Universität Bremen

Bachelorarbeit

im Studiengang Informatik - Schwerpunkt Logiksynthese und Verifikation

zur Erlangung des akademischen Grades Bachelor of Science

Thema:	Effiziente Implementierung von Binären Entscheidungsdiagrammen
Autor:	Rune Krauß <krauss@uni-bremen.de> Matrikelnummer: 4258388
Version vom:	15. August 2018
1. Betreuer:	Prof. Dr. Rolf Drechsler
2. Betreuer:	Dr. Sabine Kuske

Vorwort

Die Manipulation von Booleschen Funktionen ist ein wichtiger Bestandteil des rechnergestützten Konstruierens von digitalen Schaltungen. Diese Arbeit beschreibt ein Softwarepaket, um Boolesche Funktionen in Form eines reduzierten geordneten Binären Entscheidungsdiagramms effizient zu manipulieren, das auf einer effizienten Implementierung vom ITE-Operator basiert. Dazu wird eine Hashtabelle benutzt, um eine kanonische Form des binären Entscheidungsdiagramms zu gewährleisten. Die Verschmelzung dieses Datentyps mit Knoten in eine hybride Datenstruktur führt dazu, dass die Speichernutzung erheblich verbessert wird. Weiterhin erfolgt die Implementierung eines hashbasierten Caches für den ITE-Operator, um weiterhin die Speichernutzung zu verringern. Darüber hinaus wird der Cache insbesondere dadurch verbessert, dass spezielle Regeln für komplementäre Kanten zum Einsatz kommen, wodurch äquivalente Funktionen erkannt werden können. Für die Regulierung der Speichernutzung gibt es eine automatische Speicherbereinigung mit geringen Kosten, um dem Hauptproblem des Speicherplatzes bezüglich binärer Entscheidungsdiagramme entgegenzuwirken. Abschließend demonstrieren experimentelle Ergebnisse die Korrektheit dieses Paketes und werden für einen Vergleich mit einem – dem neuesten Stand der Technik entsprechenden – Paket herangezogen.

Prof. Dr. Rolf Drechsler danke ich für viele hilfreiche Gespräche und Diskussionen sowie die umfassende Betreuung. Weiterhin möchte ich mich bei M. Sc. Marcel Walter und Dipl.-Inf. Kenneth Schmitz für ihr Engagement in Form von intensiven Gesprächen, konstruktiver Kritik und Verbesserungsvorschlägen zu meinen Erkenntnissen bedanken. An dieser Stelle möchte ich auch an Dr. Sabine Kuske ein Danke für ihr Feedback zu dieser Arbeit ausrichten.

Mein innigster Dank gilt meiner Verwandtschaft, die immer für mich da ist und die Erreichung meiner Ziele zu jeder Zeit unterstützen.
Diese Arbeit widme ich euch in Liebe.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

ADT	Abstrakter Datentyp
ALU	Arithmetisch-logische Einheit
BDD	Binary Decision Diagram
BP	Branchingprogramm
CAD	Computer-Aided Design
CE	Complement Edge
CPU	Central Processing Unit
CT	Computed-Table
DC	Don't Care
DD	Decision Diagram
DFS	Depth First Search
DNF	Disjunktive Normalform
DOT	Graph Description Language
DRAM	Dynamic Random Access Memory
FPGA	Field Programmable Gate Array
GCF	Generalized co-factor
IC	Integrierter Schaltkreis
ID	Identifikator
ITE	If-Then-Else
IWLS	International Workshop on Logic and Synthesis
KNF	Konjunktive Normalform
L1	Level-1-Cache
L2	Level-2-Cache
LSB	Least Significant Bit
MR	Memory Ratio
NTM	Nichtdeterministische Turingmaschine
OS	Betriebssystem
ROBDD	Reduced Ordered Binary Decision Diagram
ROM	Read-Only Memory
SAT	Satisfiability
SBDD	Shared Binary Decision Diagram
TM	Turingmaschine
TR	Time Ratio
UML	Unified Modeling Language
UT	Unique-Table
VLSI	Very-Large-Scale-Integration

1 Einleitung

Die Repräsentation und Manipulation von Booleschen Funktionen ist bedeutsam für Algorithmen in Applikationen. Somit können Probleme bei dem Very-Large-Scale-Integration (VLSI) CAD, d. h. computergestützten Schaltkreisentwurf, als Sequenz von Operationen über eine Menge von Booleschen Funktionen ausgedrückt werden. Beispiele hierfür sind unter anderem die kombinatorische Logik-Verifikation [MWB88], symbolische Simulation [Cho88] und das Model Checking [Kra17].
Es ist daher erstrebenswert, ein Softwarepaket zu entwickeln, das die Erstellung von Variablen als auch Operationen wie `and`, `or`, `not` usw. auf Funktionen erlaubt. Zudem sollte das Paket auch u. a. einen Tautologietest, d. h. eine immer wahr berechnende Funktion für alle Eingaben gewährleisten (siehe Kapitel 2.4.5 auf Seite 31). Der Tautologietest ist co-NP-vollständig, d. h. das Komplement liegt in der Klasse NP (siehe Kapitel 2.1 auf Seite 11). Diese enthält alle Entscheidungsprobleme, die von einer nichtdeterministischen Turingmaschine (NTM) hinsichtlich der Eingabelänge in Polynomialzeit gelöst werden können. Ist ein solcher Algorithmus nicht effizient implementiert, so kann die Berechnung aller bekannten Lösungen eine gewisse Zeit erfordern, die mit der Anzahl von Variablen im Worst Case exponentiell wächst.
Es ist also ratsam, eine durchdachte Repräsentation und effiziente Algorithmen zur Manipulation zu wählen, um diesen Fall zu verhindern. Werden bspw. zur Darstellung Boolescher Funktionen Normalformen wie die Disjunktive Normalform (DNF) oder Konjunktive Normalform (KNF) verwendet, so haben einige Funktionen eine exponentielle Größe [BMH84]. Sie sind weiterhin nur für wenige Eingänge einsetzbar, da sie viel Speicherplatz benötigen [Hei14, S.394-395]. Um eine arbiträre Kompaktheit sowie Effizienz der Manipulation zu gewährleisten, gibt es ein Binäres Entscheidungsdiagramm (BDD) bzw. Branchingprogramm (BP), das gut erforscht ist [FH78, S.227-240]. BDDs sind eine heuristische Methode, um eine große Menge an Zuständen zu repräsentieren und sind typischerweise übersichtlich, wenn der Zustandsraum Symmetrien enthält. Dies ist genau dann der Fall, wenn ein System ähnliche Module enthält. Zu erwähnen ist, dass BDDs in einigen, aber nicht in allen Fällen effizienter als bspw. Wertetabellen sind [Sie07, S.46].
Um z. B. die Tautologie in konstanter Zeit berechnen zu können, wird die Kanonizität benötigt, d. h. eine eindeutige Darstellung, die ein reduzierter geordneter BDD (ROBDD) bietet. Somit muss nur ein ROBDD zu der untersuchenden Funktion erstellt und mit 1 bzw. 0 verglichen werden. Durch die Kanonizität wird ebenfalls der Äquivalenztest bzw. Test auf Erfüllbarkeit problemlos möglich (siehe Kapitel 2.4.1 ab Seite 26), wozu lediglich die ROBDDs auf Isomorphie geprüft werden müssen bzw. nachgeschaut werden muss, ob die Wurzel das 0-Blatt ist. Darüber hinaus ist es für ein allgemeines BDD unklar, wie eben genannte Operationen `and` etc. effizient realisiert werden können [Gün02]. Für ein ROBDD sind diese - wegen Einschränkungen -

hingegen polynomiell beschränkt. Zu beachten ist, dass die Größe eines ROBDD im Worst Case ebenfalls exponentiell sein kann, was von der Variablenordnung abhängt, die optimal gewählt werden muss (siehe Kapitel 2.3 ab Seite 19).
Hinzuzufügen ist, dass es bereits einige BDD-Pakete wie CUDD[1] oder CacBDD[2] gibt, die jedoch auch Schwachstellen aufweisen [Kra17]. So konnte bspw. beim Model Checking mit der webbasierten Schnittstelle „IscasMC“ herausgefunden werden, dass eine arbiträre Abhängigkeit zum Model besteht, was sich wiederum auf die Performanz von BDD-Paketen auswirkt. Das Ziel ist es daher, ein generisches Paket zu erstellen, das – von der Berechnung als auch dem Speicherverhalten her – effizient ist. In dieser Arbeit wird eine dementsprechende Implementierung eines ROBDD-Paketes mit verschiedenen Hashtabellen und komplementären Kanten beschrieben, die in Kapitel 3 ab Seite 33 ersichtlich ist. Die dazu benötigten Grundlagen werden in Kapitel 2 ab Seite 11 erklärt. Anschließend erfolgt die Erläuterung zu einer Implementierung eines neuen Ansatzes, um die Kollisionsbehandlung in einer Hashtabelle durchzuführen, der in Kapitel 3.4.2 ab Seite 53 zu finden ist.

[1] CUDD, `http://vlsi.colorado.edu/~fabio/`
[2] CacBDD, `http://www.kailesu.net/CacBDD/`

2 Grundlagen

Wie bereits in der Einleitung auf Seite 9 erwähnt, gibt es viele verschiedene Möglichkeiten, Boolesche Funktionen zu repräsentieren. Mithilfe von BDDs können Boolesche Funktionen effizient dargestellt werden. So können sie z. B. dazu eingesetzt werden, um große Primimplikantenmengen (siehe Kapitel 2.1.2 auf Seite 13) unter Verwendung der charakteristischen Funktion effizient darzustellen [MS99, S.106-108]. Im Gegensatz dazu sind Normalformen wie bspw. die DNF oder KNF in der Praxis nur für sehr wenige Eingänge einsetzbar, da sehr viel Speicherplatz benötigt wird (siehe Kapitel 2.1.3 auf Seite 15). Formeln in KNF oder DNF lassen sich direkt in Second-Level Schaltungen realisieren, BDDs sind ein Beispiel für mehrstufige Schaltkreise (siehe Kapitel 2.1.6 auf Seite 16) bzw. eine Datenstruktur für Logiksynthese und Verifikation, womit Entscheidungs- bzw. Berechnungsprobleme wie der Äquivalenztest oder die Synthese (siehe Kapitel 2.4.1 ab Seite 26) effizient lösbar sind.
Die Grundfrage ist dabei, welche Anforderungen an Datenstrukturen für Boolesche Funktionen gestellt werden. Bei der Verifikation bspw. soll für einen Schaltkreis überprüft werden, ob er eine gewünschte Funktion realisiert, die ebenfalls durch eine Spezifikation gegeben ist. Für beide Funktionen kann nun ein BDD berechnet werden, inklusive einer anschließenden Überprüfung auf Äquivalenz, wobei dieses Vorgehen – wie auch der Tautologietest – im Allgemeinen für Formeln als auch Schaltkreise co-NP-vollständig ist [Sie07, S.6]. Die Erwartungshaltung sollte daher sein, dieses Verfahren für praktisch relevante Funktionen effizient durchführbar zu gestalten.
Insgesamt gesehen werden also viele verschiedene Anforderungen an Boolesche Funktionen gestellt, wobei die wichtigsten Operationen in Kapitel 2.4 ab Seite 24 diskutiert werden.

2.1 Formale Definitionen

In dieser Arbeit werden verschiedene Operationen in Bezug auf Entscheidungs- bzw. Berechnungsprobleme algorithmisch realisiert als auch die Größenordnung der Rechenzeit angegeben, wobei ein Algorithmus die Beschreibung einer Methode zur Lösung eines Problems kennzeichnet. Die Fälle umfassen dabei den Worst-, Best- und Average Case.
Hierzu wird die sog. *asymptotische Komplexität* benutzt, die den Aufwand in Abhängigkeit von der Größe der Eingabe angibt. Sei hierzu also $f : \mathbb{N} \to \mathbb{R}_0^+$ eine Funktion. Die Ordnung von f ist die Menge $O(f(n)) = \{t : \mathbb{N} \to \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : t(n) \leq c \cdot f(n)\}$, die das Wachstum von f charakterisiert, d. h. sie enthält alle Funktionen, deren Graph maximal so stark wächst wie der Graph von f. Es wird dabei im Wesentlichen die Zeitkomplexität (Anzahl der benötigten Programmschritte)

betrachtet, da sie unmittelbar eine obere Schranke für die Platzkomplexität (Größe des benötigten Speicherplatzes) impliziert [MT98, S.118]. Eine Menge $M \subseteq \mathbb{R}$ heißt nach oben beschränkt, falls es ein $K \in \mathbb{R}$ gibt, sodass $x \leq K \forall x \in M$. K heißt *obere Schranke* von M. Die kleinste obere Schranke von M wird als *Supremum* bezeichnet. $M \subseteq \mathbb{R}$ heißt wiederum *nach unten beschränkt*, falls es ein $k \in \mathbb{R}$ mit $x \geq K \forall x \in M$ gibt. Dabei wird k als *untere Schranke* von M bezeichnet. Die kleinste untere Schranke von M heißt *Infimum*.
Allgemein betrachtet ist ein Entscheidungsproblem eine Abbildung $D : IN \to BOOL$. D heißt *entscheidbar*, falls eine Turingmaschine (TM) existiert, sodass sie in einem Endzustand stoppt, falls $D(x) = JA$, wobei $x \in IN$. Ansonsten hält sie in einem Nicht-Endzustand.
Es gilt $D \in NP$, falls die TM Polyzeit-beschränkt und nichtdeterministisch ist, d. h. es können zu jedem Berechnungszustand mehrere Folgezustände existieren. Sollte die TM Polyzeit-beschränkt und deterministisch sein, so gilt $D \in P$. P und NP kennzeichnen diesbezüglich Komplexitätsklassen. Viele Probleme sind darüber hinaus nicht nur in NP, sondern auch NP-hart, wodurch NP-Vollständigkeit gekennzeichnet ist. D heißt NP-hart, falls $D' \geq_p D \forall D' \in NP$, wobei $\geq_p$ eine Polynomialzeitreduktion kennzeichnet. Es gilt also $D' \geq_p D$, falls es eine totale berechnende Funktion $trans : IN' \to IN$ mit polynomiell beschränkter Berechnungslänge gibt, sodass $D'(x) = JA \Leftrightarrow D(trans(x)) = JA$. Es werden also alle Ja- bzw. Nein-Instanzen auf das andere Problem bei einer Reduktion abgebildet. Intuitiv gesagt sind Probleme, die NP-vollständig sind, wahrscheinlich nicht effizient lösbar.
Entscheidbarkeit ist weiterhin ein Spezialfall von Berechnungsproblemen, die als Funktionen $\mathbb{N}^k \to \mathbb{N}$ (k sei die Stelligkeit) kodiert werden, wobei die Ausgabe eine Menge von Objekten ist.

2.1.1 Boolesche Funktionen

Sei $f \subseteq N \times M$ eine *Relation*. Dann heißt f, falls $\forall x \in N, y, z \in M (x, y) \in f \wedge (x, z) \in f \Rightarrow y = z$ rechtseindeutig und $\forall x \in N \exists y \in M : (x, y) \in f$ linkstotal ist, *Abbildung* von N nach M. N kennzeichnet die Quell- und M die Zielmenge der Abbildung. Die Abbildung f heißt *injektiv*, falls $\forall x, y \in N : f(x) = f(y) \Rightarrow x = y$ und *surjektiv*, falls $\forall y \in M \exists x \in N : f(x) = y$ besteht. Gelten beide Eigenschaften, so wird die Abbildung als *bijektiv* bezeichnet. Sei $A \subseteq N$, dann heißt $f(A) = \{f(x) \mid x \in A\}$ *Bild* von A. Ist $B \subseteq M$, dann heißt $f^{-1}(B) = \{x \in N \mid f(x) \in B\}$ *Urbildmenge* von B.
Abbildungen $f : \{0,1\}^n \to \{0,1\}^m$ $(n, m \in \mathbb{N})$ werden als *Boolesche Funktionen* bezeichnet. Für die Eigenschaften gilt:

- $(\mathbb{B}^n, f)$ heißt *vollständig*, gdw. f surjektiv ist.
- $(\mathbb{B}^n, f)$ heißt *eindeutig*, gdw. f injektiv ist.

