Java Klassenbibliothek zur Darstellung und Manipulation von Booleschen Funktionen, basierend auf BDD's

Studienarbeit

Fachbereich Informatik Universität Hamburg

vorgelegt von Christian Langmann

Februar 1998

1 Einleitung

1.1 Überblick

In dieser Arbeit wird eine Java Klassenbibliothek zur Darstellung und Manipulation von Booleschen Funktionen basierend auf „binary decision diagrams“ (BDD’s) vorgestellt. Außerdem wird die Funktionsweise des Paketes anhand von Codebeispielen erläutert. Das Paket kann unter “http://tech-www.informatik.uni-hamburg.de/applets/bdd“ bezogen werden.
Die Strukturen und Methoden zur Repräsentation von BDD’s basieren auf den Algorithmen, die Bryant 1986 eingeführt hat [Bryant 86]. Obwohl BDD’s, welche Boolesche Funktionen repräsentieren, im schlechtesten Fall eine exponentielle Größe haben, gibt es für viele in der Praxis relevante Funktionen eine deutlich akzeptablere Größe. Die hier vorgestellten Algorithmen haben einen Zeitaufwand, der proportional zu der Größe der Graphen ist, auf denen gearbeitet wird. Zusätzlich zu den von Bryant eingeführten Strukturen sind in diesem Paket Methoden zur Visualisierung von BDD’s enthalten, die hier ebenfalls vorgestellt werden.
Im folgenden werden die Grundlagen der Booleschen Algebra und einige mögliche Repräsentationen von Booleschen Funktionen wie Wertetabellen, konjunktive und disjunktive Normalformen und BDD’s erläutert [DeMicheli 94]. In Abschnitt 1.3 wird ein kurzer Abriß über den Aufbau dieser Arbeit gegeben.

1.2 Einleitung

Die Boolesche Algebra wurde von G. Boole 1853 entwickelt [Duden 93].

Definition 1.1 [Duden 93]:

Eine Menge M mit zwei Verknüpfungen · und + heißt Boolesche Algebra, wenn für alle $x, y, z \in M$ gilt:

a) Assoziativgesetz:
   $x \cdot (y \cdot z) = (x \cdot y) \cdot z$;
b) Assoziativgesetz:
   $x + (y + z) = (x + y) + z$;
c) Kommutativgesetz:
   $x \cdot y = y \cdot x$;
d) Kommutativgesetz:
   $x + y = y + x$;
e) Absorptionsgesetz:
   $x \cdot (x + y) = x$;
f) Absorptionsgesetz:
   $x + (x \cdot y) = x$;
g) Distributivgesetz:

$x \cdot (y+z) = (x \cdot y) + (x \cdot z)$;

h) Distributivgesetz:
$x + (y \cdot z) = (x+y) \cdot (x+z)$;

i) Existenz eines neutralen Elements:
es gibt ein Element $0 \in M$ mit $0 \cdot x = 0$ und $0 + x = x$ für alle $x \in M$;

j) Existenz eines neutralen Elements:
es gibt ein Element $1 \in M$ mit $1 \cdot x = x$ und $1 + x = 1$ für alle $x \in M$;

k) Existenz des komplementären Elements:
zu jedem $x \in M$ existiert genau ein $y \in M$ mit $x \cdot y = 0$ und $x + y = 1$.

Ein Beispiel für eine Boolesche Algebra sind die Verknüpfungen „AND“ ($\wedge$) und „OR“ ($\vee$).

Definition 1.2 [Duden 93]:

Eine Boolesche Funktion, ist eine Abbildung $f:\{0,1\}^n \rightarrow \{0,1\}$, wobei $\{0,1\}^n$ die Menge aller n - Tupel über $\{0,1\}$ ist.

Die Boolesche Algebra ist ein Meilenstein in der Informatik. Alle grundlegenden Schaltungen in heutigen digitalen Computern folgen ihren Gesetzen. Auch viele Probleme der Künstlichen Intelligenz und der Kombinatorik können durch Boolesche Funktionen beschrieben werden.
Alle diese Anwendungen würden davon profitieren, wenn effiziente Algorithmen zur Repräsentation und Manipulation von Booleschen Funktionen verfügbar wären.

Es gibt viele verschiedene Darstellungsformen für Boolesche Funktionen. Drei besonders wichtige Tabellen, logische Ausdrücke und binary decision diagrams [DeMicheli 94] sollen im folgenden vorgestellt werden.

Die einfachste Darstellungsformen sind Wertetabellen. Sie sind eine Auflistung aller Variablen mit ihren möglichen Belegungen und den dazugehörigen Funktionswerten (siehe Tabelle 1). Diese Darstellungsform ist sehr ineffizient, da die Größe der Wertetabellen bei n Eingangsvariablen auf 2^n anwächst, also exponentiell zur Anzahl ihrer Eingangsvariablen.

a	***b***	***c***	
0	0	0	**0**
0	0	1	**0**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

Tabelle 1: Beispiel einer Wertetabelle für die Funktion $(a \wedge b) \vee (b \wedge c) \vee (a \wedge c)$

Die zweite Darstellungsform sind logische Ausdrücke mit den Operatoren nicht, und, oder xor, darunter die konjunktive („$(a \vee b) \wedge (a \vee c) \wedge (b \vee c)$“) bzw. die disjunktive Normalform („$(a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$“).

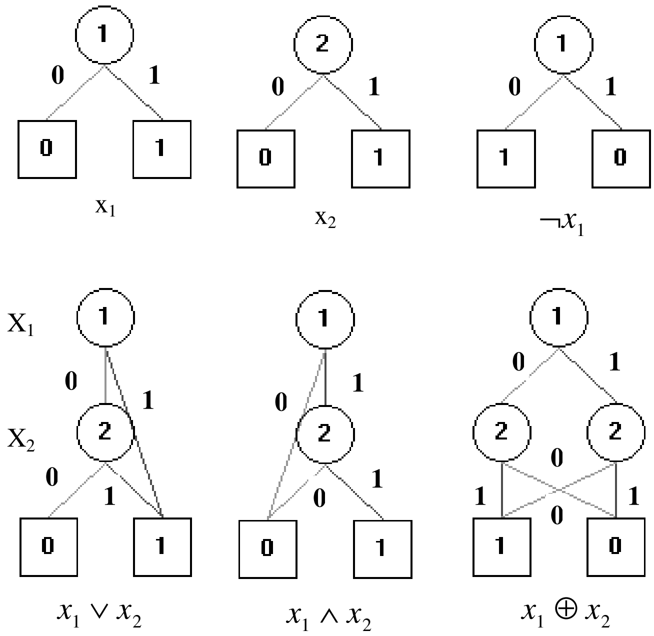


Abbildung 1 : Beispiele für OBDD's für einige elementare Boolesche Operationen

Definition 1.3 [Mertsching 96]:

Ein Boolescher Ausdruck der Form $K_1 \vee K_2 \vee \ldots \vee K_n$ heißt disjunktive Normalform, falls die K_i paarweise verschiedene Konjunktionsterme sind ($n \in N$).

Definition 1.4 [Mertsching 96]:

Ein Konjunktionsterm ist ein Boolescher Ausdruck der Form $x_1 \wedge x_2 \wedge \ldots \wedge x_n$, wobei die x_i entweder logische Variablen oder negierte logische Variablen sind ($n \in N$).

Definition 1.5 [Mertsching 96]:

Ein Boolescher Ausdruck der Form $D_1 \wedge D_2 \wedge \ldots \wedge D_n$ heißt konjunktive Normalform, falls die D_i paarweise verschiedene Disjunktionsterme sind ($n \in N$).

Definition 1.6 [Mertsching 96]:

Ein Disjunktionsterm ist ein Boolescher Ausdruck der Form $x_1 \vee x_2 \vee \ldots \vee x_n$, wobei die x_i entweder logische Variablen oder negierte logische Variablen sind ($n \in N$).

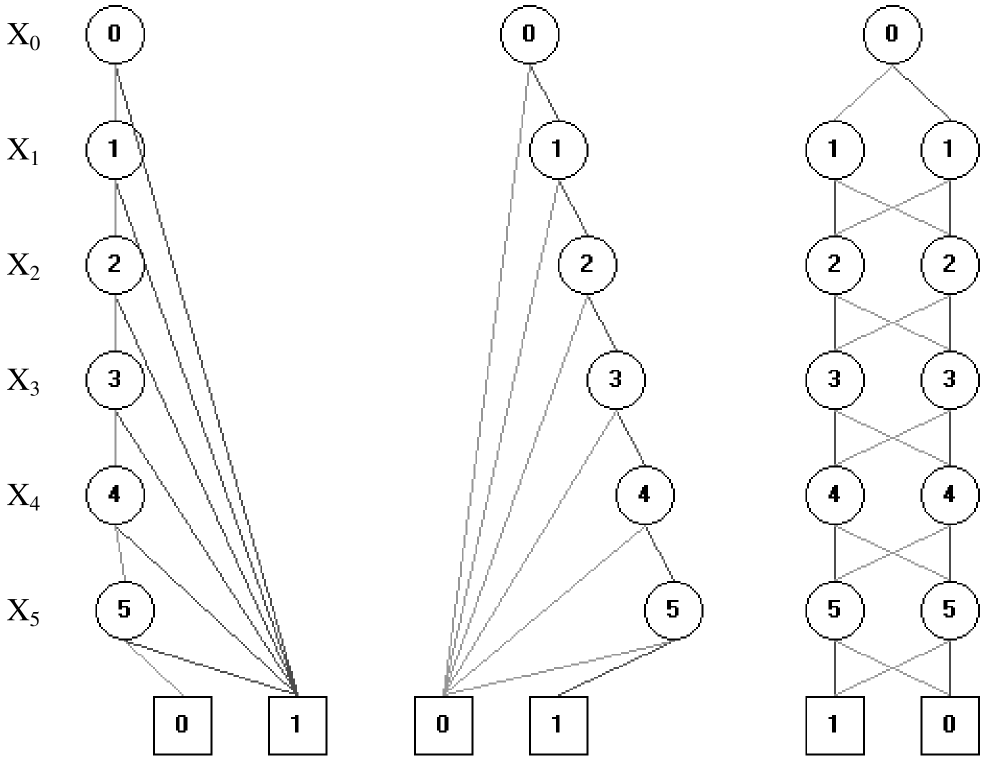


Abbildung 2: Typische OBDD's für die Booleschen Funktionen AND, OR und XOR angewandt auf jeweils 6 Variablen ($x_0 \ldots x_5$).

Es gibt zwar auch andere Formen, die befriedigende Darstellungen erzielen, doch haben auch diese wie die Normalformen entscheidende Nachteile.

1. Diese Darstellungsformen sind zwar in ihrem Umfang nicht immer exponentiell zu ihren Eingabeparametern, aber sie sind nicht kanonisch und somit nicht eindeutig.
2. Einige häufige Funktionen („even" und „odd - parity") können nach wie vor nur mit exponentiellem Aufwand dargestellt werden.
3. Operationen auf logischen Ausdrücken ziehen häufig einen exponentiellen Aufwand nach sich.

