere Testresultate als *Q-DLL-BJ*	Ist ein Lernalgorithmus, der einer Erweiterung des *Q-DLL-BJ* Algorithmus ist. Es werden gewonnene Informationen über die Eingabeformel gespeichert, damit diese an anderen Stellen des Suchbaums genutzt werden können, um Pfade, deren Erfüllbarkeit dadurch bekannt war, zu überspringen.
Quaffle	Schneller als *Q-DLL-LN*	Ähnlich wie der *Q-DLL-LN* Algorithmus. Jedoch handelt es sich hier um einen iterativen Algorithmus.

Alles in allem kann sich keiner der hier beschriebenen Algorithmen als klarer Gewinner hervorheben, da zu viele unterschiedliche Faktoren, wie beispielsweise die Eingabeformel selbst, eine Rolle spielen. Jedoch macht sich die Tendenz bemerkbar, dass Erweiterungen des *Q-DLL* Algorithmus meistens besser sind als der ursprüngliche Algorithmus.

QBF in SAT
Ein weiterer Ansatzpunkt an dem geforscht wird, ist die Transformierung von QBFs in äquivalente aussagenlogische Formeln, deren Gültigkeiten im Anschluss mit einem SAT-Solver überprüft werden[JG13]. Ein Algorithmus der auf diesem Verfahren basiert ist Gegenstand dieser Arbeit. Die einzelnen Schritte des Algorithmus sind in Kapitel 3 detailliert beschrieben und wie sich das Verhalten im Vergleich zu den anderen Verfahren verhält in Kapitel 4.5.

Inkrementelle QBF-Solver
In [EG11] wird das Vorgehen inkrementeller *QBF*-Solver beschrieben. Diese führen, stark abhängig von der eingegebenen Formelmenge, zu besseren Ergebnissen als die bisher vorgestellten *QBF*-Solver. Der Inkrementelle *QBF*-Solver verwendet Gemeinsamkeiten unterschiedlicher Probleminstanzen zur Leistungssteigerung. Der Algorithmus erhält als Eingabe eine Menge an Formeln und versucht gelernte Terme nicht nach der Gültigkeit der Formel zu löschen, sondern für zukünftige Eingaben wieder zu verwenden.

Probabilistische Algorithmen
Wie bereits in 2.4.2 beschrieben, beschränkt sich die Vorgehensweise der meisten Verfahren hauptsächlich darin, etablierte *SAT*-Solver auf *QBF*-Solver zu übertragen. Interessanterweise beschränkt man sich dabei hauptsächlich auf deterministische Algorithmen, obwohl sich im Anwendungsbereich der SAT-Solver etliche probabilistische Methoden bewährt haben. Natürlich stellt sich hier die Frage, weshalb probabilistische Methoden nicht auf *QBF*-Solver übertragen werden. Die Antwort darauf ist, das dies am exponentiell wachsendem Zertifikat liegt. Ein probabilistisches Verfahren, das auf stochastischer, lokaler Suche basiert, wird in [Ros03] beschrieben.

Parallele Algorithmen
Die letzte Kategorie beinhaltet parallele Algorithmen. Ziel hierbei ist es *QSAT* parallel zu entscheiden (siehe [Kul09]). Um unterschiedliche Aufgaben gleichzeitig abzuarbeiten werden Threads erzeugt. Beispielsweise könnte man im Suchbaum des *Q-DLL* Algorithmus an Entscheidungsstellen immer einen Thread erzeugen, um Teilbäume parallel abzuarbeiten. Leider führt die Generierung vieler Threads zu enormen Verwaltungsaufwand, der die Zeitersparnis oftmals zunichtemacht[Kul09].

Kapitel 3

Lösungsverfahren für QBFs

In diesem Kapitel werden Lösungsverfahren zur Überprüfung der Gültigkeit von QBFs beschrieben. Insgesamt werden drei unterschiedliche Verfahren vorgestellt. Das erste und einfachste Verfahren ist das *QTrial* Verfahren. Es ist ein naiver Algorithmus, der durch Ausprobieren versucht die Gültigkeit einer Formel zu überprüfen. Die Beschreibung dieses Lösungsverfahren ist in Abschnitt 3.1 dargelegt. Das zweite Verfahren *QBFtoSAT* verfolgt den Ansatzpunkt QBFs in äquivalente aussagenlogische Formeln zu transformieren und anschließend deren Gültigkeit mit einem SAT-Solver zu überprüfen siehe (3.2). In Abschnitt 3.3 ist das dritte und letzte Verfahren namens *QDPLL* dargelegt. Dieses basiert auf dem *DPLL*-Verfahren für QBFs. Bei diesem Verfahren ist die Klassifizierung von Klauseln als PK-Wahre, PK-Falsche und PK-Offene Klausel äußerst relevant. Nach Beschreibung der einzelnen Lösungsverfahren erfolgt eine kurze Begründung ihrer Korrektheit. Alle Schritte sämtlicher Verfahren werden beispielhaft an einer Formel durchgeführt. Zudem wird durch Umbenennung garantiert, das QBFs niemals den gleichen Namen für quantifizierte Variablen verwenden.

Bestandteil der nächsten Kapitel wird die Umsetzung der hier beschriebenen Verfahren in Computerprogramme und ihrer experimentellen Analyse mithilfe der Programmiersprache Java sein. Die theoretische Komplexität der QBF-Lösung ist durch $\mathcal{PSPACE}$ auf exponentielle Laufzeit festgelegt. Ziel ist hier keine Laufzeitoptimierung, sondern experimentell die Laufzeit verschiedener Beispiele miteinander zu vergleichen.

3.1 Verfahren 1: QTrial

Der Algorithmus *QTrial* erhält als Eingabe eine QBF F, die rekursiv ausgewertet wird. Die Ausgabe des Algorithmus ist abhängig von F entweder *True* oder *False*. Beim Auswerten der Formel werden systematisch alle relevanten Pfade entlang des Syntaxbaumes ausprobiert siehe „*LazyEvaluation*“ [Sch02]. Um Platz und Zeit zu sparen, werden nicht notwendige Pfade nicht beschritten. Falls beispielsweise von der Formel $F = F_1 \vee F_2$ der Formelteil F_1 bereits $True$ ausgewertet wurde, muss der rechte Formelteil F_2 nicht mehr ausgewertet werden, da F sowieso $True$ ist. Im Algorithmus 3.1.1 sind die notwendigen Handlungsschritte der *NNF*-Transformation dargelegt. Das ganze wird durch Beispiel 3.1.1 verdeutlicht.

3.1.1 Der Algorithmus QTrial

Algorithmus QTrial
Eingabe: QBF **F**
Ausgabe: True oder **False**

Auswerten **(F)**:

1. $IF \quad (F = True)$
   $return(True);$
2. $IF \quad (F = False)$
   $return(False);$
3. $IF \quad (F = F_1 \wedge F_2) \mapsto (Auswerten \quad [F_1]);$
   $IF \quad (F_1 = True)$
   $return(Auswerten \quad [F_2]);$
   $ELSE$
   $return(False);$
4. $IF \quad (F = F_1 \vee F_2) \mapsto (Auswerten \quad [F_1]);$
   $IF \quad (F_1 = True)$
   $return(True);$
   $ELSE$
   $return(Auswerten \quad [F_2]);$
5. $IF \quad (F = F_1 \implies F_2) \mapsto (Auswerten \quad [F_1]);$
   $IF \quad (F_1 = False)$
   $return(True);$
   $ELSE$
   $return(Auswerten \quad [F_2]);$
6. $IF \quad (F = F_1 \iff F_2) \mapsto (Auswerten \quad [F_1] \; und \; Auswerten \quad [F_2]);$
   $IF \quad (F_1 = F_2)$
   $return(True);$
   $ELSE$
   $return(False);$
7. $IF \quad F = \neg F_1 \mapsto (Auswerten \quad [F_1]);$
   $IF \quad (F_1 = False)$
   $return(True);$
   $ELSE$
   $return(False)$
8. $IF \quad (F = \forall x.F_1 \mapsto \underbrace{Auswerten \quad F_1[\frac{True}{x}]}_{\sigma});$
   $IF \quad \sigma = False;$
   $return(False);$
   $ELSE$
   $return(\; Auswerten \quad F_1[\frac{False}{x}] \;);$
9. $IF \quad (F = \exists x.F_1 \mapsto \underbrace{Auswerten \quad F_1[\frac{True}{x}]}_{\sigma});$
   $IF \quad \sigma \quad = True;$
   $return(True);$
   $ELSE$
   $return(\; Auswerten \quad F_1[\frac{False}{x}] \;);$

Beispiel 3.1.1.

Gegeben sei folgende QBF: $F = \exists x\ \exists y\ \forall z\ ((x \wedge \neg y) \vee \neg z)$

$Schritt\ 1 : Auswerten(\exists x.\ \exists y.\ \forall z.\ ((x \wedge \neg y) \vee \neg z))$
$Schritt\ 2 : Auswerten(\exists y.\ \forall z.\ ((True \wedge \neg y) \vee \neg z))$
$Schritt\ 3 : Auswerten(\forall z.\ ((True \wedge \neg True) \vee \neg z))$
$Schritt\ 4 : Auswerten((True \wedge \neg True) \vee \neg True))$
$Schritt\ 5 : Auswerten(\neg True \vee \neg True)$
$Schritt\ 6 : Auswerten(False \vee \neg True)$
$Schritt\ 7 : Auswerten(False)$
$Schritt\ 8 : return(False)$

Zurück zu $Schritt\ 3$ *Allquantor und Variable* z *mit* $False$ *belegen:*

$Schritt\ 3 : Auswerten(\forall z.\ ((True \wedge \neg True) \vee \neg z))$
$Schritt\ 4 : Auswerten(((True \wedge \neg True) \vee \neg False))$
$Schritt\ 5 : Auswerten(False \vee \neg False))$
$Schritt\ 6 : Auswerten(\neg False)$
$Schritt\ 7 : Auswerten(True)$
$Schritt\ 8 : return(True)$

3.1.2 Vor- und Nachteile

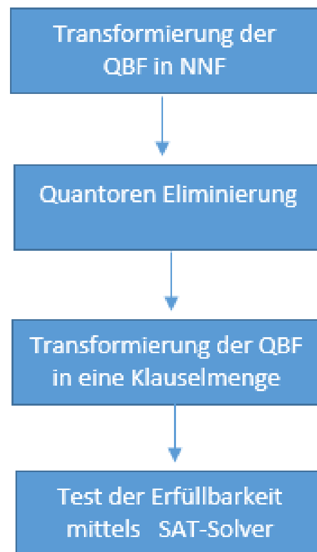
Der Platzverbrauch ist gut (d.h linear in der Größe der Eingabeformel), da der Algorithmus wie bei einer Tiefensuche immer nur einen Pfad entlang läuft, bis er den nächsten ausprobiert. Die Laufzeit des Verfahrens hingegen wird vermutlich schlechte Resultate liefern, da bei diversen Formeln alle Pfade nacheinander durchprobiert werden müssen und dafür exponentiell viel Zeit verbraucht wird. Jedoch besteht die Hoffnung, dass durch Abschneiden nicht relevanter Pfade Zwischenschritte eingespart werden können und somit die Geschwindigkeit des Verfahrens sich insgesamt verbessert. Konkrete Testergebnisse und die Analyse wie sich das Verfahren im Vergleich zu den anderen Verfahren verhält, sind in Abschnitt 4.5.2 dargelegt.

3.1.3 Korrektheit

Die Korrektheit des Verfahrens ist offensichtlich, da genau die Regeln der Semantik, die in Kapitel 2 beschrieben sind, nachgeahmt werden.

3.2 Verfahren 2: QBFtoSAT

Das *QBFtoSAT*-Verfahren besteht aus den folgenden vier Schritten:



Der erste Schritt des Verfahrens besteht darin, die eingegebene QBF in Negationsnormalform (*NNF*) zu überführen. Im Anschluss darauf können die Quantoren aus der Formel eliminiert werden. Jede beliebige QBF kann in NNF transformiert werden. Hierfür müssen folgende zwei Schritte durchgeführt werden: Im ersten Schritt werden alle Implikationen und Äquivalenzen aus der QBF entfernt. Im zweiten Schritt werden alle Negationen nach innen geschoben. Die notwendigen Handlungsschritte zur Transformierung werden im untenstehenden Algorithmus 3.2.1 *Transformation einer QBF in NNF* detailliert erläutert. Das ganze wird durch Beispiel 3.2.1 verdeutlicht.

3.2.1 NNF-Transformierung

Algorithmus: Transformation einer QBF in NNF

Eingabe: QBF F

Ausgabe: F in NNF

Schritte des Algorithmus:

Schritt 1: Eliminierung der Implikationen und Äquivalenzen aus F:

1.$F_1 \implies F_2 \mapsto \neg F_1 \vee F_2$

2.$F_1 \iff F_2 \mapsto (F_1 \implies F_2) \wedge (F_2 \implies F_1)$

Schritt 2: Negationen nach innen schieben:

1. $\neg(F_1 \wedge F_2) \mapsto \neg F_1 \vee \neg F_2$
2. $\neg(F_1 \vee F_2) \mapsto \neg F_1 \wedge \neg F_2$
3. $\neg\neg(F_1) \mapsto F_1$
4. $\neg(\forall x.F_1) \mapsto \exists x.\neg F_1$
5. $\neg(\exists x.F_1) \mapsto \forall x.\neg F_1$

Beispiel 3.2.1. *Gegeben sei die QBF* $F = \exists x.\ \exists y.\ \forall z.\ (\neg(x \iff \neg y) \vee \neg z)$, *die in NNF transformiert werden soll. Dabei müssen folgende Schritte durchgeführt werden:*

Schritt 1: Entfernung der Äquivalenz/Implikation

Eingabeformel $F = \exists x.\ \exists y.\ \forall z.\ (\neg(x \iff \neg y) \vee \neg z)$
Schritt $1.1: F = \exists x.\ \exists y.\ \forall z.\ (\neg((x \implies \neg y) \wedge (\neg y \implies x)) \vee \neg z)$
Schritt $1.2: F = \exists x.\ \exists y.\ \forall z.\ (\neg((\neg x \vee y) \wedge (y \vee x)) \vee \neg z)$

Schritt 2: Negationen nach innen schieben

Schritt $2.1: F = \exists x.\ \exists y.\ \forall z.\ ((\neg(\neg x \vee y) \vee \neg(y \vee x)) \vee \neg z)$
Schritt $2.2: F = \exists x.\ \exists y.\ \forall z.\ (((x \wedge \neg y) \vee (\neg y \wedge \neg x)) \vee \neg z)$
F *liegt in NNF vor.*

Im zweiten Schritt des Verfahrens müssen alle Quantoren aus der Formel entfernt werden. Das Eliminieren eines Existenzquantors ist relativ einfach. Dieser kann samt seiner Bindungsvariablen einfach aus der Formel gestrichen werden. Das Entfernen eines Allquantors hingegen ist mit deutlich mehr Aufwand verbunden. Sei die Formel $G = \forall x.F$ gegeben. Um den Allquantor aus G zu entfernen, muss zum einen der Allquantor mit Bindungsvariable x aus der Formel entfernt und zum anderen die Teilformel F, auf die der Allquantor verwiesen hat, konjunktiv mit der komplementären Formel $\overline{F}$ verknüpft werden. Somit wird aus $G = \forall x.F \mapsto G = F \wedge \overline{F}$. Falls die Formel Konjunktionen oder Disjunktionen enthält, wird rekursiv die linke und anschließend rechte Teilformel auf Quantoren überprüft($\psi(F_1) \otimes \psi(F_2)$). Falls die Teilformeln Quantoren enthalten geht man bei der Eliminierung analog vor. Der Algorithmus 3.2.2 erläutert die notwendigen Handlungsschritte für die Quantoreneleminierung. Das ganze verdeutlicht das Beipspiel 3.2.2.

3.2.2 Quantoreneliminierung

Algorithmus: Eliminierung der Quantoren

Eingabe: QBF F in *NNF*

Ausgabe: Aussagenlogische Formel $\psi(F)$ in NNF

Schritte des Algorithmus:

Sei ψ die Transformation der QBF:

◇ $\psi\ (\forall X.F) \mapsto \psi(F \wedge \overline{F})$ *wobei* F *die Formel* $F[\neg X/X]$ *ist*

◇ $\psi(\exists X.F) \mapsto \psi(F)$

◇ $\psi(F_1 \otimes F_2) \mapsto \psi(F_1) \otimes \psi(F_2)$ *wobei* $\otimes \in \{\wedge, \vee\}$

◇ $\psi(\neg X) := \neg X$

Beispiel 3.2.2.

Gegeben sei die QBF $F = \exists x.\ \exists y.\ \forall z.\ (((x \wedge \neg y) \vee (\neg y \wedge \neg x)) \vee \neg z)$ *in NNF, dessen Quantoren eliminiert werden sollen. Hierfür müssen folgende Schritte durchgeführt werden:*

Schritt 1: Eliminierung der Existenzquantoren

Eingabeformel $F = \exists x.\ \exists y.\ \forall z.\ (((x \wedge \neg y) \vee (\neg y \wedge \neg x)) \vee \neg z)$
Schritt $1.1 F = \exists y.\ \forall z.\ (((x \wedge \neg y) \vee (\neg y \wedge \neg x)) \vee \neg z)$
Schritt $1.2 F = \forall z.\ (((x \wedge \neg y) \vee (\neg y \wedge \neg x)) \vee \neg z)$

Schritt 2: Eliminierung des Allquantors

Schritt $2.1 : F = (\underbrace{((x \wedge \neg y) \vee (\neg y \wedge \neg x)) \vee \neg z)}_{F} \wedge \underbrace{(((x \wedge \neg y) \vee (\neg y \wedge \neg x)) \vee z}_{\overline{F}})$
$\rightarrow$ *QBF F liegt in NNF und hat keine Quantoren mehr.*

Bevor im letzten Schritt die Erfüllbarkeit der aussagenlogischen Formel F mittels eines SAT-Solvers überprüft werden kann, muss diese ausmultipliziert werden. Folgende Formel $A \vee (B \wedge C) \iff (A \vee B) \wedge (A \vee C)$ ist eine Tautologie und kann als Umformung verwendet werden. Darüber hinaus wird das Kommutativgesetz benötigt. Als Resultat erhält man eine Klauselmenge ϕ. Die Klauselmenge ϕ dient im letzten Schritt als Eingabe für den SAT-Solver. Die notwendigen Handlungsschritte zur Transformierung werden in Algorithmus 3.2.3 *Generierung einer Klauselmenge* ϕ erläutert. Das ganze wird durch Beispiel 3.2.3 verdeutlicht.

3.2.3 Übersetzung in aussagenlogische Klauseln

Algorithmus: Generierung einer Klauselmenge ϕ

Eingabe: Aussagenlogische Formel F

Ausgabe: Klauselmenge ϕ

Schritte des Algorithmus:

Wende die untenstehende Regel solange auf QBF F an, bis nicht mehr anwendbar, wobei falls notwendig die Kommutativitätsregel angewandt wird:

Distributivität der Disjunktion:
$A \vee (B \wedge C) \iff (A \vee B) \wedge (A \vee C)$

Beispiel 3.2.3. *Gegeben sei die QBF* $F = (((x \wedge \neg y) \vee (\neg y \wedge \neg x)) \vee \neg z) \wedge (((x \wedge \neg y) \vee (\neg y \wedge \neg x)) \vee z)$*, die in eine Klauselmenge* ϕ_F *transformiert werden soll. Dabei wir das Webinterface zur DPLL-Prozedur Vorlesung [uDDS13] verwendet:*

Die berechnete Klauselmenge in KNF ist:
$K = [[x, z, -y], [x, -y, -z], [z, -x, -y], [z, -y], [-x, -y, -z], [-y, -z]]$
Fuer die berechnete Klauselmenge existiert ein Modell: $[-y]$

Im letzten Schritt von Verfahren 2 dient die Klauselmenge ϕ_F als Eingabe für den *SAT*-Solver. Falls die Ausgabe des Solvers erfüllbar ist, so ist die QBF äquivalent zu *True* ansonsten zu *False*.

Beispiel 3.2.4.
Gegeben sei die im vorherigen Schritt berechnete Klauselmenge $K = [[x, z, -y], [x, -y, -z], [z, -x, -y], [z, -y], [-x, -y, -z], [-y, -z]]$*. Die Klauselmenge wird in einen SAT-Solver gegeben und man erhält folgende Ausgabe:*

Verified 6 original clauses.
restarts : 1
conflicts : 0 (0 /sec)
decisions : 3 (0.00 % random) (3 /sec)
propagations : 3 (3 /sec)
conflict literals : 0 (nan % deleted)
CPU time : 1 s

SATISFIABLE
v -1 -2 -3 0

3.2.4 Vor- und Nachteile

Ein Nachteil des *QBFtoSAT*-Verfahrens besteht darin, das die Formel unter Umständen exponentiell viel Platz einnimmt, da die Eliminierung jedes Allquantors die Formelgröße verdoppelt. Ein Vorteil ist, dass am Ende des Verfahrens ein schneller aussagenlogischer Solver verwendet werden kann und das keine Verdopplung des Platzes bei der Eliminierung von Existenzquantoren entsteht. Zu beachten ist, dass bereits die *NNF*-Berechnung zu exponentiellen Platz führen kann. Dies liegt an der Eliminierung von Biimplikationen. Konkrete Ergebnisse bezüglich der Laufzeit können von den Testresultaten aus Abschnitt 4.5 entnommen werden.

3.2.5 Korrektheit

In diesem Abschnitt wird die Korrektheit des Verfahrens begründet. Zunächst wird der Begriff der Ausdehnung einer Interpretation definiert:

Definition 3.2.5 (Ausdehnung). *Eine Interpretation I' ist eine* Ausdehnung *einer Interpretation I genau dann, wenn $I'(x) = I(x)$ für alle $x \in \mathrm{Dom}(I)$ und* $\mathrm{Dom}(I') \supseteq \mathrm{Dom}(I)$

Das folgende Lemma zeigt, dass jede gültige QBF in *NNF* durch das *QBFto-SAT*-Verfahren in eine erfüllbare aussagenlogische Formel transformiert wird.