- $(\mathbb{B}^n, f)$ heißt *kanonisch*, gdw. f bijektiv ist.
- $(\mathbb{B}^n, f)$ heißt *operationell abgeschlossen*, gdw. $\forall a, b \in \mathbb{B}^n \; \exists$ Darstellungen $u, v, w \in \mathbb{B}^n$ mit
  - $f(u) = f(a) \cdot f(b)$
  - $f(v) = f(a) + f(b)$
  - $f(w) = f(a)'$

$\mathbb{B}_{n,m}$ beschreibt die Menge aller Booleschen Funktionen von $\mathbb{B}^n \to \mathbb{B}^m$. Eine Abbildung $f : D \to \mathbb{B}^m, D \subset \mathbb{B}^n$ heißt *(partielle) Boolesche Funktion* in n Variablen. Sei $m = 1$, dann gilt $ON(f) := \{x \in \{0,1\} \mid f(x) = 1\}$ (Erfüllbarkeitsmenge) und $OFF(f) := \{x \in \{0,1\} \mid f(x) = 0\}$ (Nichterfüllbarkeitsmenge). Ist f eine partielle Boolesche Funktion, so bezeichnet $def(f) := \{x \in \{0,1\}^n \mid f(x)$ definiert$\}$ den *Definitionsbereich* von f. Hingegen kennzeichnet $dc(f) := \{x \in \{0,1\}^n \mid f(x)$ nicht definiert$\}$ den *don't care-Bereich* von f.
Weiterhin werden Boolesche Funktionen $f \in \mathbb{B}_{n,m}$ als m-dimensionale Vektoren $(f_1, \ldots, f_m)$ Boolescher Funktionen $f_i \in \mathbb{B}_n, 1 \leq i \leq m$ aufgefasst.
Eine Boolesche Funktion $f(x_1, \ldots, x_n)$ heißt genau dann *symmetrisch*, wenn der Funktionswert durch einen Vektor $w = (w_0, \ldots, w_n), w_i \in \{0,1\}$ angegeben werden kann, wobei $w_i = 1 \Leftrightarrow f(a_1, \ldots, a_n) = 1$ und $|\{a_j, j = 1 \ldots n : a_j = 1\}| = i$. Der Funktionswert ist also nur von der Anzahl der Einsen im Argument, nicht aber von deren Position abhängig.
Hinsichtlich der Eigenschaften gibt es insgesamt $2^{m \cdot 2^n}$ Boolesche Funktionen in $B_{n,m}$ und 2^{2^n} Boolesche Funktionen in $\mathbb{B}_n$, wobei wenige davon – die wenigstens 2^{n-1} Bits zur Darstellung in einer beliebigen Form benötigen – exponentiell gegen 1 für $n \to \infty$ konvergieren [Het02, S.17]. Es kann also nur erreicht werden, dass möglichst viele praktisch relevante Funktionen kompakt darstellbar sind (siehe Kapitel 2 auf Seite 11).

2.1.2 Boolesche Ausdrücke

Boolesche Funktionen können in Form von *Booleschen Ausdrücken* dargestellt werden, die auf den Regeln der *Booleschen Algebra* basieren. Sei hierzu M eine endliche Menge, auf der zwei binäre Operationen $\cdot, +$ und eine unäre Operation $\sim$ definiert sind. Das Quadrupel $(M, \cdot, +, \sim)$ heißt Boolesche Algebra, falls $\forall x, y, z \in M$ die folgenden Axiome gelten:

- Kommutativität: $x + y = y + x$ sowie $x \cdot y = y \cdot x$
- Assoziativität: $x + (y + z) = (x + y) + z$ sowie $x \cdot (y \cdot z) = (x \cdot y) \cdot z$
- Distributivität: $x + (y \cdot z) = (x + y) \cdot (x + z)$ sowie $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$

- Absorption: $x + (x \cdot y) = x$ sowie $x \cdot (x + y) = x$
- Auslöschung: $x + (y \cdot \sim y) = x$ sowie $x \cdot (y + \sim y) = x$

Viele weitere Gesetze wie bspw. die *De Morganschen Gesetze* $(x+y) = \sim x \cdot \sim y$ sowie $\sim (x \cdot y) = \sim x + \sim y$ sind zudem aus diesen Axiomen ableitbar.
Auf Basis dieser Regeln können nun Boolesche Ausdrücke definiert werden. Sei hierzu $A = X_n \cup \{0, 1, +, \cdot, \sim, (,)\}$ ein Alphabet, wobei $X_n = \{x_1, \ldots, x_n\}$ und $n \in \mathbb{N}$ gilt. Die Menge $BE(X_n)$ ist eine Teilmenge von A^*, die wie folgt induktiv definiert ist:

- $0, 1, x_1, \ldots, x_n$ sind Boolesche Ausdrücke.
- Sind g und h Boolesche Ausdrücke, so auch die Disjunktion ($+$ bzw. $\vee$), Konjunktion ($\cdot$ bzw. $\wedge$) und Negation ($\sim$ *bzw.* $\neg$).

Darüber hinaus gilt $\vee < \wedge < \neg$, d. h. z. B. hat die Negation eine höhere Präzedenz als die Konjunktion. Boolesche Ausdrücke sind eine vollständige Darstellung von $\mathbb{B}_n$, jedoch sind sie nicht eindeutig. Jedem Booleschen Ausdruck kann ein *Operatorbaum* (siehe Abbildung 1) zugeordnet werden, der zu jedem Ausdruck angibt, wie er korrekt ausgewertet werden kann:

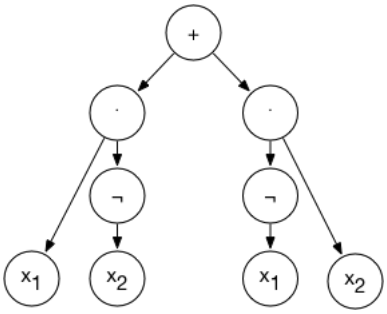

Abbildung 1: Operatorbaum für $(x_1 \neg x_2) + (\neg x_1 x_2)$

Bezüglich der Interpretation Boolescher Ausdrücke definiert also jeder Ausdruck eine Boolesche Funktion $\gamma : BE(X_n) \to \mathbb{B}_n$, wobei $\gamma(0) = 0$ sowie $\gamma(1) = 1$, d. h. $\forall x \in \mathbb{B}^n$ gilt:

- $\gamma((g + h)) = \gamma(g) + \gamma(h)$
- $\gamma((g \cdot h)) = \gamma(g) \cdot \gamma(h)$
- $\gamma((\sim g)) = \sim \gamma(g)$

Die Booleschen Ausdrücke x_i, x_i' heißen *Literale*. Ein *Monom* ist eine Konjunktion von Literalen und wird als *Minterm* m bezeichnet, falls $m(a) = \prod_{i=1}^{n} x_i^{a_i}$, wobei $a \in \mathbb{B}^n$. Die Disjunktion von paarweise verschiedenen Monomen wird als *Polynom* bezeichnet, d. h. $f = \sum_{a \in ON(f)} m(a)$, wobei $a \in \mathbb{B}^n$. Die primären Kosten sind gleich der Anzahl der Monome in f. Die sekundären Kosten hingegen sind gleich der Anzahl der Literale in Addition zu der Anzahl der Monome in f. Ein *Implikant* einer Booleschen Funktion g ist

ein Monom q mit $\gamma(q) \leq g$ und ein *Primimplikant* von g ist ein maximaler Implikant g, d. h. er ist in keinem anderen Implikanten vollständig enthalten. Zwei Boolesche Ausdrücke e_1, e_2 sind darüber hinaus äquivalent, wenn $\gamma(e_1) = \gamma(e_2)$ gilt.

2.1.3 Disjunktive und konjunktive Normalform

Die DNF bzw. KNF repräsentieren eine Boolesche Funktion $f \in \mathbb{B}_n$ als Summe von Produkten bzw. Produkt von Summen mittels $f(x) = \bigvee_{a \in f^{-1}(1)} m_a(x)$ respektive $\bigwedge_{a \in f^{-1}(0)} s_a(x)$, wobei $s_a(x) = \bigvee_{i=1}^{n} x_i^{\overline{a_i}}$ (Maxterm) und $m_a(x) = \bigwedge_{i=1}^{n} x_i^{a_i}$ (Minterm). Die DNF ist kanonisch durch die Disjunktion aller Minterme. Hingegen wird die KNF durch die Konjunktion aller Maxterme als kanonisch bezeichnet. Diesbezügliche Beispiele sind im Zusammenhang mit BDDs in Kapitel 2.2 auf Seite 18 zu finden.
Es sei erwähnt, dass durch diese Darstellung einige Nachteile auftreten. So haben viele intuitiv einfache Boolesche Funktionen, wie etwa die Paritätsfunktion $f_n = x_1 \oplus x_2 \oplus \cdots \oplus x_n$ auf n Variablen, damit eine exponentielle Darstellung. Diese symmetrische Funktion liefert 1, wenn eine ungerade Anzahl von Eingängen den Wert 1 besitzen, ansonsten gibt sie 0 zurück. Hinsichtlich der Darstellung als DNF/KNF würde es 2^{n-1} Min- bzw. Maxterme geben [Het02, S.21].

2.1.4 Wahrheitstafeln

Ein Bitstring $b[0 : 2^n - 1]$ kann durch die Boolesche Funktion $f(b)(x) = b\left[\sum_{i=0}^{n-1} x_i 2^i\right] \forall x \in \mathbb{B}^n]$, wobei $f(b) \in \mathbb{B}_n$, interpretiert werden. Dieser Bitstring heißt *Wahrheitstafel* der Booleschen Funktion $f(b)$. Bitstrings der Länge 2^n sind zusammen mit dieser Interpretation als Boolesche Funktionen kanonische Darstellungen von $\mathbb{B}_n$. Im Zusammenhang mit BDDs ist ein Beispiel dazu in Kapitel 2.2 auf Seite 18 ersichtlich.

2.1.5 Schaltkreise

Jedem Schaltkreis eine Boolesche Funktion zugeordnet werden, was auch umgekehrt gilt [Loo07]. Die Berechnung einer solchen Funktion wird als *Simulation* bezeichnet, was in Kapitel 2.4.1 auf Seite 26 beim Äquivalenztest demonstriert wird. Ein *Schaltkreis* oder *logisches Netzwerk* $SK = (X_n, G, typ, in, out, Y_m)$ ist ein gerichteter Graph über $STD := \{\neg, \wedge, \vee, \oplus, \Leftrightarrow, NAND, NOR\}$, wobei:

- $G = (V, E)$ zykelfrei
- $X_n := (x_1, \ldots, x_n)$ primäre Eingänge
- $Y_m := (y_1, \ldots, y_m)$ primäre Ausgänge
- $in : V \backslash S \to S^*, in(m) \in S^{indeg(m)}$ Eingänge

- $out : V \backslash S \rightarrow S^*, out(m) \in S^{outdeg(m)}$ Ausgänge
- $(\{0,1\} \cup X_n \cup Y_m) \subset V$ konstante Signale

Die Kosten entsprechen der Anzahl der Modulknoten $C(SK)$ und die Tiefe $depth(SK)$ wird durch die Anzahl der Modulknoten auf dem längsten Pfad von einem primären Eingang zu einem primären Ausgang beschrieben. Die Darstellung über logische Netzwerke ist vollständig, aber nicht eindeutig.
Ein *Pfad* bzw. *Weg* von v nach v' der Länge n ($n \in \mathbb{N}$) in G ist eine alternierende Knoten-Kanten-Folge $p = v_0 e_1 v_1 e_2 \dots e_n v_n$ mit:

- $v_0, \dots, v_n \in V$
- $e_1, \dots, e_n \in E$
- $v = v_0$
- $v' = v_n$

Insbesondere sei noch einmal hervorgehoben, dass der Schaltkreis kreisfrei ist, d. h. es gilt zusätzlich $v_0 \neq v_n$. In Kapitel 2.1.6 auf Seite 16 sind diesbezüglich Beispiele in Form von Multiplexern zu finden.

2.1.6 Binäre Entscheidungsdiagramme (BDDs)

Ein *BDD* ist ein gerichteter, azyklischer Graph bzw. Baum $G = (V \cup \Phi \cup \{1\}, E)$, der eine Boolesche Funktion repräsentiert sowie kreisfrei und zusammenhängend ist. Seine Knoten sind in drei Untermengen unterteilt. V ist die Menge aller internen Knoten. Diese Knoten haben je zwei ausgehende Kanten. Jeder Knoten $v \in V$ besitzt eine Bezeichnung $l(v) \in S_F$, wobei $S_F = \{x_1, \dots, x_n\}$ die Menge der Variablen ist, von denen f abhängt. $|f|$ kennzeichnet die Knotenanzahl unterhalb von f, f' sei die Negation von f. Das 1-Blatt ist der Endknoten, das keine ausgehenden Kanten besitzt. Φ ist die Menge der Funktionsknoten, die in einer eindeutigen Beziehung zu den Komponenten von f stehen. Diese haben keine eingehenden Kanten und je nur eine ausgehende Kante. Die ausgehenden Kanten von den Funktionsknoten können die Eigenschaft des Komplements besitzen. Die ausgehenden Kanten von einem internen Knoten werden mit t und e (komplementierend) bezeichnet. Mit dem *Tripel* $(l(v), t, e)$ wird ein interner Knoten und seine beiden ausgehenden Kanten bezeichnet, wobei $l(v)$ die *Entscheidungsvariable* darstellt. Dieses bestimmt dabei f eindeutig. Die *top-Variable* eines Knotens f sei die Variable, nach der f zerlegt wurde. Wird eine Menge von Knoten betrachtet, so ist sie das Minimum der top-Variablen der einzelnen Knoten, d. h. $top(l(v), t, e) = min\{top(f), top(t), top(e)\}$. Der Knoten zur eigentlichen Repräsentation wird *Formel* genannt. Die Variablen in S_F sind geordnet. Ist der Kno-

ten v_j ein Nachfolger des Knotens v_i, dann ist $l(v_i) < l(v_j)$. Für die repräsentierte Funktion gilt:

1. Die Funktion des Endknotens ist die konstante Funktion 1.
2. Ist die Kante kein Komplement, so ist die Funktion einer Kante die Funktion des Knotens, auf den sie zeigt. Ansonsten ist die Funktion der Kante das Komplement der Funktion des Zielknotens.
3. Die Funktion eines internen Knotens $v \in V$ ist gegeben durch $(l(v) \wedge f_t) \vee (\neg l(v) \wedge f_e)$ (Shannon-Zerlegung), wobei f_t, f_e die Funktionen der Kanten (t, e) von v sind. Die Funktion wird also mit Kofaktoren dargestellt.
4. Die Funktionen des Funktionsknotens $\phi \in \Phi$ ist die Funktion der davon ausgehenden Kante.