Die dritte Darstellungsform sind BDD's.

BDD's sind gerichtete azyklische Graphen, wobei die Entscheidungen den Knoten zugeordnet werden. Nach [Akers 78] wird die Entscheidung durch die Auswertung der Variablen vollführt. [Bryant 86] und [Brace et al. 90] zeigte, daß effiziente Algorithmen unter der Annahme einer Variablenordnung, durchgeführt werden können. Daher werden sie auch „Ordered Binary Decision Diagrams", kurz OBDD's, genannt. Die Variablenordnung bestimmt, welchen Ebenen des BDD's die Variablen zugeordnet werden.

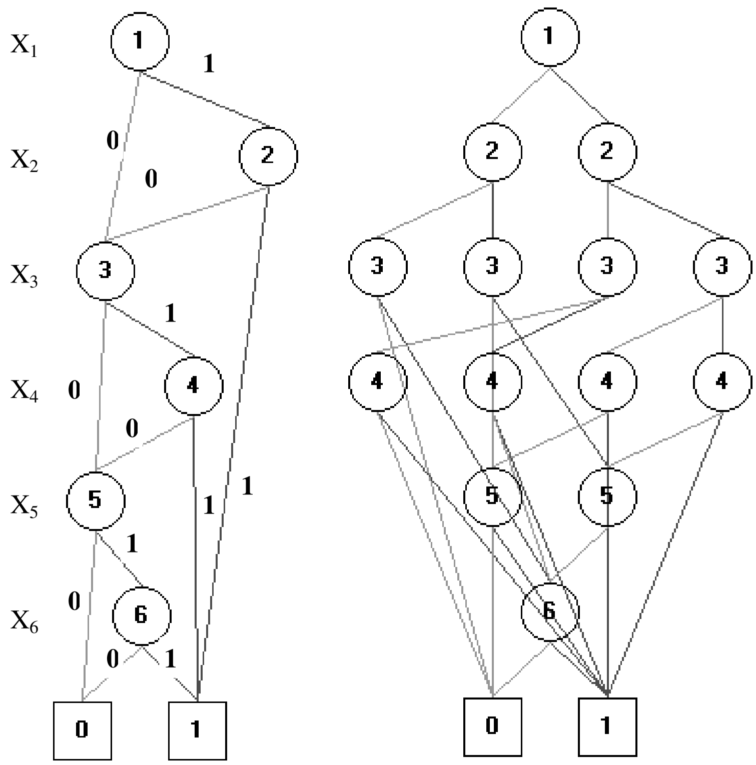


Abbildung 3: Einfluß der Variablenordnung anhand der BDD's für die Fuktionen
$(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6)$ **(links) und** $(x_1 \wedge x_4) \vee (x_2 \wedge x_5) \vee (x_3 \wedge x_6)$ **(rechts)**

Definition 1.7 [Bryant 86]:

Ein OBDD ist ein gerichteter azyklischer Graph mit einer Knotenmenge V. Jeder Knoten v, außer den Blättern, hat als Attribute einen $index(v) \in \{0,1,\ldots,n\}$, der einer Variablen aus der Menge $\{x_0, x_1, \ldots, x_n\}$ zugeordnet ist, und zwei Söhne $leftchild, rightchild \in V$. Die Blätter haben einen Wert $value(v) \in \{0,1\}$.

Für jeden Knoten v, sofern er kein Blatt ist, gilt $index(v) < index(leftchild)$ und $index(v) < index(rightchild)$.

Die Funktionsweise von OBDD's soll mit Hilfe von Abbildung 1 erklärt werden:

Jede Knotenebene, bis auf die der terminalen Knoten, repräsentiert eine Variable. In Abbildung 1 entsprechen runde Knoten mit dem Wert 1 der Variablen x_1 und runde Knoten mit dem Wert 2 der Variablen x_2. Anhand der Graphen ist zu sehen, daß die Variable x_1 in der Ordnung vor der Variablen x_2 steht. Die terminalen Knoten repräsentieren die Werte „0" bzw. „1". Um bei einer möglichen Variablenbelegung den dazugehörigen Funktionswert zu erhalten, werden folgende Schritte durchlaufen:

1. Es wird mit dem Ursprungsknoten begonnen.
2. Ist der Knoten ein terminaler Knoten, so ist der Funktionswert gleich dem Wert des terminalen Knoten und das Verfahren beendet.
3. Ist die Belegung der Variablen, die den Knoten repräsentiert, gleich 0, so wird zu dem Knoten entlang der Kante gegangen, die 0 repräsentiert. In den Beispielen ist die 0 - Kante grün.
4. Wenn die Belegung der Variablen 1 ist, wird der Kante gefolgt, die 1 repräsentiert. In den Beispielen ist diese Kante rot.
5. Angekommen bei dem Knoten, der durch die entsprechende Kante mit dem letzten Knoten verbunden ist, wird wieder bei Schritt 2 begonnen.

OBDD's bieten folgende Vorteile:

1. Sie stellen eine kanonische Repräsentation dar.
   Dadurch sind Vergleiche zwischen BDD's sehr einfach und effizient durchzuführen..
2. Operationen auf OBDD's können in polynomialer Zeit zur Knotenanzahl durchgeführt werden. Dieses bietet einen enormen Vorteil, sofern die Knotenanzahl nicht exponentiell mit der Variablenanzahl wächst.

In Abbildung 1 sind BDD's für zwei Variablen x_1 und x_2 dargestellt. Außerdem werden dort die Booleschen Operationen „nicht", „und", „oder" und das „exklusiv-oder" auf diese Variablen angewendet und als BDD's dargestellt.
Die oben genannten Vorteile von OBDD's bedeuten nicht, daß NP-Probleme, wie das „Erfüllbarkeitsproblem" [Schöning 92], nun einfach gelöst werden können, denn die Größe von OBDD's hängt von der Variablenordnung ab. Ein Beispiel für eine unterschiedliche Variablenordnung und deren Auswirkung zeigt Abbildung 3. Bei diesem Beispiel sind Entscheidungen mit 0 an einem Knoten durch eine grüne Kante und Entscheidungen mit 1 durch eine rote Kante dargestellt.

[Brace et al. 90] beschreiben, wie man mit Hilfe von einem if-then-else (ITE) Operator und einem Hashing-Mechanismus besonders effizient ein BDD-Paket implementieren kann. Der ITE-Operator definiert folgende Boolesche Funktion mit drei Eingabefunktionen f, g und h: Wenn f dann g, sonst h. Dies ist äquivalent zu:
$$ITE(f,g,h) = f \wedge g \vee \neg f \wedge h$$

Für einige Funktionen, wie z.B. einen Multiplizierer, gibt es unabhängig von der Variablenordnung keine effiziente Darstellungsform mit OBDD's. Dennoch ist es möglich, für viele der gängigen Booleschen Funktionen effiziente OBDD's zu finden.
Die im folgenden beschriebenen Algorithmen sind die Grundlagen für ein in Java geschriebenes OBDD-Paket nach [Bryant 86].

1.3 Aufbau der Studienarbeit

Diese Studienarbeit gliedert sich in fünf Teile. Der erste Teil ist die Einleitung. Im zweiten Teil, Kapitel 2, wird die Architektur des OBDD-Paketes beschrieben und mit Hilfe von OMT veranschaulicht.
Der dritte Teil, Kapitel 3, verdeutlicht die Benutzung des OBDD-Paketes durch mehrere Fallbeispiele, unter anderem durch einen 2-Bit-Addierer.
Im vierten Teil, Kapitel 4 und 5, erfolgt die Erläuterung der beiden wichtigsten Klassen des Paketes und die Beschreibung ihrer Methoden.
Der letzte Abschnitt, Kapitel 6, soll einen Ausblick auf mögliche Verbesserungen mit dem Ziel der Effizienzsteigerung geben.
In Kapitel 7 befindet sich die Literaturliste.

2 Architektur

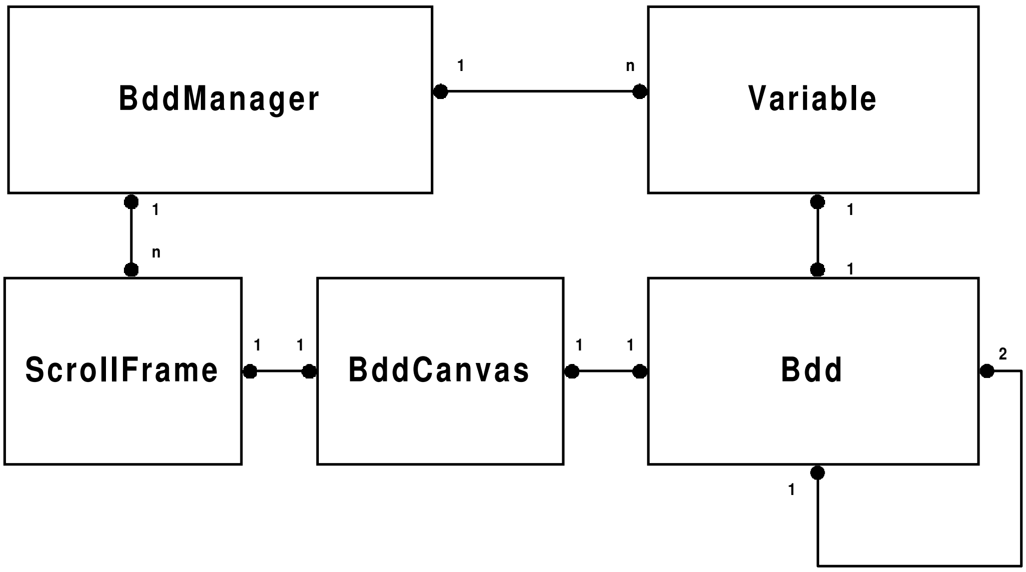


Abbildung 4: OMT - Diagramm des OBDD-Paketes.

2.1 Überblick

Im folgenden werden kurz die wichtigsten Klassen des OBDD-Java-Pakets vorgestellt und ihre Klassen mittels der Object Modeling Technique (OMT) [Rumbaugh et al. 94] veranschaulicht.

2.2 Diskussion der Klassenauswahl

Die zwei wichtigsten Klassen des OBDD-Paketes sind zum einen der BDD-Manager und die eigentliche BDD-Klasse. Da bei OBDD's die Variablenordnung von entscheidender Bedeutung ist, benötigt man eine Klasse, die diese kapselt und dem Benutzer eine wohl-definierte Schnittstelle bietet, um eine Variablenordnung zu erstellen und auf die Variablen gemäß dieser Ordnung zuzugreifen. Diese Benutzerschnittstelle bietet der BDD-Manager, der eine Liste aller Variablen enthält und durch diese ihre Ordnung beschreibt (siehe Abbildung 4 und 5).
Die zweite Klasse beschreibt die Struktur eines Knoten eines BDD's. Nach Definition 1.5 benötigt dieser folgende Attribute: einen Index (index), zwei Knoten (leftchild und rightchild) die seine Söhne beschreiben und einen Wert (value) der den Blättern zugeordnet ist (siehe Abbildung 5). Die restlichen Attribute werden für die weiteren Algorithmen benötigt und in Kapitel 5 genauer erläutert. Desgleichen werden dort ihre Methoden und ihre Funktionsweise beschrieben.