Lemma 3.2.6. *Für jede (evtl. offene) QBF F in NNF gilt: Wenn es eine Interpretation I gibt mit $I(F) = True$, dann gibt es eine Ausdehnung I' von I mit $I'(\psi(F)) = True$*

Beweis: O.B.d.A. nehmen wir an, dass F frisch benannt ist, d.h. alle gebundenen Variablen sind paarweise disjunkt und verschieden von den freien Variablen. Sei I eine Interpretation für F mit $I(F) = True$. Wir verwenden die Induktion über das Maß (qt(F), Tiefe von F), wobei die Paare lexikographisch geordnet sind.
Als Induktionsbasis betrachten wir alle Paare der Form $(0, M)$, d.h. $qt(F) = 0$ und die Tiefe von F ist beliebig. Dann ist die Formel F quantorfrei, und somit ist F eine aussagenlogische Formel, d.h. es gilt $\psi(F) = F$ und daher $I(\psi(F)) = True$ (d.h. $I' = I$ erfüllt die Aussage).
Für den Induktionsschritt sei $qt(F) > 0$, und die Tiefe von F sei M. Wir betrachten die Fälle für F

- $F = \forall x.F_1$. Dann gilt $I(F_1[\frac{True}{x}] = 1)$ und $I(F_1[\frac{False}{x}] = 1)$ und deshalb ist $I(\overline{F_1}[\frac{True}{x}] = 1)$. Sei $I_0 = I \cup \{x \mapsto True\}$ dann gilt $I_0(F_1 \wedge \overline{F_1}) = True$.

  Da $qt(F_1 \wedge \overline{F_1}) < qt(\forall x.F_1)$ liefert die Induktionsbehauptung eine Ausdehnung I'_0 von I_0 mit $I'_0(\psi(F_1 \wedge \overline{F_1})) = True$. Da $I'_0(\psi(\forall x.F_1)) = I'_0(\psi(F_1) \wedge \psi(F_1)) = I'_0(\psi(F_1 \wedge \overline{F_1})) = True$ und I_0 eine Ausdehnung von I und I'_0 die Interpretation I_0 ausdehnt, erfüllt $I' = I'_0$ die Aussage.

- $F = F_1 \wedge F_2$. Aus $I(F_1 \wedge F_2) = True$ folgt $I(F_1) = True$ und $I(F_2) = True$. Da $qt(F_1 \wedge F_2) = qt(F_1) = qt(F_2)$ und Tiefe $(F_1) <$ Tiefe $(F_1 \wedge F_2)$ und Tiefe $(F_2) <$ Tiefe $(F_1 \wedge F_2)$ kann die Induktionshypothese auf F_1 bzw. F_2 angewendet werden. Diese liefert zwei Ausdehnungen I'_1 und I'_2 jeweils von I mit $I'_1(\psi(F_1)) = True$ und $I'_2(\psi(F_2)) = True$. Da I'_1 und I'_2 Ausdehnungen von I sind, gilt $I(x) = I'_1(x) = I'_2(x)$ für alle $x \in FV(F)$. Da die gebundenen Variablen in F_1 und F_2 paarweise disjunkt sind, gilt außerdem $(\mathrm{Dom}(I'_1) \setminus FV(F)) \cap (\mathrm{Dom}(I'_2) \setminus FV(F)) = \emptyset$ und wir können I' daher bilden als:

$$I'(x) := \begin{cases} I(x) & \text{wenn } x \in FV(F) \\ I'_1(x) & \text{wenn } x \in (FV(\psi(F_1)) \setminus FV(F)) \\ I'_2(x) & \text{wenn } x \in (FV(\psi(F_2)) \setminus FV(F)) \end{cases}$$

  Nun lässt sich leicht nachrechnen, dass die Restriktion von I' auf $\mathrm{Dom}(I'_1)$ genau I'_1 ergibt und analog ergibt die Restriktion von I' auf $\mathrm{Dom}(I'_2)$ genau I'_2. Daher gilt $I'(\psi(F_1) \wedge \psi(F_2)) = True$ und daher auch $I'(\psi(F_1 \wedge$

$F_2)) = True$. Schließlich ist I' eine Ausdehnung von I und erfüllt damit alle Anforderungen der Aussage.

- $F = F_1 \vee F_2$. Aus $I(F_1 \vee F_2) = True$ folgt, dass $I(F_1) = True$ oder $I(F_2) = True$ gilt. O.b.d.A sei $I(F_1) = True$. Da $qt(F_1) = qt(F)$ und die Tiefe von $F_1 \leq$ als die Tiefe von F ist, kann die Induktionshypothese auf F_1 angewendet werden. Diese liefert eine Ausdehnung I'_1 von I mit $I'_1(\psi(F_1)) = True$.

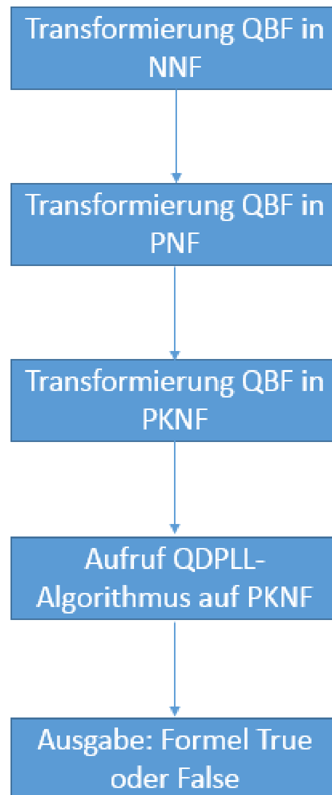
  Sei
  $$I'(x) = \begin{cases} I'_1(x) & \text{falls } x \in Dom(I'_1) \\ False & \text{sonst } (x \in FV(\psi(F_2)) \setminus FV(F)) \end{cases}$$
  Dann gilt $I'(\psi(F_1) \vee \psi(F_2)) = True = I'(\psi(F_1 \vee F_2))$. Da I'_1 eine Ausdehnung von I ist, ist auch I' eine Ausdehnung von I und daher erfüllt I' alle Anforderungen der Aussage.

- $F = \exists x.F_1$. Aus $I(\exists x.F_1) = True$ folgt $I(F_1[\frac{True}{x})] = True$ oder $I(F_1[\frac{False}{x})] = True$. O.B.d.A sei $I[F_1[\frac{True}{x}]) = True$. Sei
  $$I_0(y) = \begin{cases} I(y), & \text{wenn } y \in \mathrm{Dom}(I) \\ True, & \text{wenn } y = x \end{cases}$$
  Dann gilt $I_0(F_1) = True$. Da $qt(F_1) < qt(F)$ kann die Induktionshypothese für F_1 (mit Interpretation I_0) angewendet werden. Diese liefert eine Ausdehnung I' von I_0 mit $I'(\psi(F_1)) = True$. Da I_0 eine Ausdehnung von I ist, ist I' ebenfalls eine Ausdehnung von I. Da $I'(\psi(\exists x.F_1)) = I'(\psi(F_1)) = True$ erfüllt I' alle Anforderungen der Aussage. □

Analog kann man zeigen: Wenn $\psi(F)$ eine erfüllbare aussagenlogische Formel ist, dann ist F gültig. Auf den Beweis wird verzichtet. Dieser kann in [uMJU] nachgelesen werden.

3.3 Verfahren 3: QDPLL

Das *QDPLL*-Verfahren besteht aus den folgenden Schritten:



Die ersten drei Schritte des Verfahrens sind Vorverarbeitungsschritte. Nach erfolgreicher Durchführung dieser Schritte kann der eigentliche *QDPLL*-Algorithmus auf die *PKNF*-Formel aufgerufen werden.

3.3.1 Vorverarbeitung

Im ersten Schritt der Vorverarbeitung wird die Eingabeformel zunächst in *NNF* überführt. Diese Umformung wurde bereits beim *QBFtoSAT*-Verfahren verwendet und ist in Abschnitt 3.2.1 detailliert beschrieben. Anschließend wird die Formel in *PNF* überführt. In dieser Darstellungsform befinden sich alle Quantoren im Präfix der Formel und der Rumpf bildet eine aussagenlogische Formel. Der Algorithmus 3.3.1.1 „*Transformierung QBF in PNF*“ zeigt diesen Umformungsschritt.

3.3.1.1 PNF-Transformierung

Algorithmus: Transformierung QBF in PNF

Eingabe: QBF in NNF

Ausgabe: QBF in Pränex-Normalform PNF

Schritte des Algorithmus:

Wende folgende Regeln solange auf QBF F an, bis keine mehr anwendbar ist, dann liegt die Formel in Pränex-Normalform vor:

1. $(\exists x.F) \wedge G \rightarrow \exists x'.(F[\frac{x'}{x}] \wedge G)$
2. $(\exists x.F) \vee G \rightarrow \exists x'.(F[\frac{x'}{x}] \vee G)$
3. $F \vee (\exists x.G) \rightarrow \exists x'.(F \vee G[\frac{x'}{x}])$
4. $F \wedge (\exists x.G) \rightarrow \exists x'.(F \wedge G[\frac{x'}{x}])$

5. $(\forall x.F) \wedge G \rightarrow \forall x'.(F[\frac{x'}{x}] \wedge G)$
6. $(\forall x.F) \vee G \rightarrow \forall x'.(F[\frac{x'}{x}] \vee G)$
7. $F \vee (\forall x.G) \rightarrow \forall x'.(F \vee G[\frac{x'}{x}])$
8. $F \wedge (\forall x.G) \rightarrow \forall x'.(F \wedge G[\frac{x'}{x}])$

wobei x' eine neue Variable ist.

3.3.1.2 PKNF-Transformierung

Im letzten Schritt der Vorverarbeitung muss der Rumpf der Formel als Konjunktion von Klauseln, in eine sogenannte *PKNF* überführt werden. Hierfür kann der Algorithmus *QBFtoSAT* aus 3.2.3 auf den Rumpf der Formel aufgerufen werden. Als Resultat erhält man eine QBF in *PKNF*.

Bevor der eigentliche *QDPLL*-Algorithmus beschrieben werde kann, ist die folgende Klassifizierung der Klauseln und das Lemma zur Klassifizierung unverzichtbar(3.3.2[uMJU]).

3.3.2 Klassifizierung von Klauseln / Lemma zur Klassifizierung

Klassifizierung der Klauselmengen

1. Eine Klausel ist **PK-Wahr**, wenn sie ein Literal 1 enthält oder eine Variable sowohl positiv als auch negativ ist.
2. Eine Klausel ist **PK-Falsch**, wenn 1 nicht gilt und wenn die Klausel keine existenziell quantifizierte Variable enthält. Z.B. eine Klausel mit nur allquantifizierten Variablen ist falsch, wenn sie keine Tautologie enthält.
3. Andere Klauseln nennt man **PK-Offen**.

Lemma 3.3.1.

⋄ *Wenn eine QBF F in Pränex-Normalform eine PK-Wahre Klausel enthält, dann kann diese gestrichen werden(durch 1 ersetzt werden), was eine Formel F' ergibt, unter Erhaltung der Gültigkeit der Formel, d.h. F gültig gdw F' gültig.*

⋄ *Wenn eine QBF F in Pränex-Normalform eine PK-Falsche Klausel enthält, dann ist F nicht gültig.*

Der Beweis zum Lemma 3.3.1 kann aus [uMJU] entnommen werden.

3.3.3 Algorithmus QDPLL

Der *QDPLL*-Algorithmus benötigt als Eingabe eine QBF in *PKNF* und versucht dessen Gültigkeit zu überprüfen.

Algorithmus: **QDPLL-Algorithmus**

Eingabe: QBF $F = Q_1\ x_1.\dots Q_n\ x_n.C$**, wobei** $Q_i \in \{\forall, \exists\}$ **und** C quantorfrei ist.

Funktion: $QDPLL(Q_1\ x_1.\dots Q_n\ x_n.C)$

```
if C = ∅ // C ist eine leere Menge
   then return true; // gültig
endif
if C enthält eine PK-Falsche Klausel
   then return false; // nicht gültig
endif
if C enthält eine PK-Wahre Klausel K then
   Sei C' die Klauselmenge C' = C \ K
     then return (QDPLL( Q1 x1.... Qn xn.C')); // rekursiver Aufruf
endif

Sei Q1 x1 der erste Quantor aus dem Quantorpräfix Q1 x1.... Qn xn
if Q1 = ∀ then
   Sei C' = C, wobei alle Klauseln die x1 enthalten entfernt
   werden und aus den verbleibenden Klauseln die Literale
   ¬x1 entfernt werden.

   Sei C'' = C, wobei alle Klauseln die ¬x1 enthalten entfernt
   werden und aus den verbleibenden Klauseln die Literale
   x1 entfernt werden.
     return  (QDPLL(Q2 x2.... Qn xn.C') ∧
QDPLL(Q2 x2.... Qn xn.C''))
endif
if Q1 = ∃ then
   Sei C' = C, wobei alle Klauseln die x1 enthalten entfernt
   werden und aus den verbleibenden Klauseln die Literale
   ¬x1 entfernt werden.

   Sei C'' = C, wobei alle Klauseln die ¬x1 enthalten entfernt
   werden und aus den verbleibenden Klauseln die Literale
   x1 entfernt werden.
     return  (QDPLL(Q2 x2.... Qn xn.C') ∨
QDPLL(Q2 x2.... Qn xn.C''))
endif
if Quantorliste= ∅ // Quantorliste ist leer
   then return ERROR Formel ist nicht geschlossen;
```

Die ersten beiden If-Bedingungen des QDPLL-Algorithmus sind Abbruchbedingungen.

1.Abbruchbedingung `if` $C = \emptyset$
Falls die Klauselmenge C leer ist, wird die erste Abbruchbedingung ausgeführt. Das Programm hat dann als Ausgabe den Wert $True$. Somit ist die QBF F gültig und der QDPLL-Algorithmus endet.

2.Abbruchbedingung `if` C **enthält eine PK-Falsche Klausel**
Falls die Klauselmenge C eine PK-Falsche Klausel enthält, wird die zweite Abbruchbedingung ausgeführt. Das Programm hat dann als Ausgabe den Wert $False$. Somit ist die QBF F ungültig und der QDPLL-Algorithmus endet.

3.Bedingung `if` C **enthält eine PK-Wahre Klausel** K
Falls die Klauselmenge C eine PK-Wahre Klausel K enthält, wird die dritte Bedingung ausgeführt. Die dritte Bedingung besteht aus zwei Schritten. Im ersten Schritt wird die Klausel K aus C entfernt und somit eine neue Klauselmenge $C' = C \setminus K$ erzeugt. Im zweiten Schritt findet ein rekursiver Aufruf mit der neuen Klauselmenge C' statt $\mapsto$ `return` $(QDPLL(Q_1\, x_1 \ldots Q_n\, x_n.C'))$.

Falls jedoch die Klauselmenge C' nicht leer ist, keine PK-Falschen und PK-Wahren Klauseln enthält sind folgende Schritte relevant, wobei $Q_{1\ x_1}$ der erste Quantor aus dem Quantorpräfix $Q_{1\ x_1} \ldots Q_{n\ x_n}$ sei.

4.Bedingung if `if` $Q_1 = \exists$
Falls es sich bei dem Quantor aus dem Quantorpräfix um einen $\exists$-Quantor handelt, wird die vierte Bedingung ausgeführt. Diese sorgt dafür, dass zwei Klauselmengen C' und C'' erzeugt werden. Die Klauselmenge C' enthält alle Klauseln von C, außer diejenigen Klauseln die x_1 enthalten. Aus den verbleibenden Klauseln wird das Literal $\neg x_1$ entfernt. Die Klauselmenge C'' enthält alle Klauseln von C, außer diejenigen Klauseln die $\neg x_1$ enthalten. Aus den verbleibenden Klauseln wird das Literal x_1 entfernt. Als letztes erfolgt der rekursive Aufruf `return` $(QDPLL(Q_2\, x_2 \ldots Q_n\, x_n.C') \vee QDPLL(Q_2\, x_2 \ldots Q_n\, x_n.C''))$.

5.Bedingung if `if` $Q_1 = \forall$
Falls es sich bei dem Quantor aus dem Quantorpräfix um einen $\forall$-Quantor handelt, wird analog wie Bedingung 4 vorgegangen. Der einzige Unterschied ist, dass im rekursiven Aufruf die beiden Formelteile konjugiert werden `return` $(QDPLL(Q_2\, x_2 \ldots Q_n\, x_n.C') \wedge QDPLL(Q_2\, x_2 \ldots Q_n\, x_n.C''))$.

6.Bedingung if `if` **Quantorliste**$= \emptyset$ Falls die Quantorliste leer ist, handelt es sich um keine geschlossene Formel und eine Fehlermeldung wird ausgegeben.

Beispiel 3.3.2.
Sei QBF $F = \exists x_1.\forall x_2.\forall x_3.((x_1 \vee x_2 \vee x_3 \vee \neg x_1) \wedge (\vee \neg x_2 \vee x_3))$ *in PKNF. Nun wird der QDPLL -Algorithmus auf F angewandt, um zu bestimmen, ob die Formel gültig ist oder nicht.*

1. *QDPLL(*$\exists x_1.\forall x_2.\forall x_3.((x_1 \vee x_2 \vee x_3 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2 \vee x_3))$
2. *Prüfe ob Klauselmenge C leer ist.* $\rightarrow$ *C ist nicht leer.*
3. *Prüfe, ob Klauselmenge C PK-Falsche Klauseln enthält. Keine PK-Falschen Klauseln in C vorhanden.*
4. *Prüfe, ob Klauselmenge C PK-Wahre Klauseln enthält.*
   $\rightarrow$ *F enthält PK-Wahre Klausel* $K = (x_1 \vee x_2 \vee x_3 \vee \neg x_1)$, *somit muss die Klausel K aus F gestrichen werden* $\cancel{(x_1 \vee x_2 \vee x_3 \vee \neg x_1)}$.
5. *Nun erfolgt der rekursive Aufruf return QDPLL(*$\exists x_1.\forall x_2.\forall x_3.(\neg x_2 \vee x_3))$. *Da keine PK-Wahren bzw. PK-Falschen Klauseln vorhanden sind wird nun die Quantorliste überprüft.*
6. *Der erste Quantor aus der Quantorliste ist ein* $\exists$ *Quantor. Nun wird Klauselmenge* C' *und* C'' *erzeugt und es erfolgt folgender rekursiver Aufruf return (QDPLL(*$\forall x_2.\forall x_3.\underbrace{(\neg x_2 \vee x_3)}_{C'})\vee$ *QDPLL(*$\forall x_2.\forall x_3.\underbrace{(\neg x_2 \vee x_3)}_{C''}))$
7. *return(*$false \vee false$*)*
8. $False \rightarrow$ *Formel F ist ungültig.*

3.3.4 Vor- und Nachteile

Das *QDPLL*-Verfahren ist ein sehr direktes Verfahren und orientiert sich stark an dem *DPLL*-Algorithmus aus Abschnitt 2.4.1. Bei Verwendung der schnellen *CNF*-Methode kommt es beim Entfernen von Äquivalenzen zur Verdopplung der Formelgröße. Anders beim *QBFTtoSAT*-Verfahren, wo die Verdopplung der Formelgröße auch durch die Schachtelung von Allquantoren entsteht. Ein weiterer Vorteil ist, das bereits etliche Formeln im *QDimacs*-Format vorliegen und somit das Verfahren ohne aufwendige Vorverarbeitung direkt ausgeführt werden kann. Leider hat das Verfahren gegenüber *QTrial* den Nachteil, das die Formeln immer in *PKNF* transformiert werden müssen. Dies steigert die Laufzeit enorm. Konkrete Ergebnisse können aus den Testresultaten in Abschnitt 4.5 entnommen werden.

3.3.5 Korrektheit

Im folgenden wird die Korrektheit der einzelnen Schritte des *QDPLL*-Verfahrens begründet:

Im ersten Schritt des Verfahrens wird überprüft, ob die Klauselmenge leer ist. Falls dies der Fall ist, ist die QBF ungültig. Dieser Schritt ist korrekt, weil die leere Klauselmenge per Definition wahr ist. Im zweiten Schritt wird die Klauselmenge auf PK-Falsche Klauseln überprüft und falls vorhanden wird $False$ bzw. QBF ungültig ausgegeben. Sei $K = \{L_1, \dots, L_n\}$ eine PK-Falsche Klausel, wobei die Literale $L_1, \dots, L_n$ aus K die Variablen $x_1, \dots, x_n$ enthalten. Da alle Variablen der Klausel K allquantifiziert in der QBF sind, genügt es eine Interpretation der Variablen $\{x_1, \dots, x_n\}$ anzugeben, die die Klausel K falsch macht. Die folgende Interpretation I leistet dies:

$$I(x_i) := \begin{cases} 0, & \text{wenn } L_i = x_i \\ 1, & \text{wenn } L_i = \neg x_i \end{cases}$$

Zu beachten ist noch, dass die Interpretation I wohl-definiert ist, da eine PK-Falsche Klausel definitionsgemäß kein Literal sowohl positiv als auch negativ enthält.

Der dritte Schritt des Verfahrens, C enthält eine PK-Wahre Klausel, ist ebenfalls korrekt und diese kann folgendermaßen begründet werden. Angenommen K ist eine PK-Wahre Klausel der Formel F. Dann kann dies zwei Gründe haben, denn eine Klausel kann PK-Wahr sein, wenn sie ein Literal 1 enthält, oder eine Variable sowohl positiv als auch negativ vorkommt. Daher unterscheiden wir die beiden folgenden Fälle:

1. **Fall:** $K = \{1, L_1, \dots, L_n\}$
2. **Fall:** $K = \{\neg x, x\} \cup K'$

Da im ersten Fall alle Literale in K miteinander disjunktiv verknüpft sind und K eine 1 enthält, ist die Belegung der restlichen Literale irrelevant und die Klausel K ist wahr und kann gelöscht werden. Genauso im zweiten Fall wird die Klausel immer wahr unabhängig davon ob die Variable x $\forall$-quantifiziert oder $\exists$-quantifiziert ist: Betrachtet man die Semantik von QBFs, so sieht man, dass bei Auswertung der Klausel K die Interpretation der Variablen von K fest ist. Da jede Interpretation I die Klausel K wahr macht, folgt die Korrektheit.
Die Korrektheit der weiteren Schritte des Algorithmus lässt sich wie folgt begründen: Die erste quantifizierte Variable x_1 wird (wie in der Semantik von QBFs) per Fallunterscheidung analysiert:
Wird die Variable x_1 als `True` interpretiert, so können alle Klauseln, die x_1 enthalten, gelöscht werden, da diese nun PK-wahr sind. Alle Vorkommen von $\neg x_1$ können aus den Klauseln gelöscht werden, da sie $\neg$`True` = `False` entsprechen und durch die disjunktive Verknüpfung der Literale in den Klauseln nicht zu deren (Un-)/Gültigkeit beitragen.
Wird die Variable x_1 als `False` interpretiert, wird analog vorgegangen: Alle Klauseln, die $\neg x_1$ enthalten, sind PK-Wahr und können entfernt werden und alle Vorkommen des Literals x_1 werden aus den Klauseln entfernt.