Wichtige Komplexitätsmaße für BDDs sind die Größe, d. h. die Anzahl der inneren Knoten sowie die Tiefe zur Kennzeichnung des längsten Pfades von der Wurzel zu einem Blatt.
BDDs lassen sich weiterhin unmittelbar durch ein Multiplexer-Netzwerk (Selektionsschaltungen) realisieren [PKK91]. Dabei wird ein BDD $1:1$ in Schaltungsstrukturen umgesetzt, die aus $2:1$-Multiplexern bestehen, was die Abbildung 2 zeigt: Jeder Kno-

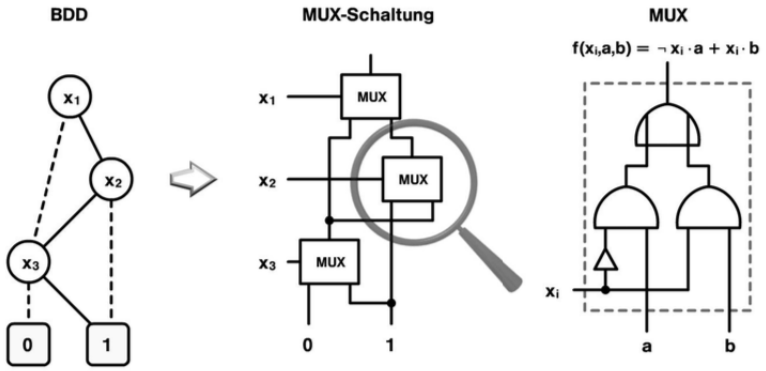

Abbildung 2: BDD zu Multiplexer [Het02, S.46]

ten steht für einen Multiplexer, wobei sie mittels einem Bottom-up Verfahren erzeugt werden. An der Wurzel ist demnach der Ausgang zu finden.
Allgemein schalten Multiplexer von vielen Eingängen auf einen Ausgang, d. h. es wird $mux_n : \mathbb{B}^{2n+1} \to \mathbb{B}^n$ berechnet, wobei:

$$mux_n(a_{n-1}, \ldots, a_0, b_{n-1}, \ldots, b_0, x_i) = \begin{cases} (a_{n-1} \ldots a_0) & x_i = 1 \\ (b_{n-1} \ldots b_0) & x_i = 0 \end{cases}$$

Die Kosten betragen $C(mux_n) = 3n + 1$, für die Tiefe gilt $depth(mux_n) = 3$.
Kaskadierte $2:1$-Multiplexer mit n Variablen an den Auswahl- oder Adresseingängen können zudem zu einem Multiplexer mit n Adresseingängen und 2^n Dateneingängen zusammengefasst werden. Dieser entspricht einem Nur-Lese-Speicher (ROM) mit 2^n Bits,

was eine Grundstruktur der Zellen von Field Programmable Gate Arrays (FPGAs) kennzeichnet, welche programmierbare integrierte Schaltungen (ICs) darstellen.
Unter bestimmten Voraussetzungen – die nachfolgend (siehe Kapitel 2.3 ab Seite 19) genauer beschrieben werden – sind BDDs kanonisch, d. h. dass die Repräsentation einer gegebenen Funktion eindeutig ist.

2.2 Beispiele zu Darstellungen Boolescher Funktionen

Nachfolgend zeigt die Abbildung 3 nun Beispiele bekannter Boolescher Operationen $x_1 \wedge x_2$, $x_1 \vee x_2$ und $x_1 \oplus x_2$:

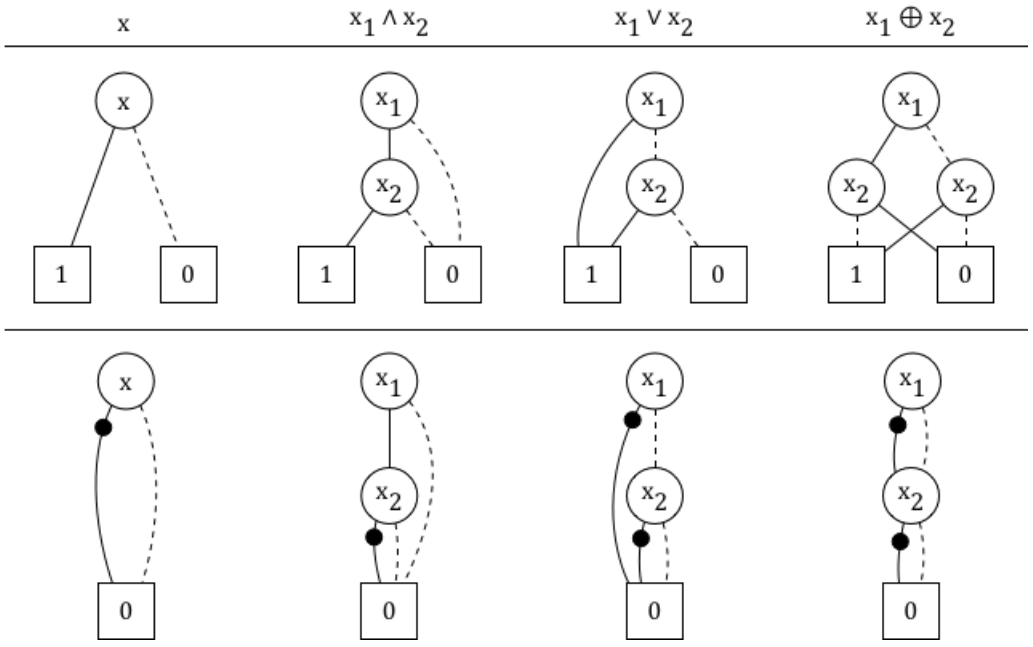

Abbildung 3: BDDs von bekannten Operationen

Hinweis: Zur Vereinfachung wurde der Startknoten f weggelassen. Eine gestrichelte Linie entspricht der Kante e, eine durchgezogene Linie kennzeichnet hingegen die Kante t. Die ausgefüllten Kreise stehen wiederum für Output-Inverter hinsichtlich komplementärer Kanten, die in Kapitel 3.6 auf Seite 57 beschrieben werden.

Um den Funktionswert zu einer gegebenen Variablenbelegung zu finden, muss (ausgehend vom Wurzelknoten) jedem Variablenknoten der t-Kante gefolgt werden, wenn die Variable den Wert 1 hat, ansonsten der e-Kante, was einen Weg kennzeichnet (siehe Kapitel 2.1.5 auf Seite 15).
Wie bereits in der Definition 2.1.6 auf Seite 16 erwähnt – was hier gut zu beobachten ist – gilt auch die Kreisfreiheit sowie der Zusammenhang, da $\exists$ ein Weg von v nach v' für je zwei Knoten $v, v' \in V$. Der erreichte Endknoten gibt dann den jeweiligen Funktionswert an. Letztendlich resultiert durch die Iteration eine sog. Wahrheitstafel (siehe

Kapitel 2.1.4 auf Seite 15), die folgende Tabelle 1 darstellt:

Tabelle 1: Wahrheitstabellen von bekannten Operationen

x_1	x_2	$x_1 \cdot x_2$	$x_1 + x_2$	$x_1 \oplus x_2$
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

Der Bitstring 00 entspricht somit z. B. dem Funktionswert 0 bei x_1x_2. Anhand eines BDDs kann also auch eine DNF bzw. KNF abgeleitet werden. Gemäß der Definition aus Kapitel 2.1.3 auf Seite 15 wäre somit bspw. die DNF von $x_1 \oplus x_2$ $(x_1 \neg x_2) + (\neg x_1 x_2)$. Die KNF wäre durch $(x_1 + x_2) \cdot (\neg x_1 \neg x_2)$ bestimmt.
Hinzuzufügen ist, dass die negierte Funktion die gleiche Struktur hat, es sind lediglich die Einsen und Nullen der Wertzuweisungen vertauscht, was in Kapitel 3 auf Seite 33 weitergehend erläutert wird.

2.3 Reduzierte geordnete BDDs (ROBDDs)

Ein Entscheidungsdiagramm (DD) heißt *frei*, wenn es auf jedem Pfad von der Wurzel zu einem Blatt jede Variable höchstens einmal als Markierung vorkommt. Es heißt wiederum *geordnet*, wenn auf jedem Pfad von der Wurzel zu einem Blatt die Variablen in der gleichen Reihenfolge abgefragt werden:

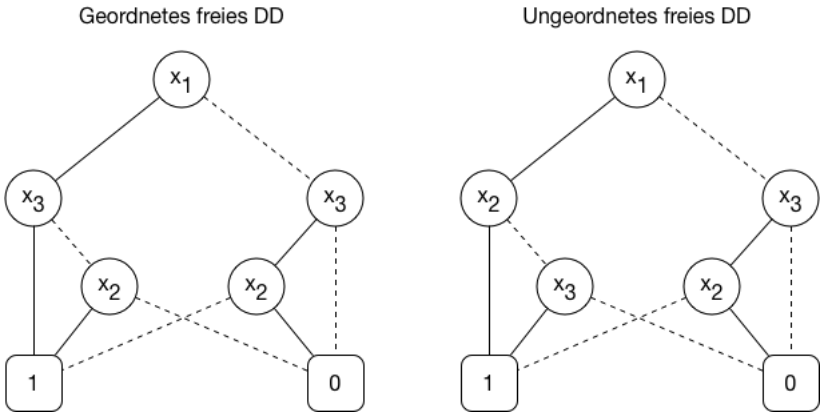

Abbildung 4: Geordnetes und ungeordnetes freies DD

Bei beiden Diagrammen aus der Abbildung 4 geht hervor, dass sie frei sind, da auf jedem Pfad jede Variable höchstens einmal vorkommt. Das zweite Diagramm ist allerdings ungeordnet, da dort die Variablen nicht in der gleichen Reihenfolge abgefragt werden.
Eine Boolesche Funktion ist von n Variablen abhängig, was für ein BDD bedeutet, dass er mindestens $2^{n+1} - 1$ Knoten hat. Wie bereits in der Einleitung auf Seite 9

erwähnt, sollen Informationen aber kompakt dargestellt werden. Es kann vorkommen, dass sich beim Erstellen eines BDDs identische Kofaktoren ergeben, d. h. es gibt zwei interne Knoten, die dieselbe Funktion repräsentieren. Es kann aber auch passieren, dass redundante Knoten auftreten. Ein Knoten ist genau dann redundant, wenn seine t- als auch seine e-Kante auf denselben nächsten Knoten zeigen. Ein BDD heißt also *reduziert*, wenn es keinen nichtterminalen Knoten v mit dem Index x und $(f_v)_x = (f_v)'_x$ gibt und auch keine verschiedenen Knoten v, w existieren, die gleich markiert sowie deren Kinder (falls vorhanden) jeweils identisch sind. Sowohl die Isomorphie- als auch die Redundanz-Regel sind in der Abbildung 5 ersichtlich:

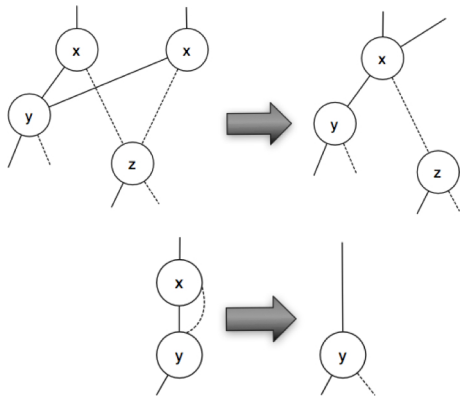

Abbildung 5: Reduktionsregeln bei BDDs

Durch die wiederholte Verschmelzung von isomorphen Teilgraphen und Entfernen redundanter Knoten entsteht letztendlich ein reduziertes BDD, was anhand der Abbildung 6 demonstriert wird:

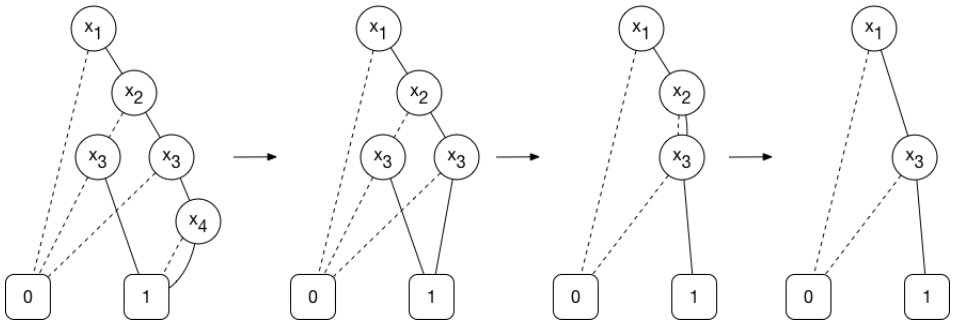

Abbildung 6: Anwendung der Reduktionsregeln bei einem BDD

Hier gehen besonders die Abhängigkeiten der Regeln hervor. So ist gelegentlich erst eine Regel nach vorherigen Reduktionsschritten anwendbar. Insgesamt gesehen genügt es, solange Reduktionsregeln anzuwenden, bis keine Regel mehr anwendbar ist. Es sollte jedoch verhindert werden, dass zu häufig für einen Knoten geprüft wird, ob eine Regel angewendet werden kann. Es empfiehlt sich hier ein rekursiver Algorithmus, da so für Knoten endgültig entschieden werden kann, ob sie gelöscht oder mit einem anderen Knoten verschmolzen werden. Diese Frage stellt sich genau dann, wenn auf die Nachfolger keine Regel mehr angewendet werden kann [Art10]:

```
program reduce()
  input: OBDD G in Bezug auf f mit der Ordnung x[1], ..., x[n]
  output: ROBDD G'
  v := Innerer Knoten von g
  l0 := 0-Blatt
  l1 := 1-Blatt
  v.id := Nummer von v
  id := Nummer
  v.i := Index der Variablen, mit der v markiert ist
  v.low := low-Kind
  v.high := high-Kind
  vec := Liste, welcher Knoten die weiteren Nummern darstellt
  x := Markierte Knoten
  marked := Liste mit markierten Knoten
  foreach (G) do
    merge(l0.all)
    merge(l1.all)
    l0.id := 0
    vec[l0.id] := l0
    l1.id := 1
    vec[l1.id] := l1
    marked := (x[1], ..., x[n])
  end foreach
  id := 1
  for (i := n to 1) do
    while (marked) do
      if (v.low.id = v.high.id) then
        v.id := v.low.id
        marked.del(v)
      end if
    end while
    // Key von v ist (v.low.id, v.high.id)
    sort_lex(marked)
    // Verschmelze hintereinander stehende Knoten
    foreach (marked) do
      id := id + 1
      v[1].id := id
      vec[v[1].id] := v[1]
      v[1].0 := vec[v[1].low.id]
      v[1].1 := vec[v[1].high.id]
      v[2].id := v[1].id
      ...
      v[l].id := v[1].id
    end foreach
  end for
end program
```

Code 1: Implementierung von `reduce`

Gemäß dem Code 1 können Knoten des OBDDs nicht direkt gelöscht werden, da es keine rückwärts gerichteten Zeiger gibt. Wenn also ein Knoten gelöscht werden soll, so müssen die Zeiger von v auf den Nachfolger davon umgesetzt werden. Wenn die Knoten v, w verschmolzen werden sollen, so müssen die Zeiger wiederum von v auf w umgelegt werden. Hier wird das Löschen derartig realisiert, indem v die id von w erhält. Aufgrund dessen, dass nun mehrere Knoten mit derselben Nummer existieren können, wird in einem Vektor vec gespeichert, welcher Knoten mit der jeweiligen Nummer die Übrigen repräsentiert. Hervorzuheben – in Bezug auf den Redundanztest – ist, dass lediglich eine Traversierung durch das OBDD ausreicht, was durch die Eigenschaften von ROBDDs bedingt ist [Sie07, S.44]. Wenn G keinen mit x_i markierten Knoten besitzt, so wird für $x_i = 0, x_i = 1$ derselbe Funktionswert berechnet. Demnach hängt f aber nicht maßgeblich von x_i ab, was automatisch auch für alle Subfunktionen gilt. Das ROBDD enthält somit auch keinen mit x_i markierten Knoten. Sollte f im Umkehrschluss maßgeblich von x_i abhängen, so gilt $f_{x_i=0} \neq f_{x_i=1}$ und das ROBDD enthält einen mit x_i markierten Knoten.
Bei Implementierungen (siehe Kapitel 3.3.1 ab Seite 38) wird direkt bei der Synthese darauf geachtet, dass Reduktionen unmittelbar bei der Erstellung von Knoten angewendet werden können, um mehrfache Traversierungen zu verhindern.
Alle Operationen – bis auf das Sortieren – benötigen pro Knoten nur eine konstante Zeit, d. h. $O(1)$. Für das Sortieren kann Bucketsort verwendet werden, womit in der Liste dann Knoten mit den gleichen Nachfolgern gefunden werden können [Art10]. Bucketsort ist daher ein Sortierverfahren, das für bestimmte Werte-Verteilungen einer Eingabeliste wie folgt sortiert:

1. Die Elemente werden auf die „Buckets“ verteilt (Partitionierung).
2. Jeder „Bucket“ wird mit einem weiteren Sortierverfahren sortiert.
3. Der Inhalt der sortierten „Buckets“ wird konkateniert.