BddManager
maxVar : int Variablenliste : Vektor lookahead : String bddAusdruck : String tokens : String
not and or xor depth nodes bddEquals bddEval addVariableLast getVariableAt getVariableNameAt getVariablewithName createBddByFormel showFormel showBdd variablenaufbau existVariable parse expression termXor termAnd termNot factor match lexikalischeAnalyse

Bdd
leftchild : Bdd rightchild : Bdd index : int value : int terminalIndex : int id : int xpoint : int ypoint : int isxyset : boolean
not and or xor depth nodes equals eval makeSubgraph reduce apply applyStep moveNiceBdd showNiceBdd

BddCanvas
bdd : Bdd bddFormel : String knotenDurchmesser : int knotenAbstand : int obererAbstand : int breite : int hoehe : int tiefe : int masstab : int moveRow : int xdistance : int
verdoppel verschiebe halbiere getMoveRow mouseDown mouseDrag mouseUp paint layout

Variable
var : Bdd varname : String
getvar setvar getvarName setvarName

Abbildung 5: Klassendiagramme des OBDD-Paketes.

2.3 Implementierung

Bei der Implementierung wurde sich im Rahmen der Studienarbeit an die grundlegenden Strukturen und Methoden von [Bryant 86] gehalten. Zusätzlich wurden Methoden („showNiceBdd“ und „moveNiceBdd“) zur Visualisierung und Manipulation („depth“, „nodes“, „equals“, „eval“) von BDD’s implementiert. In der Methode „applyStep“ wird die Klasse „Set“ benutzt, in ihr werden Informationen abgespeicher um nicht mehrmals ein Paar von Subgraphen zu evaluieren. Bevor ein Paar von Subgraphen evaluiert wird muß nur in der Klasse Set nachgeschaut werden, ob das Resultat schon berechnet wurde. Zur Sortierung wird der Algorithmus „Quick Sort“ [Wirth 86] benutzt. Zu weiteren Details siehe Kapitel 5.

Abbildung 5 zeigt die OMT-Klassendiagramme für die wesentlichen Klassen des OBDD-Paketes. Dies sind im einzelnen:

`BddManager`:

Er ist die eigentliche Benutzerschnittstelle zur Benutzung des OBDD-Pakets. Er hält Informationen wie die Variablenordnung bereit und stellt alle möglichen Funktionen zur Manipulation von OBDD's zur Verfügung. Zusätzlich integriert er einen Parser der es ermöglicht String-Ausdrücke zu verarbeiten und anhand einer vorgegebenen Grammatik diese zu OBDD's zu verarbeiten (siehe Kapitel 4.7). Diese Klasse wird genauer in Kapitel 4 vorgestellt.

`Bdd`:

Ein Knoten eines OBDD's konstruiert nach [Bryant 86]. Diese Klasse implementiert die Algorithmen zur Manipulation und Visualisierung von OBDD's, genauere Erläuterungen finden sich in Kapitel 5.

Set:

Eine Klasse die Paare von Schlüsseln (key) und BDD's vorhält. Sie wird in der Methode „apply“ verwendet um ein „Caching“ zu ermöglichen. Zur Sortierung bietet diese Klasse die Methode „quickSortbyKey“.

ElementofSet:

Diese Klasse definiert die Elemente der Klasse Set. Sie besteht aus einem Paar „key“ und Bdd.

Folgende Klassen werden zur Darstellung am Bildschirm benötigt:

`BddCanvas`:

Dieser Canvas dient zur Visualisierung von BDD's, er stellt verschiedene Methoden zur Verfügung um visualisierte OBDD's in ihrer Darstellungsweise zu manipulieren.

`ScrollFrame`:

Ein Scroll Frame für das JDK1.0.2. Die Klassen BddCanvas und ScrollFrame werden in Kapitel 4.8 erläutert.

3 Fallbeispiele

3.1 Überblick

Dieses Kapitel soll anhand zweier Probleme die Benutzung des OBDD-Paketes beschreiben. Im ersten Beispiel werden zwei Boolesche Ausdrücke miteinander verglichen und ausgewertet. Im zweiten Beispiel wird ein Volladdierer mit OBDD's aufgebaut und im dritten Abschnitt zu einem 2-Bit Addierer erweitert.

3.2 Problembeschreibung: Vergleich von Booleschen Ausdrücken

Gegeben seien die folgenden zwei Booleschen Ausdrücke:

$$\neg(a \wedge b \wedge \neg(c \wedge d) \vee \neg(a \wedge b) \wedge c \wedge d) \vee \neg(a \wedge b)$$

$$(a \wedge b \wedge c \wedge d) \vee \neg a \vee \neg b$$

Der zweite Ausdruck soll eine disjunktive Normalform des ersten sein. Dieses wollen wir mit Hilfe des OBDD-Paketes überprüfen und anschließend zur Bestätigung die beiden BDD's, die diese Ausdrücke repräsentieren, auf dem Bildschirm ausgeben. Zusätzlich soll folgende Variablenbelegung ausgewertet werden:

$a = 0$, $b = 0$, $c = 1$, $d = 0$

3.3 Problemlösung: Vergleich von Booleschen Ausdrücken

Zunächst muß ein BDD-Manager erzeugt werden, der in diesem Fall mindestens vier Variablen enthält (siehe Abschnitt 4.2):

```
BddMangaer bddm = new BddManager(4);
```

Die Ausdrücke können nun in eine dem Parser verständliche Syntax gebracht werden (siehe Abschnitt 4.7):

```
String bddString1 = "!(a*b*!(c*d)+!(a*b)*c*d)+!(a*b)";
String bddString2 = "a*b*c*d+!a+!b";
```

Nun können die beiden BDD's mit der Methode „createBddByFormel“ erzeugt werden:

```
Bdd bdd1 = bddm.createBddByFormel(bddString1);
Bdd bdd2 = bddm.createBddByFormel(bddString2);
```

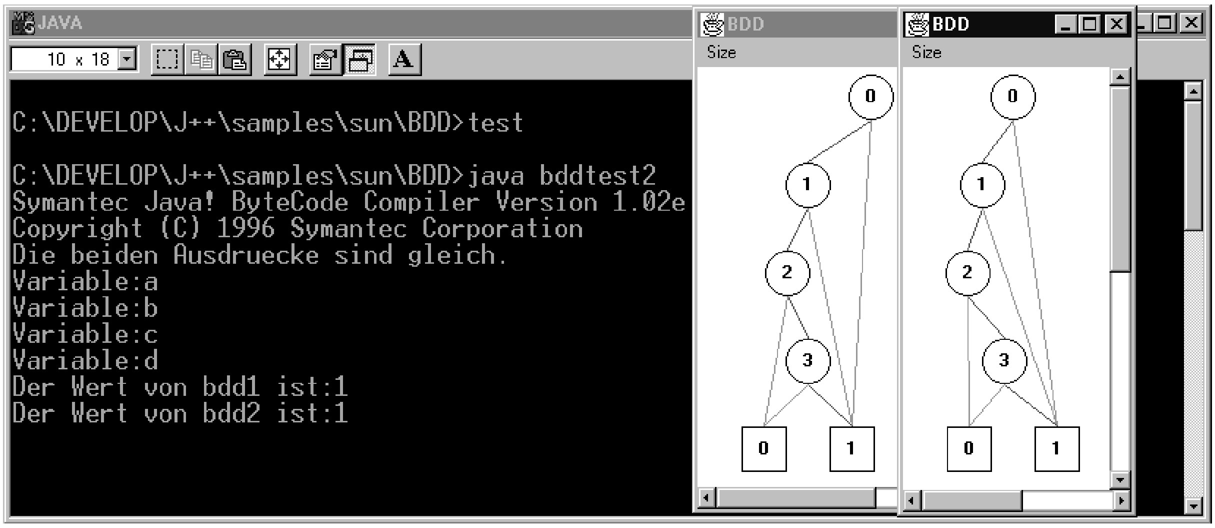


Abbildung 6: Bildschirmausgabe für das erste Fallbeispiel.

Um zu überprüfen, ob die beiden erzeugten BDD's gleich sind, benutzen wir die Methode „bddEquals" (siehe Abschnitt 4.5) und geben das Ergebnis auf dem Bildschirm aus.

```
If (bddm.bddEquals(bdd1, bdd2))
{
        System.out.println("Die beiden Ausdruecke sind gleich.");
}
else
{
        System.out.println("Die beiden Ausdruecke sind nicht gleich.");
}
```

Anschließend sollen die beiden BDD's noch auf dem Bildschirm dargestellt werden. Dazu benutzen wir die Methode „showBdd":

```
bddm.showBdd(bdd1);
bddm.showBdd(bdd2);
```

Um die oben gegebene Variablenbelegung auszuwerten, wird die Methode „bddEval" benutzt und ein Array mit einer der Variablenbelegung entsprechenden Ordnung übergeben. In diesem Fall wird die Variablenordnung und somit die Belegung des übergebenen Arrays an „bddEval" mittels der for-Schleife über alle Variablen ermittelt. Dieser Aufwand ist hier nötig, da die Variablenordnung durch den Parser bestimmt wurde:

```
int values[] = new int[4];
for (int i=0;i<4;i++)
{
        System.out.println("Variable:" + bddm.getVariableNameAt(i));
```

c_0	a_0	b_0	Summe (S_1)	Übertrag (c_1)
0	**0**	**0**	**0**	**0**
0	**0**	**1**	**1**	**0**
0	**1**	**0**	**1**	**0**
0	**1**	**1**	**0**	**1**
1	**0**	**0**	**1**	**0**
1	**0**	**1**	**0**	**1**
1	**1**	**0**	**0**	**1**
1	**1**	**1**	**1**	**1**

Tabelle 2: Wertetabelle für einen Volladdierer.

```
        if (bddm.getVariableNameAt(i)=="a")
              values[i]=0;
        else if (bddm.getVariableNameAt(i)=="b")
              values[i]=1;
        else if (bddm.getVariableNameAt(i)=="c")
              values[i]=1;
        else if (bddm.getVariableNameAt(i)=="d")
              values[i]=0;
}
int wert1 = bddm.bddEval(bdd1,values);
int wert2 = bddm.bddEval(bdd2,values);
System.out.println("Der Wert von bdd1 ist:" + wert1);
System.out.println("Der Wert von bdd2 ist:" + wert2);
```

Das Resultat dieser Zeilen wird in Abbildung 6 gezeigt.