Schließlich wird entsprechend der Semantik in zwei Fälle unterschieden:

- Ist die erste quantifizierte Variable existenzquantifiziert, so genügt es, wenn eine der beiden möglichen Belegungen von x_1 die Formel wahr macht. Daher werden die beiden rekursiven Aufrufe durch ein Oder verbunden.
- Ist die erste quantifizierte Variable allquantifiziert, so muss die Formel für beide Belegungen von x_1 wahr werden. Daher werden die beiden rekursiven Aufrufe durch ein Und verbunden.

3.4 Optimierung der Verfahren

Die Verfahren können durch zwei unterschiedliche Optimierungsarten verbessert werden. Einerseits kann die Performance des *QDPLL*-Algorithmus gesteigert werden, indem sogenannte PK-Unit Klauseln berechnet werden und somit die Formel im Besten Fall kleiner wird. Andererseits kann man in einem weiteren Optimierungsansatz, der sowohl *QDPLL* als auch *QBFtoSAT* betrifft, zusätzlich die *CNF*-Berechnung optimieren.

3.4.1 Erkennen und Behandeln von PK-Unit-Klauseln

Wir definieren zunächst, wann eine Klausel als PK-Unit bezeichnet wird:
Eine Klausel K wird als PK-Unit Klausel bezeichnet [uMJU], gdw. wenn sie genau ein existenzielles Literal hat (x oder $\neg x$), keine Konstanten enthält und alle anderen (universellen) Literale der Klausel haben eine Variable y, die rechts von x im Quantorpräfix steht.

Für eine PK-Unit-Klausel genügt es, nur den Fall den Fall $F[1/x]$ (wenn x das Literal ist) bzw. $F[0/x]$ (wenn $\neg x$ das Literal ist) zu betrachten. Algorithmisch können wir daher entsprechende Operationen auf der Klauselmenge durchführen:

- Falls x das Literal ist: Entferne alle Klauseln, die x enthalten, und lösche alle Vorkommen von $\neg x$.
- Falls $\neg x$ das Literal ist: Entferne alle Klauseln, die $\neg x$ enthalten, und lösche alle Vorkommen von x.

Begründung der Korrektheit:
Gegeben sei eine PK-Unit Klausel $K = \{L_x, L_{y_1}, \ldots, L_{y_n}\}$, wobei L_x $\{x$ oder $\neg x\}$ und L_y y_i oder $\neg y_i$ entspricht und der Quantorpräfix der Formel $\ldots \exists x \ldots \forall y_1 \ldots \forall y_n \ldots$ lautet. Sei daher $F := \ldots \exists x \ldots \forall y_1 \ldots \forall y_n \ldots . (\{K\} \cup C')$. Sei $v = 1$ für den Fall, dass x das Literal in K ist und $v = 0$ für den Fall, dass $\neg x$ das Literal in K ist.

Für die Korrektheit der Behandlung von PK-Unit-Klauseln sind zwei Richtungen zu zeigen:

- Wenn $F[v/x]$ gültig ist, dann ist auch F gültig.
  Das ist offensichtlich, da x existenziell-quantifiziert ist und daher die Belegung von x gewählt werden darf.

- Wenn F gültig ist, dann ist $F[v/x]$ gültig.
  Sei F gültig. Betrachte die Semantik für QBFs: Die Auswertung von F muss irgendwann $I(\forall y_1 \dots \forall y_n \dots (\{K\} \cup C'))$ berechnen, wobei die Interpretation I die Werte der Variablen festlegt, die links von y_1 stehen. Da $y_1, \dots, y_n$ all-quantifziert sind, muss $I'(\dots(\{K\} \cup C')) = 1$ gelten, für alle I', die I um die Belegung der Variablen $y_1, \dots, y_n$ erweitern. Insbesondere muss dies auch für I' gelten mit:

  $$I'(y_i) := \begin{cases} 0, & \text{wenn } L_{yi} = y_i \\ 1, & \text{wenn } L_{y_i} = \neg y_i \end{cases}$$

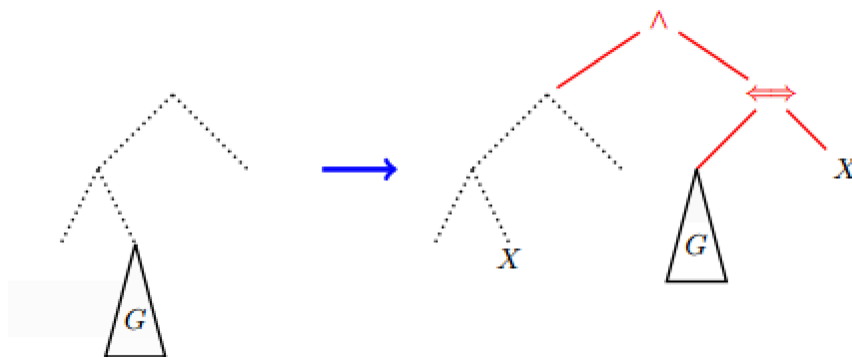
  Da F gültig ist muss $I'(K) = I'(x) = 1$ gelten. Dies ist jedoch nur möglich für:

  $$I'(x) := \begin{cases} 0, & \text{wenn } L_x = x \\ 1, & \text{wenn } L_x = \neg x \end{cases}$$

  Daher zeigt I' auch, dass $F[v/x]$ gültig ist.

3.4.2 Schnelle CNF-Methode

Die Berechnung der *CNF* in Lösungsverfahren 2 und 3 kann optimiert werden. Das exponentielle Anwachsen bei der *CNF*-Berechnung ist exponentiell in der Tiefe der Formel. Mit einem Trick lässt sich die Formel vorverarbeiten sprich flach klopfen. Dabei werden komplexe Subformeln iterativ durch neue Variablen abgekürzt. Sei $F[G]$ eine Formel mit der Subformel G. Dann wird daraus $(X \iff G) \wedge F[X]$, wobei X eine neue aussagenlogische Variable ist (siehe Abbildung 3.4.2). Das ganze ist graphisch in Abbildung 4.1 dargelegt. Der Algorithmus zur schnellen *CNF*-Berechnung kann aus [uMJU] entnommen werden.
Abbildung 3.4.2 [uMJU]



Kapitel 4

Implementierung und experimentelle Analyse

In diesem Kapitel wird die Implementierung der Inferenzverfahren und die experimentelle Analyse vorgestellt. Bevor auf die Implementierung eingegangen werden kann, wird im ersten Schritt die objektorientierte Programmiersprache Java erläutert (siehe 4.1). Dabei wird die Entwicklung, Verwendung, Programmierkonzepte und weitere relevante Eigenschaften Javas dargelegt. Zudem werden die für die Implementierung notwendigen Datenstrukturen definiert. Anschließend wird in 4.2 die Klassenhierarchie und Formeldarstellung diskutiert. Dabei wird eine zweite auf Listen basierende Formeldarstellung herangezogen und begründet warum diese den Anforderungen nicht gerecht wird und syntaktisch ungültige Formeln erzeugt. Daraufhin werden in 4.3 die Implementierungen der drei Lösungsverfahren beschrieben. Insbesondere wird auf ihre Methoden eingegangen und signifikante Merkmale hervorgehoben. Im nächsten Abschnitt 4.4 werden Ein- und Ausgabeschnittstellen definiert. Dabei sind die textuelle Formelausgabe, das Einlesen von Formeln und die Konstruktion einer Graphische Benutzeroberfläche zum Lösen von QBFs Bestandteil. Zum Schluss werden in 4.5 die Ergebnisse der experimentellen Analyse erörtert. Hierbei werden die Verfahren mit verschiedenen QBF-Instanzen auf ihre Laufzeit und Lösungsverhalten getestet.

4.1 Programmiersprache Java und verwendete Bibliotheken

Java ist eine objektorientierte Programmiersprache mit der Anwendungen entwickelt werden können. Java ist weitverbreitet und beinhaltet ähnliche Konzepte, die auch in anderen Programmiersprachen verwendet werden (siehe [KH14]). Ziel ist es hier dem Leser einen groben Einblick in die Sprache zu gewähren. Dieses Wissen ist für das Verständnis der Klassenhierarchie in 4.2 und der Implementierung der Lösungsverfahren in 4.3 notwendig. Hierfür werden im folgenden Besonderheiten, Anwendungsgebiete und Datenstrukturen Javas beschrieben (siehe Abschnitt 4.1.3. Als Installationspaket stellt die Firma Oracle ein kostenloses *Standard Development Kit*, kurz SDK genannt,

zur Verfügung. Das meistgenutzte und hier verwendete SDK ist das *Java Development Kit*(JDK). Das JDK beinhaltet die Laufzeitumgebung namens *Java Runtime Environment*(JRE), den Java Compiler und weitere wichtige Entwicklungswerkzeuge (siehe [Ull12]). Alle in Kapitel 3 vorgestellten Lösungsverfahren wurden mit Java in der Entwicklungsumgebung Eclipse programmiert und getestet. Hierfür wurde die Version Eclipse Luna Version 4.4 genutzt. Laut [Ste07] ist Eclipse mit knapp 65% Marktanteil Marktführer im Bereich der IDEs. Eclipse bietet einige sehr nützliche Hilfsmittel, die das Programmieren enorm erleichtern. Eine bessere Übersicht bei der Bearbeitung mehrerer Quellcodes, das automatische Kompilieren und Ausführen von Programmen oder der Hinweis auf unterschiedliche Fehler durch farbliche Markierung im Quelltext sind einige dieser Erleichterungen. Zudem unterstützt Eclipse nicht nur die Sprache Java, sondern ist für etliche weitere Programmier– bzw. Skriptsprachen gut geeignet (siehe [Kü07]).

4.1.1 Entwicklung von Java

Die Entwicklung der Sprache Java ist äußerst spannend und wurde maßgeblich durch die Entwicklung des World Wide Webs geprägt. Das ursprüngliche Ziel, das mit der Entwicklung Javas verfolgt wurde, bestand darin elektronische Haushaltsgeräte zu programmieren. Nichtsdestotrotz wurde diese Entwicklung durch bestimmte Ereignisse in andere Richtungen gelenkt. Die Tabelle 2.4.3 stellt signifikante Ereignisse der Entwicklung dar (siehe [Dar09]).

Tabelle 2.4.3

Jahr	Ereignis
1991	Sun Microsystems gründet Green-Projekt, um EDV Trends aufzuspüren.
Ende 1991	Entwicklung des Java Vorläufer namens OAK(Object Application Kernel). Mit OAK war die Entwicklung robuster Anwendungen möglich.
1993	Marc Andreessen entwickelt graphischen Browser NCSA Mosaic.
1994	WWW wird immer wichtiger / Sun erkennt Potenzial von WWW und entwickelt WebRunner mit Hilfe von OAK. Kein großer Erfolg von WebRunner.
1995	Entwicklung der Programmiersprache Java, die zuvor OAK hieß.
1996	Erste Version JDK(Java Development Kit) verfügbar. Programmierer haben Möglichkeit Java-Applikationen und Web-Applets zu erstellen.
1996	Neue Version des Netscape-Browsers, die Java unterstützt erscheint. Dies führt zum Durchbruch Javas.
1997-2014	Java etablierte sich zur Programmiersprache des WWW und verbreitete sich rasant. Mehr als neun Millionen Entwickler weltweit und somit einer der erfolgreichsten Programmiersprachen überhaupt. Ob in Laptops, Spielkonsolen, Supercomputern, Mobiltelefonen oder dem Internet, Java kommt überall zum Einsatz.

4.1.2 Verwendung von Java

Das Einsatzgebiet der Programmiersprache Java ist äußerst vielfältig und erstreckt sich vom Einsatz in Küchengeräten bis hin zum Einsatz in wissenschaftlichen Supercomputern. Die untenstehende Statistik gibt konkrete Informationen zur Verwendung Javas wieder.

Statistik zur Verwendung von Java laut [Ora15a]:

- 97% aller Unternehmensdesktops nutzen Java.
- 89 % aller Desktops (oder Rechner) in den USA nutzen Java.
- 9 Millionen Java-Entwickler weltweit.
- Entwicklungsplattform Nummer 1.
- 3 Milliarden Mobiltelefone nutzen Java.
- 100% aller Blu-Ray-Player werden mit Java ausgeliefert.
- 5 Milliarden Java-Karten im Gebrauch.
- 125 Millionen Fernsehgeräte nutzen Java.
- 5 der Top 5 Original Equipment Manufacturer liefern Java ME aus.

4.1.3 Java und seine Eigenschaften

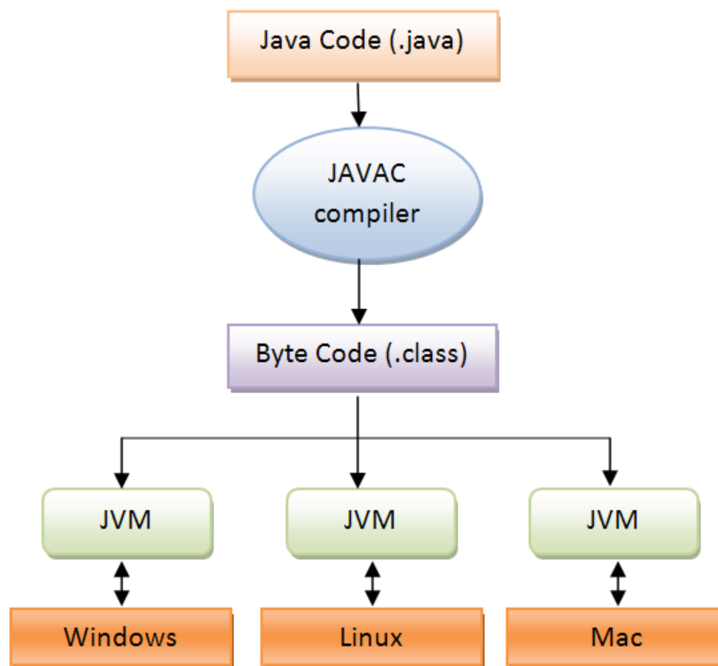
Java ist eine einfache und objektorientierte Programmiersprache. Die Einfachheit der Sprache macht sich vor allem durch die Wahl der Sprachkonstrukte bemerkbar, die das Programmieren erleichtern. Dies kommt zustande, da Java die komplexe Zeigerarithmetik aus C/C++ nicht mit übernommen hat. Bestandteil von Java sind essentielle objektorientierte Konzepte wie Klassen, Objekte und Vererbung [Ull14] [KR10].

Folgendes Zitat fasst die Merkmale von Java äußerst kompakt und konkret zusammen [KH14]:

> Java soll eine einfache, objektorientierte, verteilte, interpretierte, robuste, sichere, architekturneutrale, portable, performante, nebenläufige, dynamische Programmiersprache sein.

Im folgenden werden einige Schlüsselmerkmale Javas näher betrachtet. Ein Schlüsselmerkmal Javas ist die Plattformunabhängigkeit (siehe Abbildung 4.1.3). Unter Plattformunabhängigkeit versteht man die Eigenschaft, dass ein und das selbe Programm auf unterschiedlichen Plattformen und unter verschiedenen Betriebssystemen laufen kann. Beim Kompilieren von Java-Code wird plattformunabhängiger Bytecode erzeugt, der von einer *Java Virtual Machine* (JVM) ausgeführt werden kann. Die Plattformunabhängigkeit hat zum einen robustere Programme zur Folge, aber andererseits erhebliche Geschwindigkeitsverluste (siehe [KR10]).

Abbildung 4.1.3 [Pla]



Eine weitere Schlüsseleigenschaft ist das **Multi-Threading**. Mit Multi-Threading ist es möglich, dass mehrere Vorgänge parallel ausgeführt werden können. Zudem beinhaltet Java eine sehr umfangreiche **Klassenbibliothek** mit über 200 Paketen und einer riesigen Anzahl an Entwicklern. Eine weitere Eigenschaft die Java ausmacht, ist der hohe Sicherheitsstandard. Gerade in der heutigen Zeit spielen sicherheitsrelevante Aspekte eine zentrale Rolle und sind innerhalb der Softwareentwicklung ausschlaggebend. Das von Java verwendete Security-Modell garantiert den sicheren Programmablauf auf verschiedensten Ebenen. Auf Compiler Ebene wird der Bytecode bevor er zur JVM gelangt genau überprüft. Auf Programmebene sorgt der Security-Manager, dass der Zugriff auf Systemressourcen wie z.B. auf das Dateisystem oder den Netzwerk-Ports nur mit entsprechenden Rechten erlaubt wird [Kar15].

4.1.4 Wichtige Konzepte

Ein wichtiges Konzept von Java bildet das objektorientierte Programmierparadigma. Die objektorientierte Sichtweise auf die Welt, auch kurz *OOP* genannt, ist im Vergleich zu konventionellen Programmiersprachen unterschiedlich. Für die *OOP* besteht die Welt aus Objekten die miteinander kommunizieren. Ein Objekt ist die Ausprägung einer Klasse und Klassen sind abstrakte Beschreibungen von Objekten. Man sagt auch, dass Klassen Baupläne von Objekten

sind (siehe [Kok99]). Ein zweites wichtiges Konzept bilden abstrakte Klassen. Eine abstrakte Klasse ist eine Klasse von der keine Objekte erzeugt werden können. Sie dienen als Superklassen und geben die Struktur einer Klassenhierarchie vor. In den abgeleiteten Klassen werden eine oder mehrere abstrakte Methoden implementiert. Erst wenn alle Methoden einer abstrakten Klasse implementiert sind, ist die Klasse konkret. Abstrakte Methoden unterscheiden sich von herkömmlichen Methoden dahingehend, das sie nur die Deklaration der Methoden, jedoch keine konkrete Implementierung besitzen d.h. ihr Rumpf ist leer. Abstrakte Klassen sowie abstrakte Methoden werden durch das Schlüsselwort *abstract* eingeleitet [Sim07].

Beispiel 4.1.1.
Ein Programm soll verschiedene geometrische Figuren wie beispielsweise Rechtecke, Kreise, Quadrate usw. verwalten. Diese Figuren sind sich zwar sehr ähnlich, aber unterscheiden sich beispielsweise bei der Berechnung des Flächeninhalts oder des Umfangs. Daher muss jede Methode speziell definiert werden. Der untenstehende Codeabschnitt zeigt die abstrakte Oberklasse geoFigur und eine konkretere Klasse Kreis.

```
Abstrakte Superklasse geoFigur

public abstract class geoFigur{
public abstract double umfang();
public abstract double flaeche();

}

Konkrete Implementierung der Klasse Kreis

public class Kreis extends Figur{
    public double r;
    public Kreis(double radius){
        this.r=radius;
     }
    public double flache(){
       return Math.PI * r * r;
     }
public double umfang() {
      return 2 * Math.PI * r;
     }
}
```

Weitere relevante Konzepte der OOP, die hier nicht näher beschrieben werden, sind die Polymorphie, Vererbung und Datenkapselung (siehe [Bec08]).

4.1.5 Verwendete Bibliotheken

Java besitzt eine äußerst umfangreiche Bibliothek mit mehr als 200 Paketen [Ora15b]. Ein Paket besteht aus einer Sammlung von kleineren Programmen, um häufig wiederkehrende Aufgaben zu erledigen. Beispielsweise beinhaltet das Paket java.net z.B. Klassen zum Aufbau von Client- und Serversystemen, die sich über TCP beziehungsweise IP mit dem Internet verbinden lassen wie in [Ull14] beschrieben. Im Folgenden werden Pakete beschrieben, die für die Implementierung der Inferenzverfahren genutzt wurden.

Paket java.util Iteratoren
Ein Iterator ist ein Zeiger, mit dem die Elemente einer Containerklasse durchlaufen werden können. Er befindet sich im Paket java.util. Mit der Methode *next()* kann das erste Element des Containers zurückgegeben werden. Um zu schauen, ob der Container noch weitere Elemente enthält, kann die Methode *hasNext()* verwendet werden. Diese gibt den Wahrheitswert *True* zurück, falls ein weiteres Element vorhanden ist. Als weiterer Iteratorentyp steht der ListIterator zur Verfügung. Mit diesem sind folgende Handlungen möglich [KH14]:

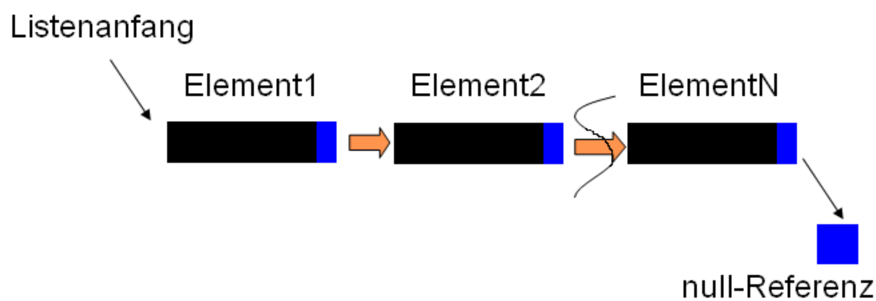
1. Vorwärts- und Rückwärtsiterationen
2. Rückgbe des Index des aktuellen Elementes
3. Einfügen eines Elementes an einer gegebenen Position

Paket java.util HashMap
HashMap ist eine Java-Klasse aus dem Paket *java.util* (siehe [Sch11]). Das Paket *java.util* ist Bestandteil von vielen Anwendungen und enthält Klassen für Datenstrukturen, Zeitangaben und Internationalisierungen. Eine *HashMap* ist eine Datenstruktur, in der viele Elemente unsortiert abgespeichert werden können. Jedes Element einer *HashMap* verfügt über einen Schlüssel, mit dem effizient auf das Element zugegriffen werden kann. Man spricht in diesem Kontext von Schlüssel-Werte Paaren. Ein Schlüssel kann ein beliebiges Objekt sein, mit der Beschränkung, dass ein und derselbe Schlüssel niemals für zwei Elemente existiert. Beim Einfügen eines Schlüssel-Werte Paares wird für jeden Schlüssel intern ein Hashwert berechnet, sprich eine Adresse in einer Tabelle zugeordnet, in dem das Element bzw. der Wert abgelegt wird. Java besitzt ein internes Verfahren, um Hashwerte optimal zu berechnen. Mit dem Konstruktor *HashMap()* lässt sich eine leere *HashMap* generieren. Weitere elementare Methoden sind *put*, mit der Elemente in eine *HashMap* eingefügt und die Methode *remove*, mit der Elemente aus der *HashMap* entfernt werden können. Zum Schlüsselvergleich werden zunächst die Hashcodes mit der Methode *hashcode()* verglichen. Bei gleichem Hashcode wird im Anschluss die Methode *equals()* aufgerufen. Beide Methoden können in den entsprechenden Klasse überschrieben werden. Dasselbe gilt für den Elementvergleich bei Hashsets.