Die Ausgabedaten werden also gesondert gespeichert, womit die Eingabedaten nicht überschrieben werden. Der gesamte Algorithmus benötigt dementsprechend dann eine lineare Rechenzeit.
Nach dem Satz von Bryant sind ROBDDs kanonische Darstellungen Boolescher Funktionen [Bry86, S.677-691]. Als Beispiel soll hierzu im Folgenden die Funktion
$f(x_1, x_2, x_3, x_4) = \neg x_1 x_2 + x_2(x_3 + x_4)$ mit der Variablenordnung $\pi : x_1 < x_2 < x_3 < x_4$ dienen, die aus der Abbildung 7 hervorgeht. Die jeweilige Subfunktion infolge der Shannon-Zerlegung (siehe Kapitel 2.1.6 auf Seite 16) ist an den Knoten gekennzeichnet:

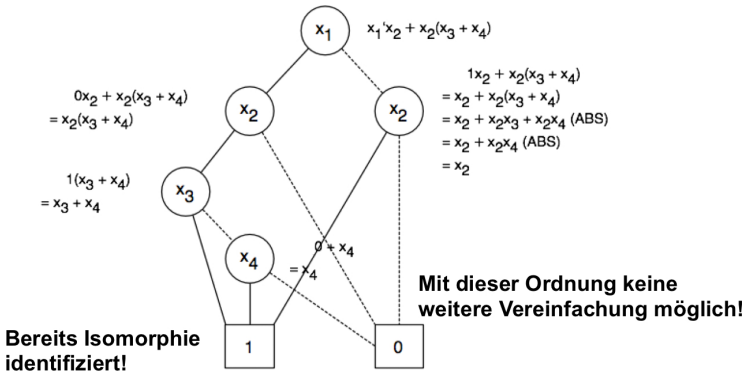

Abbildung 7: ROBDD für die Funktion $f = \neg x_1 x_2 + x_2(x_3 + x_4)$

Diese Minimalitätsaussage bezieht sich jedoch nur auf eine Funktion f mit fester Variablenordnung. Die nachfolgende Abbildung 8 zeigt die Funktion f mit einer anderen Variablenordnung $\pi : x_1 < x_3 < x_4 < x_2$:

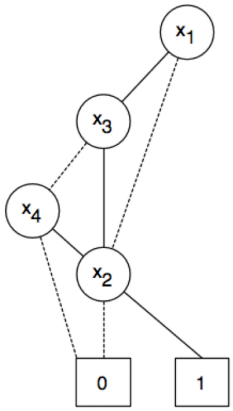

Abbildung 8: ROBDD für f mit einer anderen Ordnung

Die Größe hängt insgesamt von der Variablenordnung ab, die exponentiell sein kann [Pfe10]. Zur Verdeutlichung dieses Sachverhalts wird die Funktion $f \in \mathbb{B}_{2n}$ mit $f(x_1, x_2, x_3, x_4, \ldots, x_{2n-1}, x_{2n}) = (x_1 + x_2)(x_3 + x_4)\ldots(x_{2n-1} + x_{2n})$, wobei $n = 3$ angenommen. Das dazugehörige ROBDD ist der Abbildung 9 auf Seite 24 zu entnehmen:

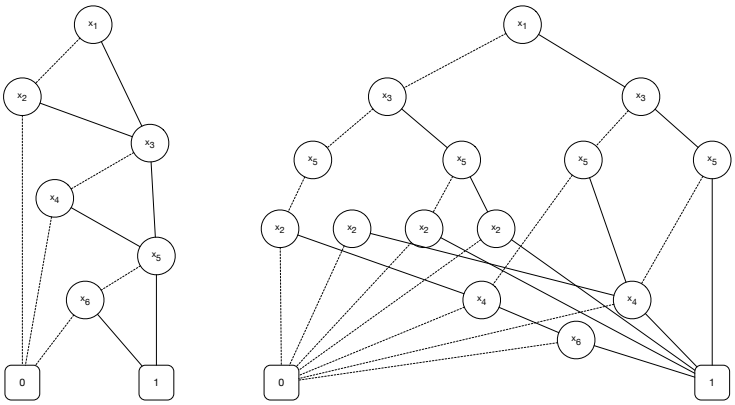

Abbildung 9: Lineares und exponentielles Wachstum von Knoten

Das linke ROBDD besitzt die Variablenordnung $\pi : x_1 < x_2 < x_3 < x_4 < x_5 < x_6$. Es ist zu erkennen, dass es sich hier um ein lineares Wachstum der Knoten in Bezug auf die Variablenanzahl handelt, da es $2n + 2$ Knoten gibt. Wird hingegen die Variablenordnung $\pi : x_1 < x_3 < x_5 < x_2 < x_4 < x_6$ gewählt, so ist ein exponentielles Wachstum ersichtlich. Das ROBDD bläht sich hierzu bis zur Tiefe n zu einem vollständig binären Baum auf, wobei jeder Knoten die Informationen zu allen vorherigen Variablenbelegungen besitzt. Somit können diese Knoten also nicht zusammengefasst werden und es gibt 2^{n+1} Knoten.
Eine gute Ordnung zu finden ist nicht immer leicht. So benötigen nahezu alle Booleschen Funktionen sogar mit einer optimalen Variablenordnung mindestens $\frac{2^n}{2n}$ Knoten [LL92]. In der Praxis finden jedoch hauptsächlich Boolesche Funktionen eine Anwendung, die relativ einfach aufgebaut sind. Somit gelingt es i. d. R. eine optimale Ordnung zu finden, sodass eine Darstellung durch ein ROBDD effizient möglich ist [Gün02]. Insgesamt betrachtet ist das Problem, eine optimale Variablenordnung zu finden, jedoch NP-vollständig, kann allerdings mit verschiedenen Heuristiken in Polyzeit behandelt werden [BW99].
Für viele Funktionen wie z. B. XOR liefern BDDs jedoch eine kompakte Repräsentation, selbst wenn sie mittels KNF/DNF nicht kompakt darstellbar sind. Es kann zudem in konstanter Zeit geprüft werden, ob eine Funktion (dargestellt durch ein BDD) erfüllbar (SAT), d. h. $sat(f) \Leftrightarrow \exists a \in B^n$, sodass $f(a) = 1$ für $f \in B_{n,1}$ (siehe Kapitel 2.4.3 auf Seite 28) oder eine Tautologie (siehe Kapitel 2.4.5 auf Seite 31) ist [BMM01]. Im Allgemeinen ist SAT nach dem Satz von Cook und Levin NP-vollständig, wobei die Laufzeit exponentiell ist, da alle 2^n Belegungen systematisch erzeugt werden müssen [KM07].

2.4 Operationen auf BDDs

Wie bereits in Kapitel 2 auf Seite 11 erwähnt, werden viele Anforderungen an Boolesche Funktionen gestellt.

Wenn Boolesche Funktionen realisiert bzw. analysiert werden, so wird auf Datenstrukturen gearbeitet, die als solche Funktionen interpretiert werden können. Mit ROBDDs sollen diesbezüglich praktisch wichtige Boolesche Funktionen effizient dargestellt und Berechnungen darauf ausgeführt werden. Besonders sollen effiziente Berechnungen ermöglicht werden, die mit anderen Darstellungen von Booleschen Funktionen wegen dem Bedarf der Rechenzeit und des Speicherplatzes nicht möglich sind. So gibt es bspw. den Äquivalenztest oder die Synthese, worunter ein Entscheidungs- bzw. Berechnungsproblem verstanden wird. Mithilfe von BDDs sind solche – im Gegensatz zu Formeln oder Schaltkreisen – effizient lösbar. Es sollen daher im Folgenden wichtige Operationen in ihrem Verfahren und ihrer Komplexität (siehe Kapitel 2.1 auf Seite 11) untersucht werden. So gibt es bspw. die Booleschen Operatoren wie `and`, `or` und `not`, die bezüglich ROBDDs in der Laufzeit polynomiell beschränkt sind, was in Kapitel 3.3.1 auf Seite 38 gezeigt wird. Es können jedoch zu anderen Darstellungsformen Unterschiede festgestellt werden, die in der Tabelle 2 erläutert sind [Mol07]:

Tabelle 2: Vergleich der operationellen Komplexitäten für Darstellungen

Darstellung	**Beschreibung**
Wahrheitstafel	Die Laufzeit ist proportional zu 2^n, d. h. linear in der Größe zur Wahrheitstafel.
Boolesche Ausdrücke	Hierzu können die beiden Operatorbäume (siehe Kapitel 2.1.2 auf Seite 13) kopiert werden. Anschließend erfolgt eine Verknüpfung über eine neue Wurzel. Insgesamt besteht eine polynomielle Laufzeit.
DNF	Die Or-Operation wird durch eine Konkatenation der DNFs umgesetzt, die And-Operation hingegen durch das Ausmultiplizieren davon. Diese Operation ist wie auch die Not-Operation, die durch Anwendung der Regel von De Morgan sowie anschließendem Ausmultiplizieren der Klauseln umgesetzt wird, recht teuer. Insgesamt besteht dennoch eine polynomielle Laufzeit.
KNF	Hierbei sind die Or- und Not-Operation recht teuer. Die OR-Operation wird durch das Ausmultiplizieren der KNFs umgesetzt, die NOT-Operation hingegen durch Anwendung der Regel von De Morgan sowie anschließendem Ausmultiplizieren der Monome. Bei der And-Operation erfolgt lediglich eine Konkatenation der KNFs. Alle aufgeführten Operationen sind dabei polynomiell beschränkt.
Schaltkreise	Die operationellen Komplexitäten entsprechen den Booleschen Ausdrücken.

Auch Umwandlungen wie z. B. die Überführung einer KNF zu einer DNF kann die Formel exponentiell vergrößern, was nicht verhindert werden kann [Sch10]. Im Hinblick auf die verschiedenen Darstellungen gibt es im Laufzeitverhalten also bereits bei den Basisoperationen erhebliche Unterschiede. Weiterhin soll daher auch bezüglich weiterer

nützlicher Operationen ein Vergleich zu anderen Darstellungen für Boolesche Funktionen gemacht werden.

2.4.1 Äquivalenztest

Hardware- und Softwaresysteme finden eine immer größer werdende Verbreitung wie z. B. in eingebetteten Systemen wie Handys oder Autos. Zudem werden Systeme immer komplexer, was z. B. mit der Betriebssystementwicklung zusammenhängt [Bau08]. Aufgrund dessen, dass Fehler hohe Kosten verursachen, hat die Verifikation von Hardware- und Softwaresystemen eine enorme Bedeutung.
BDDs eignen sich – wie bereits in Kapitel 2 auf Seite 11 erwähnt – zur Verifikation von Schaltungen. Eine zentrale Aufgabe dabei ist der sog. Äquivalenztest. Angenommen, es gibt zwei Schaltkreise, die in der Abbildung 10 präsentiert werden:

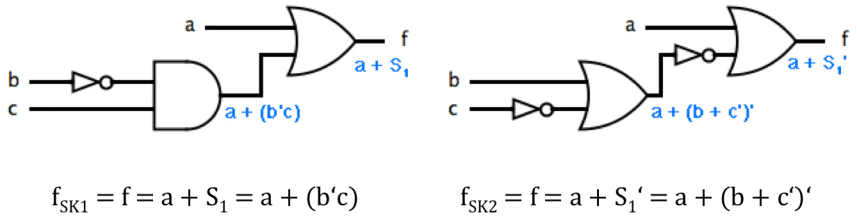

Abbildung 10: Äquivalenz von Schaltkreisen

Mithilfe eines ROBDDs kann nun eine Aussage zur möglichen Äquivalenz dieser getroffen werden, indem es für die jeweiligen Schaltkreise erstellt wird. Die daraus resultierenden ROBDDs sind in Abbildung 11 ersichtlich:

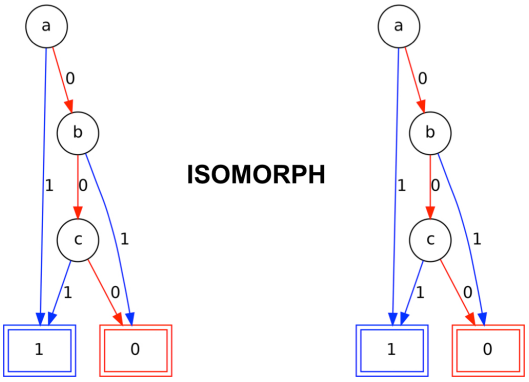

Abbildung 11: Äquivalenztest mithilfe von BDDs

Demnach sind beide Schaltkreise isomorph zueinander. Durch diesen Test können also unter anderem Schaltungen vereinfacht werden. Allgemein formuliert gilt also:

Eingabe:	OBDDs G_a und G_b, wobei $a, b \in \mathbb{B}_n$.
Ausgabe:	`JA`, falls $a = b$, `NEIN` sonst.
Verfahren:	Es werden zunächst einmal G_a und G_b zu G'_a und G'_b reduziert. Anschließend werden G'_a und G'_b auf Isomorphie getestet.
Komplexität:	$O(\|G_a\| + \|G_b\|)$, da die Reduktion sowie der Isomorphietest in linearer Zeit möglich sind, weil OBDDs Graphen mit einer Quelle und markierten Knoten sowie Kanten sind (siehe Kapitel 2.3 auf Seite 19). So können G'_a und G'_b simultan mit einer DFS durchlaufen werden, wobei getestet werden muss, ob die erreichten Knoten die gleiche Markierung haben.

Wenn a und b gemeinsam in einem reduzierten OBDD dargestellt werden bzw. in derselben Unique-Table (siehe Kapitel 3.4 auf Seite 47), so ist der Äquivalenztest in konstanter Zeit möglich. Es genügt hierbei der Test, ob die Zeiger für a und b auf denselben Knoten zeigen [MS99, S.66].
Bei Wahrheitstafeln hingegen ist die Laufzeit proportional zu 2^n, d. h. linear in der Größe der Funktionstafeln. Bezüglich Booleschen Ausdrücken, der KNF/DNF und Schaltkreisen sind hierfür keine effizienten Verfahren bekannt.[3]

2.4.2 Auswertungsproblem

Allgemein betrachtet ist hierzu eine Formel sowie eine Belegung gegeben und es stellt sich die Frage, ob diese Belegung zu einem Funktionswert von 1 am Ausgang führt. Bezogen auf OBDDs gilt:

Eingabe:	OBDD G für $f \in \mathbb{B}_n$, eine Belegung $x \in \{0,1\}^n$.
Ausgabe:	`JA`, falls $f(x) = 1$, `NEIN` sonst.
Verfahren:	Das OBDD wird von der Wurzel bis zu einem Blatt gemäß der Belegung durchlaufen. Anschließend wird die Markierung des erreichten Blattes ausgegeben. Dieser Vorgang wird auch als *Evaluation* bezeichnet.
Komplexität:	$O(n)$, da jede Variable höchstens einmal getestet werden darf. Bezüglich des Speicherplatzes gilt natürlich dennoch $O(\|G\|)$ für den dazugehörigen Aufbau.

Hinzuzufügen ist, dass bei diesem Test nicht die Möglichkeit besteht, mithilfe eines ROBDDs für eine konstante Laufzeit zu sorgen, was durch folgende Abbildung 12 auf Seite 28 verdeutlicht werden soll:

[3] Molitor, vgl. [Mol07]

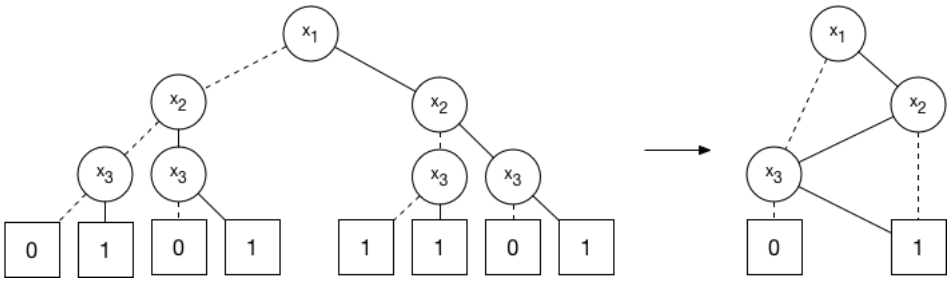

Abbildung 12: BDDs zur Verdeutlichung des Auswertungsproblems

Zu erkennen ist ein OBDD sowie ROBDD für die Boolesche Funktion $f(x_1, x_2, x_3) = x_1 \cdot \neg x_2 + x_3$, wobei $f \in \mathbb{B}_3$. Angenommen, die Belegung sei $x = (1, 1, 1)$, so würde die Funktion $1 \cdot 0 + 1$ zu 1 auswerten. Bezüglich dem OBDD ist dies zu erkennen, indem gemäß des Verfahrens den Kanten entlang gelaufen wird. Hinsichtlich dem dazugehörigen ROBDD kann jedoch nicht anhand der Wurzel abgesehen werden, wie die Auswertung anhand der jeweiligen Belegung sein wird, da weiterhin die Variablen gemäß der Belegung abgelaufen werden müssen.
Im Vergleich zu den anderen Darstellungen für Boolesche Funktionen gilt ebenfalls ein linearer Aufwand, weil die Anzahl der Aufrufe zur Auswertung durch die Größe der Formel beschränkt ist [Mol07].