3.4 Problembeschreibung: Volladdierer

Im zweiten Abschnitt soll ein Volladdierer gebaut und mit Hilfe von BDD's visualisiert werden. Zum besseren Verständnis ist in Tabelle 2 die Wertetabelle eines Volladdierers zu sehen und im folgenden die Booleschen Funktionen für den Übertrag und die Summe [Schiffmann et al. 92].

$$S_1 = c_0 \oplus a_0 \oplus b_0$$
$$c_1 = (c_0 \wedge a_0) \vee (c_0 \wedge b_0) \vee (a_0 \wedge b_0)$$

3.5 Problemlösung: Volladdierer

Anhand der Wertetabelle ist zu erkennen, daß 3 Variablen benötigt werden. Also wird zuerst ein BDD - Manager mit drei Variablen erzeugt.

```
BddManager bddm = new BddManager(3);
for (int i=0;i<3;i++) {
        bddm.addVariableLast();
}
```

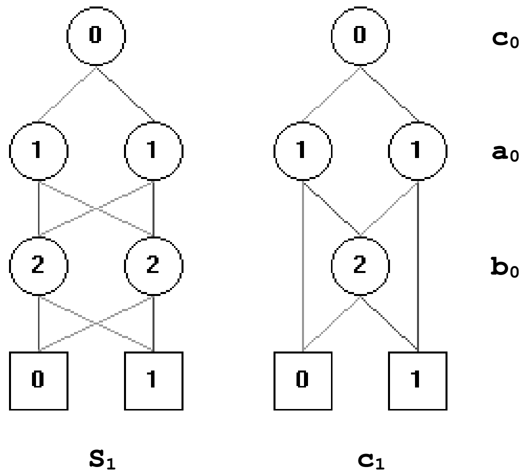


Abbildung 7: Zwei BDD's, die einen Volladdierer darstellen: einer die Summe (links), der andere den Übertrag (rechts).

Nun kann zunächst der Summen - BDD erzeugt werden:

```
x1 = bddm.getVariableAt(1);
x2 = bddm.getVariableAt(2);
ueber = bddm.getVariableAt(0);

Bdd tmp = bddm.xor(ueber, x1);
Bdd summe = bddm.xor(tmp, x2);
```

Als nächstes wird der Übertrag - BDD erzeugt:

```
Bdd and1 = bddm.and(ueber, x1);
Bdd and2 = bddm.and(ueber, x2);
Bdd and3 = bddm.and(x1, x2);
tmp = bddm.or(and1,and2);
Bdd uebertrag = bddm.or(tmp,and3);
```

Abschließend werden die beiden BDD's auf dem Bildschirm ausgegeben:

```
bddm.showBdd(summe);
bddm.showBdd(uebertrag);
```

Das Resultat sieht man in Abbildung 7.

3.6 Problembeschreibung: Erweiterung zum 2-Bit Addierer

Im letzten Abschnitt soll der Volladdierer zu einem 2-Bit Addierer erweitert und mit Hilfe von BDD's visualisiert werden.

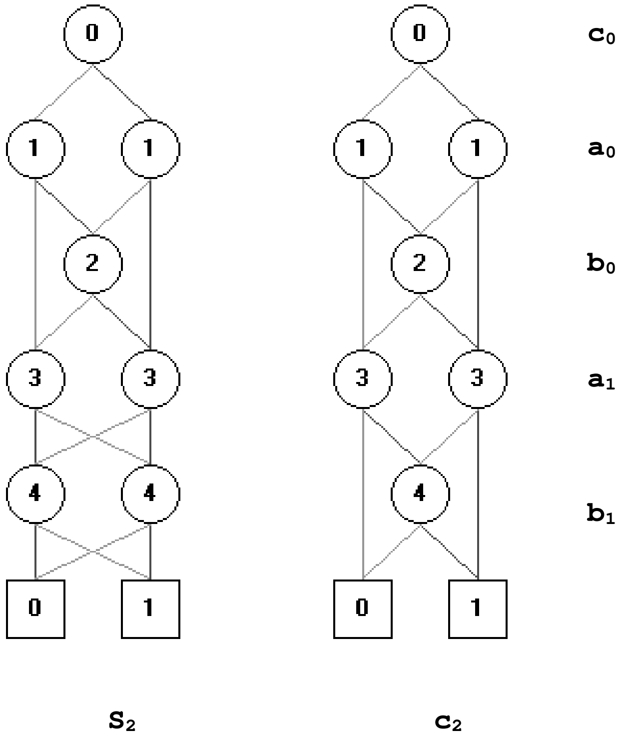


Abbildung 8: Das zweite Summenglied (links) und das zweite Übertragglied (rechts) eines 2 - Bit Addierers

3.7 Problemlösung: Erweiterung zum 2 - Bit Addierer

Als erstes die Definition eines 2-Bit Addierers:

$$S_1 = c_0 \oplus a_0 \oplus b_0$$
$$c_1 = (c_0 \wedge a_0) \vee (c_0 \wedge b_0) \vee (a_0 \wedge b_0)$$

$$S_2 = c_1 \oplus a_1 \oplus b_1$$
$$c_2 = (c_1 \wedge a_1) \vee (c_1 \wedge b_1) \vee (a_1 \wedge b_1)$$

Im folgenden die von uns benutzte Variablenordnung:

1. $c_0=x_0$
2. $a_0=x_1$
3. $b_0=x_2$
4. $a_1=x_3$
5. $b_1=x_4$

Da wir bei dem Problem des 2-Bit Addierers 2 Variablen mehr benötigen als beim Volladdierer, muß ein BDD-Manager mit 5 Variablen erzeugt werden.

```
BddManager bddm = new BddManager(3);
for (int i=0;i<5;i++) {
       bddm.addVariableLast();
```

```
}
```

Anschließend werden wie in Abschnitt 4.5 zwei BDD's erzeugt, die einen Volladdierer repräsentieren.

```
Bdd x1,x2,ueber;
x1 = bddm.getVariableAt(1);
x2 = bddm.getVariableAt(2);
ueber = bddm.getVariableAt(0);

Bdd tmp = bddm.xor(ueber, x1);
Bdd summe = bddm.xor(tmp, x2);

Bdd and1 = bddm.and(ueber, x1);
Bdd and2 = bddm.and(ueber, x2);
Bdd and3 = bddm.and(x1, x2);
tmp = bddm.or(and1,and2);
Bdd uebertrag = bddm.or(tmp,and3);
```

Als nächstes müssen der zweite Übertrag und die zweite Summe anhand desjenigen BDD's erzeugt werden, der den ersten Übertrag repräsentiert.

```
x1 = bddm.getVariableAt(1);
x2 = bddm.getVariableAt(2);
ueber = uebertrag;

tmp = bddm.xor(ueber, x1);
Bdd summe2 = bddm.xor(tmp, x2);

and1 = bddm.and(ueber, x1);
and2 = bddm.and(ueber, x2);
and3 = bddm.and(x1, x2);
tmp = bddm.or(and1,and2);
Bdd uebertrag2 = bddm.or(tmp,and3);

bddm.showBdd(summe2);
bddm.showBdd(uebertrag2);
```

Das Resultat sieht man in Abbildung 8.
In Abbildung 9 sieht man ebenfalls einen 2 - Bit Addierer, aber mit einer anderen Variablenordnung. Es folgt die Variablenordnung für Abbildung 9:

1. $c_0=x_0$
2. $a_0=x_1$
3. $b_0=x_4$
4. $a_1=x_2$
5. $b_1=x_3$

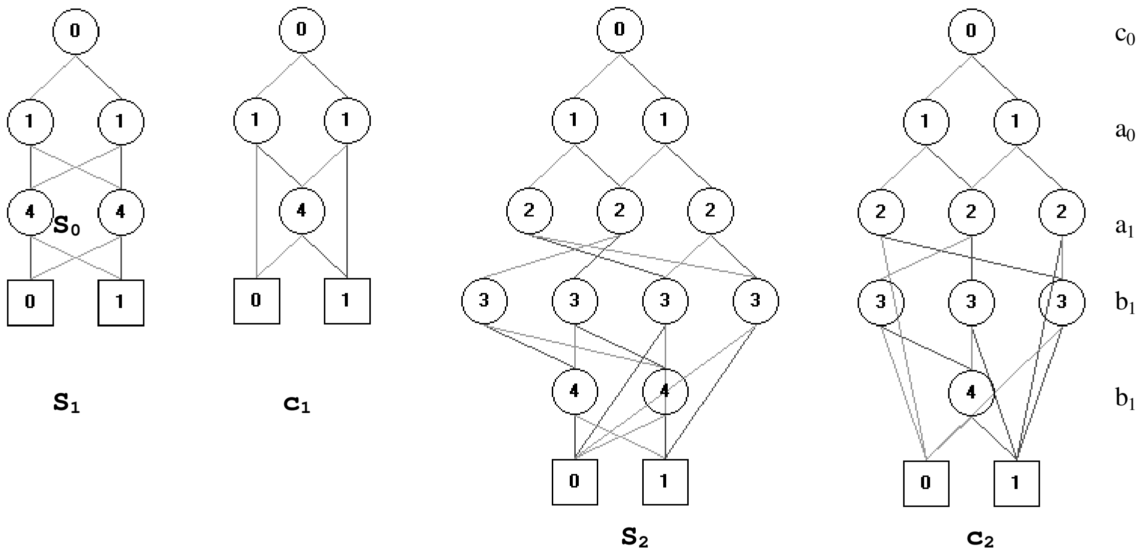


Abbildung 9: Ein 2 - Bit Addierer mit ungünstiger Variablenordnung. Schon im ersten Summen- und Übetragsglied wird auf die vierte Variable zugegriffen. (Variablenordnung ist rechts aufgeführt)

Zum Abschluß noch eine allgemeine Methode, die einen n-Bit-Addierer aufbaut und darstellt.