Paket jave.util HashSet
Die Datenstruktur *Set* wird in Java durch das Interface *java.util.Set* abgebildet(siehe [uBP15a]). Die Elemente einer *Set* unterliegen keiner bestimmten Reihenfolge. Zudem kann eine Set keine doppelten Einträge enthalten. Eine *HashSet* ist eine spezielle Art von *Set*, die die Elemente in einer schnellen hashbasierten Datenstruktur verwaltet. Dabei wird der Hashcode intern von Java berechnet, sodass die Elemente effizient in die Datenstruktur eingefügt werden können. Die Klasse *HashSet* verfügt über keine Methoden, um direkt auf die Elemente zuzugreifen. Um trotzdem Zugriff auf die Elemente zu gewährleisten, verwendet man Iteratoren.

Paket java.util LinkedList
Die Datenstruktur *LinkedList* ist eine Java-Klasse aus dem Paket *java.util* (siehe [uBP15b]). Die Listenelemente einer LinkedList sind mit dem jeweiligen Vorgänger bzw. Nachfolger verbunden. Das letzte Element der Liste verweist auf die null-Referenz (siehe Abbildung 4.1.5).

Abbildung 4.1.5



Das Einfügen bzw. Löschen eines Elementes funktioniert schneller als die gleichartigen Operationen bei einer *ArrayList* Datenstruktur. Dafür beansprucht das Suchen eines Elementes mehr Zeit. Im schlimmsten Fall müssen sogar alle Elemente der Liste durchlaufen werden. Genau wie bei *HashSets* kann man mit einem Iterator die Liste durchlaufen, um einzelne Elemente auszugeben. Anders als bei *HashSets* hat man direkten Zugriff auf die Elemente und kann mit einer for-Schleife die einzelnen Elemente ausgeben.

java.util Scanner

Mit der Klasse *Scanner* aus dem Paket *java.util* ist es möglich Eingaben von der Konsole, aus Dateien oder Strings zu lesen [Ora15b]. Die Eingabe wird als Folge von Tokens interpretiert, die durch Leerzeichen, Tabs oder Zeilenvorschübe getrennt sind. Wichtige Methoden sind *next()* und *nextInt()*. Die Methode *next()* liest eine Zeichenkette ein und die Methode *nextint()* liest einen Integer-Wert ein.

java.lang Klasse Math
Mathematische Funktionen befinden sich in der der Klasse *Math*. Das Verwenden der *Math*-Klasse benötigt keinen zusätzlichen import. Sämtliche Methoden der Klasse sind static und lassen sie sich somit durch Vorstellen des Klassennamens aufrufen (siehe [Ora15b]).

```
Die Methode Math.abs gibt den Absoluten Betrag einer Zahl wieder
public class Test{

   public static void main(String args[]){

      double a = -100;

      System.out.println(Math.abs(a));
   }
}

Result: 100.0
```

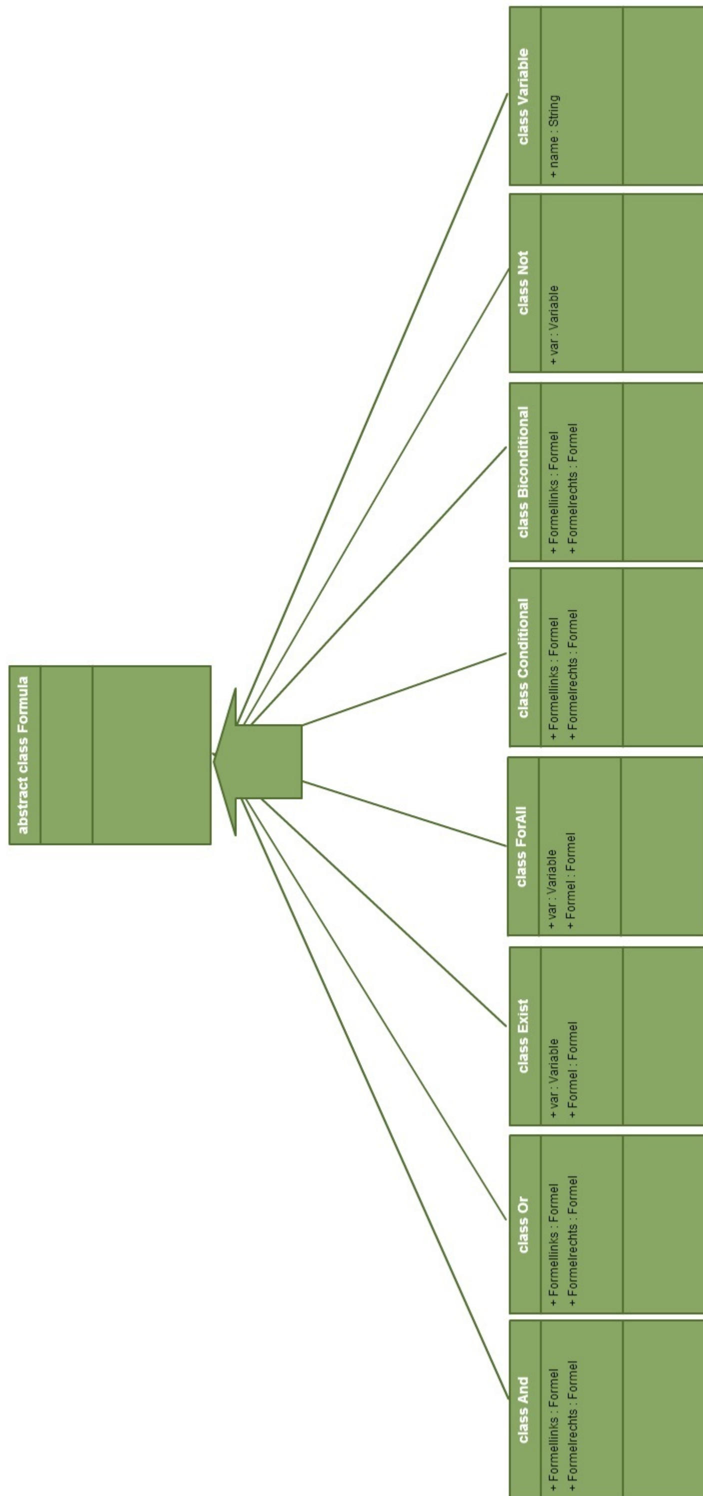
4.2 Klassenhierarchie und Formeldarstellung

Für den Entwurf eines geeigneten Klassenkonzeptes muss zunächst das gemeinsame Verhalten beliebiger QBFs extrahiert werden. Die Gemeinsamkeiten sind deutlich und daher einfach festzustellen. QBFs können Kombinationen aus Quantoren, Variablen und Junktoren sein, die nach bestimmten syntaktischen Regeln gebildet werden dürfen. In Java liegt es daher nahe für jeden Junktor und Quantor eigene Klassen zu definieren. Natürlich muss es auch eine Klasse Variable geben, um aussagenlogische Variablen abzubilden. Um jedoch das gemeinsame Verhalten darzustellen, bietet Java das Konzept abstrakter Klassen. Die Idee hierbei ist es, gemeinsames Verhalten in eine abstrakten Oberklasse unterzubringen und mit dieser Klasse die Hierarchie zu beginnen. Die konkreten Unterklassen implementieren somit die Methoden der abstrakten Oberklasse und bestimmen so ihr Verhalten. Hier ist die abstrakte Oberklasse die Klasse Formel und die davon abgeleiteten Unterklasse sind die folgenden:

- *class And* ($\wedge$)
- *class Or* ($\vee$)
- *class Conditional* ($\implies$)
- *class Biconditional* ($\iff$)

- $class\ Not$ ($\neg$)
- $class\ Exist$ ($\exists$)
- $class\ Forall$ ($\forall$)
- $class\ Variable$

Im nächsten Schritt des Entwurfes müssen die Attribute der von Formel abgeleiteten Unterklassen definiert werden. Die Unterklassen *And, Or, Conditional* und *Biconditional* entsprechen zweistelligen Junktoren die zwei Formelteile miteinander verknüpfen. Daher haben diese Klassen als Attribute Formellinks und Formelrechts vom Typ Formel. Die Quantorklassen *Exist* und *ForAll* verwenden die Attribute var vom Typ Variable, der ihrer Quantorvariablen entspricht und das Attribut F vom Typ Formel, der ihren Rumpf darstellt. Die Unterklasse *Not* stellt einen einstelligen Junktor dar, der sich auf eine Formel bezieht. Daher besitzt dieser das Attribut F vom Typ Formel. Die letzte Unterklasse Variable beinhaltet als Attribut einen Namen vom Typ String, der einer konkreten Variable bzw. Aussage entspricht. Mit diesem Klassenkonzept kann jede beliege QBF in Java generiert werden. Die konkreten Methoden werden in der Implementierung beschrieben. Das untenstehende UML-Diagramm stellt die Klassenhierarchie mit konkreten Attributen dar.



abstract class Formula
class Variable
+ name : String
class Not
+ var : Variable
class Biconditional
+ Formellinks : Formel
+ Formelrechts : Formel
class Conditional
+ Formellinks : Formel
+ Formelrechts : Formel
class ForAll
+ var : Variable
+ Formel : Formel
class Exist
+ var : Variable
+ Formel : Formel
class Or
+ Formellinks : Formel
+ Formelrechts : Formel
class And
+ Formellinks : Formel
+ Formelrechts : Formel

Ein alternatives Konzept um Formeln darzustellen, das nicht auf abstrakten Klassen basiert, könnte folgendermaßen aussehen:

```
1. Junktornamen werden als String gespeichert
2. Formeln können in eine Liste gespeichert werden -> Liste von Argumenten
3. Variablen können in eine Liste gespeichert werden -> Liste von Variablen
```

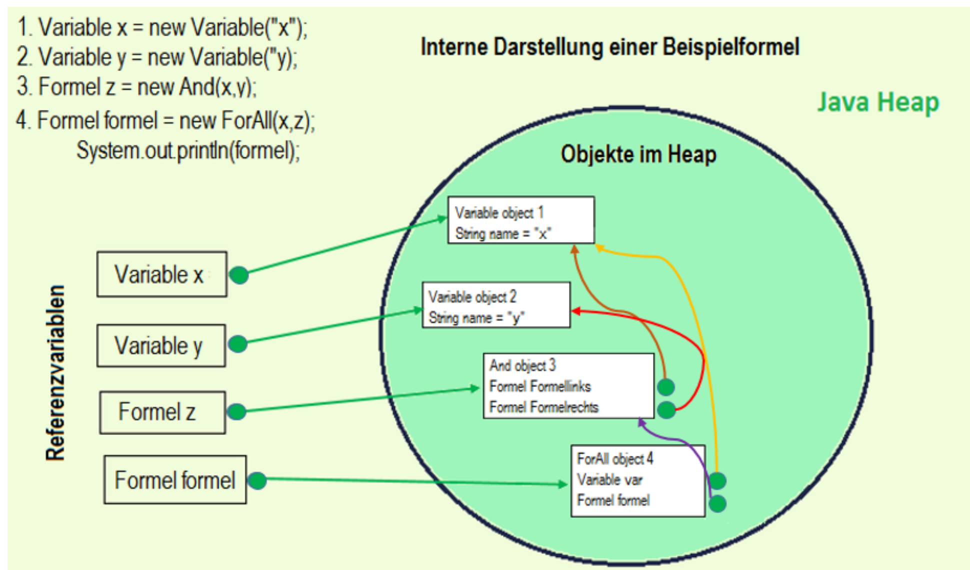
Im Vergleich zum ersten Ansatz ist dieses Darstellungskonzept viel einfacher aufgebaut. Die Einfachheit liegt in der der nicht aufwendigen Programmierung. Zudem lassen sich Änderungen, aufgrund der einheitlichen Darstellung, deutlich schneller durchführen. Leider birgt die Darstellungsform aber auch den Nachteil, das sich syntaktisch nicht korrekte Formeln erzeugen lassen.

Alles in allem ist im ersten Ansatz zwar der Programmieraufwand höher und das Konzept deutlich komplexer, jedoch können keine syntaktisch ungültigen Formeln generiert werden. Deshalb ist dieses Konzept vorzuziehen.

4.2.1 Interne Formeldarstellung in Java

Um ein genaueres Verständnis der Formeldarstellung zu erhalten, sollte die interne Speicherorganisation bei der Erzeugung von Formeln, in diesem Fall der Formel $F = \forall x.(x \wedge y)$, näher betrachtet werden. Bei der Generierung der Formel F werden Referenzvariablen, Objekte und Pointer zwischen diesen gesetzt. Die untenstehende Abbildung gibt das Zusammenspiel aus diesen Komponenten graphisch wieder 4.2.1.

Abbildung 4.2.1



1. **Speicher:** Generierung von Referenzvariable x vom Typ Variable und von Variable-Objekt1. Referenzvariable und Objekt werden miteinander verknüpft.

2. **Speicher:** Generierung von Referenzvariable y vom Typ Variable und von Variable-Objekt2. Referenzvariable und Objekt werden miteinander verknüpft.

3. **Speicher:** Generierung von Referenzvariable z vom Typ Formel und von And-Objekt3. Referenzvariable z und And-Objekt werden miteinander verknüpft. Zudem referenzieren die Attribute des And-Objekts Formellinks auf Variable-Objekt1 und Formelrechts auf Variable-Objekt2

4. **Speicher:** Generierung von Referenzvariable *formel* vom Typ Formel und von ForAll-Objekt4. Referenzvariable *formel* und ForAll-Objekt werden miteinander verknüpft. Zudem referenzieren die Attribute des ForAll-Objekts var auf Variable-Objekt1 und formel auf And-Objekt3.

4.3 Implementierung der Inferenzverfahren

Im folgenden Abschnitt befinden sich die Beschreibungen zur Implementierung der Inferenzverfahren *QTrial*, *QBFtoSAT* und *QDPLL*. Alle drei Verfahren sind in Java programmiert und verwenden die in Abschnitt 4.1.5 beschriebenen Datenstrukturen. Zu Beginn wird in 4.3.1 die Implementierung des naiven QTrial-Verfahrens erläutert. Das *QTrial*-Verfahren versucht eine Lösung, durch simples Ausprobieren der Formel, mit unterschiedlichen Belegungen, zu finden. Im Anschluss darauf wird in 4.3.2 die Implementierung des *QBFto-SAT*-Verfahrens beschrieben. Hierbei spielen insbesondere die Schritte zur Transformierung einer QBF in eine aussagenlogische Formel und der SAT-Solver *SAT4J* eine zentrale Rolle. Zum Schluss des Abschnitts wird auf die Implementierung des *QDPLL*-Verfahrens eingegangen. Beim *QDPLL*-Verfahren stehen die Klassifizierung der Klauseln und die Optimierung des Verfahrens durch PK-Unit Berechnung und Verwendung der schnellen *CNF* im Vordergrund (siehe 4.3.2).

4.3.1 Implementierung QTrial-Verfahren

Das *QTrial*-Verfahren besteht aus einem rekursiven Algorithmus (siehe Abschnitt 3.1), der durch simples Ausprobieren versucht, die Gültigkeit einer QBF zu bestimmen. Dieser Algorithmus wurde folgendermaßen implementiert:

Die Klasse Formel enthält die Methode $formelAuswerten()$, mit der eine beliebige QBF rekursiv ausgewertet werden kann. Die Auswertung erfolgt entlang des Syntaxbaums und hat als Resultat einen Wahrheitswert, der entweder *True* oder *False* sein kann. Die Methode ist ebenfalls in sämtlichen Unterklassen implementiert und unterscheidet sich in ihrer Auswertungsstrategie. Wann welcher Formelteil ausgewertet wird, hängt von den logischen Bedingungen ab. Diese sind im Grundlagenteil in Abschnitt 2 beschrieben. Falls die Formel eine Konjunktion enthält, wie es beispielsweise bei der Formel $F = F_1 \wedge F_2$ der Fall

ist, wird *formelAuswerten()* der Klasse *And* aufgerufen. Diese wertet zunächst rekursiv den linken Formelteil F_1 aus. Wenn das Resultat der Auswertung *True* ist wird auch der rechte Formelteil F_2 ausgewertet. Falls F_2 auch *True* ausgewertet wird, ist das Gesamtergebnis von F *True*. In allen anderen Fällen ist das Resultat der Auswertung *False*. Dies wird durch folgenden Programmcode illustriert:

```
public Wahrheitswert formelAuswerten() {
  Wahrheitswert links = this.FormelLinks.formelAuswerten();
    if (links.wahrheitswert == true) {
      Wahrheitswert rechts = this.FormelRechts.formelAuswerten();
      return rechts;
     }
    else {
      return new Wahrheitswert(false);
     }
}
```

Die Implementierung von *formelAuswerten()* der Klassen *Or*, *Conditional* und *Biconditional* unterscheidet sich nur marginal von der obigen. Bei den Klassen *Or* und *Conditional* wird als erstes der linke Formelteil ausgewertet. Abhängig vom Resultat der Auswertung wird entweder direkt *True* ausgegeben oder der rechte Formelteil ausgewertet. Natürlich könnte man auch als erstes den rechten und anschließend den linke Formelteil auswerten. Der Code für die Auswertung ist der folgende:

```
public Wahrheitswert formelAuswerten() {
    Wahrheitswert links = this.FormelLinks.formelAuswerten();
     if (links.wahrheitswert == true) {
       return new Wahrheitswert(true);
     } else {
       Wahrheitswert rechts = this.FormelRechts.formelAuswerten();
       return rechts;
     }
}
```

Bei der Klasse *Biconditional* muss sowohl der rechte als auch linke Formelteil ausgewertet werden. Die Auswertung wird durch folgenden Code beschrieben:

```
public Wahrheitswert formelAuswerten() {
   Wahrheitswert links = this.FormelLinks.formelAuswerten();
   Wahrheitswert rechts = this.FormelRechts.formelAuswerten();
   if (links.wahrheitswert == rechts.wahrheitswert) {
     return new Wahrheitswert(true);
    } else {
     return new Wahrheitswert(false);
    }
}
```

Einen weiteren Fall stellt *formelAuswerten()* der Klasse *Not* dar. Angenommen *formelAuswerten()* wird auf die Formel $F = \neg F_1$ aufgerufen. Dann wird die Teilformel F_1 rekursiv ausgewertet. Das Resultat von F ist immer das Komplement vom Wahrheitswert der Auswertung von F_1. Dies wird durch folgenden Code dargestellt:

```
public Wahrheitswert formelAuswerten(){
  Wahrheitswert formel = this.formel.formelAuswerten();
  if(formel.wahrheitswert==true){
  return new Wahrheitswert(false);
  }
  else{
  return new Wahrheitswert(true);
  }
}
```

Die bisherige Art Formeln auszuwerten unterschied sich für quantorfreie Formeln nur geringfügig. Die Komplexität der Auswertung nimmt für Formeln mit Quantoren deutlich zu. Daher werden für die Klassen *Exist* und *ForAll* zusätzliche Funktionalitäten benötigt. Eine dieser Funktionalitäten ist die *calcBel(Variable var, boolean b)* Methode, die als Argumente eine Variable und ein boolean erhält und in der Lage ist die Variable mit entsprechendem Wahrheitswert zu belegen. Die Belegung funktioniert durch folgenden Code:

```
public Formel calcBel(Variable var, boolean b) {
   if (this.var.name == var.name) {
    return new Exist(this.var, this.f);
   } else {
    Formel neuF = this.f.calcBel(var, b);
    return new Exist(this.var, neuF);
   }
}
```

Falls eine Formel einen Existenz-Quantor enthält, wie beispielsweise die Formel $F = \exists x.F_1$, wird zuerst der Rumpf F_1 mit $x = True$ ausgewertet. Bevor jedoch die Auswertung erfolgen kann, muss vorher die Variable x mit *True* belegt werden. Die *calcBel()* Methode sorgt dafür, dass die Variable x im gesamtem Rumpf der Formel auf *True* gesetzt wird. Danach kann *formelAuswerten()* auf *F.calcBel(this.var, true)* aufgerufen werden. Falls das Resultat der Auswertung $True$ ergibt, ist F gültig. Ansonsten wird die Variable x mit *False* belegt durch Aufruf von *calcBel()* mit den Argumenten *this.var* und *x=False* und anschließend wieder *formelAuswerten()* aufgerufen. Das Resultat dieser Auswertung spiegelt die Gültigkeit von F wieder. Dies ist ein Ausschnitt des dazugehörigen Codes:

```
//  formelAuswerten()- Methode der Klasse Exist
public Wahrheitswert formelAuswerten(){
// Formel wird mit True belegt und ausgewertet
  Wahrheitswert trueBelegung = this.f.calcBel(this.var,true)
```

```
   .formelAuswerten();
   if(trueBelegung.wahrheitswert==true){
return new Wahrheitswert(true);
   }
   else{
// Formel wird mit False belegt und ausgewertet
  Wahrheitswert falseBelegung = this.f.calcBel(this.var,false)
   .formelAuswerten();
  return falseBelegung;
    }
}
```

Die Auswertung einer Formel mit Allquantoren funktioniert analog, bis auf den Unterschied, dass die Auswertung des Rumpfes sowohl mit der Belegung $x=$ *True* als auch mit $x =$*False* *True* ergeben muss, damit die Formel gültig ist.