2.4.3 Erfüllbarkeit

In Kapitel 2.3 auf Seite 19 wurde erwähnt, dass durch die Kanonizität eines OBDDs der Erfüllbarkeitstest in konstanter Zeit möglich ist, was im Folgenden algorithmisch analysiert werden soll. Ein BDD ist erfüllbar, gdw. es nicht aus dem 0-Blatt besteht:

Eingabe:	OBDD G für $f \in \mathbb{B}_n$.
Ausgabe:	`JA`, falls $\exists$ eine Belegung $x \in \mathbb{B}^n : f_G(x) = 1$, `NEIN` sonst.
Verfahren:	Das OBDD wird von der Quelle mit einer DFS durchlaufen, bis ein 1-Blatt erreicht wird. Der dazu gefundene Pfad gibt eine Eingabe x mit $f(x) = 1$ an. Ansonsten gibt es keine derartige Eingabe.
Komplexität:	$O(\|G\|)$, da im schlechtesten Fall jeder Knoten und jede Kante betrachtet werden muss.

Ein dementsprechender Algorithmus 2 gemäß der Beschreibung des Verfahrens kann wie folgt aussehen, wobei die Variablenordnung $\pi : x_1, \ldots, x_n$ angenommen wird:

```
1 program sat()
2   input: G (in Form eines OBDDs)
3   output: Status, ob es einen Funktionswert '1' gibt
4   if ( isTerminal(G) ) then
5     return l(G)
6   end if
7   count := sat( low(G) )
8   if (count != 0) then
```

```
9       return !l(G) * count
10    end if
11    return l(G) * sat( high(G) )
12  end program
```

Code 2: Implementierung von `sat`

Hierbei steht $l(G)$ für die Markierung des jeweiligen Knotens. Falls sich das low-Kind als 0-Blatt herausstellt, so wird eine Traversierung (ausgehend von der Wurzel zu den Blättern) zum high-Kind eines Knotens gemacht. In der Tiefe des Graphen entspricht dies einer Konjunktion von Literalen und beträgt $O(n)$.
Sollte G jedoch reduziert sein, so wird der Test dahingehend modifiziert, dass geprüft wird, ob die Quelle von $G \neq$ dem 0-Blatt ist. Wird damit zusammenhängend die Abbildung 2.4.2 auf Seite 27 betrachtet, so kann festgestellt werden, dass bei dem ROBDD die Quelle nicht dem 0-Blatt entspricht. Daher ist die dazugehörige Funktion erfüllbar. Dieser adaptierte Test ist – im Gegensatz zum Auswertungsproblem – also in konstanter Zeit möglich [Sie07, S.37].
Im Hinblick auf Schaltkreise, allgemeine Boolesche Ausdrücke sowie KNFs sind keine effiziente Verfahren bekannt, um SAT zu lösen [Mol07]. Dagegen kann in Form einer DNF dieses Problem ebenfalls in Polyzeit gelöst werden, da eine Überprüfung dahingehend ausreicht, ob mindestens ein Monom (siehe Kapitel 2.1.2 auf Seite 13) existiert. Weiterhin besteht bei Wahrheitstafeln eine Laufzeit, die proportional zu 2^n, d. h. linear in der Größe der Wahrheitstafeln ist.

2.4.4 Anzahl erfüllender Belegungen

Ähnlich zum Auswertungsproblem in Kapitel 2.4.2 auf Seite 27 ist dieses Problem auch in linearer Zeit – unabhängig ob ein OBDD/ROBDD vorliegt – berechenbar [Sie07, S.37-38]. Im Endeffekt geht es darum, wie viele erfüllende Belegungen es gibt, weshalb ein Berechnungsproblem vorliegt:

Eingabe:	OBDD G für $f \in \mathbb{B}_n$.
Ausgabe:	Die Berechnung von $\lvert f_G^{-1}(1)\rvert$.
Verfahren:	Markiere zunächst die Quelle mit 2^n und die verbliebenen Knoten mit -1. Durchlaufe nun G in der vorliegenden Ordnung. Sobald ein Knoten v erreicht wird, der mit einer Eingabe in markiert ist, so wird zu den Markierungen beider Nachfolger der Wert $\frac{in}{2}$ addiert. Letztendlich wird das Gesamtergebnis durch die Summe der Markierungen der 1-Blätter bestimmt.
Komplexität:	$O(\lvert G\rvert)$, da jeder Knoten und jede Kante betrachtet werden muss.

Im Endeffekt soll für jede Kante bzw. jeden Knoten berechnet werden, für wie viele Eingaben sie durchlaufen werden. Die Quelle wird für 2^n Eingaben traversiert, die 1-

Blätter hingegen für $|f^{-1}(1)|$ Eingaben. Sei ein innerer Knoten mit v bezeichnet, der mit einer Variable x_i markiert ist. Zudem steht *in* für die Eingaben, wofür v durchlaufen wird. Zunächst werden immer jeweils die „Kinder“ von v betrachtet, anschließend werden die Ergebnisse zusammengezählt und nach oben propagiert. Somit wird x_i nicht vor v getestet, was der folgende Algorithmus 3 zeigt:

```
1  program satCount()
2    input: G (in Form eines OBDDs)
3    output: Anzahl der Belegungen zum Wert '1'
4    if ( isTerminal(G) ) then
5      return l(G) * pow(2, n)
6    end if
7    e := satCount( low(G) )
8    t := satCount( high(G) )
9    return (e + t) / 2
10 end program
```

Code 3: Implementierung von `satCount`

Wenn G ein OBDD-Knoten mit dem Variablenindex `index(G)` ist, dann gibt es zwei Mengen von Zuweisungen, die für eine erfüllende Belegung sorgen. Sollte demnach bspw. `index(G) = 0` gelten, so wird die Anzahl derartig ermittelt, indem deren Zuweisungen `count(low(G))` gefunden werden, die `low(G)` wahr werden lassen. Analog gilt dies ebenfalls für die Menge `index(G) = 1`. Folgende Abbildung 13 verdeutlicht diesen Sachverhalt:

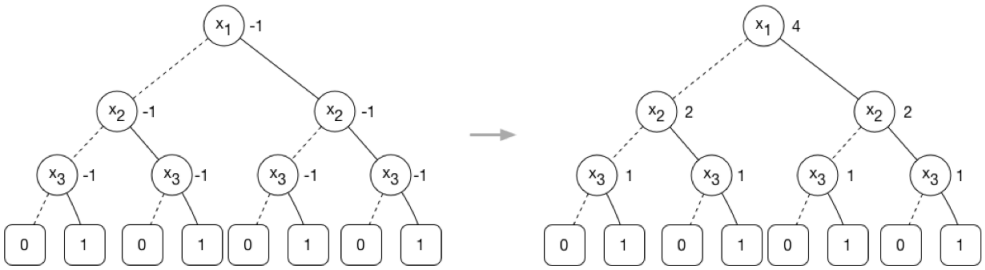

Abbildung 13: BDD zur Verdeutlichung der Ermittlung von erfüllenden Belegungen

Dementsprechend gibt es hierbei vier erfüllende Belegungen für G. Hervorzuheben ist, dass – wenn ein ROBDD vorliegen würde – natürlich i. d. R. weniger Kanten passiert werden müssen. Dennoch bleibt die Laufzeit linear in der Größe des BDDs.
Bei anderen Darstellungen für Boolesche Funktionen – bis auf Wahrheitstafeln – ist die Laufzeit nicht polynomiell beschränkt [Mol07]. Bei Wahrheitstafeln gilt ebenfalls eine lineare Laufzeit, wobei hervorzuheben ist, dass sich dort die Platzkomplexität in Grenzen halten muss. So würden bspw. 100 Variablen in 2^{100} Zeilen resultieren, weshalb die Variablenanzahl n nicht zu groß sein darf.

2.4.5 Tautologietest

In der Einleitung auf Seite 9 wurde erwähnt, dass der Tautologietest mithilfe eines ROBDD in konstanter Zeit möglich ist. Im Folgenden soll dieses Entscheidungsproblem ebenfalls algorithmisch analysiert werden. Ein BDD ist eine Tautologie, gdw. es nur aus einem 1-Blatt besteht:

Eingabe:	OBDD G für $f \in \mathbb{B}_n$.
Ausgabe:	`JA`, falls f von jeder Belegung erfüllt wird (gültig), `NEIN` sonst.
Verfahren:	Es müssen alle Pfade gesucht werden, die von der Wurzel zu einem 1-Blatt führen. Diese können mithilfe von einem rekursiven Algorithmus berechnet werden, der bei der Wurzel startet. Für jeden inneren Knoten v wird dieselbe Methode für beide Nachfolger sukzessive aufgerufen. Wird ein 1-Blatt erreicht, so wird der Pfad rekonstruiert und die getesteten Variablen derartig belegt, dass dieser Pfad durchlaufen wird.
Komplexität:	$O(\|G\|)$, da jeder Knoten und jede Kante betrachtet werden muss.

Eine Implementierung für das Verfahren kann in Form von Code 4 folgendermaßen aussehen:

```
1  program checkTautology()
2    input: G (in Form eines OBDDs)
3    output: Status, ob alle Belegungen zum Wert '1' auswerten
4    x := Variablen
5    w := Alle Belegungen zu '1'
6    if (G = 0) then
7      return []
8    else if (G = 1) then
9      return [{}]
10   else
11     return [{checkTautology(low(G)).add(w.front((map(x[G], 0))))},
12       {checkTautology(high(G)).add(w.front((map(x[G], 1))))}]
13 end program
```

Code 4: Implementierung von `checkTautology`

Es wird eine Variablenordnung von $\pi : x_1, \ldots, x_n$ angenommen. Eine Rückgabe `{[]}` würde z. B. bedeuten, dass eine Sequenz mit einer leeren Belegung zur Erfüllung einer Formel zurückgegeben wird.

Wenn G jedoch reduziert ist, so muss lediglich geprüft werden, ob die Wurzel von G das 1-Blatt ist. Dies ist trivial und daher in konstanter Laufzeit möglich [Sie07, S.38]. Ein Beispiel hierfür wäre die Boolesche Funktion $f(x_1, x_2, x_3) = x_1 + x_2 + \neg x_1 + x_3$, wobei $f \in \mathbb{B}_3$. Offensichtlich ist diese Funktion eine Tautologie, da das ROBDD lediglich das 1-Blatt beinhaltet.

Im Hinblick auf die Implementierung (siehe Kapitel 3.3.1 auf Seite 38) können durch

diese Operation weiterhin zusätzliche ITE-Aufrufe gespart werden, was dort genauer erläutert wird.
Im Vergleich zu anderen Darstellungen bleibt zu erwähnen, dass der Tautologietest für Schaltkreise, Boolesche Ausdrücke sowie die DNF nur in exponentieller Zeit entscheidbar ist [Mol07]. So müssen z. B. für Boolesche Ausdrücke alle 2^n Belegungen systematisch erzeugt werden. Für die KNF hingegen ist dieser Test in polynomieller Laufzeit entscheidbar, da diese genau dann nicht erfüllbar ist, wenn sie mindestens eine Klausel enthält.

3 Implementierung von ROBDDs

Dieses Kapitel handelt von den Implementierungsdetails des ROBDD-Paketes. Dementsprechend werden die einzelnen Bestandteile wie Knoten oder die Synthese sowie die damit zusammenhängenden Datenstrukturen und Operatoren erläutert.
Das Hauptproblem von ROBDDs ist insbesondere der Speicherplatz, wobei das Auslagern von Hauptspeicher langsam ist, was wiederum die Effizienz bei Berechnungen verringert [Sie07, S.37-45]. Es ist daher unausweichlich, besonders in diesen Bereichen selbst kleinere Optimierungen vorzunehmen, da dies bereits darüber entscheiden kann, ob eine bestimmte Berechnung vollendet wird. In diesem Zusammenhang gilt es auch zu beachten, dass verschiedene Operationen nacheinander ausgeführt werden und die Zwischenergebnisse sehr groß sein können. Ein Beispiel hierfür liefert die Berechnung eines ROBDDs für eine Funktion, die durch einen Schaltkreis beschrieben wird, weil dazu für jeden Baustein eine Synthese ausgeführt wird (siehe Kapitel 3.3.1 auf Seite 38). Es ist dabei nicht möglich, für einzelne Syntheseoperationen obere Schranken für die Rechenzeit anzugeben bzw. ist auch die Umformung eines Schaltkreises in ein ROBDD ein NP-hartes Problem [Sie07, S.46].
Im Folgenden wird daher die Beschreibung einer effizienten generischen Implementierung eines ROBDD-Paketes gegeben, das zudem bestimmte Eigenschaften von Rechnerarchitekturen ausnutzt. So wurde bspw. in Kapitel 2.4 auf Seite 24 erwähnt, dass die Basisoperationen von ROBDDs polynomiell beschränkt sind. Allgemein formuliert kann diesbezüglich die Negation derartig realisiert werden, dass die Blätter $0, 1$ vertauscht werden, was eine explizite Durchführung kennzeichnet und insgesamt eine lineare Laufzeit aufweist, was unmittelbar aus $not(f) = ite(f, 0, 1)$ (siehe Tabelle 4 auf Seite 39) folgt [Gün02]. Konkretisiert kann die Operation `not` durch eine geschickte Implementierung auch in konstanter Zeit realisiert werden, indem festgelegt wird, in welcher Art und Weise die Markierung der Blätter zu interpretieren ist [MIY90]. Dieses Konzept der sog. komplementären Kanten wird in Kapitel 3.6 auf Seite 57 vorgestellt. Diese Technik und viele weitere Konzepte wie z. B. die Synthese sind in den Bestandteilen (siehe Tabelle 3 auf Seite 34) des Paketes zu finden und werden zusammenhängend in den nächsten Kapiteln erläutert.

Tabelle 3: Bestandteile des ROBDD-Paketes

Bestandteil	**Beschreibung**
Knoten	Die Knoten dienen zur Repräsentation von OBDDs, die den elementaren Datentypen kennzeichnen. Die Datenstruktur wird bspw. zum Zusammensetzen von OBDDs benutzt und im Zusammenhang mit der Speicherbereinigung erläutert.
Manager	Kennzeichnet ein Objekt, das alle verfügbaren Operationen eines BDDs verwaltet (siehe Kapitel 3.3.1 ab Seite 38). Dies betrifft z. B. die Erstellung von Variablen, Minimierung oder Synthese, aber auch Visualisierung des jeweiligen Entscheidungsgraphen.
Unique-Table	Diese Tabelle stellt die Kanonizität des erstellten BDDs sicher und kennzeichnet eine dynamische Hashtabelle. Aus diesem Grund wird auch ein schneller Zugriff auf Knoten bzw. Kanten gewährleistet. Dieses Konstrukt wird in Kapitel 3.4 ab Seite 47 vorgestellt.
Computed-Table	Mithilfe dieser Tabelle (siehe Kapitel 3.5 ab Seite 55) werden bereits durchgeführte Operationen gespeichert. Die Tabelle wird also als Cache dafür eingesetzt, sodass diese Operationen nicht neu berechnet werden müssen. Daher ist sie maßgeblich für eine effiziente Synthese.
Speicherbereinigung	Die Knoten bzw. Kanten besitzen einen Referenzzähler und zählen dabei die eingehenden Zeiger auf sich selbst. Sollte der Zähler eines Objektes also eine Zahl > 0 halten, so wird es in dem BDD an arbiträren Stellen in Form einer Referenz benutzt. Besitzt der Zähler hingegen den Wert 0, so wird das jeweilige Objekt nicht mehr benutzt und kann gelöscht bzw. bereinigt werden. Die Inhalte der Speicherbereinigung sind in Kapitel 3.8 ab Seite 64 ersichtlich.