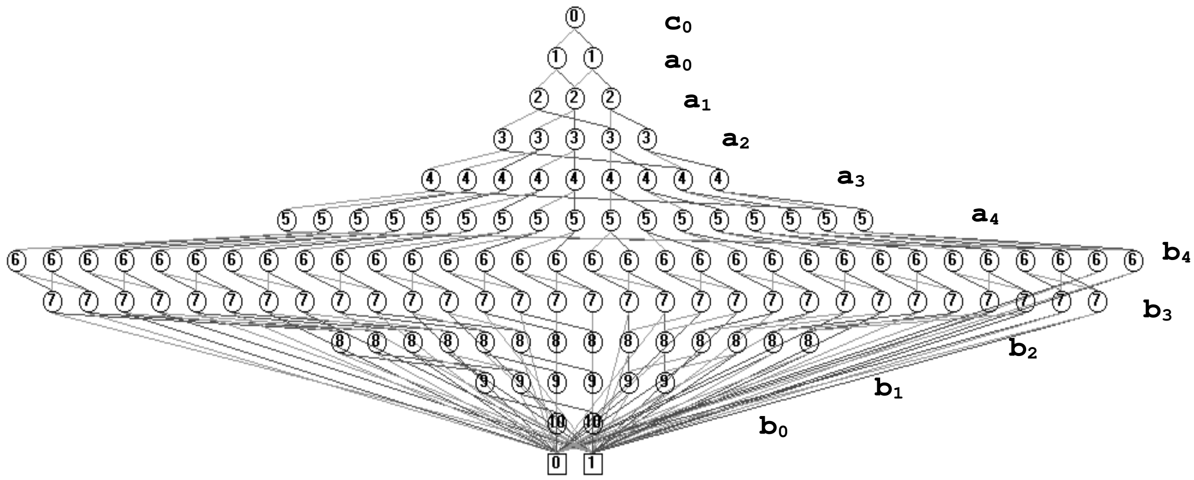
```
void n_Bit_Adder(int n) {
 BddManager bddm = new BddManager(2*n+1);
 for (int i=0;i<(2*n+1);i++) {
       bddm.addVariableLast();
 }
 Bdd summe[] = new Bdd[n];
 Bdd uebertrag[]= new Bdd[n];
 Bdd x1,x2,ueber,tmp,and1,and2,and3;

 for (int i=0;i<n;i++) {
       x1 = bddm.getVariableAt(2*i+1);
       x2 = bddm.getVariableAt(2*i+2);
       if (i==0)
             ueber = bddm.getVariableAt(0);
       else
             ueber = uebertrag[i-1];

       //Aufbau des Summen-BDD's
       tmp = bddm.xor(ueber, x1);
       summe[i] = bddm.xor(tmp, x2);

       //Aufbau des Übertrag-BDD's
       and1 = bddm.and(ueber, x1);
       and2 = bddm.and(ueber, x2);
       and3 = bddm.and(x1, x2);
       tmp = bddm.or(and1,and2);
       uebertrag[i] = bddm.or(tmp,and3);

       //Ausgabe vom Summen-BDD und Übertrag-BDD
       bddm.showBdd(summe[i]);
       bddm.showBdd(uebertrag[i]);
 }
}
```



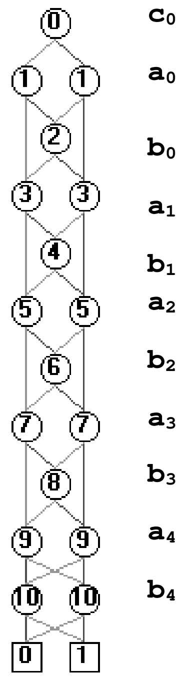


Abbildung 10: Beispiel für die Auswirkungen der Variablenordnung bei einem Summenglied eines 5-Bit Addierers

4 Der BDD-Manager

4.1 Überblick

Der BDD-Manager dient zur Verwaltung der Variablenliste und ihrer Ordnung.
Im vorliegenden Kapitel werden die verschiedenen Methoden zur Bearbeitung des BDD-Managers vorgestellt und anschließend ihre Benutzung mit einem Beispiel beschrieben. Um einen BDD-Manager zu erzeugen, ruft man seinen Konstruktor auf und übergibt ihm die maximale Anzahl der zu verwaltenden Variablen.
Im folgenden ein Beispiel, welches einen BDD-Manager erzeugt, der 30 Variablen verwalten kann:

```
//Erzeugung eines BDD_Manager mit maximal 30 Variablen
bddm = new Bddmanager(30);
```

4.2 Erzeugung von Variablen

In diesem Abschnitt werden die Methoden vorgestellt, welche neue Variablen der Variablenliste hinzufügen. Anschließend werden diese Methoden mittels eines Beispiels veranschaulicht.

public void addVariableLast(String variablenname):
Fügt eine Variable mit übergebenem Namen am Ende der VariablenListe zu.

public void addVariableLast():
Fügt eine Variable am Ende der VariablenListe zu.

Ein Beispiel, in welchem ein BDD-Manager und 15 Variablen mit Namen x0 bis x14 sowie 15 weitere Variablen ohne Namen erzeugt werden:

```
//Erzeugung eines BDD_Manager mit maximal 30 Variablen
bddm = new Bddmanager(30);
//Schleife zum initialisieren:
for (int i=0;
i<15;i++)
  {
        bddm.addVariableLast("x["+i+"]");
  }
for (int i=0;
i<15;i++)
  {
        bddm.addVariableLast();
  }
```

4.3 Abfragen der Variablenliste

Im folgenden werden die Methoden, welche zum Zugriff auf die Variablenliste benötigt werden, kurz beschrieben und ihre Funktionsweise durch ein Beispiel verdeutlicht.

public Bdd getVariableAt(int i):
„getVariableAt“ holt den BDD mit Index i aus der Variablenliste.

public Bdd getVariableWithName(String variablenname):
Mit ihr wird ein BDD mit übergebenem Variablenname aus der Variablenliste geholt.

Um die erste Variable aus dem BDD-Manager in unserem Beispiel zu erhalten,
gibt es also zwei Möglichkeiten:

```
bddm.getVariableAt(0);
```

oder

```
bddm.getVariableWithName("x0");
```

public String getVariableNameAt(int i):
Diese Methode holt den Variablennamen von der Variablen mit dem Index i in der Variablenliste.

Der Aufruf

```
String name=bddm.getVariableNameAt(0);
```

würde in unserem Beispiel den String „x0“ an die Variable name zurückgeben.

4.4 Manipulation von BDD’s

Die folgenden Methoden dienen zur Manipulation von BDD’s. Sie repräsentieren Boolesche Funktionen. Am Ende dieses Abschnittes folgt ein kleines Beispiel für die Anwendung dieser Methoden.

public Bdd not (Bdd bdd):
Sie gibt die Negation eines BDD`s zurück.

public Bdd and (Bdd bdd1,Bdd bdd2):
Sie gibt die UND-Verknüpfung der BDD’s bdd1 und bdd2 zurück.

public Bdd or (Bdd bdd1,Bdd bdd2):
Sie gibt die ODER-Verknüpfung zweier BDD`s zurück.

```
C:\DEVELOP\J++\samples\sun\BDD>java bddtest2
Symantec Java! ByteCode Compiler Version 1.02e
Copyright (C) 1996 Symantec Corporation

Erzeuge den ersten Bdd aus Abbildung 2
Er hat 7Ebenen mit,  8Knoten,

Erzeuge den zweiten Bdd aus Abbildung 2
Er hat 7Ebenen mit,  16Knoten,

C:\DEVELOP\J++\samples\sun\BDD>
```

Abbildung 11 zeigt die Ausgabe des Programms aus Abschnit 4-5.

public Bdd xor (Bdd bdd1,Bdd bdd2):
Sie gibt die XOR-Verknüpfung zweier BDD`s zurück.

Das folgende Beispiel bezieht sich auf den oben definierten BDD-Manager bddm und die dazu definierten 30 Variablen:

```
//Erzeugung eines BDD,der die Und-Veknüpfung zweier Variablen
//repräsentiert
Bdd bdd_und =
 bddm.getVariableWithName("x1").and(bddm.getVariableWithName("x2"));

//Erzeugung eines BDD,der die Oder-Veknüpfung zweier Variablen
//repräsentiert
Bdd bdd_oder =
 bddm.getVariableWithName("x1").or(bddm.getVariableWithName("x2"));

//Erzeugung eines BDD,der die Und-Veknüpfung zweier Variablen
//repräsentiert
Bdd bdd3_xor =
 bddm.getVariableWithName("x1").xor(bddm.getVariableWithName("x2"));

//Erzeugung eines BDD,der die Nicht-Veknüpfung der Variable bdd_und //
//repräsentiert
Bdd bdd_not = bddm.not(bdd_und);
```

4.5 Auswertung von BDD’s

Die folgenden Methoden dienen zur Auswertung von BDD’s. Sie werden in diesem Abschnitt kurz vorgestellt und am Ende durch ein Beispiel ihre Anwendung beschrieben.

public int depth (Bdd bdd):
Sie gibt die Anzahl aller Ebenen (inklusive der Blätter) eines BDD’s zurück.

public int nodes (Bdd bdd):
Diese Methode gibt die Knotenanzahl eines BDD`s zurück.

public boolean bddEquals (Bdd bdd1,Bdd bdd2):
Sie vergleicht zwei BDD`s: sind sie gleich, wird „true“ zurückgegeben.

public int bddEval (Bdd bdd,int[] values):
Diese Methode wertet einen BDD für eine bestimmte Variablenbelegung aus.

Das folgende Beispiel erzeugt die beiden BDD’s aus Abbildung 3 und ruft die verschiedenen Methoden zur Auswertung auf. In Abbildung 11 ist die Ausgabe beim Durchlauf des Programms zu sehen.

```
public static void main (String argv[]){

//Variablenanzahl des BDD-Manager
nvars=10;
//Erzeugung des BDD-Managers mit nvars Variablen.
bddm = new BddManager(nvars);
//Füllen des Variablenvektors
for (int i=0;
      i<nvars;i++)
      {
            bddm.addVariableLast("x"+i);
      }

//Aufbau des ersten BDD’s
System.out.println("\nErzeuge den ersten BDD aus Abbildung 2");

Bdd bddAbb2 = bddm.and(bddm.getVariableAt(0), bddm.getVariableAt(1));

bddAbb2 = bddm.or(bddAbb2,
            bddm.and(bddm.getVariableAt(2),bddm.getVariableAt(3)));

bddAbb2 = bddm.or(bddAbb2,
            bddm.and(bddm.getVariableAt(4),bddm.getVariableAt(5)));

// Ausgabe der Tiefe und der Knotenanzahl des ersten BDD’s
System.out.println("Er hat "
                        + bddm.depth(bddAbb2) + "Ebenen mit,  "
                        + bddm.nodes(bddAbb2) + "Knoten,  ");

//Und nun den zweiten BDD durch Zugriff über den Variablennamen aufbauen
System.out.println("\nErzeuge den zweiten BDD aus Abbildung 2");
Bdd tmp =   bddm.and(bddm.getVariableWithName("x0"),
                  bddm.getVariableWithName("x3"));

tmp =       bddm.or(tmp,bddm.and(bddm.getVariableWithName("x1"),
                  bddm.getVariableWithName("x4")));

tmp =       bddm.or(tmp,bddm.and(bddm.getVariableWithName("x2"),
                  bddm.getVariableWithName("x5")));

// Ausgabe der Tiefe und der Knotenanzahl des zweiten BDD’s
System.out.println("Er hat "
                        + bddm.depth(tmp) + "Ebenen mit,  "
                        + bddm.nodes(tmp) + "Knoten,  ");
}
```

4.6 Darstellung von BDD's

Die Methoden in diesem Abschnitt werden benötigt, um BDD's auf dem Bildschirm darzustellen.

public void showBdd(Bdd bdd):
Diese Methode zeigt einen übergebenen BDD in einem eigenem Frame auf dem Bildschirm an.

Die BDD's aus Abbildung 3 könnte man mit folgenden beiden Zeilen am Ende des eben beschriebenen Beispiels darstellen.

```
bddm.showBdd(bddAbb2);
bddm.showBdd(tmp);
```

public void showFormel(String formelString):
Diese Methode zeigt einen BDD an, der aus einer übergebenen Formel generiert wird. Sie benutzt einen Parser um Boolesche Ausdrücke auszuwerten.