4.3.2 Implementierung QBFtoSAT-Verfahren

Die Implementierung des *QBFtoSAT*-Verfahrens orientiert sich an Abschnitt 3.2. Der erste Schritt des Verfahrens beinhaltet die Transformation einer QBF in *NNF*. Diese Umwandlung ist durch die Methode *nnf()*, die sowohl in der abstrakten Oberklasse Formel, als auch in jeder Unterklasse von Formel implementiert wurde, realisiert worden. Die Methode ist rekursiv und ruft abhängig von der betrachteten Subformel die *nnf()* Prozedur der entsprechenden Unterklasse auf. Hierbei müssen allerdings verschiedene Fälle in Betracht gezogen werden. Der erste Fall der auftreten kann, ist der Aufruf der *nnf()*-Methode der Klasse *And*. Dies geschieht, wenn die Formel eine Konjunktion enthält. Hierbei wird rekursiv die *nnf()*-Methode auf die linke und rechte Teilformel aufgerufen. Analog ist das Vorgehen, falls die Formel eine Disjunktion beinhaltet. Der Programmcode ist der folgende:

```
// Rekursive Methode zur Generierung der Negationsnormalform
public Formel nnf() {
      Formel links = (this.FormelLinks.nnf());
      Formel rechts = (this.FormelRechts.nnf());
      return new And(links, rechts);
}
```

Bei Formeln, die Quantoren enthalten wird *nnf()* auf den Rumpf der Formel *this.f.nnf* aufgerufen. Dies ist bei den Klassen *Exist* und *ForAll* der Fall. Das ganze wird durch folgenden Programmcode beschrieben:

```
public Formel nnf() {
      Formel body = (this.f.nnf());
      Variable v = new Variable(this.var.name);
      return new Exist(v, body);
}
```

Falls in der Formel Implikationen bzw. Biimplikationen auftauchen, muss vor Aufruf der *nnf()*-Methode eine Umformung durchgeführt werden. Dabei wird die Formel gemäß Äquivalenzgesetze umgewandelt, sprich Implikationen und Äquivalenzen werden entfernt. Um Beispielsweise eine Implikation aufzulösen wird ein *Or*-Objekt erzeugt, dessen linker Formelteil auf ein *Not*-Objekt referenziert. Eine Biimplikation wird in eine Formel mit zwei Implikationen umgewandelt und dann analog wie eine IImplikation behandelt. Der entsprechende Programmcode ist folgender:

```
public Formel nnf() {
    return (new Or(new Not(this.FormelLinks),this.FormelRechts))
    .nnf();
}
```

Als letztes muss die *nnf()*-Methode der Klasse *Not* betrachetet werden. Diese ist komplexer und unterscheidet sich abhängig von der betrachteten Subformel:

Fall 1: Die Formel enthält eine Konjunktion, Disjunktion, Implikation oder Biimplikation: Es werden zwei *Not*-Objekte erzeugt. Der linke Formelteil referenziert auf das erste und der rechte Formelteil auf das zweite *Not*-Objekt. Als letztes wird auf beide veränderte Formelteile *nnf()* aufgerufen.

Fall 2: Die Formel enthält einen Quantor: Es wird ein *Not*-Objekt erzeugt. Der Rumpf der Formel referenziert auf das *Not*-Objekt und zum Schluss wird die Methode auf den Rumpf aufgerufen.

Fall 3: Die Formel enthält eine Negation. In diesem Fall wird ein *Not*-Objekt erzeugt und *nnf()* rekursiv auf die innere Formel angewandt.

Fall 4: Die Formel enthält eine Variable. In diesem Fall wird kein Objekt erzeugt, sondern der Name der Variablen ausgegeben.

Die Programmcodes für die entsprechenden Fälle sind die Folgenden:

```
public Formel nnf() {
     // Fall 1
     if (this.formel instanceof And) {
          Formel links = (new Not(((And) this.formel).FormelLinks)).nnf();
          Formel rechts = (new Not(((And) this.formel).FormelRechts)).nnf();
          return new Or(links, rechts);
      }
     // Fall 2
     if (this.formel instanceof Exist) {
          Formel body = (new Not(((Exist) this.formel).f)).nnf();
          Variable v = new Variable(((Exist) this.formel).var.name);
          return new ForAll(v, body);
      }
      // Fall 3
      if (this.formel instanceof Not) {
         Formel nfFormel = ((Not) this.formel).formel.nnf();
         return (nfFormel);
```

```
    }
    // Fall 4
    if (this.formel instanceof Variable) {
        return new Not(new Variable(((Variable) this.formel).name));
     }
}
```

Im zweiten Schritt des Verfahrens müssen die Quantoren aus der Formel eliminiert werden. Dies funktioniert mittels der rekursiven *quantorDelete()*-Methode. Sie ist in allen Klassen, außer den Klassen Conditional und Biconditional implementiert, da die Formel bereits in *NNF* vorliegt und somit keine Implikationen bzw Biimplikation enthält. Wie bei der *NNF*-Transformierung greift abhängig von der Formel die entsprechende *quantorDelete()*-Methode der jeweiligen Klasse. Falls die Formel eine Konjunktion oder Disjunktion enthält wie beispielsweise die Formel $F = F_1 \wedge F_2$, wird sowohl auf den linken, als auch rechten Formelteil *quantorDelete()* aufgerufen. Dies ist der entsprechende Programmcode:

```
public Formel quantorDelete() {
    Formel links = this.FormelLinks.quantorDelete();
    Formel rechts = this.FormelRechts.quantorDelete();
    return new And(links, rechts);
}
```

Falls die Formel eine Negation enthält, wird die Methode auf deren Subformel *this.formel* aufgerufen. Der Programmcode ist der folgende:

```
public Formel quantorDelete() {
    return new Not(this.formel.quantorDelete());
}
```

Wenn die Formel einen Existenzquantor enthält, kann dieser einfach aus der Formel gestrichen werden. Anschließend muss die Methode auf die $\exists$-Quantor freie Formel angewandt werden. Der folgende Programmcode beschreibt das Vorgehen:

```
public Formel quantorDelete() {
    return this.f.quantorDelete();
}
```

Das Entfernen eines Allquantors ist mit deutlich mehr Aufwand verbunden. Um beispielsweise einen Allquantor aus der Formel $\forall X.(F)$ zu entfernen, muss eine neue Formel erzeugt werden. Die neu erzeugte Formel besteht aus einem *And*-Objekt, dessen linker Formelteil auf die Formel F ohne Allquantor und der rechte Formelteil auf die Formel F' referenziert. F' wird durch die Methode *calcQuer(Variable var)* berechnet. Bei der Berechnung von F' wird die die gesamte Formel F durchlaufen und parallel alle freien Vorkommen von $\neg x$ durch x und x durch $\neg x$ ersetzt. Nach der Umformung kann *quantorDelete()* auf F und F' aufgerufen werden. Zu beachten ist, dass zusätzlich die Namen der Variablen aus Formel F vor Aufruf von *quantorDelete()* umbenannt werden müssen. Für die Umbenennung ist die *rename()*-Methode verantwortlich. Diese Umbenennung wird durchgeführt, damit sich die Variablen in F und F' unterscheiden und nicht überschrieben werden. Folgender Programmcode stellt das Vorgehen dar:

```
public Formel quantorDelete() {
//Umbenennung der Formel und Eliminierung der Quantoren
  Formel q = this.f.rename(new variableMapping()).quantorDelete();
  Formel qQuer = this.f.rename(new variableMapping())
  .calcQquer(this.var).quantorDelete();
  return new And(q, qQuer);
 }
```

Der nächste Schritt des Verfahrens ist die Generierung der konjunktiven Normalform. Die Transformierung der Formel in konjunktiver Normalform ist durch die Methode *cnf()* realisert. Die Methode ist in jeder Unterklasse von Formel implementiert und funktioniert rekursiv. Genau wie die vorherigen Methoden wird abhängig von der betrachteten Teilformel die entsprechende *cnf()*-Methode ausgeführt. Falls eine Konjunktion vorliegt wie beispielsweise in der Formel $Z = (F \wedge G) \vee H$, dann wird Z gemäß Distributivgesetz zu $Z' = (F \vee H) \wedge (G \vee H)$ umgeformt. Anschließend wird *cnf()* auf die Formel aufgerufen. Natürlich muss der Algorithmus sich merken, welche Teilformeln bereits verändert wurden. Daher wird nach jeder Veränderung die *hasChanged()*-Methode aufgerufen. Diese setzt den Wahrheitswert, falls eine Teilformel geändert wurde. Als Resultat gibt die Methode ein Paar Objekt zurück bestehend aus der in cnf vorliegenden Formel und dem Wahrheitswert *haschanged*, der die Änderung der Teilformel beinhaltet. Das ganze wird durch folgenden Programmcode illustriert:

```
4.A
public Paar cnf() {
    Paar links = this.FormelLinks.cnf();
    Paar rechts = this.FormelRechts.cnf();
    return new Paar(new And(links.f, rechts.f), links.hasChanged()
    || rechts.hasChanged()); //
 }
```

Die Implementierung der *cnf()*-Methoden der übrigen Klassen sind der obigen $4.A$ sehr ähnlich und können aus dem beiliegendem Programm entnommen werden.

Im letzten Schritt des Verfahrens kann die vorverarbeitete QBF nun dem *SAT*-Solver *SAT4J* übergeben werden. In der so vorliegenden Form kann die Formel jedoch nicht übergeben werden. *SAT4J* erwartet die Eingabe analog zum Dimacs-Format d.h. Klauseln werden durch positive bzw. negative Zahlen ersetzt. Eine Klausel wird durch eine negative Zahl ersetzt, wenn es in negierter Form vorliegt und ansonsten durch eine positive Zahl. Die Abbildung von Variablen auf Zahlen wird durch die Methode *getIntMap()* berechnet und als *HashMap⟨ String, Integer ⟩* dargestellt. Die *getIntMap()* Methode läuft rekursiv durch die Formel und sammelt alle Variablennamen auf. Zusätzlich muss für *SAT4J* eine sogenannte *ISolver*-Instanz erzeugt werden und die Klauseln zur Instanz hinzugefügt werden. Dies wird durch die Methode *getClauses()* bewerkstelligt. Sie erhält die von der *getIntMap* berechneten HashMap als Argument und läuft rekursiv durch die Formeln. Dabei fügt sie jede Klausel in der Darstellung als Vektor von Zahlen der Solverinstanz hinzu.

Das *SAT4J* Framework wirft bei unerfüllbaren Formeln eine Exception. Daher muss diese abgefangen werden. Die Implementierung von *getClauses()* setzt dies um, in dem sie einen Wahrheitswert als Ergebnis liefert der genau dann *True* ist, wenn eine Exception auftritt, sprich die Formel unerfüllbar ist. Falls der Wahrheitswert *False* ist, wird auf die Solverinstanz die *SAT4J*-Methode *isSatisfiable()* aufgerufen. Auf diese Art kann die Erfüllbarkeit der Klauselmenge überprüft werden. Insgesamt führt die Methode *toSolver()* den *QBFtoSAT*-Algorithmus aus. Der entsprechende Programmcode ist der folgende:

```
static boolean toSolver(Formel f) throws TimeoutException{
   // Berechnung der Normalform
   Formel normalForm == f.nnf().quantorDelete().cnf().f;

   //Abbildung Variablen auf Zahlen
   HashMap<String, Integer> x = new HashMap<String, Integer>();
   HashMap<String, Integer> values = normalForm.getIntMap(x);

   // Erzeugung einer ISolver Instanz
   ISolver solver = SolverFactory.newDefault();

   // Hinzufügen der Klauseln zur ISolver Instanz
   boolean wert = normalForm.getClauses(solver,values);
   if(wert=true){
      return(false);
   }
   else{
    return(solver.isSatisfiable());
    }
   }
 }
```

4.3.3 Implementierung QDPLL-Verfahren

Bevor die Beschreibung des *QDPLL*-Verfahrens erfolgen kann, muss die Transformation von Formeln in *PKNF* gezeigt werden. Um eine Formel in *PNKF* zu überführen sind folgende drei Vorverarbeitungsschritte notwendig. Im ersten Schritt muss die Formel in *NNF* überführt werden. Diese Umformung funktioniert ebenfalls mit der *nnf()*-Methode aus Verfahren (siehe 4.3.2). Aufgrund der bereits ausführlichen Beschreibung der Methode wird an dieser Stelle nicht näher darauf eingegangen.

```
Formel wird in NNF überführt
   Formel F;
   F.nnf();
```

Im zweiten Schritt erfolgt die Quantorverschiebung. Hierbei müssen alle Quantoren in den Präfix der Formel verschoben werden. Die Verschiebung ist durch die Methode *quantorMove()* realisiert. Die Methode arbeitet rekursiv und ist in allen Unterklassen von Formel, bis auf den Klassen *Conditional* und *Biconditional* implementiert, da *F* bereits in *NNF* vorliegt.

In den Klassen *And* und *Or* wird *quantorMove()* zunächst rekursiv auf den linken und rechten Formelteil aufgerufen. Nach erfolgreichem Aufruf befindet sich sowohl linker als auch rechter Formelteil in *PNF*. Das ganze verdeutlicht folgender Programmcode:

```
public Formel quantorMove(){
    Formel links = this.FormelLinks.quantorMove();
    Formel rechts = this.FormelRechts.quantorMove();
}
```

Im nächsten Schritt müssen die beiden Formelteile zu einer einzelnen Formel in PNF zusammengeführt werden. Damit dies gelingt, müssen zunächst die Quantoren aus den Formeln extrahiert werden. Programmiertechnisch werden mittels einer *while*-Schleife die Formeln durchlaufen und die Quantoren des linken und anschließend rechten Formelteils in einer *LinkedList⟨Formel⟩* abgespeichert. Dabei wird für jeden Quantor die entsprechende Teilformel durch die *rename()*-Methode umbenannt und der Quantor samt Bindungsvariablen in die Liste hinzugefügt. Dies garantiert, dass keine gleichen Variablennamen vorhanden sind und somit die Zuordnung der Quantoren eindeutig ist. Folgender Programmcode illustriert das Vorgehen:

```
public Formel quantorMove(){
   LinkedList<Formel> liste = new LinkedList<Formel>();
   while(links instanceof Exist || links instanceof ForAll){
   if(links instanceof Exist){
     Formel renamedLinks = ((Exist) links).rename(new variableMapping());
     links = ((Exist) renamedLinks).f;
     liste.push( new Exist(((Exist)renamedLinks).var, new Wahrheitswert(true)));
    }
   if(links instanceof ForAll){
     Formel renamedLinks = ((ForAll) links).rename(new variableMapping());
     links = ((ForAll) renamedLinks).f;
     liste.push( new ForAll(((ForAll)renamedLinks).var, new Wahrheitswert(true)));
 }  ........
```

Zum Schluss wird die *LinkedList* durchlaufen und die einzelnen Listenelemente aus der Liste entfernt und an den Anfang der Formel gesetzt. Dies wird durch folgenden Programmcode dargestellt:

```
ListIterator<Formel> iterator =liste.listIterator();
    Formel result = new And(links,rechts);
    while( ! liste.isEmpty()){
      Formel listContent= liste.pop();
      if(listContent instanceof ForAll){
      ((ForAll)listContent).f= result;
       result = listContent;
      }
      if(listContent instanceof Exist){
      ((Exist)listContent).f= result;
       result = listContent;
      }
```

```
 }
    return result;
```

Analog ist *quantorMove()* in der Klasse *Or* implementiert, bis auf den Unterschied, dass linker und rechter Formelteil disjunktiv verknüpft werden. Siehe folgenden Codeabschnitt:

```
public Formel quantorMove(){
   Formel links = this.FormelLinks.quantorMove();
   Formel rechts = this.FormelRechts.quantorMove();
   LinkedList<Formel> liste = new LinkedList<Formel>();
   while(links instanceof Exist || links instanceof ForAll){....
```

In der Klasse *Not* wird die Methode rekursiv auf ihr Formelattribut aufgerufen. Dabei ist der Code der Folgende:

```
public Formel quantorMove(){
    Formel formel = this.formel.quantorMove();
    LinkedList<Formel> liste = new LinkedList<Formel>();
    while(formel instanceof Exist || formel instanceof ForAll){ .....
```

Der Rekursionsanker der Methode befindet sich in der Klasse Variable, wo die entsprechende Formel als Resultat ausgegeben wird. Siehe folgenden Code:

```
public Formel quantorMove(){
     return this;
}
```

Im letzten Schritt der Vorverarbeitung wird der quantorfreie Rumpf der Formel in konjunktive Normalform überführt. Hierfür wird die Methode *cnf()*, die bereits in 4.3.2 eingesetzt wurde, verwendet. Die Beschreibung der Methode befindet sich in Abschnitt 4.3.2. Zusätzlich wird eine weitere Methode *cnf2()* benötigt. *cnf2()* durchläuft den Quantorpräfix einer Formel und ruft *cnf* auf, wenn sie ihren Rumpf erreicht. Der folgende Code beschreibt die Methode:

```
public Formel cnf2() {
    if (this.f instanceof ForAll || this.f instanceof Exist) {
      return new Exist(this.var, this.f.cnf2());
    } else {
      return new Exist(this.var, this.f.cnf().f);
        }
    }
```

Die Vorverarbeitung ist beendet und Formeln können nun in *PKNF* überführt werden.

Bevor jedoch Formeln in *PKNF* an den *QDPLL* gegeben werden können, muss ihre Repräsentation verändert werden. Um die Repräsentation der Formeln zu verändern müssen alle Klauseln, wie bereits bei Verfahren 4.3.2 angewendet, in Zahlen überführt werden. Dies funktioniert mit der Methode *getIntMap()*. Dabei

wird der Quantorpräfix der Formel in eine *LinkedList* von Quantoren eingefügt und der quantorfreie Rumpf, der als *CNF* vorliegt, wird als Menge von Zahlen in eine HashSet⟨Integer⟩ eingefügt. Gleichzeitig wird für jede Klausel die Klassifizierung gemerkt. Die Klassifizierung von Klauseln als PK-Wahr, PK-Falsch oder PK-Offen ist der bedeutende Schritt des *QDPLL* Verfahren (siehe 3.3.2). Anschließend müssen alle PK-Wahren Klauseln aus der Klauselmenge entfernt werden. Das Entfernen funktioniert mit Hilfe der Methode *deleteAllPkTrue()*. Dabei wird die Klauselmenge durchlaufen und mit der Methode *remove()* entsprechende Einträge löscht. Der zugehörige Programmcode ist der folgende:

```
void deleteAllPkTrue() {
  // Iterator über die Schlüssel der Klauseln, mit
      Kopie der Schlüsselmenge
   Iterator<Integer> clause_keys = Util.deepCopySet
   (this.klauselmenge.keySet())
   .iterator();
    while (clause_keys.hasNext()) {
      // Nächster Schlüssel
      Integer key = clause_keys.next();
      // Zugehörige Klausel
      HashSet<Integer> clause = this.klauselmenge.get(key);
      // Iterator über die Klausel (als Kopie)
      Iterator<Integer> iterate_clause = Util.deepCopySet(clause)
      .iterator();
      while (iterate_clause.hasNext()) {
      // Lösche Originalklausel, wenn sie das aktuelle Literal
      auch negiert enthält
       if (clause.contains(iterate_clause.next() * -1)) {
       this.klauselmenge.remove(key);
          }
      }
   }
 }
```

Ein interessantes Detail hierbei ist, dass für die Iteratoren tiefe Kopien der Schlüsselmengen bzw. der Klauseln verwendet wurden. Der Grund liegt darin, dass Java Modifikationen der Elemente einer Collection verbietet, während ein Iterator über diese Collection verwendet wird. Die Lösung hierbei war es, den Iterator daher über eine Kopie der ursprünglichen Collection laufen zu lassen. Eventuell hätte es hier auch andere Programmiermöglichkeiten gegeben, die das Löschen direkt über den Iterator durchführen (siehe [Ora15b]).