Hinzuzufügen ist, dass das Paket in der Programmiersprache C++ entwickelt wurde und sequenziell ist. Dementsprechend entspricht die Implementierung einem objektbasierten bzw. imperativen Paradigma, wobei einige der geschickten Implementierungstricks durch einen maschinennahen Ansatz umgesetzt sind. Der Grund hierfür liegt darin, dass die Rechenzeiten für einige Operationen somit verbessert werden konnte (siehe Kapitel 4 auf Seite 67), was sich wiederum positiv auf die Verifikation von Schaltkreisen auswirkt. Das jeweilige Zusammenspiel der Klassen sei im Folgenden über die Unified Modeling Language (UML) [IKRR10] in Form eines Klassendiagramms 14 auf Seite 35 ersichtlich:

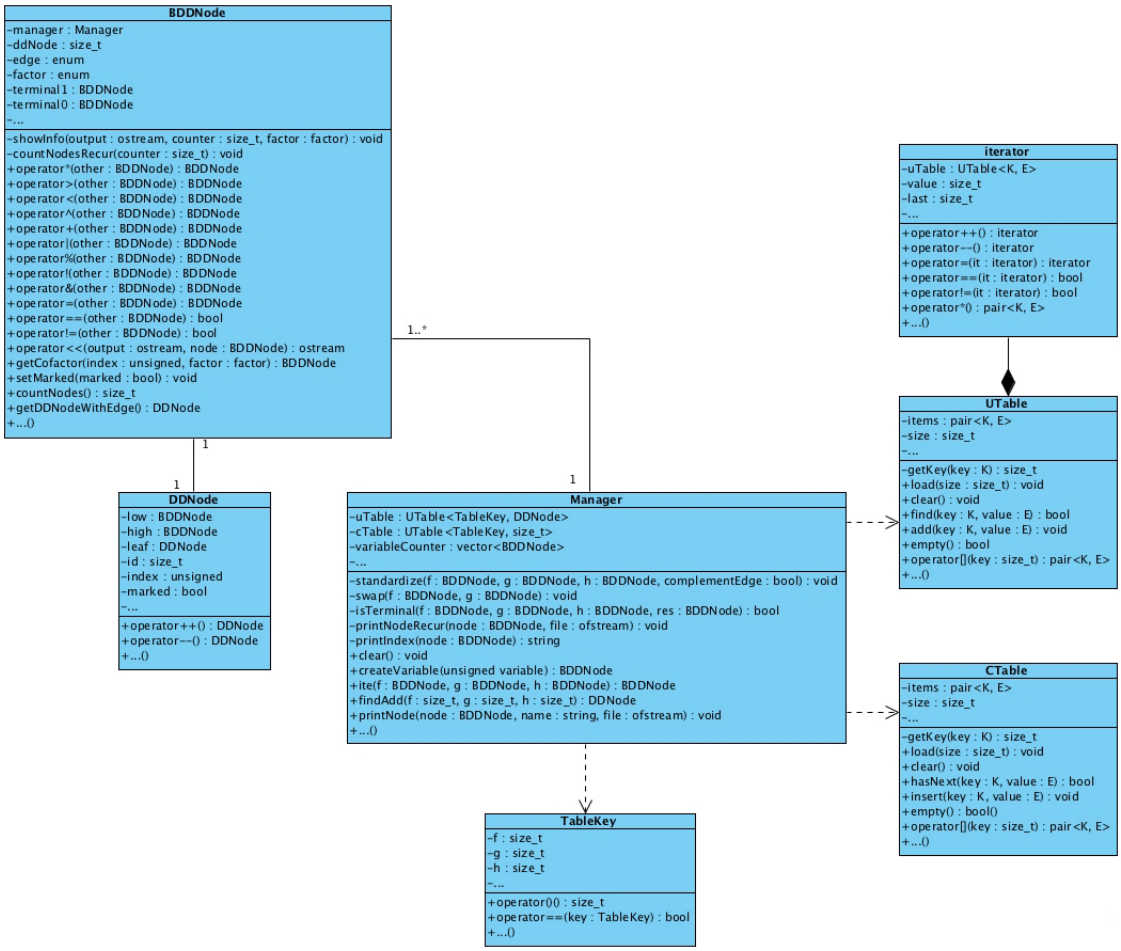

Abbildung 14: Zusammenspiel der Objekte im Paket

Hinweis: Aufgrund der besseren Übersichtlichkeit wurde auf die Aufführung der Konstruktoren, Destruktoren, Akzessoren und Kennzeichnung, ob eine Adresse oder Wert übergeben wird, verzichtet.

Der Programmablauf passiert phasenweise und die `load`-Methoden sind dazu da, die jeweiligen Objekte in einen endgültigen Zustand zu versetzen. Für die Tabellen wie die `CTable` oder `UTable` bedeutet es somit, dass dadurch Speicher allokiert wird. Anderseits sorgen die `clear`-Methoden dafür, dass der Speicher korrekt freigegeben sowie der Referenzzähler `id` der Klasse `DDNode` verringert wird und steht daher in einem unmittelbaren Zusammenhang zur beschrieben Speicherverwaltung (siehe Kapitel 3.8 auf Seite 64). Grundsätzlich greift der `Manager` also bei dessen Initialisierung auf die `CTable` sowie `UTable` zu und reserviert den angegebenen Speicher bezüglich der Knoten. Zudem werden die angegebenen Variablen in einem Vektor gespeichert, wobei die Anzahl in `variableCounter` festgehalten wird. Die damit verbundene Klasse `BDDNode` bietet Operationen wie z. B. die Konjunktion an, die zur Ausführung an den `Manager` für die Synthese weitergeleitet werden. Weiterhin besitzt der `BDDNode` wiederum aufgrund der automatischen Speicherbereinigung eine bidirektionale Beziehung zu einem

`DDNode`, der im Unterschied dazu die jeweiligen Attribute hält. Dadurch wird es ermöglicht, den Referenzzähler `id` der Klasse `DDNode` auch automatisch zu dekrementieren, weil dieser durch den `BDDNode` eingehüllt wird. An dieser Stelle sei erwähnt, dass keine sog. *Smart-Pointer* aus C++ dafür verwendet werden können, weil es keine Möglichkeit gibt, auf die Bits des Zeigers zuzugreifen. Knoten, deren `id = 0` ist, werden nicht mehr benötigt und können gelöscht werden, weshalb die innere Klasse `iterator` existiert, der eine Komposition zur Klasse `UTable` darstellt. Es können somit also Knoten in der `UTable` durchlaufen werden, um festzustellen, welche Knoten noch valide sind. Darüber hinaus wird die Klasse `TableKey` vom `Manager` dazu benutzt, um Schlüssel für deren Zugriff in der Hashtabelle über ein Modulo-Verfahren (siehe Kapitel 3.4.1 auf Seite 49) zu generieren. Es gilt dabei, dass Schlüssel (Tripel) auf Knoten in Form von Paaren abgebildet werden. Sollte bezüglich der `UTable` eine Kollision bei `add` auftreten, so wird der jeweilige Knoten in eine Überläuferliste bzw. einen gemeinsamen Slot eines Vektors abgelegt. Insgesamt basiert das Paket daher auf Indizes anstelle von Zeigern, da keine Zeiger auf Nachfolger bestehen und der Zugriff über Slots mittels `operator[]()` geregelt wird. Sollte nämlich ein 64-Bit System vorliegen, so beträgt die Größe eines Zeigers 64 anstatt 32 Bits. Um diesen Overhead bei 64-bit Rechnern zu reduzieren, können anstelle von Zeigern Integer-Indizes verwendet werden, um Knoten zu referenzieren. Letztendlich sei darauf hingewiesen, dass zur Repräsentation ganzer Zahlen der Datentyp `size_t` verwendet wird, da bei dieser Applikation mit Speichergrenzen gearbeitet wird. Somit wird garantiert, dass sowohl auf 32- als auch 64 bit Systemen genug Kapazität zur Verfügung steht, um die angeforderten Knoten zu repräsentieren. Konkret betrachtet, steht dieser Datentyp für einen primitiven Datentypen. Wenn jedoch direkt ein primitiver Datentyp verwendet werden würde, so kann die Performanz auf manchen Systemen schlechter sein. Der Grund hierfür liegt darin, dass mehr Maschineninstruktionen bestehen würden [Gar15].
Ein Beispiel für eine Programmausführung sowie dessen Ausgabe in Form einer Visualisierung ist im Anhang auf Seite 81 zu finden. Zur Darstellung der Graphen wird in dem Zusammenhang eine Beschreibungssprache für Graphen (DOT) [EKN15] benutzt. Sämtlicher Quellcode bezüglich der entwickelten abstrakten Datentypen (ADTs) – der im Zusammenhang mit den dazugehörigen erläuterten Konstrukten steht – wird in den nachfolgenden Kapiteln beschrieben und erläutert.

3.1 Gemeinsame Darstellung mehrerer Funktionen

In Kapitel 2 auf Seite 11 sind aufgrund der Einfachheit Boolesche Funktionen in unterschiedlichen BDDs repräsentiert worden. In einem BDD können jedoch auch mehrere Funktionen dargestellt werden, womit es verschiedene Startknoten gibt, was als SBDD bezeichnet wird. Hierzu soll das folgende Beispiel in Form einer Abbildung 15 diesen

Zusammenhang verdeutlichen:

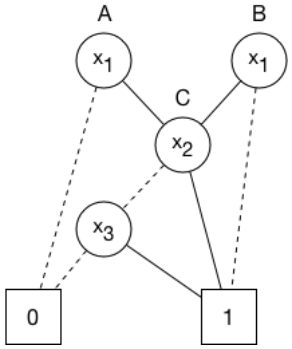

Abbildung 15: Beispiel eines SBDDs

Der Knoten A repräsentiert hier die Funktion $f_A(x_1, x_2, x_3) = x_1(x_2 + x_3)$, hingegen stellen die Knoten B und C die Funktionen $f_B(x_1, x_2, x_3) = x_1'(x_2 + x_3)$ sowie $f_C(x_2, x_3) = x_2 + x_3$ dar.
Ein SBDD ist somit eine Ansammlung von OBDD-Knoten, die dieselbe Variablenordnung besitzen. Darüber hinaus können mehrere Funktionen gemeinsame OBDD-Knoten haben. Eine Folge daraus ist, dass Speicherplatz gespart werden kann, womit dem Hauptproblem von BDDs (siehe Kapitel 3 auf Seite 33) entgegengewirkt wird [MIY90]. Zudem können aber auch mehrfach identische Berechnungen verhindert werden, was in Kapitel 3.4 ab Seite 47 genauer erläutert wird. Im weiteren Verlauf werden daher nur noch SBDDs betrachtet.

3.2 Der ROBDD-Knoten

Für die Implementierung der Knoten ist ein wichtiger Faktor, dass Anwendungen für ROBDDs gewöhnlicherweise mit Millionen von Instanzen davon zur Laufzeit arbeiten. Daher müssen Knoten so kompakt wie möglich realisiert werden.
Eine der wichtigsten Eigenschaften für die Implementierung von OBDDs ist die Eindeutigkeit in Form von ROBDDs, die in Kapitel 2.3 auf Seite 19 beschrieben ist. Damit zusammenhängend stellt jeder Knoten genau eine Boolesche Formel mit den Variablen dar, die für die Beschriftung von Subknoten zuständig sind. Wenn nun ein Knoten v in einem ROBDD die Beschriftung $l(f)$ und die „Kinder“ g, h besitzt, so wird der Knoten v mit $ite(f, g, h)$ (siehe Code 6 auf Seite 42) als eindeutig bezeichnet. Der damit verbundene Einsatz einer Unique-Table wird in Kapitel 3.4 auf Seite 47 erläutert. Der ADT eines Knotens benötigt also einen Verweis auf sein low-Kind `low` und high-Kind `high` sowie eine Variablenmarkierung `index`, die für eine Synthese benötigt wird, um zu einem Knoten aufgrund der Referenzen anderer zu gelangen.
Weiterhin ist es wichtig, einen Referenzzähler `id` einzusetzen, um sich die Anzahl der Verweise auf einen Knoten zu merken. Dadurch kann dann entschieden werden, ob ein

Knoten nicht mehr benötigt wird, um dann den dazu reservierten Speicher freizugeben (siehe Kapitel 3.8 auf Seite 64).
Um die Verwaltung der Unique-Table zu unterstützen bzw. deren Speicherbedarf zu reduzieren, ist außerdem ein Verweis zur Referenzierung eines Knotennachfolgers bezüglich der Überläuferliste notwendig. Wie bereits in Kapitel 3 auf Seite 33 erwähnt, wird dieser Mechanismus direkt in der Unique-Table realisiert, sodass sich verkettete Überläufer in einem Slot bzw. Vektor befinden. Dadurch liegen die Knoten hinsichtlich des Speichers eng beieinander und durch ihre Position wird Zeiger-Arithmetik möglich, womit ein konstanter Zugriff auf alle Felder besteht. Zudem entfällt ein Attribut im Knoten, der auf den Nachfolger zeigen würde. Dadurch werden die Unique-Table als auch der Knoten miteinander kombiniert, was in Kapitel 3.4 auf Seite 54 erläutert wird. Das zugrunde liegende Prinzip der Kollisionsbehandlung wird wiederum in Kapitel 3.4.2 auf Seite 53 erläutert.
Das Attribut `marked` wird bei der Traversierung durch den Graphen benötigt und von Methoden wie z. B. `printNode` des Managers verwendet, um das BDD zu visualisieren. Dadurch wird wiederum ausgeschlossen, dass Knoten doppelt gezählt werden.
Insgesamt gesehen, darf ein Platzbedarf von approximiert 22 Bytes angenommen werden, wenn die Anzahl möglicher Variablen auf 2^{16} begrenzt ist [Het02, S.81-82]. Dementsprechend gelten zwei Bytes für die Markierung und jeweils vier Bytes für weitere Daten, wobei die genaue Zusammenstellung ebenfalls in Kapitel 3.4.3 auf Seite 54 erläutert wird.

3.3 Synthese von ROBDDs

Eine *Synthese* umfasst die Berechnung der Darstellung einer Booleschen Funktion anhand bereits vorhandener Darstellungen. Dazu werden BDDs i. d. R. erzeugt, indem bestehende BDDs „Bottom-up“ miteinander kombiniert werden. Zu Beginn werden daher primitive BDDs angelegt (beginnend bei den Blättern), die zur Darstellung von Funktionen $f = x_i \forall x_i$, die als Entscheidungsvariablen vorkommen, dienen. Diese werden dann dementsprechend kombiniert und verknüpft, d. h. synthetisiert, um die Repräsentation der gewünschten Booleschen Funktion zu erhalten.