Im folgenden Beispiel werden die beiden BDD's aus Abbildung 3 mit Hilfe der Methode „showFormel" erzeugt und dargestellt.

```
//Variablenanzahl des BDD-Manager
nvars=10;
//Erzeugung des BDD-Managers mit nvars Variablen.
bddm = new BddManager(nvars);
//Füllen des Variablenvektors
for (int i=0;
        i<nvars;i++)
        {
              bddm.addVariableLast("x"+i);
        }
System.out.println("\nErzeuge den ersten BDD aus Abbildung 2");
//Aufbau des ersten BDD's
bddm.showFormel("x0*x1+x2*x3+x4*x5");
System.out.println("\nErzeuge den zweiten BDD aus Abbildung 2");
//Und nun den zweiten BDD durch Zugriff über den Variablennamen
aufbauen
bddm.showFormel("x0*x3+x1*x4+x2*x5");
```

4.7 Parser

Dieser Abschnitt beschreibt die Funktionsweise und Grammatik des Parsers, der durch folgende Methoden implementiert wurde: „viablenaufbau", „existVariable", „parse", „expression", „termXor", „termAnd", „termNot", „factor", „match" und „lexikalischeAnalyse".

Der Parser hat die Aufgabe, übergebene Boolesche Ausdrücke in „Stringform" zu verarbeiten und auszuwerten. Es wird folgende Notation für Boolesche Verknüpfungen verwendet:

+	:	logisches oder
*	:	logisches und
!	:	logisches nicht
#	:	logisches xor
()	:	Klammerung

Dabei beruht der Parser auf folgender Grammatik:

$$expression \rightarrow expression + termXor$$
$$expression \rightarrow termXor$$

$$termXor \rightarrow termXor \# termAnd$$
$$termXor \rightarrow termAnd$$

$$termAnd \rightarrow termAnd * termNot$$
$$termAnd \rightarrow termNot$$

$$termNot \rightarrow !termNot$$
$$termNot \rightarrow factor$$

$$factor \rightarrow (expression)$$
$$factor \rightarrow value$$
$$factor \rightarrow variable$$

$$value \rightarrow 0$$
$$value \rightarrow 1$$

$$variable \rightarrow Variable$$

Zum näheren Verständnis von Grammatiken und Parsern siehe [Aho et al. 92].
In Tabelle3 stehen einige Beispiele für Strings, die durch diesen Parser akzeptiert werden, und ihre Syntax in der Booleschen Algebra.

Parser	Boolesche Algebra
a*b	$a \wedge b$
a+b	$a \vee b$
a#b	$a \oplus b$
!a	$\neg a$
(a*b)+(a*c)+(b*c)	$(a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$
(a+b)*(a+c)*(b+c)	$(a \vee b) \wedge (a \vee c) \wedge (b \vee c)$

Tabelle 3:Syntaxvergleich

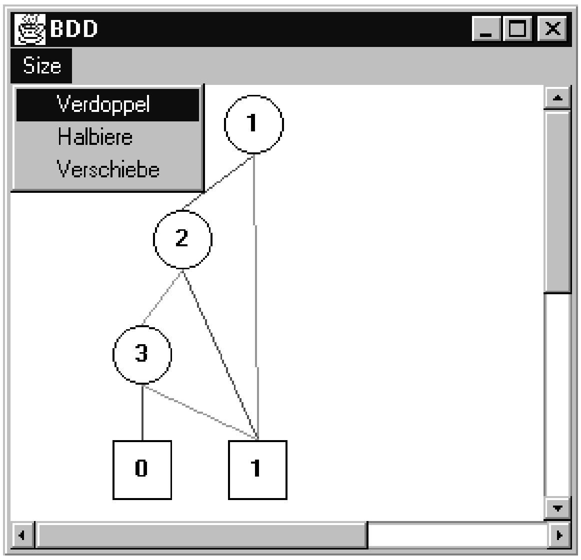


Abbildung 12 Grafische Benutzerschnitstelle

4.8 Grafische Benutzerschnittstelle

Wenn ein BDD auf dem Bildschirm visualisiert wird, erhält er ein eigenes Fenster, aus dem zusätzliche Methoden zur verbesserten Darstellungsform aufgerufen werden können.
In diesem Abschnitt werden diese Methoden vorgestellt und ihre Interaktionen mit der Klasse BddCanvas beschrieben, welche zur eigentliche Darstellung des BDD benutzt wird.

4.8.1 ScrollFrame und BddCanvas

Diese beiden Klassen werden benötigt, um die BDD-Objekte auf dem Bildschirm zu visualisieren. In diesem Abschnitt werden die beiden Klassen und ihre Strukturen beschrieben.

Die Klasse Scroll Frame ist ein Frame mit zwei Scrollbars, welcher einen BddCanvas in seiner Mitte anzeigt. Diese Implementierung war aufgrund des JDK1.0.2 notwendig, da dort keine Scrollframe-Klasse vorhanden war. Dieser Frame enthält auch ein Menü, welches dem Benutzer verschiedene Aktionen wie „Verdoppeln“, „Halbieren“ und „Verschieben“ zur verbesserten Visualisierung von BDD’s bereitstellt. Die Menüpunkte sind mit den jeweiligen Methoden im BddCanvas verknüpft.

Die Klasse BddCanvas hält folgende Parameter bereit, die zur Visualisierung von BDD’s benötigt werden:

bdd:

Diese Variable enthält den anzuzeigenden BDD (siehe Kapitel 5).

bddFormel:
: Sofern der BDD durch einen Formelstring generiert wurde, wird dieser String in bddFormel bereitgehalten (siehe 3.6).

tiefe:
: Diese Variable enthält die Tiefe des BDD's, der angezeigt werden soll, damit diese nicht jedesmal neu berechnet werden muß (siehe 4.11).

knotenDurchmesser:
: Diese Variable gibt den augenblicklichen Knotendurchmesser des BDD's an.

knotenAbstand:
: Diese Variable gibt den vertikalen Abstand zwischen zwei Knotenebenen an.

obererAbstand:
: Dieser Abstand ist der vertikale Abstand vom oberen Rand des BDD's.

breite:
: Die aktuelle Breite des BDD's entspricht der Breite des BDD's in Pixeln gemessen.

hoehe:
: Die aktuelle Höhe des BDD's, ebenfalls in Pixeln gemessen.

massstab:
: Dieser Wert gibt den Maßstab des anzuzeigenden BDD's an. Er wird mit 2 initialisiert.

moveRow und xdistance:
: Diese beiden Werte werden beim Verschieben benötigt. Sie geben zum einen an, welche Knotenebene verschoben wird, und zum anderen um welche Distanz.

originX, originY, xCursor, dx ,dy:
: Verschiedene Parameter zur Berechnung der Distanz beim Verschieben.

4.8.2 Menüpunkt „Verdoppeln“

Dieser Menüpunkt stellt den BDD in der doppelten Größe dar. Vorherige Veränderungen in der Darstellungsform, wie z.B. durch „verschieben“, gehen dabei verloren.

Es wird die Methode „Verdoppele“ in der Klasse BddCanvas aufgerufen, welche die Knotenbreite und den Knotenabstand verdoppelt und den Maßstab um eins erhöht. Anschließend wird der BDD mit den neuen Werten durch die Methode „showNiceBdd“ neu gezeichnet (siehe 4.14).

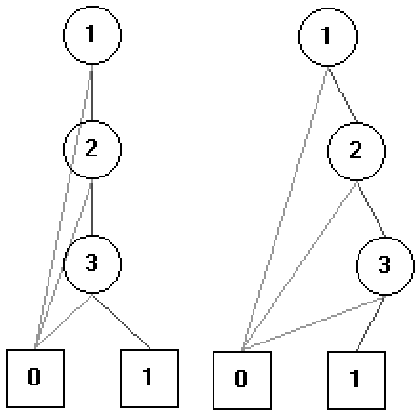


Abbildung 13 Beispiel für die Methode „Verschiebe“ ($x_1 \wedge x_2 \wedge x_3$).

4.8.3 Menüpunkt „Halbieren“

Dieser Menüpunkt arbeitet auf die gleiche Weise wie der Menüpunkt „Verdoppele“, nur halbiert er die Größe des darzustellenden BDD's.
Es wird die Methode „Halbieren“ in der Klasse BddCanvas aufgerufen, welche analog zu der Methode „Verdoppeln“ arbeitet.

4.8.4 Menüpunkt „Verschieben“

Der Menüpunkt „Verschieben“ gibt dem Benutzer die Möglichkeit, eine Ebene von Knoten zu verschieben und somit eine optimalere Darstellungsform des BDD's zu erreichen.
Es wird die Methode „Verschieben“ in der Klasse BddCanvas aufgerufen, welche alle Ebenen nach rechts verschiebt. Siehe Abbildung 13 für ein Beispiel.
Die eigentliche Verschiebung wird in der BDD-Klasse durchgeführt. Für jede Knotenebene wird die Methode „moveNiceBdd“ aufgerufen, und die Ebenen werden um 15 Pixel pro Tiefe nach rechts verschoben.

4.8.5 Verschieben mit der Maus

Neben der Methode „Verschieben“ läßt sich eine Knotenebene mit Hilfe der Maus verschieben. Hierzu wird die Methode „showNiceBdd“ verwendet.

5 Die Klasse BDD

5.1 Überblick

Der BDD-Manager verwaltet verschiedene BDD-Objekte, deren Struktur in der Klasse BDD beschrieben wird. In diesem Kapitel werden die Struktur und anschließend die Implementation der einzelnen Methoden zur Manipulation von BDD's beschrieben.

5.2 Struktur

Der BDD ist in diesem Paket als Objektstruktur implementiert, die verschiedene Eigenschaften besitzt.
In diesem Abschnitt werden die einzelnen Variablen, welche die Struktur des BDD-Objektes beschreiben, vorgestellt und ihr Zweck beschrieben.

Ein BDD enthält folgende Member-Variablen:

rightchild:
Die Variable rightchild gibt den BDD an, der am rechten Knoten des BDD's sitzt.

leftchild:
Die Variable leftchild gibt den BDD an, der am linken Knoten des BDD's sitzt.

value:
Der Wert der jeweiligen Variablen ist entweder 0 oder 1 für die Endknoen oder 3.
3 bedeutet: der Wert ist nicht gesetzt.

index:
Der Index beschreibt die Ordnung der jeweiligen Variablen, die durch den BDD-Knoten repräsentiert wird.

id:
Wird von den verschiedenen Algorithmen benötigt.

terminalindex:
Beschreibt den Wert des Index, den die Blätter eines BDD's besitzen.
Die Variable wird von den verschiedenen Algorithmen benötigt.

xPoint:
x-Parameter zur Darstellung eines Knoten eines BDD's auf dem Bildschirm.

yPoint:
y-Parameter zur Darstellung eines Knotens eines BDD's auf dem Bildschirm.