Im nächste Schritt des Verfahrens wird die Methode *evaluate()* aufgerufen, die der eigentlichen Implementierung des *QDPLL* entspricht.

```
boolean qdpll() {
// Loesche alle PK-wahren Klauseln
this.deleteAllPkTrue();
// Weiter mit evaluate
return this.evaluate();
}
```

In *evaluate()* wird zunächst überprüft, ob die Klauselmenge die Abbruchbedingungen erfüllt Die erste Abbruchbedingung tritt ein, falls die Klauselmenge PK-Falsche Klauseln enthält. Falls dies der Fall ist, wird das Programm beendet und *False* ausgegeben. Die Überprüfung auf PK-Falsche Klauseln erfolgt durch die Methode *hasPkFalse()*, die über die Schlüssel der Klauselmenge iteriert und die entsprechende Klassifizierung überprüft.

```
 // 1) Wenn es  PK-falsche Klauseln gibt,
      breche ab und gebe False zurueck
        if (this.hasPkFalse()) {
           return false;}

boolean hasPkFalse() {
// allquantifzierte Variablen:
HashSet<Integer> allListe = this.getAllquant();
// iteriere ueber die Schluessel der Klauselmenge
    terator<Integer> iterate_keys =
    (this.klauselmenge.keySet()).iterator();
boolean found_pk_false = false;
    while (iterate_keys.hasNext() && !found_pk_false) {
    Integer key = iterate_keys.next();
     HashSet<Integer> clause = this.klauselmenge.get(key);
      // iteriere ueber die aktuelle Klausel:
       .....
```

Die zweite Abbruchbedingung tritt ein, wenn die Klauselmenge leer ist. Falls dies der Fall ist wird *true* ausgegeben. Die Überprüfung auf Leerheit erfolgt mit der Methode *isEmpty()*. Hierbei handelt es sich um eine Standard Methode für HashSets mit folgendem Code:

```
// 2) Wenn die Klauselmenge leer ist,
     breche ab und gebe True zurueck
        if (this.klauselmenge.isEmpty()) {return true;}
```

Ansonsten wird überprüft, ob die Klauselmenge *PK-unit* Klauseln enthält. Falls ja, wird die Methode *unitPropagate()* auf das entsprechende Literal aufgerufen und rekursiv weiter gemacht. Die Methode *unitPropagate()* löscht alle Klauseln, die das Literal enthalten und das negierte Literal, das in allen anderen Klauseln vorkommt. Ansonsten wird am nächsten Quantor verzweigt.

```
  // ------------------
  // Ansonsten: ...
  // ------------------
  if (!quantorList.isEmpty()) {
  // Gibt es PK-Unit-Klauseln?
  Integer literal = this.hasPKunit();
  if (literal != 0) {
  // Es gibt eine PK-Unit-Klausel mit Literal literal
  // Führe UnitPropagation durch und mache rekursiv weiter
       unitPropagate(literal);
       return this.evaluate();
```

```
    }
   else {   ....
```

Mit einem Trick kann die *CNF*-Berechnung optimiert werden. Durch die Methode *flatten(Formel f)* lassen sich für tiefe Subformeln Abkürzungen einführen. Dafür wird die Formel mit der Methode *findSubformula()* durchsucht und beim Auffinden einer entsprechenden Subformel (SearchResult) eine neue Variable und ein Biimplikation-Objekt erzeugt und konjunktiv mit der Restformel verknüpft.

```
public SearchResult(boolean b, Variable var, Formel f) {
// flatten macht eine Formel flach, indem
// Abkürzungen für tiefe Subformeln eingeführt werden
  public static Formel flatten(Formel f) {
   Formel flat_formula = f;
   Boolean suche = true;
   while (suche) {
     SearchResult res = flat_formula.findSubformula(0);
     if (res.foundFormel) {
     // erzeuge  /\ x <=> G
      flat_formula = new And(new
      Biimplikation((Formel)res.newName,res.formel),flat_formula);
       }
     else {
   suche = false; // Formel ist flach
  }
  }
     return flat_formula.nnf().cnf().f;
}
```

4.4 Ein- und Ausgabeschnittstellen

In diesem Abschnitt werden Ein- und Ausgabeschnittstellen definiert. Dabei sind die textuelle Formelausgabe(4.4.1), das Einlesen von Formeln mittels einem Parser (4.4.2) und die Konstruktion einer Graphische Benutzeroberfläche(4.4.3) Bestandteil dieses Kapitels.

4.4.1 Textuelle Formelausgabe

Leider werden bislang keine Formeln auf der Konsole ausgedruckt. Dies erschwert die Implementierung der Algorithmen und das anschließende Testen enorm. Damit Formeln auf der Konsole ausgegeben werden, muss die *toString()* Methode aller Unterklassen überschrieben werden. Die *toString()* Methode der Klassen *And, Or, Conditional* und *Biconditional* muss geklammert den linken, das entsprechende logische Symbol und anschließend den rechten Formelteil ausgeben. Bei den Klassen *Exist* und *ForAll* dagegen wird zuerst das logische Symbol, die gebundene Variable und zum Schluss der Rumpf der Formel ausgeben. Die letzten beiden Varianten der *toString()* Methode befinden sich in den Klassen *Not* und *Variable*. Bei *Not* wird das logische Symbol gefolgt

von der Formel ausgedruckt und bei der Klasse Variable der Name der Variablen.

Beispiel 4.4.1.
Überschreiben der toString() Methode der Klasse AND

```
@Override  Überschreiben der toString() Methode
//Definition Unicode Symbole
String symbolAnd = "\u2227";
String bracketOpen = "\u0028";
String bracketClose = "\u0029";
// Formel wird zusammengefügt
public String toString() {
return (bracketOpen + this.FormelLinks.toString() + symbolAnd
+ this.FormelRechts.toString() + bracketClose);
};
```

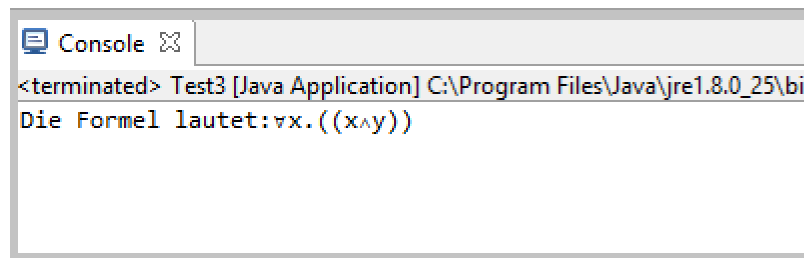
Um das logische Symbol in der Konsole auszugeben benötigt man den entsprechenden Unicode des Zeichens. Unicode ist ein internationaler Standard, der für etliche Schriftzeichen oder Textelemente, einen digitalen Code festlegt. Die Unicode Tabelle wird ständig erweitert und verfolgt das langfristige Ziel die Verwendung unterschiedlicher und inkompatibler Kodierungen zu beseitigen. Folgende Unicodes waren für das Programm notwendig und können aus der Unicode Tabelle entnommen werden (siehe [fdLaG15])

Unicode	Zeichen
0028	(
0029	)
2227	$\wedge$
2228	$\vee$
2192	$\Longrightarrow$
2194	$\Longleftrightarrow$
2203	$\exists$
2200	$\forall$
00AC	$\neg$

[Unicode Tabelle der benötigten Symbole]

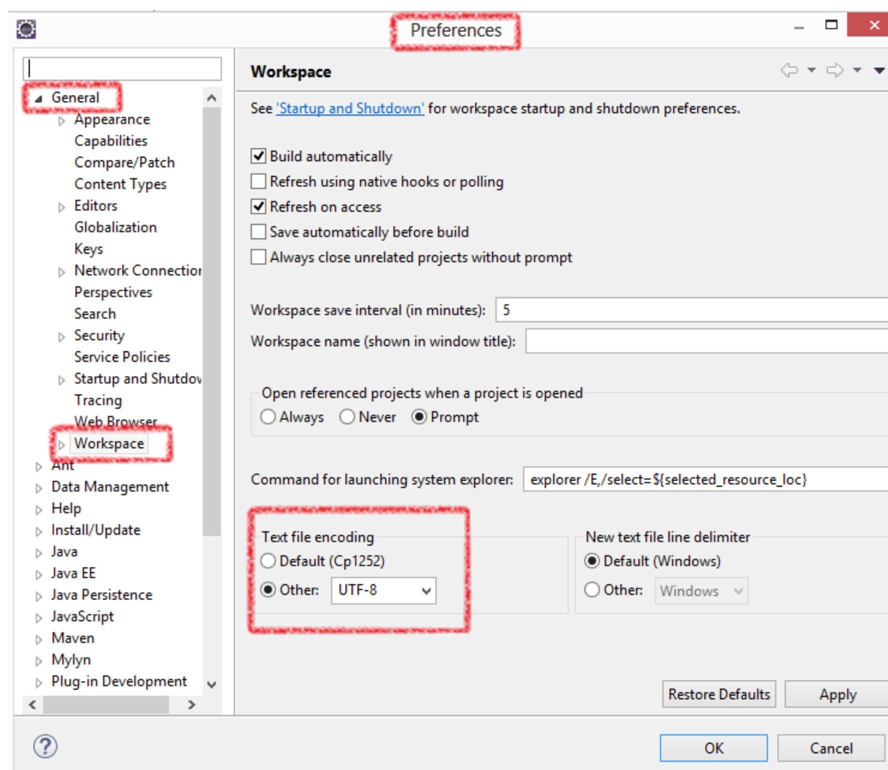
Beispiel 4.4.2. *In diesem Beispiel soll in Eclipse die Formel $\forall x.(x \wedge y)$ erzeugt und anschließend auf der Konsole ausgegeben werden.*

```
public static void main(String[] args) {
Variable x = new Variable("x");
Variable y = new Variable("y");
Formel z = new And(x,y);
Formel formel = new ForAll(x,z);
System.out.println(formel);
}
Konsolenausgabe:
```



[Screenshot Konsolenausgabe]

Hinweis Exkurs: Damit die Unicode Symbole korrekt auf der Konsole ausgedruckt werden, muss die Kodierung von Eclipse auf UTF-8 umgestellt werden. In Eclipse gelangt man zu den entsprechenden Einstellungen, wenn man innerhalb der Hauptnavigation unter *Window* unten auf *Preferences* klickt. Dort den Punkt *General* aufklappen, um zum *Workspace* zu gelangen. Innerhalb des Menüpunkts *Workspace* ganz unten im Bereich *Text file encoding* auf UTF-8 umstellen.



Nun wird eine einfache Möglichkeit präsentiert Formeln über eine graphische Benutzeroberfläche einzugeben. Die Eingabeformeln werden automatisch geparst und entsprechende Algorithmen können per Knopfdruck angewandt werden.

4.4.2 Einlesen von Formeln

Das Parsen der Formeln erfolgt automatisch. Dies wurde programmiertechnisch mittels ANTLR[1] realisiert [Par12]. ANTLR ist ein objektorientierter Parsergenerator zum Lesen, Verarbeiten, Ausführen oder Übersetzen von strukturiertem Text-oder Binärdateien. Aus einer Grammatik erzeugt ANTLR einen Parser der Parsebäume erzeugen und durchlaufen kann. Der Parsergenerator ANTLR unterstützt LL(k)-Grammatiken mit beliebigem k. LL(k)-Grammatiken sind spezielle kontextfreie Grammatiken bei der ein Ableitungsschritt eindeutig durch k Symbole der Eingabe bestimmt ist. ANTLR ist im akademischen und industriellen Umfeld weit verstreut beispielsweise wertet Twitter über zwei Milliarden Anfragen täglich mittels ANTLR aus. In dieser Arbeit wird ANTLR verwendet um die Eingabeformel, die als String vorliegt, zu parsen. Dabei wird die Eingabeformel in eine Formel entsprechend der Definition durch die vorliegenden Java Klassen umgewandelt.

Folgende Grammatik war Grundlage für den Parser :

```
prog:   expr ;
expr:
        expr '/\\' expr       # And
    |   expr '\\/' expr  # Or
    |   expr '->' expr       # Imp
    |   expr '<->' expr  # Biimp
    |   '-' expr # Not
    |   ID                      # id
    |   '!' ID '.' expr # Exist
    |   '?' ID '.' expr # Forall
    |   '(' expr ')'            # parens


AND :   '/\\' ;
OR  :   '\\/' ;
EX:   '!' ;
FOR: '?' ;
DOT:  '.';
NOT:  '-' ;
IMP:    '->' ;
BIMP: '<->' ;
ID  :   [a-zA-Z]+ ;
WS  :   [ \t\r\n]+ -> skip ;
```

Die Installation und weitere ANTLR Eigenschaften können aus dem Buch [Par12] entnommen werden.

[1] Another Tool for Language Recognition

4.4.3 Graphische Benutzeroberfläche

Graphische Benutzeroberflächen kurz GUI genannt können relativ simpel per Drag und Drop in Eclipse designed werden. Um von dieser Design-Funktionalität Gebrauch zu machen, muss das passende Eclipse Plug-in installiert werden. Ein Plug-in, dass sich für die Gui Programmierung mit Eclipse sehr bewährt hat, ist WindowBuilder. Dieses Plug-in sowie sämtliche für die GUI Programmierung notwendigen Elemente befinden sich in den Java Swing Klassen, die im Paket javax.swing enthalten sind [Ecl15]. Folgende Swing-Elemente sind für diese Arbeit relevant [Gar11]:

Fenster und Dialoge:

- *Klasse JFrame*: Hauptfenster der GUI mit zusätzlicher Menüleiste.
- *Klasse JFileChooser*: Fenster zum Auswählen und Öffnen von Dateien.
- *Klasse JOptionPane*: Dialog, der bei Fehlermeldungen oder Benutzerbestätigungen verwendet wird.

Bedienelemente:

- *Klasse JLabel*: Nicht editierbarer Text.
- *Klasse JButton*: Schaltfläche.
- *Klasse JTextField*: Einfache einzeilige Texteingabe.
- *Klasse JTextArea*: Einfache mehrzeilige Texteingabe.

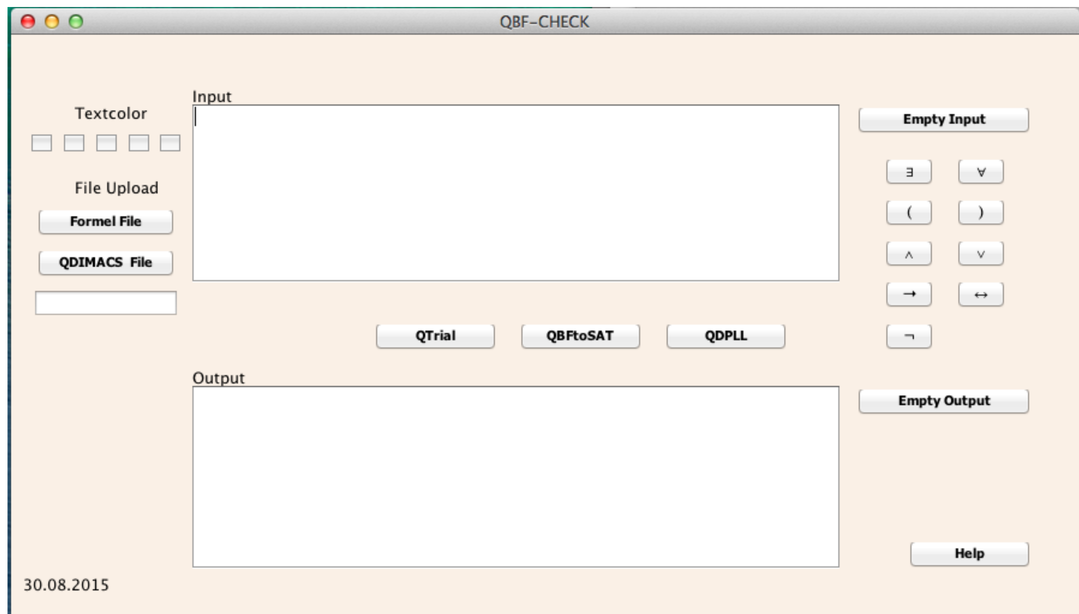
Die Steuerung des Programms erfolgt über Bedienelemente. Beim Drücken eines Buttons kann ein Ereignis ausgeführt werden. Falls ja wird ein Objekt der Klasse *ActionEvent* aus dem Paket *java.awt.event* erzeugt. Dieses Objekt kann mittels einem *ActionListeners*[2] erfasst werden. Um eine Klasse zum *ActionListener* zu machen, muss die entsprechende Methode void *actionPerformed(ActionEvent e)* implementiert werden d.h. falls ein *ActionEvent* bei einem Bedienelement wie einem Button auftritt, wird diese Methode aufgerufen. Häufig verwendete Methoden, zur Veränderung von Bedienelemente wie Textfelder oder Buttons sind unter anderem *setEnabled()*, mit der das jeweilige Bedienelement aktiviert bzw. deaktiviert werden kann, sowie die Methode *setText()*, mit der ein entsprechender Text dem Bedienelement zugewiesen werden kann.

Durch die GUI ist die Bedienung des Programms stark vereinfacht worden. Man kann Formeln auf unterschiedliche Arten eingeben und die verschiedenen Verfahren auf Knopfdruck verwenden. Die GUI stellt folgende Funktionalitäten bereit:

- Eine QBF kann direkt in das obere Textfeld eingegeben werden und anschließend automatisiert weiterverarbeitet werden. Zu beachten ist, dass das Zeichen ! für einen $\exists$-Quantor und das Symbol ? für einen $\forall$-Quantor steht.

[2] Interface ActionListener von java.util.EventListener

- Das Drücken der Schaltfläche *Upload*, bietet die Möglichkeit eine Datei, die eine Formel enthält, hochzuladen. Nach *Upload* der Datei kann diese weiter verarbeitet werden.
- Beim Drücken der Schaltflächen *QTrial*, *QBFtoSAT* und *QDPLL* werden die entsprechenden Algorithmen aufgerufen.
- Zusätzlich hat man die Möglichkeit das obere Textfeld mittels der Schaltfläche *Empty Textarea* zu leeren. Alle notwendigen Programmschritte werden im unteren Textfeld ausgegeben.



4.5 Experimentelle Ergebnisse

In diesem Abschnitt werden die experimentellen Ergebnisse dargestellt. Zunächst werden einfache Tests zur reinen Funktionalität des Programms vorgestellt. Im Anschluss darauf werden die Laufzeitmessungen in Form einer Tabelle wiedergegeben. Zum Schluss werden signifikante Merkmale, die für die Laufzeitunterschiede relevant sind, in Diagrammform gegenübergestellt.

4.5.1 Einfache Testfälle

In diesem Abschnitt werden einfache Testfälle, die die reine Funktionalität des Programms zeigen, vorgestellt:

Test 1

In das Programm wurde eine widersprüchliche Formel eingegeben und erwartet, dass das Programm dies erkennt und *false* ausgibt.

```
Eingabeformel ist geschlossen:  (!_x1.(?_x2.((_x1->_x2)/\(_x2->_x1))))
Schritt 1: Transformierung in NNF:  (!_x1.(?_x2.(((-_x1)\/_x2)/\((-_x2)\/_x1))))
Schritt 2: Eliminierung der Quantoren:((((-_x1)\/_x2)/\((-_x2)\/_x1))/\(((-_x1)\/...
Schritt 3: Transformierung in CNF: ((((-_x1)\/_x2)/\((-_x2)\/_x1))/\(((-_x1)\/...
Schritt 4: Auswertung durch den SAT-Solver:  false
```

Wie erwartet gibt das Programm *false* aus und somit ist die Formel widersprüchlich. Die Zwischenschritte des Algorithmus wurden per Hand nachgerechnet und sind alle korrekt.

Test 2

In das Programm wurde eine gültige Formel eingegeben und erwartet, dass das Programm dies erkennt und *true* ausgibt.

```
Eingabeformel ist geschlossen:  (?_x2.(!_x3.((_x3->_x2)/\(_x2->_x3))))
Schritt 1: Transformierung in NNF:  (?_x2.(!_x3.(((-_x3)\/_x2)/\((-_x2)\/_x3))))
Schritt 2: Eliminierung der Quantoren:  ((((-_x4)\/_x2)/\((-_x2)\/_x4))/\(((-_x5)...
Schritt 3: Transformierung in CNF:  ((((-_x6)\/_x2)/\((-_x2)\/_x6))/\(((-_x7)\/...
Schritt 4: Auswertung durch den SAT-Solver:  true
```

Wie erwartet gibt das Programm *true* aus und somit ist die Formel gültig. Die Zwischenschritte des Algorithmus wurden per Hand nachgerechnet und sind alle korrekt.

Test 3

Beispiel Transitivität der Implikation:

```
Eingabeformel: (?_x1.(?_x2.(?_x3.(((_x1->_x2)/\(_x2<->_x3))->(_x1->_x3)))))
Schritt 1: Transformierung in NNF:  (?_x1.(?_x2.(?_x3.(((_x3/\(-_x3))...
Schritt 2: Eliminierung der Quantoren:  ((((((_x6/\(-_x6))\/((_x6/\(-_x6))...
Schritt 4: Auswertung durch den SAT-Solver:  true
```

Test 3 zeigt, dass die aussagenlogische Implikation transitiv ist.

Test 4

Beispiel Transitivität der Äquivalenz:

```
Eingabeformel:  (?_x1.(?_x2.(?_x3.(((_x1<->_x2)/\(_x2<->_x3))->(_x1<->_x3)))))
Schritt 1: Transformierung in NNF:  (?_x1.(?_x2.(?_x3.((((_x1/\(-_x2))\/(_x2/\...
Schritt 2: Eliminierung der Quantoren:  (((((((_x1/\(-_x4))\/(_x4/\(-_x1)))...
Schritt 3: Transformierung in CNF:  ((((((((((_x1\/_x12)\/(_x12\/_x14))\/...
Schritt 4: Auswertung durch den SAT-Solver:  true
```

Test 4 zeigt, dass die aussagenlogische Biimplikation transitiv ist.

4.5.2 Laufzeitmessung

In diesem Abschnitt werden Tests vorgestellt, die zeigen sollen, wie sich die Laufzeiten der Implementierungen bei wachsender Formelgröße, sprich mehr Quantoren, Quantorwechsel und Klauseln, verhalten. Hierfür wurden diverse Formeln aus der QBFLIB Sammlung verwendet. QBFLIB ist eine Sammlung von Instanzen, Solver und Werkzeuge zur Lösung von QBFs. Über 13000 Instanzen aus unterschiedlichsten Problembereichen sind Teil dieser Sammlung. Die Testfälle beschränken sich auf Probleminstanzen unterschiedlicher Gruppen wie beispielsweise dem Paket Ling, dessen Instanzen Probleme der FPGA-Logik beschreiben oder dem Paket Ayari, eine Familie von Problemen bezogen auf Gleichheitsüberprüfungen innerhalb von Schaltungen [QBF15a]. Somit deckt das Testen einen äußerst umfangreichen Problembereich ab und ist nicht spezifisch für eine Gruppe von Problemen aussagekräftig. Die Instanzen obliegen dem QDIMACS-Format und werden wie in der Implementierung beschrieben geparst und in eine QBF-Formel transformiert. Im Anschluss darauf werden die verschiedenen Algorithmen auf die Formel angewandt. Die Instanzen aus den verschiedenen Gruppen sind zwar alle im QDIMACS-Format, jedoch nicht einheitlich. Dieses Problem wird beim Parsen, durch das Löschen von Kommentaren und anderen nicht relevanten Informationen, behoben.

4.5.2.1 Testbedingungen

Jede Instanz wurde pro Algorithmus drei mal getestet und hatte eine maximale Laufzeit von fünf Minuten. Falls nach der Laufzeit der Algorithmus immer noch am rechnen war, wurde das Resultat der Formelauswertung auf *ERROR* gesetzt. Ansonsten war das Resultat abhängig von der Instanz entweder *TRUE* oder *FALSE*.