3.3.1 Der ITE-Operator

Der ITE-Operator ist der Kern des Paketes und stellt einen Universal-Syntheseoperator dar. Bezüglich der Namensgebung steht ITE für „If f Then g Else h“:

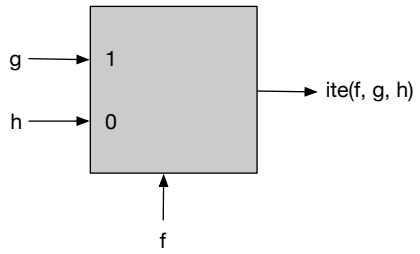

Abbildung 16: Visualisierung des ITE-Operators

Gemäß der Abbildung 16 benötigt der Operator also drei Operanden, um die Boolesche Funktion $ite(f, g, h) = f \cdot g + f' \cdot h$ berechnen zu können. Ist f eine Entscheidungsvariable, so entspricht diese Darstellung genau der Shannon-Zerlegung (siehe Kapitel 2.1.6 auf Seite 16). Boolesche Funktionen können also mittels eines Aufrufs des ITE-Operators geschrieben werden.
In einem BDD werden Boolesche Operatoren realisiert. Alle Booleschen Operatoren aus der Menge $\mathbb{B}_{\not\nu}$ und $\mathbb{B}_{\not\in}$ (bei mehreren Variablen muss explizit eine Funktionskonkatenation stattfinden) können auf diesen ternären Operator direkt zurückgeführt werden [Cas94], was die folgende Tabelle 4 verdeutlicht:

Tabelle 4: ITE Operator

Nummer	**Name**	**Ausdruck**	**Äquivalente Form**
0000	0	0	0
0001	$and(f, g)$	fg	$ite(f, g, 0)$
0010	$f > g$	$f\neg g$	$ite(f, \neg g, 0)$
0011	f	f	f
0100	$f < g$	$\neg fg$	$ite(f, 0, g)$
0101	g	g	g
0110	$xor(f, g)$	$f \oplus g$	$ite(f, \neg g, g)$
0111	$or(f, g)$	$f + g$	$ite(f, 1, g)$
1000	$nor(f, g)$	$\neg(f + g)$	$ite(f, 0, \neg g)$
1001	$xnor(f, g)$	$\neg(f \oplus g)$	$ite(f, g, \neg g)$
1010	$not(g)$	$\neg g$	$ite(g, 0, 1)$
1011	$f \geq g$	$f + \neg g$	$ite(f, 1, \neg g)$
1100	$not(f)$	$\neg f$	$ite(f, 0, 1)$
1101	$f \leq g$	$\neg f + g$	$ite(f, g, 1)$
1110	$nand(f, g)$	$\neg(fg)$	$ite(f, \neg g, 1)$
1111	1	1	1

So gilt bspw. $ite(f, g, 0) = f \cdot g + f' \cdot 0 = f \cdot g$ für die Konjunktion, wobei ein derartiger Aufruf direkt bei den Operatorüberladungen wie z. B. `operator*()` im `BDDNode` (siehe Abbildung 14 auf Seite 35) erfolgen kann.
Die Argumente für den ITE-Operator können dazu automatisch aus der rekursiven Formulierung infolge der Shannon-Zerlegung erhalten werden. Sei also v die top-Variable

von f, g, h. Dann gilt gemäß Shannon-Zerlegung und Anwendung des ITE-Operators:

$$\begin{aligned} ite(f,g,h) &= f \cdot g + f' \cdot h \\ &= v \cdot (f \cdot g + f' \cdot h)_v + v' \cdot (f \cdot g + f' \cdot h)_{v'} \\ &= v \cdot (f_v \cdot g_v + f'_v \cdot h_v) + v' \cdot (f_{v'} \cdot g_{v'} + f'_{v'} \cdot h_{v'}) \\ &= (v, ite(f_v, g_v, h_v), ite(f_{v'}, g_{v'}, h_{v'})) \end{aligned}$$

Jeder Knoten speichert also ein Tripel mit einer Variablen v, an welcher er zerlegt wurde sowie zwei „Kindern“, die sich rekursiv aus einem ITE-Aufruf der Kofaktoren ergeben. Die Knoten werden also durch (f, g, h) eindeutig bezeichnet. Die Kofaktoren von f, g, h nach v können dazu weiterhin bestimmt werden. Sei dazu f ein Knoten, wobei $f = (r, t, e), v \leq r$ ist. Dann gilt:

$$f_v = \begin{cases} f & \text{für } v < r \\ t & \text{für } v = r \end{cases}$$

$$f_{v'} = \begin{cases} f & \text{für } v < r \\ e & \text{für } v = r \end{cases}$$

Sollte $v < u$ gelten, so kommt die Variable v demnach nicht in f vor, da v die top-Variable von f ist. Folgender Code 5 verdeutlicht die Berechnung des Kofaktors:

```
1  program getCofactor()
2    input: OBDD f, Variable index, Factor factor (high/low)
3    output: Kofaktor des jeweiligen Operanden
4    t := BDDNode
5    e := BDDNode
6    edgeF := Kante von f
7    if ( index > index(f) ) then
8      return f
9    end if
10   if ( index = index(f) ) then
11     if ( high(factor) ) then
12       return isComplementEdge(f) ? !high(f) : high(f)
13     else
14       return isComplementEdge(f) ? !Low(f) : low(f)
15   end if
16   t := ( high(factor) ) ? getCofactor(high(f), index, high) :
17     getCofactor(low(f), index, low)
18   e := ( high(factor) ) ? getCofactor(low(f), index, high) :
19     getCofactor(low(f), index, low)
20   if (t = e) then
21     return isComplementEdge(f) ? !t : t
22   end if
23   if ( isComplementEdge(f) ^ isComplementEdge(t) ) then
24     edgeF := complement
```

```
25    else
26      edgeF := regular
27    end if
28    if ( isComplementEdge(t) ) then
29      t := !t
30      e := !e
31    end if
32    return BDDNode(index(f), getDDNode(t), getDDNode(e), edgeF)
33  end program
```

Code 5: Implementierung von `getCofactor`

Für eine komplementäre Kante (siehe Kapitel 3.6) auf Seite 57) erfolgt also bspw. die Berechnung von $f_{x_{i=0}} = f(\dots, x_{i-1}, 0, x_{i+1}, \dots)$ im Rahmen einer Traversierung, die linear durch das ROBDD beschränkt ist, wobei für jeweils eine Variable eine konstante Funktion eingesetzt wird.
Hinsichtlich des Verfahrens wird zunächst das ursprüngliche ROBDD kopiert, wobei jede Kante, die einen Knoten f mit $index(f) = x_i$ referenziert, durch eine Kante auf $low(f)$ bzw. $high(f)$ infolge der Reduktionsregeln ersetzt wird. Der Kofaktor nach x_i eines Knotens f entspricht also der Low-Kante, wenn dieser Knoten mit x_i markiert ist. Ansonsten gilt der Knoten $index(f), low(f)_{x_i=0'}, high(f)_{x_i=0}$. Wie bereits erwähnt, ist der Kofaktor einer Variable oftmals die Wurzelmarkierung. In diesem Fall entspricht die Lösung einem der beiden „Kinder“ und es besteht lediglich eine konstante Laufzeit [Gün02]. Die jeweiligen Abfragen `isComplementEdge()` kennzeichnen weiterhin eine Überprüfung, ob eine komplementäre Kante (siehe Kapitel 3.6 auf Seite 57) vorliegt. Damit die Not-Operation in konstanter Laufzeit erfolgen kann, wird in dem SBDD auch die jeweilige Negation des BDDs dargestellt. Daher müssen also auch komplementäre Kanten passiert werden können, wozu diese Abfragen dienen.
Der ITE-Operator ist also insgesamt mit der Shannon-Dekomposition verträglich. In Kapitel 2.4 auf Seite 24 wurde im Hinblick auf die Operationen ein Vergleich zu anderen Darstellungen von Booleschen Funktionen gemacht, was bezüglich der Kofaktor-Berechnung ebenfalls in der Tabelle 5 auf Seite 42 betrachtet wird [Mol07]:

Tabelle 5: Vergleich der Kofaktor-Berechnung für Darstellungen

Darstellung	**Beschreibung**
Wahrheitstafel	Die Laufzeit ist proportional zu 2^n, d. h. linear in der Größe zur Wahrheitstafel.
Boolesche Ausdrücke	Es wird zunächst der Operatorbaum (siehe Kapitel 2.1.2 auf Seite 13) kopiert. Anschließend erfolgt eine Ersetzung der mit der Variablen markierten Blätter durch die entsprechende Konstante. Zuletzt wird eine Reduzierung des Baumes „Bottom-up“ vorgenommen. Insgesamt ist das Verfahren daher polynomiell beschränkt.
DNF	Die entsprechenden Literale werden ersetzt, anschließend erfolgt eine Reduzierung der DNF. Offensichtlich gilt eine polynomielle Laufzeit.
KNF	Auch hier werden die entsprechenden Literale ersetzt, anschließend erfolgt eine Reduzierung der KNF. Offensichtlich gilt ebenfalls eine polynomielle Laufzeit.
Schaltkreise	Die Komplexität entspricht den Booleschen Ausdrücken.

Alle ein- und zweistelligen Booleschen Funktionen können direkt durch verschiedene Aufrufe des ITE-Operators geschrieben werden, wodurch die Implementierung deutlich vereinfacht wird [BRB07]. Dieser universelle Operator wird also insbesondere dafür verwendet, um OBDDs zu konstruieren und kann wie folgt rekursiv über die angesprochenen Kofaktoren (siehe Code 5 auf Seite 40) der Operanden implementiert werden:

```
1  program ite()
2    input: f, g, h (OBDDs)
3    output: OBDD ite(f, g, h)
4    cT := Computed-Table
5    uT := Unique-Table
6    top := top-Variable
7    resT, resC, f1, g1, h1, f0, g0, h0, t, e := BDDNode
8    node := DDNode
9    key := TableKey
10   complementEdge := false
11   standardize(f, g, h, complementEdge)
12   if ( isTerminal(f, g, h, resT) ) then
13     if (complementEdge = true) then
14       resT := !resT
15       return resT
16     end if
17   end if
18   key := getKey( getDDNode(f), getDDNode(g), getDDNode(h) )
19   if ( cT.hasNext(key, resC) ) then
20     if (complementEdge = true) then
21       resC := resC ^ getComplementEdge()
22     end if
```

```
23      return resC
24    end if
25    top := index(f)
26    if (index(g) > top) then
27      top := index(g)
28    end if
29    if (index(h) > top) then
30      top := index(h)
31    end if
32    f1 := getCofactor(f, top, high)
33    g1 := getCofactor(g, top, high)
34    h1 := getCofactor(h, top, high)
35    f0 := getCofactor(f, top, low)
36    g0 := getCofactor(g, top, low)
37    h0 := getCofactor(h, top, low)
38    t := ite(f1, g1, h1)
39    e := ite(f0, g0, h0)
40    if (t = e) then
41      if (complementEdge = true) then
42        t := !t
43        return t
44      end if
45    end if
46    node := uT.findAdd( top, getDDNode(t), getDDNode(e) )
47    resC := node
48    cT.insert(key, resC)
49    if (complementEdge = true) then
50      resC = resC ^ getComplementEdge()
51    end if
52    return resC
53  end program
```

Code 6: Implementierung von `ite`

Die Konzepte der Computed-Table und Unique-Table werden in den Kapiteln 3.5 ab Seite 55 sowie 3.4 ab Seite 47 genauestens erläutert.
Zunächst einmal wird der Aufruf standardisiert (Z. 11), d. h. Kombinationen, die zum gleichen Ergebnis führen, werden in Äquivalenzklassen zusammengefasst, wobei ein Repräsentant ausgewählt wird. Diese Technik wird in Kapitel 3.7 auf Seite 61 erläutert.
Danach wird auf die Terminierung des Verfahrens geprüft (Z. 12-17), was z. B. bei $ite(1, f, g) = ite(0, g, f) = ite(f, 1, 0) = f$ der Fall wäre. In einem solchen Fall kann das Ergebnis unmittelbar zurückgegeben werden.
Sollte sich der mit diesen Parametern berechnete OBDD-Knoten noch in der Computed-Table befinden, so kann ebenfalls direkt dieser Knoten als Ergebnis zurückgegeben werden (Z. 18-24). Somit werden weitere Aufrufe in diesem Rekursionsschritt gespart.
Ansonsten werden zwei Teilprobleme `t, e` für die Kofaktoren der Operanden erzeugt,

wobei die Wurzelmarkierung x_i ausgewählt wird, die in der Ordnung zuerst vorkommt. Dabei gilt, dass der höhere Index zuerst abgefragt wird (Z. 25-31).
Somit können die dazugehörigen Kofaktor-Berechnungen (siehe Code 5 auf Seite 40) für diese Variable in $O(1)$ erfolgen, da ein Kofaktor für einen Operanden eine Selbstreferenz liefert (wenn der Operand nicht von x_i abhängt) oder einem der „Kinder“ entspricht (Z. 32-37). Weiterhin wird diese Steigerung der Performanz durch die Festlegung einer identischen Variablenordnung für alle Operanden ermöglicht [Het02, S.37-38].
Nachdem beide Teilprobleme berechnet worden sind (Z. 38-39), erfolgt eine Überprüfung auf Isomorphie (Z. 40-45), was bezüglich SBDDs durch einen Vergleich der beiden Wurzeln von `t, e` in $O(1)$ erfolgt (siehe Kapitel 2.4.1 auf Seite 26). Es ist also die Anwendung der Isomorphie-Regel (siehe Abbildung 6 auf Seite 20) erforderlich. Sollte keine Isomorphie bestehen, so entspricht das Ergebnis daher einem Wurzelknoten r mit der Markierung x_i sowie den „Kindern“ $low(r) = e$ und $high(r) = t$. Die Wurzel könnte allerdings bereits existieren, womit die Redundanzregel greifen würde. Daher soll in $O(1)$ ein solcher Fall vermieden werden. Der Knoten sollte also nur erstellt werden, wenn er noch nicht existiert.
Zur Speicherung bereits vorhandener Ergebnisse wird dementsprechend die Unique-Table (Z. 46) benutzt. Die Funktion `findAdd` (siehe Code 8 auf Seite 48) prüft, ob ein vorgegebenes Tripel bereits in OBDD-Knoten realisiert ist. Im positiven Fall gibt sie eine Referenz auf den Knoten zurück, ansonsten wird mittels `add` ein neuer Knoten erzeugt und darauf ein Verweis zurückgeliefert. Der Vorgang der jeweiligen Überprüfung sowie Erstellung eines Knoten wird häufig auch in einer Funktion `mk` zusammengefasst.
Die anschließende Funktion `insert` (Z. 47-48) fügt das wiederum berechnete Teilproblem in die Computed-Table ein. Ohne diesen Cache würde der obige Algorithmus auch – vom Wurzelknoten ausgehend – alle Wege bis zu $0, 1$ durchlaufen, wobei die zusammengefassten Teilgraphen allerdings mehrmals betrachtet werden würden. Sollten durch den Einsatz einer Computed-Table wiederum alle bereits berechneten Ergebnisse gespeichert werden, dann wird der ITE-Algorithmus für jede Knotenkombination der drei OBDDs f, g, h aufgerufen.
Damit die Negation in $O(1)$ ermittelt werden kann, erfolgt zusätzlich die Berechnung komplementärer Kanten, wobei die Standardisierung die binäre Variable
`complementEdge` dazu verändern kann. Mittels `getComplementEdge` (Z. 49-51) passiert wiederum ein Zugriff auf eine solche Kante. Wenn demnach eine komplementäre Kante vorliegt, so muss das jeweilige Ergebnis negiert werden. Das Prinzip dahinter wird in Kapitel 3.6 auf Seite 57 erläutert.
Offensichtlich unterliegt der Algorithmus also dem *Divide and Conquer*-Prinzip, d. h. ein Problem wird in Teilprobleme aufgeteilt, anschließend werden diese einzeln gelöst und zusammengefasst. Auf dieses Verfahren bezogen, ist somit jeder Operator von Kofaktoren demnach ein Kofaktor von Operatoren.
Die Funktionsweise zur Synthese einer Funktion des ITE-Algorithmus soll nun anhand

der Abbildung 17 verdeutlicht werden:

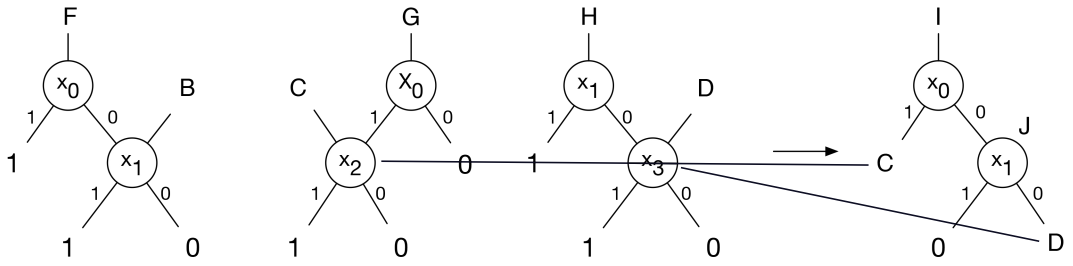

Abbildung 17: Beispiel des ITE-Algorithmus

Es sind verschiedene BDDs von unterschiedlichen Booleschen Funktionen dargestellt. Dabei sind F, G, H, I, J, B, C, D als Zeiger zu verstehen. Der durch den Aufruf $ite(F, G, H)$ berechnete OBDD I gemäß Abbildung 17 ergibt sich nachstehend:

$$\begin{aligned}
I &= ite(F, G, H) \\
&= (x_0, ite(F_{x_0}, G_{x_0}, H_{x_0}), ite(F_{x_0'}, G_{x_0'}, H_{x_0'})) \\
&= (x_0, ite(1, C, H), ite(B, 0, H)) \\
&= (x_0, C, (x_1, ite(B_{x_1}, 0_{x_1}, H_{x_1}), ite(B_{x_1'}, 0_{x_1'}, H_{x_1'})) \\
&= (x_0, C, (x_1, ite(1, 0, 1), ite(0, 0, D))) \\
&= (x_0, C, (x_1, 0, D)) \\
&= (x_0, C, J)
\end{aligned}$$