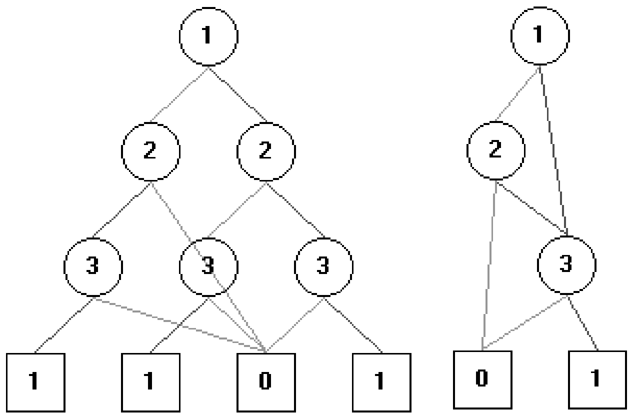


Abbildung 14 Beispiel für die Methode "reduce" bei der Funktion $(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$

isxySet:
Wenn sich an den Parametern zur Darstellung etwas geändert hat, so ist gegebenenfalls eine Neuberechnung der x- und y-Koordinaten erforderlich.

5.3 MakeSubgraph

Diese Methode macht aus der Baumstruktur des BDD's einen Vektor (Marker), so daß die einzelnen Elemente direkt angesprochen werden können. Der Algorithmus traversiert die Graphenstruktur des BDD durch wiederholten rekursiven Aufruf der Methode „makeSubgraph" und trägt dabei die einzelnen Knoten in einen übergebenen Vektor ein.

5.4 Reduce

Die Methode „reduce" wandelt den Graphen in einen reduzierten Graphen um, der dieselbe Funktion beschreibt. Dieser Algorithmus vergibt an jeden Knoten nach Beendigung eine eindeutige ID. [Bryant 86]
Der Algorithmus benutzt die Bottom-Up Methode, d.h. er arbeitet von den Blättern bis zum Wurzel-Knoten mit folgenden Methoden:

1) Zwei Blätter haben die gleiche ID genau dann, wenn sie den gleichen Wert (value) haben.

2) Es werde angenommen, daß alle Knoten mit einem Index größer i schon mit einer ID versehen seien. Wenn wir nun an einen Knoten v mit Index i eine ID vergeben wollen, so darf er nur dann eine schon vergebene ID bekommen, wenn einer der folgenden beiden Bedingungen eintritt.
   Wenn die ID des linken Knoten (leftschild.id) gleich der ID des rechten Knoten (rightchild.id) ist, bekommt der Knoten mit dem Index i die gleiche ID wie der linke Knoten.

Zweitens, wenn ein Knoten u mit demselben Index i existiert und die ID vom linken Knoten von u gleich der ID vom linken Knoten von v (u.leftchild.id = = v.leftchild.id) ist und die ID vom rechten Knoten von u gleich der ID vom rechten Knoten von v ist (u.rightchild.id == v.rightchild.id), dann sind die beiden Subgraphen u und v isomorph und die ID von v muß gleich der ID von u gesetzt werden.

Zur Veranschaulichung siehe Abbildung 14.

5.5 Die Verknüpfungen and, or und xor

Diese Verknüpfungen werden durch folgende Wertetabellen als Array dargestellt:

a	b	$a \wedge b$
0	**0**	0
0	**1**	0
1	**0**	0
1	**1**	1

a	b	$a \vee b$
0	**0**	0
0	**1**	1
1	**0**	1
1	**1**	1

a	b	$a \oplus b$
0	**0**	0
0	**1**	1
1	**0**	1
1	**1**	0

Sie rufen die Methode „apply“ auf, die ein Array benötigt, welches die jeweilige Definition der Operation beinhaltet (für „and“ also 0,0,0 und 1).

5.6 Die Methode not

Es gilt $\neg a = a \oplus 1$, wie sich leicht durch zwei Wertetabellen zeigen läßt.

a	$\neg a$
0	1
1	0

a	$a \oplus 1$
0	1
1	0

Daher wird die Methode „not“ durch ein „xor“ implementiert.

5.7 Apply und ApplyStep

Diese Methoden sind die Basis zur Erstellung der eigentlichen Repräsentation einer Funktion gemäß ihrer Operatoren in einem Booleschen Ausdruck. Sie wurden nach Bryant [Bryant 86] implementiert.
Sie benötigen zwei BDD’s f und g sowie einen binären Operator, dargestellt durch ein Array, um aus diesen drei Parametern den reduzierten BDD zu erzeugen, der diesen Booleschen Ausdruck darstellt.

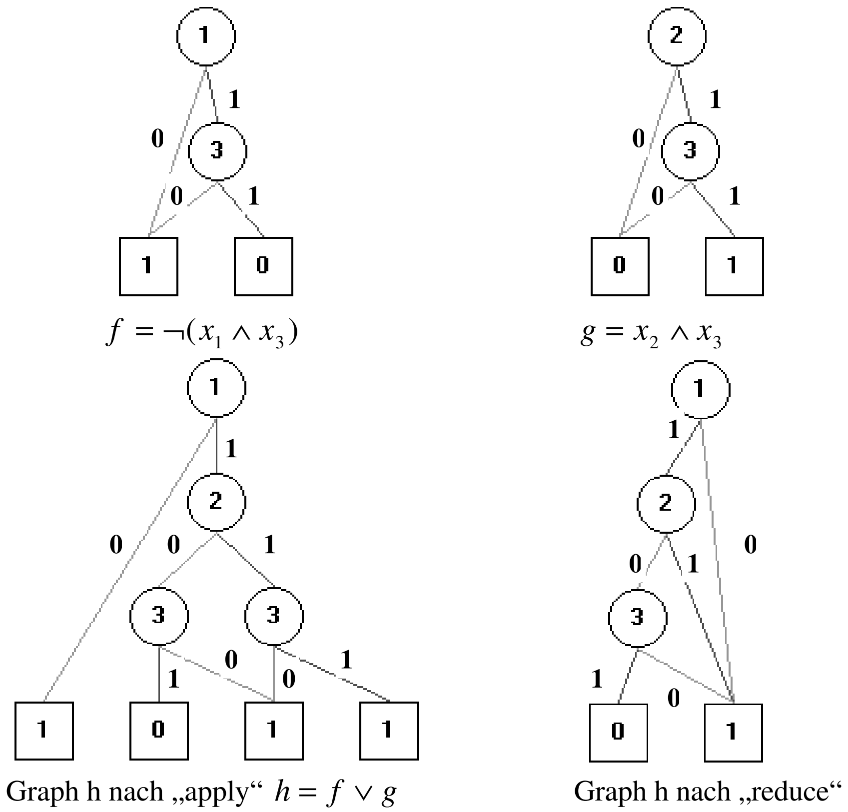


Abbildung 15 Beispiel für die Methode „apply" angewendet auf f, g und der Booleschen Operation oder.

Der Algorithmus beginnt bei den Ursprungsknoten der beiden übergebenen BDD’s und erzeugt die resultierenden Knoten anhand der Knoten einer Ebene von f und g, also anhand zweier Knoten mit gleichem Index.
Angenommen wir haben zwei BDD’s mit den Ursprungsknoten v1 und v2, so müßten wir vier verschiedene Fälle betrachten.

Fall 1: Beide Knoten sind Blätter, d.h. ihr Index ist gleich dem Wert von terminalIndex.

In diesem Fall ist der resultierende Knoten wieder ein Blatt mit einem Wert (value), der sich aus dem Ergebnis der Booleschen Operation bezüglich der Werte von v1 und v2 ergibt.

Fall 2: Einer der beiden Knoten ist kein Blatt, und die Indizes von v1 und v2 sind gleich.

In diesem Fall ist der resultierende Knoten ein Knoten mit Index i.
Der linke Knoten (leftchild) des resultierenden Knotens berechnet sich durch rekursiven Aufruf der Methode „applyStep“ mit dem linken Knoten von v1 und dem linken Knoten von v2.

Der rechte Knoten (rightchild) des resultierenden Knotens berechnet sich analog ebenso aus einem rekursiven Aufruf von „applyStep“ mit dem rechten Knoten von v1 und dem rechten Knoten von v2.

Fall 3: v1 hat den Index von i und v2 hat einen grösseren Index als v1.

In diesem Fall entsteht ebenso ein resultierender Knoten mit Index i, aber da v2 unabhängig von der Variablen mit dem Index i ist, gilt folgendes:
Der linke Knoten des Resultierenden berechnet sich aus dem rekursivem „applyStep“ mit dem linken Knoten von v1 und v2.
Der rechte Knoten des Resultierenden berechnet sich analog aus rekursivem „applyStep“ mit dem rechten Knoten von v1 und v2.

Fall 4: wie Fall3 nur vertauscht.

Das Hauptproblem dieses Algorithmus ist sein exponentielles Komplexitätsmaß, da jeder Aufruf eines inneren Knotens mit „applyStep“ zwei weitere rekursive Aufrufe dieses Algorithmus nach sich zieht. Es gibt zwei Ansätze, um dieses Komplexitätsmaß zu verbessern.

Der erste geht von dem Problem aus, daß der Algorithmus ein gegebenes Paar von Subgraphen mehrmals auswertet, was aber nicht erforderlich wäre.
Daher wird wird eine Hashtabelle (BddHash hier als zweidimensionales Array) in Form von Einträgen (v1,v2,u) gehalten, welche besagen, daß der Aufruf von apply mit den beiden BDD’s mit Ursprungsknoten v1 und v2 als Resultat einen BDD mit Ursprungsknoten u ergibt.
Bevor jetzt der rekursive Aufruf von apply geschieht, wird in der Hashtabelle nach einem Eintrag der beiden Knoten gesucht. Wenn dieser existiert, kann das Ergebnis u sofort zurückgegeben werden. Ansonsten wird, wie oben beschrieben, weiter verfahren und ein neuer Eintrag in der Hashtabelle angelegt. Nach [Bryant 86] ergibt sich so eine Verbesserung des Komplexitätsmaß zu $O(|G1|*|G2|)$.

Als zweite Verbesserung des Algorithmus wird überprüft, ob sich das Ergebnis der Operation möglichst frühzeitig bestimmen läßt. Dies wäre zum Beispiel bei einem „oder“ möglich, da der Wert von einem beliebigen Knoten v1 mit dem Wert 1 immer zu einem Blatt mit dem Wert 1 abgeschlossen werden kann.

5.8 Equals

Diese Methode überprüft ob zwei BDD’s gleich sind.
Sie traversiert die beiden BDD’s gleichzeitig beginnend mit dem Ursprungsknoten durch wiederholten rekursiven Aufruf der Methode „equals“. Dabei wird bei jedem Knoten jeder Ebene überprüft, ob ein gleicher Knoten im zweiten BDD existiert. Ist dies für alle Knoten gewährleistet, so sind die beiden BDD’s gleich.
Zwei Knoten sind genau dann gleich, wenn sowohl der Index als auch der Wert beider Knoten gleich sind.