Die Tests wurden auf folgenden Systemen durchgeführt:

```
------------------
System Information
------------------
System 1:
   Operating System: Windows 8 64-bit (6.2, Build 9200) (9200.win8_gdr.130531-1504)
   System Manufacturer: ASUSTeK COMPUTER INC.
   System Model: K55VJ
   BIOS: K55VJ.201
   Processor: Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz (4 CPUs), ~2.5GHz
   Memory: 4096MB RAM
   Available OS Memory: 3982MB RAM

System 2:
   Operating System: OpenSUSE 13.1 64-bit
   Processor: Intel(R) Core(TM) i5 CPU @ 3.6GHz (4 CPUs)
   Memory: 8GB RAM
```

Hinweis: Das Testen auf anderen Systemen kann zu anderen Messergebnissen führen. Dies hängt einerseits von der Leistungsfähigkeit und andererseits von der Auslastung des Rechners ab.

Die Programmausführung wurde zu verschiedenen Zeitpunkten gemessen, sodass sich die Zeiten für

- die Anwendung des $QDPLL$-Algorithmus mit Optimierung,
- die Anwendung des $QDPLL$-Algorithmus ohne Optimierung,
- die Anwendung des $QBFtoSAT$-Algorithmus und
- die Anwendung des $QTRIAL$-Algorithmus

aus den Messpunkten berechnen ließen. Die Messwerte beinhalten allerdings nicht die Zeit, die für die Vorverarbeitung notwendig ist. Sämtliche Laufzeiten sind in Millisekunden angegeben. Auf die Messung des Platzverbrauches wurde verzichtet.

4.5.2.2 Messergebnisse

Die Tabelle beinhaltet nicht nur die Laufzeiten und Resultate der diversen Instanzen, sondern auch weitere wesentliche Informationen, die für eine korrekte und vollständige Analyse notwendig sind. Folgende weitere Informationen sind notwendig:

- Anzahl Quantorwechsel
- Anzahl Quantorvariablen
- Anzahl ∀-Quantoren
- Anzahl ∃-Quantoren

Die gemessenen Laufzeiten der verschiedenen Instanzen sind in der untenstehenden Tabelle 4.5.1 angegeben. Signifikante Zellen sind fett gedruckt hervorgehoben. Verschiedene Messwerte wie beispielsweise die Anzahl der Quantorwechsel gegenüber der Laufzeit der Verfahren oder die Anzahl der Instanzen die speziell nur von einem Algorithmus gelöst wurden, werden im Anschluss darauf als Diagramme gegenüber gestellt. Außerdem wird auf die Frage, ob die Laufzeit der Verfahren bei Zunahme von Quantorwechseln steigt, eingegangen. Die Diagramme verdeutlichen die Zusammenhänge die aus der Tabelle gewonnen werden können und lassen weitere Schlussfolgerungen zu, die im Fazit 5.2 vermerkt sind.

T:= TRUE ; **F**:= FALSE ; **E**:= ERROR

Tabelle 4..5.1

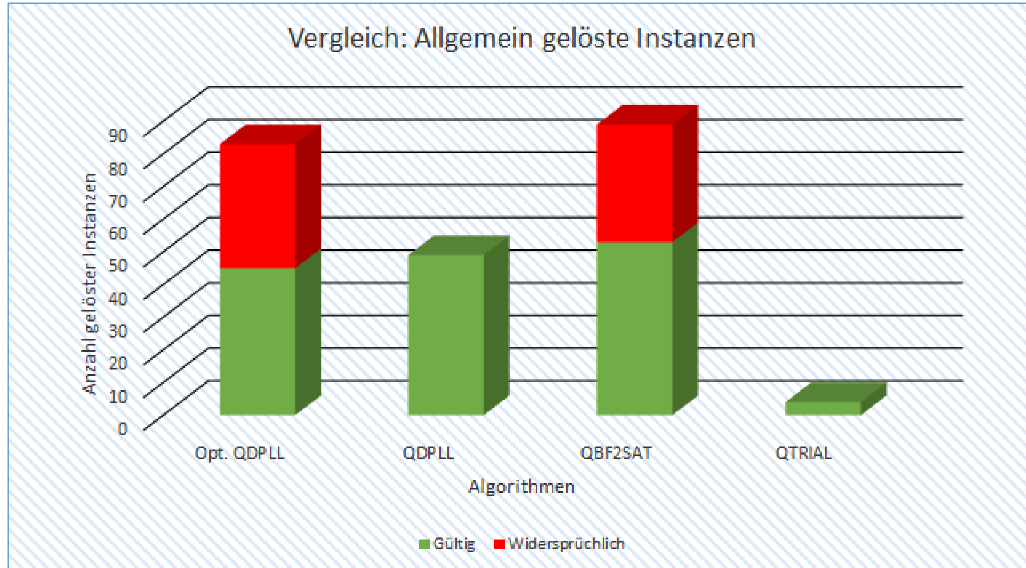
Instanzen	Quantorwechsel	Anzahl ∀-Variablen	Anzahl ∃-Variablen	Anzahl Klauseln	Laufzeit Opt. QDPLL	Resultat Opt. QDPLL	Laufzeit QDPLL	Resultat QDPLL	Laufzeit QBF2SAT	Resultat QBF2SAT	Laufzeit QTrial	Resultat QTrial
1	9	4	260	631	2092	F	-	E	20	F	-	E
2	9	4	260	631	3975	T	-	E	43	T	-	E
3	9	4	260	631	1335	F	-	E	40	F	-	E
4	13	6	506	1375	30965	F	-	E	84	F	-	E
5	13	6	506	1375	70966	T	-	E	213	T	-	E
6	13	6	506	1375	16348	F	-	E	-	F	-	E
7	17	8	832	2407	173160	F	-	E	-	E	-	E
8	17	8	832	2407	96425	F	-	E	-	E	-	E
9	13	9	595	1502	39455	F	-	E	-	E	-	E
10	13	9	595	1502	24626	F	-	E	-	E	-	E
11	3	3	38	88	45	T	75	T	12	T	-	E
12	3	9	107	268	353	T	2019	T	1679	T	-	E
13	3	18	239	619	266820	T	-	E	-	E	-	E

14	3	5	78	196	5391	T	31340	T	39	T	-	E
15	3	13	194	511	-	E	-	E	193346	T	-	E
16	3	17	322	844	169132	T	-	E	-	E	-	E
17	3	1	25	70	31	T	18	T	8	T	114	E
18	3	2	50	142	56	T	80	T	21	T	-	E
19	3	3	105	304	160	T	178	T	33	T	-	E
20	3	1	14	37	12	T	32	T	5	T	50966	E
21	3	2	46	130	52	T	84	T	5	T	-	E
22	3	3	92	265	118	T	180	T	26	T	-	E
23	3	1	16	43	15	T	14	T	7	T	51	E
24	3	2	49	139	55	T	76	T	10	T	-	E
25	3	3	102	295	145	T	173	T	17	T	-	E
26	3	3	68	187	274	T	435	T	13	T	-	E
27	3	6	197	565	38474	T	64478	T	198	T	-	E
28	3	9	469	1372	-	E	-	E	6090	T	-	E
29	3	9	410	1177	-	E	-	E	1185	T	-	E
30	3	9	412	1183	-	E	-	E	3185	T	-	E
31	3	3	80	223	186	T	433	T	15	T	-	E
32	3	6	206	592	10772	T	23500	T	144	T	-	E
33	3	9	406	1183	-	E	-	E	8039	T	-	E
34	3	4	239	688	631	T	669	T	49	T	-	E
35	3	10	640	1873	-	E	-	E	15576	T	-	E
36	3	3	108	313	148	T	203	T	39	T	-	E
37	3	3	79	220	278	T	526	T	16	T	-	E
38	3	6	201	577	33137	T	64076	T	205	T	-	E
39	3	9	386	1123	-	E	-	E	5118	T	-	E
40	3	1	73	208	68	T	84	T	7	T	-	E
41	3	4	416	1210	3981	T	1795	T	126	T	-	E
42	3	3	73	202	383	T	471	T	13	T	-	E
43	3	6	227	655	97959	T	90696	T	223	T	-	E
44	3	9	487	1426	-	E	-	E	10387	T	-	E
45	3	9	237	646	-	E	-	E	804	T	-	E
46	3	7	784	2278	-	E	-	E	432	T	-	E
47	3	7	817	2377	-	E	-	E	1585	T	-	E
48	3	8	272	766	-	E	-	E	765	T	-	E
49	3	12	209	547	-	E	-	E	1195	T	-	E
50	3	16	207	478	23098	T	-	E	-	E	-	E
51	3	6	165	406	316	T	729	T	116	T	-	E
52	3	6	167	412	316	T	744	T	106	T	-	E
53	3	6	200	508	473	T	991	T	116	T	-	E
54	3	1	19	52	23	T	25	T	4	T	25	E
55	3	2	140	412	158	T	186	T	29	T	-	E
56	3	3	483	1438	3239	T	1908	T	70	T	-	E
57	3	2	349	918	1661	F	-	E	60	F	-	E
58	5	4	698	1857	18440	F	-	E	50	F	-	E
59	7	6	1047	2796	71812	F	-	E	50	F	-	E
60	3	2	349	918	1658	F	-	E	70	F	-	E

61	3	4	698	1857	18453	F	-	E	22	F	-	E
62	3	6	1047	2796	71911	F	-	E	85	F	-	E
63	3	1	228	596	418	F	-	E	4	F	-	E
64	5	2	456	1200	3626	F	-	E	9	F	-	E
65	7	3	684	1804	13446	F	-	E	19	F	-	E
66	3	1	228	596	417	F	-	E	3	F	-	E
67	3	2	456	1200	3624	F	-	E	11	F	-	E
68	3	3	684	1804	13464	F	-	E	37	F	-	E
69	3	2	349	918	1697	F	-	E	52	F	-	E
70	5	4	698	1857	19070	F	-	E	43	F	-	E
71	7	6	1047	2796	71723	F	-	E	45	F	-	E
72	3	2	349	918	1666	F	-	E	80	F	-	E
73	3	4	698	1857	19176	F	-	E	22	F	-	E
74	3	6	1047	2796	71881	F	-	E	82	F	-	E
75	3	2	350	921	1701	F	-	E	8	F	-	E
76	5	4	700	1860	19421	F	-	E	43	F	-	E
77	7	6	1050	2799	72768	F	-	E	77	F	-	E
78	3	2	350	921	1697	F	-	E	8	F	-	E
79	3	4	700	1860	19423	F	-	E	80	F	-	E
80	3	6	1050	2799	72652	F	-	E	108	F	-	E
81	3	2	346	899	1732	F	-	E	7	F	-	E
82	5	4	692	1816	19470	F	-	E	46	F	-	E
83	7	6	1038	2733	74252	F	-	E	78	F	-	E
84	3	2	346	899	1732	F	-	E	8	F	-	E
85	3	4	692	1816	19520	F	-	E	85	F	-	E
86	3	6	1038	2733	73794	F	-	E	104	F	-	E
87	3	12	87	190	424	T	7160	T	485	T	-	E
88	3	9	303	874	2223	T	21154	T	386	T	-	E
89	3	1	51	148	50	T	61	T	5	T	-	E
90	3	2	289	859	800	T	385	T	21	T	-	E
91	3	3	933	2788	65135	T	8203	T	94	T	-	E
92	3	6	582	1711	11471	T	13226	T	244	T	-	E
93	3	1	53	154	48	T	60	T	4	T	-	E
94	3	2	244	724	436	T	295	T	18	T	-	E
95	3	3	755	2254	24898	T	5136	T	149	T	-	E
96	3	8	381	1102	6058	T	17874	T	981	T	-	E

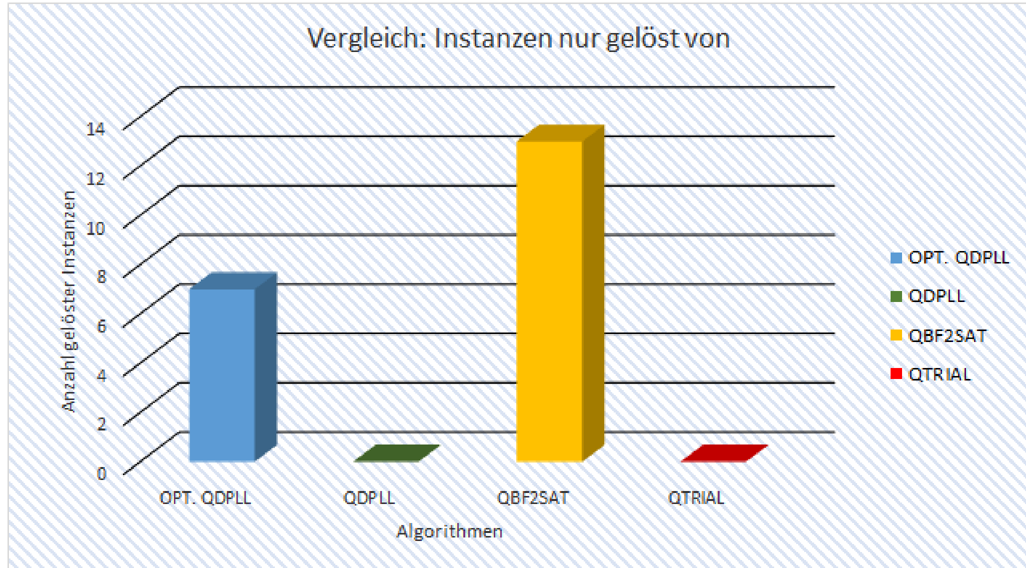
Tabelle 4.1

Diagramm 1.1



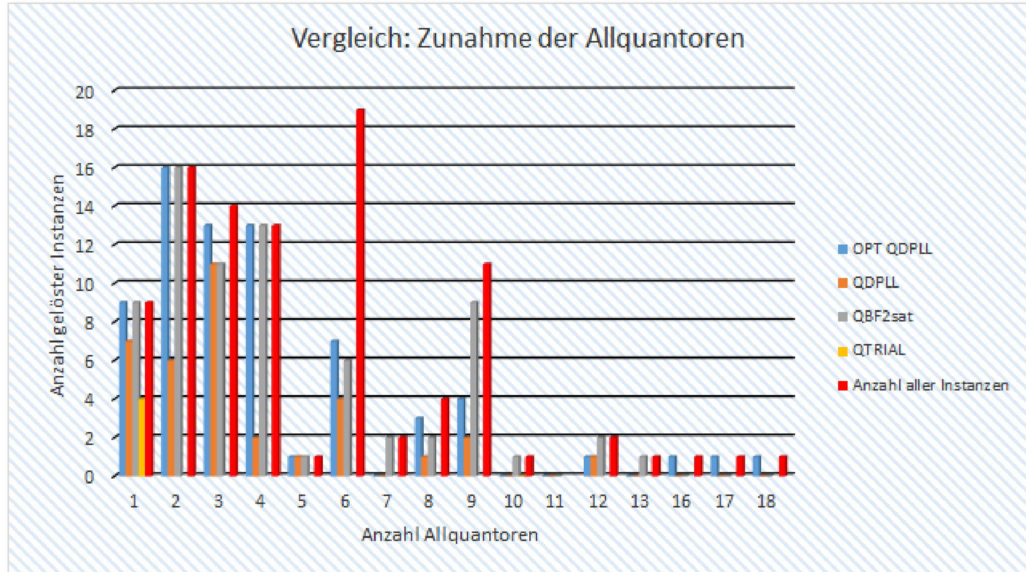
Das Diagramm 1.1 zeigt die Anzahl der gelösten Instanzen durch die unterschiedlichen Algorithmen. Dabei wurde zwischen gültigen und widersprüchlichen Formeln unterschieden. Aus dem Diagramm 1.1 wird ersichtlich, dass der *QTrial*-Algorithmus äußerst schlecht abschneidet und lediglich eine geringe Anzahl an gelösten Instanzen vorzuweisen hat. Die Resultate des *QDPLL* befinden sich im mittleren Bereich, aber sind deutlich besser als die des *QTrial*-Algorithmus. Weiterhin ist zu erwähnen, dass die gelösten Instanzen durch den *QDPLL*-Algorithmus alle gültig sind. Die besten Testresultate liefern der optimierte *QDPLL* und der *QF2SAT*. Beide lösen über 80 Instanzen, wobei der *QBF2SAT* mehr gültige als widersprüchliche Formeln löst.

Diagramm 1.2



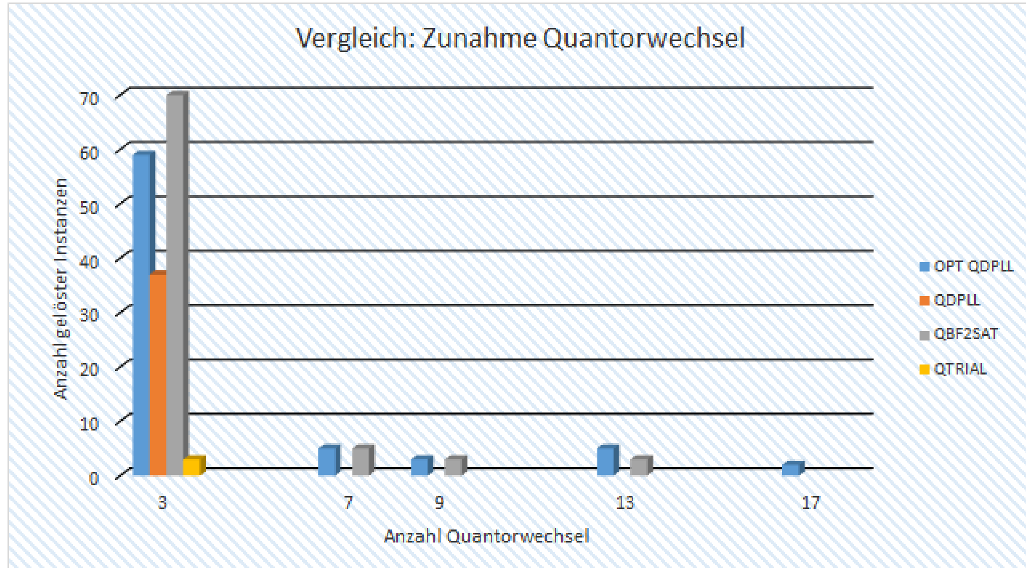
Manche Instanzen können nur von einen bestimmtem Algorithmus gelöst werden. Die Fragestellung, welche Instanzen von genau nur einem Algorithmus gelöst wurde, wird in diesem Diagramm dargelegt. Dabei ist ganz offensichtlich, dass der *QTrial* und der *QDPLL*-Algorithmus keine einzige Instanz lösen konnten, die nicht auch vom optimierten *QDPLL* als auch vom *QBF2SAT* gelöst werden konnte. Der optimierte *QDPLL* und der *QBF2SAT* enthalten Instanzen, die speziell nur von diesen gelöst werden konnten. Das interessante hierbei ist, dass *QBF2SAT* fast doppelt so viele Instanzen wie der optimierte *QDPLL* löst.

Diagramm 1.3



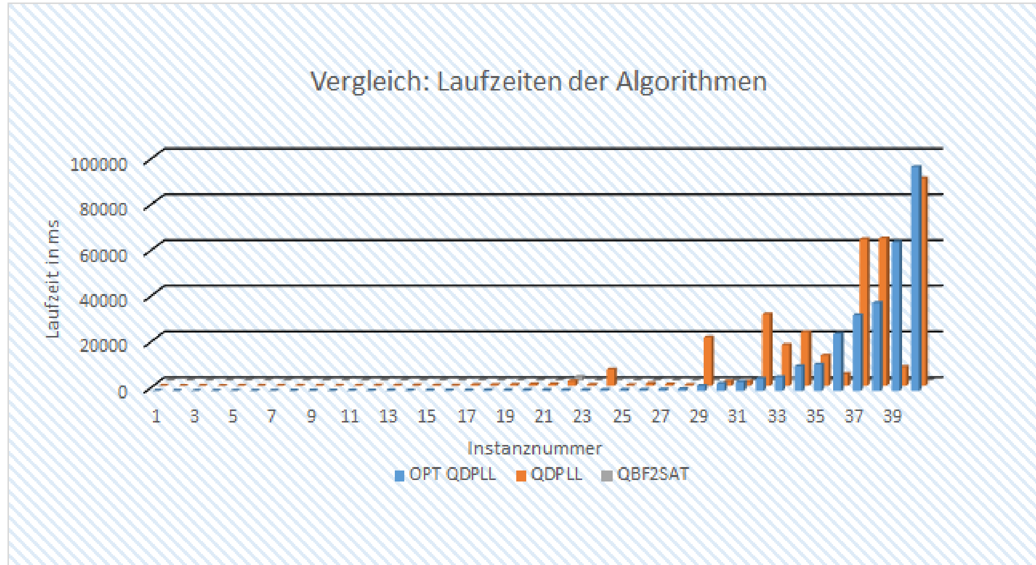
Das Diagramm 1.3 zeigt die Anzahl der gelösten Instanzen in Abhängigkeit zur Anzahl der Allquantoren. Das Testergebnis ist ein wenig unfair, da die Anzahl der Instanzen in den jeweiligen Allquantor Kategorien nicht gleich ist. Trotzdem zeigt sich die Tendenz, dass bei Zunahme der Allquantoren, eine Abnahme der Anzahl der gelösten Instanzen erfolgt. Im Detail wird deutlich, dass der *QTrial*-Algorithmus bereits nach Hinzunahme eines zweiten Allquantors aufgibt. Im Allgemeinen kann dieser nur Instanzen mit einem einzigen Allquantor lösen. Der *QDPLL*-Algorithmus hingegen schlägt sich wesentlich besser. Jedoch kann dieser auch nur Instanzen mit maximal zwölf Allquantoren lösen und davon auch nicht alle. Alles in allem erzielen der optimierte *QDPLL* und der *QBF2SAT*-Algorithmus die besten Testresultate. Sie schaffen es Instanzen in jeder Allquantor Kategorie zu lösen.

Diagramm 1.4



In Diagramm 1.4 wurde die Fragestellung erörtert, wie sich die Anzahl der gelösten Instanzen in Abhängigkeit zur Anzahl der Quantorwechsel verhält. Leider gibt es im Diagramm einen sprunghaften Fall zwischen den Werten 3 und 7. Der Bereich zwischen drei und sieben Quantorwechseln wird leider nicht getestet, da die QBF Datenbank keine Instanzen mit vier und sechs Quantorwechsel geeigneter Größe enthalten hat. Für die Verfahren *QTrial* und *QDPLL* lässt sich aus dem Diagramm ebenfalls ein Resultat ziehen. Bereits bei weniger als sieben Quantorwechsel können sie keine Instanz mehr lösen.

Diagramm 1.5



Das letzte Diagramm 1.5 zeigt die Laufzeit der verschiedenen Algorithmen bei verschiedenen Instanzen. Der *QTrial*-Algorithmus wird hierbei außer Acht gelassen, da dieser nicht genügend Instanzen lösen konnte. Laufzeitunterschiede sind beim *QBF2SAT* kaum bemerkbar, da dieser eine Instanz entweder löst oder nicht löst. Bei den anderen beiden Verfahren erkennt man, dass die Laufzeiten ansteigen und der optimierte *QDPLL* schneller ist. Jedoch muss man dazu sagen, dass der optimierte *QDPLL* an manchen Stellen wie beispielsweise bei Instanznummer 35 langsamer ist.

Kapitel 5

Zusammenfassung und Ausblick

5.1 Zusammenfassung

Das Ziel der Arbeit bestand darin, Algorithmen zum schnellen Lösen von quantifizierten booleschen Formeln in Java zu implementieren. Kapitel 1 enthält eine Einleitung in die Thematik und einen groben Überblick der einzelnen Themenbereiche der Arbeit. Im darauf folgendem Kapitel 2 werden notwendige Grundlagen für die Lösungsverfahren erörtert. Dabei werden zu Beginn die Grundlagen der Aussagenlogik beschrieben und betont, das dieses logische System die Grundlage der Quantifizierten-Aussagenlogik bildet. Daraufhin wird die Quantifizierte-Aussagenlogik präsentiert, wobei vor allem dessen Erfüllbarkeitsproblem $QSAT$ im Fokus steht.

Anschließend werden die theoretischen Komplexitätsklassen definiert und die Einordnung des $QSAT$-Problems in die Klasse $\mathcal{PSPACE}$ vorgenommen. Danach werden Bestandteile, Anwendungsgebiete und Beispiele für Solver gegeben. Hierbei wird speziell auf den *DPLL*-Algorithmus, der Grundlage der meisten Solver ist, eingegangen. Zum Schluss des Kapitels wird dem Leser ein aktueller Stand der Technik gewährt, worin alternative Vorgehensweisen zur Lösung von $QSAT$ vorgestellt werden. In Kapitel 3 werden Lösungsverfahren zur Überprüfung der Gültigkeit von QBFs beschrieben. Insgesamt werden drei unterschiedliche Verfahren vorgestellt. Das erste und einfachste Verfahren ist das *QTrial*-Verfahren. Es ist ein naiver Algorithmus, der durch Ausprobieren versucht die Gültigkeit einer Formel zu überprüfen. Das zweite Verfahren $QBFtoSAT$ verfolgt den Ansatzpunkt QBFs in äquivalente aussagenlogische Formeln zu transformieren und anschließend deren Gültigkeit mit einem SAT-Solver zu überprüfen. Das dritte und letzte Verfahren namens *QDPLL* basiert auf dem *DPLL*-Verfahren für QBFs. Bei diesem Verfahren ist die Klassifizierung von Klauseln als PK-Wahre, PK-Falsche oder PK-Offene Klausel äußerst relevant.

Im letzten Kapitel 4 werden wichtige Eigenschaften, Anwendungsgebiete, Konzepte sowie Datenstrukturen der Programmiersprache Java dargelegt.

Daraufhin wird das abstrakte Klassenkonzept und die Formeldarstellung des Programms erläutert. Anschließend wird die Implementierung der unterschiedlichen Lösungsverfahren mit Codesequenzen vorgestellt. Zum Schluss des Kapitels werden die Ergebnisse der experimentellen Analyse erörtert. Dort werden die Verfahren mit verschiedenen QBF-Instanzen auf ihre Laufzeit und Lösungsverhalten getestet.

5.2 Fazit und Ausblick

Die Ziele der Arbeit wurden erfolgreich umgesetzt und ein funktionsfähiges Programm steht als Ergebnis zur Verfügung, welches QBFs auf Erfüllbarkeit testet. Das erste Ziel bestand darin, unterschiedliche *QSAT*-Verfahren auszuwählen und allgemein verständlich darzustellen. Hierfür wurden drei Verfahren ausgewählt, die sich in ihrem Vorgehen grundsätzlich unterscheiden. Das erste Verfahren, das ausgesucht wurde, heißt *QTrial* und diente als Einstieg in die QBF-Lösung. Hierbei handelt es sich um einen naiven und einfach zu implementierenden Algorithmus. Im Folgenden wurde der QBF-in-SAT Algorithmus namens *QBFtoSAT* selektiert. Dieser verwendet am Ende seines Verfahrens einen schnellen *SAT*-Solver. *QDPLL* ist ein Algorithmus basierend auf dem *DPLL*-Verfahren und Bestandteil des dritten Verfahrens. *QDPLL* ist eine adaptierte Variante, die direkt auf den QBFs arbeitet.

Weitere Ziele lagen in der Implementierung der Verfahren in der Programmiersprache Java und der objektorientierten Modellierung von Formeln. Alle Verfahren wurden in Java implementiert und für die Formeldarstellung wurde ein objektorientiertes Modell entworfen. Dieses besteht aus einer abstrakten Oberklasse *Formel* und weiteren abgeleiteten Unterklassen. Die Formeldarstellung scheint geeignet zu sein, da sich nur syntaktisch korrekte Formeln generieren lassen. Jedoch hat diese Darstellungsform auch den Nachteil, das sie äußerst umständlich zu programmieren ist.

Das nächste Ziel bestand darin, die verschiedenen Verfahren auf ihre Laufzeit zu testen und experimentell miteinander zu vergleichen. Hierbei kamen folgende Erkenntnisse zum Vorschein: *QTrial* ist nur für kleine Formeln mit wenig Quantorwechseln geeignet. Zudem hat sich gezeigt, dass das optimierte *QDPLL*-Verfahren in seiner Performance besser ist, als das entsprechende Verfahren ohne Optimierung. Außerdem sollte *QBFtoSAT* nur bei Formeln mit möglichst wenig Allquantoren verwendet werden, da bei der Eliminierung eines Allquantors sich die Formelgröße verdoppelt. Diese Erkenntnisse decken sich mit den Annahmen aus Kapitel 3.

Das letzte Ziel dient für didaktische Zwecke d.h. es soll beim Umgang und dem Lösen von QBFslhelfen Ziel war es eine Schnittstelle in Form einer GUI zu entwickeln um diese dann in Lehrveranstaltungen einzusetzen. In der GUI können zum einen Formeln direkt eingegeben oder hochgeladen werden und zum anderen lassen sich per Knopfdruck die unterschiedlichen Algorithmen ausführen. Dieses Ziel wurde erfolgreich umgesetzt

Als Ausblick könnte man versuchen die nicht zufriedenstellende Darstellungsform zu optimieren. Eine mögliche Variante wäre, dass man die Junktoren *And* und *Or* nicht binär sonder n-när darstellt d.h mit einer Liste von beliebig vielen Argumenten. Dies würde eine effizientere Implementierung an einigen Stellen wie beispielsweise bei der Berechnung der Klauselnormalform, die mit Listen von Argumenten arbeitet, bewirken. Desweiteren ist das Finden von Testfällen äußerst schwierig. Daher kann die experimentelle Analyse, aufgrund der betrachteten Testfälle, kein endgültiges Ergebnis liefern. Das generische Erzeugen von Formeln hätte in diesem Fall nur wenig Sinn gemacht. Daher sollten in Zukunft mehr Testfälle öffentlich bereitgestellt werden. Für Formeln die nicht in Normalform vorliegen, könnten sich die Ergebnisse der Laufzeiten eventuell verändern. Der Grund liegt darin, das *QTrial* direkt auf den Formeln arbeiten kann, wobei *QBFtoSAT* und *QDPLL* unter Umständen die Formel exponentiell vergrößern. Leider wurde dafür keine Benchmark gefunden. Daher ist als Ausblick die Forderung nach einem Benchmark Set.

Literaturverzeichnis

[BA02] Christel Baier and Alexander Asteroth. *Theoretische Informatik - eine Einführung in Berechenbarkeit, Komplexität und formale Sprachen mit 101 Beispielen.* Pearson Studium, München, 1. aufl. edition, 2002.

[Bec06] Bernhard Beckert. Vorlesung Normalformen Logik für Informatiker. `formal.iti.kit.edu/~beckert/teaching/Logik-SS06/05AussagenlogikNormalformen_Teil1.pdf`, 2006.

[Bec08] Peter Becker. Vor- und Nachteile Java. `http://www2.inf.fh-bonn-rhein-sieg.de/~pbecke2m/progjava/abstrKlassen1.pdf`, 2008.

[bil] bildSolver. Bildsol. `http://www.bildagentur-illustrationen.de/wp-content/uploads/2010/08/illu,http://cdn.vectorstock.com/i/composite/57,92/music-robot-vector-1185792.jpg,http://previews.123rf.com/images/michaeldb/michaeldb0809/michaeldb080900029/3593973-car-buying-a-black-used-or-new-auto-inside-a-shopping-cart-proceed-to-`

[Bos98] Rainer Bosch. Diplomarbeit Theorembeweisen für Qbf und die Anwendung zum Planen. `http://www.informatik.uni-ulm.de/ki/Edu/Diplomarbeiten/rbosch-dipl.html`, 1998.

[Bre10] Gerhard Brewka. Teil III: Komplexitätstheorie. `http://www.informatik.uni-leipzig.de/~brewka/papers/Komplexit.pdf`, 2010.

[Bre15] Universität Bremen. Mathematische und logische grundlagen der Linguistik. `http://www.fb10.uni-bremen.de/khwagner/grundkurs2/kapitel3.aspx`, 2015.

[Dar09] TU Darmstadt. Allgemeine Informatik /Java Grundbegriffe. `http://www.ke.tu-darmstadt.de/vl/pub/AI2-11/Material/AI-2-1-Java.pdf`, 2009.

[Das05] Jürgen Dassow. *Logik für Informatiker.* Vieweg Teubner Verlag, Wiesbaden, 2005.

[Dew13] A.K. Dewdney. *Der Turing Omnibus - Eine Reise durch die Informatik mit 66 Stationen.* Springer-Verlag, Berlin Heidelberg New York, 2013.

[Ecl15] Eclipse. Windowbuilder - is a powerful and easy to use bi-directional Java GUI designer. https://eclipse.org/windowbuilder/, 2015.

[EG11] Armando Tacchella Enrico Giunchiglia, Massimo Narizzano. Clause Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas. http://adsabs.harvard.edu/abs/2011arXiv1111.0860G, 2011.

[ES04] Niklas Een and Niklas Sorenson. The boolean satisfaction and optimization library in Java. http://sat4j.org/allabout.php, 2004.

[fdLaG15] Staatliches Studienseminar für das Lehramt an Gymnasien. Andere Sprachen, andere Symbole! http://www.inf-schule.de/information/darstellunginformation/binaerdarstellungzeichen/konzept_unicode, 2015.

[Gar11] Angelo Gargantini. Windowbuilder Pro Tutorial. http://cs.unibg.it/scandurra/material/INF3B_1112/windowbuilderTutorial.pdf, 2011.

[GM09] Carsten Gremzow and Nico Moser. *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen.* Univerlagtuberlin, Berlin, 1. aufl. edition, 2009.

[Got03] Prof. Dr. Siegfried Gottwald. Skripte zur Vorlesung Klassische Logik I. http://www.uni-leipzig.de/logik/gottwald/KL1fin.pdf, 2003.

[Gro09] Martin Groetschel. Das Problem mit der Komplexität. https://www.zib.de/groetschel/pubnew/paper/, 2009.

[GRR+10] Frank Gurski, Irene Rothe, Jörg Rothe, Egon Wanke, Irene Rothe, Egon Wanke, and Jorg Rothe. *Exakte Algorithmen für schwere Graphenprobleme -.* Springer-Verlag, Berlin Heidelberg New York, 1. aufl. edition, 2010.

[Gö04] Kevin Göser. Hauptseminar Automatisches Beweisen: Der Davis-Putnam Algorithmus. http://www.informatik.uni-ulm.de/ki/Edu/Seminare/Automatisches.Beweisen/SS04/Ausarbeitungen/01-Goeser.pdf, 2004.

[Hen14] Christian Henning. Qbf-Solver. http://www.thi.uni-hannover.de/fileadmin/forschung/arbeiten/henning-ba.pdf, 2014.

[Idt04] Viktoria Idt. Quantifizierte Boolesche Formeln: Analyse der Erfüllbarkeit. http://www.informatik.uni-ulm.de/ki/Edu/Seminare/Automatisches.Beweisen/SS04/Ausarbeitungen/04-Idt.pdf, 2004.

[JG13] Matti Jarvisalo and Allen Van Gelder. *Theory and Applications of Satisfiability Testing SAT 2013 16th International Conference, Helsinki, Finland, July 8-12, 2013, Proceedings.* Springer, Berlin, Heidelberg, 2013.

[Jü13] Thomas Jüngling. Datenvolumenverdoppelt sich alle zwei jahre. http://www.welt.de/wirtschaft/webwelt/article118099520/Datenvolumenverdoppelt-sich-alle-zwei-Jahre.html, 2013.

[Kar15] Chris Karakas. Eigenschaften von Java. http://www.karakas-online.de/teia/JAVA/java121.htm, 2015.

[KH14] Guido Krüger and Heiko Hansen. *Java-Programmierung - Das Handbuch zu Java 8*. OReilly Germany, Köln, 2014.

[Kok99] Dr. Bernd Kokavecz. Was sind Objekte in der OOP. http://www.b.shuttle.de/b/humboldt-os/python/kapitel1/oop.html, 1999.

[Kot07] Stephan Kottler. Diplomarbeit Backdoors in SAT-instanzen. http://www.satcompetition.org, 2007.

[KR10] Marco Thomas Katja Röttinger, Prof. Dr. Andreas Schwill. Vor- und Nachteile Java. http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/SeminarDidaktik/OOP/java_vor_nachteile.html, 2010.

[Kre06] Stefan Kreuzer, Martin und Kühling. *Logik für Informatiker*. Pearson Studium, München, 1. aufl. edition, 2006.

[KT12] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Pearson Education, Amsterdam, 2012.

[Kul09] Oliver Kullmann. *Theory and Applications of Satisfiability Testing - SAT 2009 - 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*. Springer Science and Business Media, Berlin Heidelberg, 2009. aufl. edition, 2009.

[Kü07] Thomas Künneth. *Einstieg in Eclipse 3.3 - Einführung, Programmierung, Plugin-Nutzung ; [für Windows, MacOS X und Linux geeignet, RCP-, Web- und Ajax-Anwendungen entwicklen, inkl. Ant, Refactoring, Debugging, Subversion, CVS, Plug-ins ; DVD-ROM: Eclipe 3.3, Plug-ins, alle Beispiele, Video-Lektionen]*. Galileo Press, Bonn, 1. auflage edition, 2007.

[Lan12] Martin Lange. SAT-Solver. http://www.uni-kassel.de/eecs/fileadmin/datas/fb16/Fachgebiete/FMV/TIL-WS1213/aussagenlogik-satsolver.pdf, 2012.

[Let13] Theodor Lettmann. *Aussagenlogik: Deduktion und Algorithmen - Deduktion und Algorithmen*. Springer-Verlag, Berlin Heidelberg New York, 2013.

[Mil13] R. Miller. *Complexity of Computer Computations - Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of*

Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department. Springer Science and Business Media, Berlin Heidelberg, 2013.

[NBP15] Raja Natarajan, Gautam Barua, and Manas Ranjan Patra. *Distributed Computing and Internet Technology - 11th International Conference, ICDCIT 2015, Bhubaneswar, India, February 5-8, 2015. Proceedings.* Springer, Berlin, Heidelberg, 2015.

[oAI09] Theory of Artificial Intelligence. P vs NP. `http://www.informatik.uni-bremen.de/tdki/lehre/ss09/kt/Kapitel3.pdf`, 2009.

[Oma14] Süleyman Omari. Entwurf und Implementierung einer Kodierung der Allenschen Zeitlogik als Aussagenlogisches Erfüllbarkeitsproblem. `http://www.ki.informatik.uni-frankfurt.de/bachelor/abgeschlossen.html`, 2014.

[oMLaP01] Laboratory of Mathematical Logic at PDMI. Quantified Boolean Formulas Satisfiability Suggested Format. `http://logic.pdmi.ras.ru/~basolver/dimacs.html`, cited 2001.

[Ora15a] Oracle. Erfahren Sie mehr über die Java-Technologie. `http://www.java.com/de/about/`, 2015.

[Ora15b] Oracle. Java Platform, Standard Edition 7 API Specification. `http://docs.oracle.com/javase/7/docs/api/`, 2015.

[Par12] Terence Parr. *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf, Raleigh, Noth Caolina and Dallas, Texas, new edition. edition, 2012.

[Pla] Plattforumunabhängigkeit. Java plattformunabhängigkeit. `http://java.sampleexamples.com/files/2013/02/run-java-example.png`.

[QBF15a] QBFLIB. QBFLIB Iinstances. `http://www.qbflib.org/`, 2015.

[QBF15b] QBFLIB. Qdimacs Standard. `http://www.qbflib.org/qdimacs.html`, 2015.

[Ren14] Burkhardt Renz. Logik und formale Methoden. `https://homepages.thm.de/~hg11260/mat/lfm.pdf`, 2014.

[Rin99] Jussi Rintanen. Constructing Conditional Plans by a Theorem-Prover. `http://arxiv.org/pdf/1105.5465.pdf`, 1999.

[Ros03] Francesca Rossi. *Principles and Practice of Constraint Programming - CP 2003 - 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings.* Springer Science and Business Media, Berlin Heidelberg, 2003. aufl. edition, 2003.

[Rot08] Jörg Rothe. *Komplexitätstheorie und Kryptologie - Eine Einführung in Kryptokomplexität.* Springer-Verlag, Berlin Heidelberg New York, 2008.

[SC09] SAT-Competition. SAT-Competition: Benchmark Dimacs-Format. `http://www.satcompetition.org/2009/format-benchmarks2009.html`, 2009.

[Sch79] Peter Schefe. *On Foundations of Reasoning with Uncertain Facts and Vague Concepts.* Fachbereich Informatik, Universität Hamburg, 1979.

[Sch02] Prof. Dr. Uwe Schmidt. Lazy Evaluation. `http://www.fh-wedel.de/~si/seminare/ss02/Ausarbeitung/3.lazy/lazy2.htm`, 2002.

[Sch11] Johannes Schaback. Java Hashmap. `http://schabby.de/hashmap/`, 2011.

[Sch13] Professor Dr. Georg Schnitger. Theoretische Informatik I. `ttp://www.thi.informatik.unifrankfurt.de/lehre/gl1/ws1213`, 2013.

[Sim07] Dozent Nino Simunic. Vorlesung Grundlegene Programmiertechniken. `https://www.uni-due.de/imperia/md/content/computerlinguistik/prog0708_v_01.pdf`, 2007.

[Ste07] Bernhard Steppan. Eclipse: Ein Ökosystem für Entwickler-Tools. `http://www.computerwoche.de/a/eclipse-ein-oekosystem-fuer-entwickler-tools,590303`, 2007.

[SW12] Jörg Siekmann and G. Wrightson. *Automation of Reasoning - 2: Classical Papers on Computational Logic 1967-1970.* Springer Science and Business Media, Berlin Heidelberg, 2012.

[uBP15a] Bjorn und Britta Petri. Java Tutorial Hashsets. `http://www.javatutorial.org/hashset.html`, 2015.

[uBP15b] Bjorn und Britta Petri. Java Tutorial Linkedlist. `http://www.javatutorial.org/linkedlist.html`, 2015.

[uDDS13] Manfred Schmidt-Schauß und Dr. David Sabel. Skript zur Vorlesung Einführung in die Methoden der Künstlichen Intelligenz im Wintersemester 2012/2013. `http://www.ki.informatik.uni-frankfurt.de`, 2013.

[Ull12] Christian Ullenboom. *Java 7 - mehr als eine Insel - das Handbuch zu den Java SE-Bibliotheken ; [Nebenläufigkeit, Datenstrukturen und Algorithmen, Ein/Ausgabe, XML, Swing, Grafik- und Netzwerkprogrammierung, RMI, JavaServer Pages und Servlets, JDBC, Reflection u.v.m.].* Galileo Press, Bonn, 1. aufl. edition, 2012.

[Ull14] Christian Ullenboom. *Java ist auch eine Insel - Programmieren mit dem Standardwerk für Entwickler, aktuell zu Java 8.* Rheinwerk Verlag GmbH, Bonn, 11. aufl. edition, 2014.

[Uma15] Chris Umans. Cs 21: Decidability and Tractability. `http://users.cms.caltech.edu/~umans/cs21/lec24.pdf`, 2015.

[uMJU] Professor Dr. Manfred Schmidt-Schauß-Fachbereich Informatik und Mathematik J.W.Goethe-Universit˙ Automatische Deduktion Skript.

[vMF15] Hans van Maaren and John Franco. The international SAT Competitions web page. `http://www.satcompetition.org`, 2015.

[vtUCRF01] BASolver Homepage via the U.S. Civilian Research and Development Foundation. Dimacs CNF Format. `http://logic.pdmi.ras.ru/~basolver/dimacs.html`, 2001.

[Vö10] Prof. Dr. Berthold Vöcking. NP-Vollständigkeit und der Satz von Cook und Levin. `http://algo.rwth-aachen.de/Lehre/WS0910/VBuK/Folien/komplexitaet3.pdf`, 2010.

[Wec13] Gerd Wechsung. *Vorlesungen zur Komplexitätstheorie* -. Springer-Verlag, Berlin Heidelberg New York, 2013.

[Wie98] Oswald Wiener. *Eine elementare Einführung in die Theorie der Turing-Maschinen.* Springer, Berlin, Heidelberg, 1998.