5.9 Eval

Diese Methode dient zur Auswertung eines BDD's bei einer bestimmten Variablenbelegung.
Die Variablenbelegung des BDD's wird mittels eines Arrays aus Nullen und Einsen übergeben, welches die Variablenbelegung gemäß der Variablenordnung angibt.
Anhand dieses Arrays wird der Graph zu den Blättern gemäß der Variablenbelegung durchlaufen. Das heißt, bei einer Eins wird der rechte Pfad (rightchild) und bei einer Null der linke Pfad (leftchild) durchlaufen.
Sobald bei einem Blatt angekommen, wird der entsprechende Wert zurückgegeben.

5.10 nodes

Die Methode „nodes" gibt die Anzahl der Knoten eines BDD's an.
Sie traversiert den BDD mit Hilfe der Methode „makeSubgraph" und gibt die Größe des erhaltenen Vektors zurück, da diese gleich der Knotenanzahl ist. (siehe 4.3)

5.11 depth

„depth" gibt die Tiefe eines BDD an (siehe 3.4).
Es wird ebenfalls wie bei der Methode „nodes" ein Vektor mit Hilfe von „makeSubgraph" erzeugt. Anschließend wird ein Array von boolean-Werten gefüllt, und zwar werden die Variablen auf „true" gesetzt, deren Index im Vektor auftaucht.
Nun werden nur noch die Einträge gezählt, die „true" sind, und man erhält die Tiefe des BDD's.

5.12 showNiceBdd

Diese Methode stellt einen BDD auf einem übergebenen „Java-Graphics-Objekt" dar.
Sie besitzt 6 Übergabeparameter

1. Das Graphics-Objekt **g** auf dem gezeichnet werden soll,
2. einen Wert **obererAbstand,** der den Abstand des zu zeichnenden BDD von vertikal null angibt,
3. einen Wert **width,** der den Durchmesser eines Knoten angibt,
4. einen Wert **height,** der den Abstand zwischen zwei Ebenen angibt,
5. einen Wert **moveRowIndex,** der gegebenenfalls die Knoten-Ebene angibt, die verschoben werden soll,
6. und einen Wert **distance,** der gegebenenfalls angibt, um wieviel Pixel eine Knotenebene verschoben werden soll.

Zunächst wird der BDD mit Hilfe der Methode MakeSubgraph traversiert und anschließend im Vektor vSubgraph gespeichert. Um die Knoten einzelnen Ebenen zuzuordnen, wird eine Liste (knotenEbene) mit Vektoren so gefüllt, daß jeder Knoten mit Index i dem i'ten Vektor in der Liste zugeordnet wird.

Wird der BDD das erstemal dargestellt, so werden die x- und y-Koordinaten neu berechnet. Falls das Flag „isxySet“ auf true gesetzt ist, werden die Koordinaten nicht neu berechnet.

Fall1: Neuberechnung der Koordinaten:

Es wird die Ebene mit der größten Anzahl an Knoten bestimmt und ihre maximale Breite in Pixeln berechnet. Mit den gegebenen und berechneten Werten läßt sich nun jede x- und y-Koordinate eines Knoten berechnen. Es ergeben sich folgende 2 Formeln:

1. $y_{i,j} = obererAbstand + height \cdot i$; i wird der jeweiligen Knotenebene zugeordnet, j dem jeweiligen Knoten der Ebene.
2. $x_{i,j} = maxPixelWidth + width * (size(i) - 1 + 2 * (size(i) - j))$; j wird dem jeweiligen Knoten der Ebene zugeordnet, i der entsprechenden Knotenebene. „size(i)“ bestimmt die Anzahl der Knoten einer Ebene.

Fall2: Verschiebung einer Knotenebene

Dieser Fall wird nur durchlaufen, sofern der Wert von „distance“ ungleich null und der Wert von moveRowIndex zwischen 0 und der Tiefe des aktuellen BDD’s ist. Ist dies der Fall, dann werden die x-Koordinaten aller Knoten der ausgewählten Reihe („moveRowIndex“) um den angegebenen Wert (von „distance“) an Pixeln verschoben.

Fall3: Neuzeichnung des BDD’s

Der aktuelle BDD wird auf dem angegebenen Graphics-Objekt gezeichnet. Hierbei werden alle Knoten aller Ebenen nacheinander an die Position der angegebenen x- und y-Koordinaten gezeichnet. In die Mitte der Kreise wird, sofern die Größe des Fonts es zuläßt, der Index geschrieben. Handelt es sich um Blätter, so werden an die Position der x- und y-Koordinaten statt Kreise Rechtecke gezeichnet. Außerdem werden Verbindungen von jedem Knoten zu ihrem rightchild (rote Linie) und ihrem leftchild (grüne Linie) gezogen.

5.13 moveNiceBdd

Diese Methode verschiebt eine Knotenebene eines BDD’s. Sie arbeitet genauso wie der Fall 2 „Verschiebung einer Knotenebene“ in showNiceBdd (siehe Abb. 13).

Name	*Boolescher Ausdruck*	*ITE-Form*
0	0	0
1	1	1
NOT(F)	$\neg F$	$ite(G,0,1)$
AND(F,G)	$F \wedge G$	$ite(F,G,0)$
OR(F,G)	$F \vee G$	$ite(F,1,G)$
XOR(F,G)	$F \oplus G$	$ite(F,\neg G,G)$

Tabelle 4: Einige Boolesche Funktion und ihre Form als ITE-Operation

6 Ausblick

Die vorliegende Klassenbibliothek dient zur Erstellung und Visualisierung von OBDD's. Sie implementiert die originalen Algorithmen von [Bryant 86] und bildet damit eine gute Repräsentation vieler Boolescher Funktionen und eine effiziente Möglichkeit solche Funktionen auszuwerten. Darüber hinaus sollten die folgenden drei Punkte noch Berücksichtigung finden, die aber aufgrund des engen Rahmens dieser Studienarbeit nicht geleistet wurden.

1. Eine gemeinsame Verwaltung der OBDD's pro BDD-Manager.
2. Eine Verbesserung des Hashing - Mechanismus durch Einführung des ITE - Operators [Brace et al. 90].
3. Eine effizientere Repräsentation durch dynamische Variablenordnung [Rudell 93].

Durch den ITE-Operator kann jede Funktion dargestellt werden (siehe Tab. 4). Zur Erinnerung: Der ITE-Operator kann durch folgende Boolesche Funktion dargestellt werden:

$$ITE(f,g,h) = f \wedge g \vee \neg f \wedge h$$

Wenn man jede Funktion durch den ITE Operator implementiert, so ist es möglich, eine Hash - Tabelle zu verwalten, die ein Tripel (v,g,h) auf einen OBDD - Knoten F abbildet. Diese sogenannte „Unique-Table" enthält einen Eintrag für jeden Knoten eines OBDD's. Bevor ein neuer Eintrag angefordert wird, wird in der Unique-Table nachgeschaut, ob schon einer vorhanden ist. Wenn ein Eintrag existiert, wird dieser Knoten benutzt, andernfalls wird ein neuer angelegt. Mit Hilfe dieses Mechanismus läßt sich ein enormer Geschwindigkeitsvorteil erzielen, da die mehrfache Neuberechnung von Zwischenergebnissen vermieden wird.

Als zweite Möglichkeit zur Effizienzsteigerung könnte man die Variablenordnung mit Hilfe verschiedener Algorithmen dynamisch verändern [Rudell 90]. Die Ergebnisse von ungünstigen Variablenordnungen wurden schon in den Abbildungen 3, 9 und 10 dargestellt. Durch eine dynamische Variablenordnung wäre es möglich, die Größe der OBDD's möglichst gering zu halten.

7 Literaturliste

Die folgende Liste enthält Literaturangaben zu den im Text zitierten Arbeiten. Es wird auch der Titel [Marcus et al 96], genannt der mich beim Vorbereiten und Schreiben der Studienarbeit unterstützt hat. Außerdem werden folgende Java-Bücher angegeben die mir bei der Programmierarbeit geholfen haben: [Hendrich 97], [Chan et al. 96], [Cornell et al. 96].

[Bryant 86] Bryant, R. (1986): **Graph-Based Algorithms for Boolean Function Manipulation**, IEEE Transactions on Computers, Vol. C-35, No. 8, 677-691

[Duden 93] Duden (1993): **Duden Informatik Ein Sachlexikon für Studium und Praxis**, Dudenverlag, Mannheim, 2.Aufl.

[Akers 78] Akers, S. (1978)**: Binary Decision Diagrams,** IEEE Transactions on Computers, Vol. C-27, 509-516

[Brace et al. 90] Brace, K.,Rudell, R., Bryant, R. (1990)**: Efficient Implementation of a BDD Package Proceeding**, 27th Design Automation Conference, June 1990, 40-45

[Rumbaugh et al. 94] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, W., Lorensen, W. (1994): **Objektorientiertes Modellieren und Entwerfen**, Carl Hanser Verlag München Wien

[DeMicheli 94] DeMicheli, G.(1994): **Synthesis and Optimization of Digital Circuits**, McGraw-Hill New York

[Schöning 92] Schöning, U. (1992): **Theoretische Informatik kurz gefaßt**, BI Wissenschaftsverlag Mannheim

[Aho et al. 92] Aho, A., Sethi, R.,Ullman J. (1992): **Compilerbau Teil 1**, Addison Wesley, Deutschland, 2. Nachdruck

[Schiffmann et al. 92] Schiffmann, W., Schmitz, R. (1992): **Technische Informatik 1 Grundlagen der digitalen Elektronik**, Springer Verlag Berlin.

[Marcus et al .96] Marcus, D.,Lichter, H., Ludewig, J., Schneider, K. (1996): **Studien-Arbeiten ein Leitfaden zur Vorbereitung, Durchführung und Betreuung von Studien-, Diplom- und Doktorarbeiten am Beispiel Informatik**, vdf Hochschulverlag Zürich, und Teubner, Stuttgart, 2 Aufl.

[Rudell 93] Rudell, R. (1993): **Dynamic Variable Ordering for Ordered Binary Decision Diagrams**, ICCAD, Proceedings of the International Conference on Computer Aided Design, 42-47

[Mertsching 96] Mertsching, B. (1996): **Materialien zur Vorlesung Grundzüge der Informatik B**, Vorlesungsskript, Fachbereich Informatik Universität Hamburg

[Wirth 86] Wirth, N. (1986): **Algorithmen und Datenstrukturen mit Modula - 2**, B. G. Teubner Stuttgart, 3 Aufl.

[Hendrich 97] Hendrich, N. (1997): **Java für Fortgeschrittene**, Springer-Verlag Berlin Heidelberg New York

[Chan et al. 96] Chan, P., Lee, R. (1996): **The Java Class Libraries. An Annotated Reference,** Addison Wesley

[Cornell et al. 96] Cornell, G., Horstmann, C. (1996): **Java bis ins Detail. Das Buch für Experten**, Heinz Heise Verlag Hannover