Wenn z. B. $F = x_0 + x_1$, $G = x_0 \cdot x_2$ und $H = x_1 + x_3$ aus $\mathbb{B}_4$ angenommen werden, so ergibt sich bezüglich dem ITE-Operator hinsichtlich einer Überprüfung $ite(F, G, H) = (x_0 + x_1)(x_0 x_2) + x_0' x_1' (x_1 + x_3) = x_0 x_2 + x_0' x_1' x_3$.
Zu jedem Tripel (f, g, h) existieren also weitere Tripel (f', g', h'), sodass $ite(f, g, h) = ite(f', g', h')$, obwohl mindestens einer der Operatoren verschieden ist. Wenn diese verschiedenen Paare mit dem gleichen Ergebnis auf einen eindeutigen Repräsentanten abgebildet werden, so können mithilfe einer Computed-Table zusätzliche ITE-Aufrufe gespart werden.
Das Einfügen bzw. Ermitteln von Knoten in der Computed-Table bzw. Unique-Table benötigt eine konstante Zeit (siehe Kapitel 3.5 ab Seite 55 sowie 3.4 ab Seite 47). Aufgrund dessen beträgt die Zeit- und Speicherkomplexität des ITE-Algorithmus $O(|f| \cdot |g| \cdot |h|)$ (kubisch), weil der Operator höchstens einmal für jede Kombination von Knoten in f, g, h aufgerufen werden kann. Ohne die jeweiligen Tabellen würde sich der Algorithmus ansonsten für die Lösung jedes Teilproblems zweifach rekursiv für maximal jede Variable aufrufen, was eine exponentielle Laufzeit in der Anzahl der Variablen bedeuten würde.
Meistens ist es ausreichend, zu berechnen, ob ein ITE-Aufruf eine konstante Funktion

liefert, um bereits frühzeitig für eine Terminierung zu sorgen. Soll bspw. $f \Rightarrow g$ überprüft werden, so genügt es zu zeigen, dass $\neg f + g$ eine Tautologie (siehe Kapitel 2.4.5 auf Seite 31) ist, indem es mithilfe eines adaptierten ITE-Algorithmus berechnet wird. Das Ergebnis ist uninteressant, wenn es nicht konstant ist. Der folgende Algorithmus 7 verdeutlicht die damit zusammenhängende Modifizierung des ITE-Algorithmus (siehe Code 6 auf Seite 42):

```
1  program iteConstant()
2    input: f, g, h (OBDDs)
3    output: Status, ob eine konstante Funktion vorliegt
4    cT := Computed-Table
5    res := OBDD-Knoten
6    if (terminal) then
7      // 0, 1 oder NC
8      return res
9    else if ( res := cT.hasNext(f, g, h) )
10     return NC
11   else
12     top := top-Variable von (f, g, h)
13     t := iteConstant(f[top], g[top], h[top])
14     if (t != 0 and t != 1)
15       return NC
16     end if
17     e := iteConstant(f[top'], g[top'], h[top'])
18     if (t != e) then
19       return NC
20     end if
21     cT.insert(hash(f, g, h), t)
22   end if
23 end program
```

Code 7: Implementierung von `iteConstant`

Dabei entspricht `NC` „Non Constant“, zudem wurde aufgrund der Übersichtlichkeit auf die explizite Berechnung der Kofaktoren, komplementären Kanten und Standardisierung verzichtet.
Eine Funktion ist also nur dann konstant, wenn beide Kofaktoren konstant und gleich sind. Andernfalls ist sie nicht konstant und der Algorithmus wird vorzeitig abgebrochen. Weiterhin ist hierbei die Nützlichkeit für die logische Implikation zu erkennen, da bspw. $f \Rightarrow g \Leftrightarrow \mathit{iteConstant}(f, g, 1) = 1$. Eine derartige Operation kann im Vergleich zum allgemeinen ITE-Operator durchschnittlich effizienter vollzogen werden, da keine temporären Knoten konstruiert werden müssen und dieser – wie bereits erwähnt – frühzeitig abgebrochen werden kann.
Im folgenden Abschnitt wird gezeigt, wie Knoten innerhalb der beschriebenen Synthese effizient gespeichert bzw. wiederverwendet werden können.

3.4 Die Unique-Table (UT)

Die *Unique-Table* (UT) wird als Hashtabelle realisiert, womit es gelingt, eine streng kanonische Form (siehe Kapitel 2.3 auf Seite 19) im ROBDD zu erhalten. Im Endeffekt werden dort die Subfunktionen abgespeichert, die bereits vom BDD repräsentiert werden. Dadurch ist es möglich, beim Hinzufügen oder Berechnen neuer Funktionen, ohne großen Suchaufwand bereits vorhandene Knoten einzusetzen [Hof11]. Die UT sorgt also dafür, dass Isomorphien nicht auftreten können bzw. sorgt ein Vergleich der Kofaktoren (Kinder des neuen Knotens) für die Reduktion des OBDDs.
Mithilfe von Hashverfahren ist es möglich, Knotenzugriffe bzw. Existenzprüfungen effizient zu realisieren. Im Worst Case kann ein Hashverfahren aber auch sehr ineffizient sein, weil Kollisionen nicht gänzlich verhindert werden können (Hashfunktionen sind injektiv) und die Performanz der Operationen *Einfügen*, *Suchen* sowie *Entfernen* maßgeblich von einem sog. Belegungsfaktor abhängt. Im Folgenden soll daher die Wahl der jeweiligen Hashfunktion. Tabellengröße und Kollisionsbehandlung diskutiert werden.

3.4.1 Das Hashverfahren

Bei der UT spielt *Hashing* als Technik eine große Rolle. Die Grundlage für Hashing bildet die Berechnung eines Index aus einem Suchwert. Dabei enthält Hashing zwei wesentliche Komponenten:

- **Hashfunktion**: Bildet Objekte auf Natürliche Zahlen, sog. Hashcodes ab, die als numerische Schlüssel zur Identifikation der Objekte genutzt werden.
- **Hashtabelle**: Kennzeichnet eine Liste, in der Objekte eingetragen werden, wozu der Hashcode als Index genutzt wird.

Weiterhin kennzeichnet $a = \frac{n}{m}$ den sog. *Belegungsfaktor*, der die Anzahl der Datensätze n in das Verhältnis zur Größe der Hashtabelle m setzt. Der Zugriff auf ein Element ist folglich definiert durch $O(a)$. Offensichtlich ist Hashing besonders dann effektiv, wenn m im Verhältnis zu n nicht zu klein ist. Werden die Hashfunktion und die Größe der Tabelle geeignet gewählt, so beträgt der Aufwand für den Zugriff nur $O(1)$, was im Folgenden diskutiert wird und in Kapitel 3.4.2 auf Seite 50 genau beschrieben ist. Eine gute Hashfunktion ist demnach schnell berechenbar respektive verteilt die Datensätze nach Möglichkeit gleichmäßig auf den Speicherbereich. Weiterhin sorgt der Einsatz der UT im ITE-Algorithmus dafür, dass der Aufwand dieses Operators kubisch beschränkt ist (siehe Kapitel 3.3.1 auf Seite 38).
Hinsichtlich der Schlüsselwahl wird jedem Tripel (v, g, h) ein Knoten zugeordnet. Jeder Knoten besitzt also einen Eintrag in der Hashtabelle. Bevor ein neuer Knoten $f = (v, g, h)$ eingefügt wird (siehe Code 6 auf Seite 42), erfolgt eine Überprüfung, ob es

nicht schon einen entsprechenden Eintrag gibt. Der folgende Code 8 verdeutlicht diese Operation:

```
program findAdd()
  input: f, g, h (Adressen von Knoten)
  output: Adresse eines Knoten aus der UT
  key := TableKey
  ddNode := DDNode
  uTable := Unique-Table
  key := getKey(f, g, h)
  if ( !uTable.find(key, ddNode) ) then
    create(ddNode)
    uTable.add(key, ddNode)
  end if
  return ddNode
end program
```

Code 8: Implementierung von `findAdd`

Die relevante Operation hierbei, um eine dementsprechende Überprüfung zu erledigen, ist `find` und als Code 9 folgendermaßen aufgebaut:

```
program find()
  input: key, value (Knoten)
  output: Status, ob es den Knoten bereits in der UT gibt
  pos := getKey(key)
  items := Vektor mit Abbildungen von Keys auf Knoten
  nodes := size(items[pos])
  for (i to nodes) do
    if (items[pos][i].first = key) then
      value := items[pos][i].second
      return true
    end if
  end for
  return false
end program
```

Code 9: Implementierung von `find`

Wenn also bereits ein Knoten existiert, wird der Eintrag ohne Aktualisierung benutzt, ansonsten wird der jeweilige Knoten in die Hashtabelle mit aufgenommen und erhält einen Eintrag. Somit kann ein Knoten über `items[getKey(key)].push(pair(key, value))` eingetragen werden. Sollte eine Kollision auftreten, so wird der Knoten direkt als Nachfolger eingetragen.
Somit bildet das Tripel den Schlüssel, um auf einen Knoten zugreifen zu können. Daraus folgt, wenn g und h bereits eine Kanonizität vorweisen, so existiert f im ROBDD unter der Voraussetzung, dass es einen Eintrag für (v, g, h) gibt. Infolge der Verwendung des ITE-Operators und der UT folgt also automatisch die Kanonizität.

In der Praxis werden i. d. R. zwei Methoden eingesetzt [Het02, S.72-73], nämlich das Modulo- sowie Multiplikations-Verfahren, die nun beschrieben werden, wobei ein direkter Vergleich bezüglich experimenteller Ergebnisse in Kapitel 4.1 auf Seite 69 gemacht wird.

Das Modulo-Verfahren

Bei dem *Modulo-Verfahren* lautet die Hashfunktion $h(k) = k \; mod \; m$, wobei k dem Schlüssel und m der Größe der Hashtabelle entspricht, wobei m für eine gute Gleichverteilung der Elemente in der Tabelle eine Primzahl sein sollte [Knu11]. Bezogen auf Knoten, wird k durch das jeweilige Tripel f, g, h ersetzt, d. h. es gilt die Funktion $h(f, g, h) = ((g+h) >> f) \; mod \; m$. Hierbei kennzeichnet $>>$ einen bitweisen Operator bzw. Rechtsshift.
Wird der Divisionsblock einer Arithmetisch-logischen Einheit (ALU) des Prozessors (CPU) betrachtet, so kann festgestellt werden, dass dies eine zeitintensive Rechenoperation ist [Het02, S.73]. Hingegen können andere Operationen wie der Rechts-Shift um i Stellen eingesetzt werden. Dieser entspricht dabei Divisionen mit einem Divisor 2^i für ein $i \in \mathbb{N}$ in Abhängigkeit zu der Wortbreite des Rechners. Für viele Eingaben ist hierbei die Wahl einer Zweierpotenz für m nicht geeignet, da die höherwertigen Bits bei den Hash-Berechnungen ignoriert werden, was zu einer Ungleichverteilung der Schlüssel führt. Wird hingegen m als Primzahl derartig festgelegt, dass keine Mersenne-Primzahl der Form $2^i - 1$ besteht, so gibt es eine geringe Anzahl von zu erwarteten Kollisionen bei vielen Eingabeverteilungen [CLRS01, S.231].

Das Multiplikations-Verfahren

Das *Multiplikations-Verfahren* berechnet durch die Funktion $h(k) = \lfloor m \cdot (kx - \lfloor kx \rfloor) \rfloor$ die Hashadresse, wobei k den Schlüssel und $x := \frac{(\sqrt{5}-1)}{2}$ eine irrationale Zahl sowie m die Größe der Hashtabelle kennzeichnet. Um eine gleichmäßige Verteilung der Datensätze zu erreichen, wird x als Kehrwert des *goldenen Schnittes* gewählt [OW90].
Innerhalb der ALU wird die Fließkommazahl zur Multiplikation verwendet, um irrationale Zahlen zu verarbeiten [Het02, S.74]. Auch diese Methode ist zeitintensiv, jedoch können ganze Zahlen im Rechner als Bruchzahlen mit einem Dezimalpunkt vor der höchstwertigen Ziffer betrachtet werden, wobei für m dann eine Zweierpotenz gewählt wird. Somit ist die Berechnung von $h(k)$ mit einer ganzzahligen Multiplikation und einer Rechts-Shift-Operation bitweise möglich.
Infolge von verschiedenen Studien erwies sich in diesem Zusammenhang die Hashfunktion $h(g, h) = ((g + h \cdot p_1) \cdot p_2) >> (w - log_2(m))$ als besonders vorteilhaft [YBO+98]. Dabei kennzeichnet w die Wortbreite der ALU des Rechners in Bit, $p_i, i \in \{1, 2\}$

bestimmt möglichst große Primzahlen, sodass sich eine gute Gleichverteilung ergibt. Konkret sollten die Primzahlen derartig gewählt werden, sodass die aus ähnlichen Zeigern resultierenden Overflows der zwischenzeitlichen Ergebnisse aus dem Wortbereich herausfallen. Zwei Zeiger gelten hierbei als ähnlich, wenn die oberen acht Bits gleich sind. So könnten bspw. $p_1 = 12.582.917, p_2 = 4.256.249$ gewählt werden [Het02, S.74-75].
Im Vergleich zum Modulo-Verfahren (siehe Kapitel 3.4.1 auf Seite 49) liefern beide Verfahren bezüglich der Datenverteilung ähnliche Resultate, was in den dazugehörigen Experimenten im Anhang ersichtlich ist. Statistisch gesehen, gilt darüber hinaus, dass das Modulo-Verfahren keiner anderen Methode – hinsichtlich der Ergebnisqualität – nachsteht [LYD71].
Nachdem hierbei die Wahl der jeweiligen Hashfunktionen für die UT erläutert wurden, handelt der nächste Abschnitt darüber, wie mit Kollisionen in der Tabelle umgegangen wird.

3.4.2 Die Kollisionsbehandlung

Im Gegensatz zur Computed-Table (siehe Kapitel 3.5 auf Seite 55) dürfen bei der UT keine noch benötigten Einträge gelöscht werden, da hierdurch die Kanonizität verloren geht [Sie07, S.49]. Der Zugriff auf einen Knoten bzw. dem ROBDD wird also durch ein Hashverfahren mit Kollisionsbehandlung realisiert.
Ein Knoten wird mit verschiedenen Informationen – wie etwa einem Referenzzähler – gespeichert, die in Kapitel 3.2 auf Seite 37 beschrieben sind.
Eine gute Hashfunktion muss so aufbereitet sein, dass möglichst selten Kollisionen auftreten. Sollten wiederum Kollisionen auftreten, so müssen diese effizient gelöst werden. Eine *Kollision* ist dadurch gekennzeichnet, dass für unterschiedliche Objekte derselbe Hashcode berechnet worden ist. Es gibt diesbezüglich zwei Ansätze, die im Folgenden betrachtet werden, nämlich das offene Hashing und Verfahren mit Verkettung von Überläufern.

Offenes Hashing

Bei dem *offenen Hashing* werden Überläufe im Regelfall durch Sondieren behandelt, d. h. es wird ein Ausweichplatz gesucht [Pet57]. Da dies nicht vorweg absehbar ist, gibt es zusätzlich zur Hashfunktion eine sog. Sondierungsfunktion $s(j, k)$. Diese beschreibt die Reihenfolge, in der die Speicherpositionen betrachtet werden, d. h. sie liefert einen Versatz zum Hashwert, wobei Indizes $(h(k) - s(j, k))\ mod\ m$ ($j :=$ Anzahl der Fehlversuche im Bereich $0 \ldots m - 1$) durchsucht werden. Im Wesentlichen werden drei Sondierungsarten unterschieden:

- **Lineares Sondieren**: Hierbei wird die Hashtabelle linear nach Werten durchsucht, d. h. es besteht $h(k), h(k)-1, h(k)-2, \ldots, 0, m-1, \ldots, h(k)+1$, wobei eine Sondierungsfunktion $s(j,k) = j$ gilt.
  So kann bspw. die Größe der Hashtabelle $m = 7$ und die Hashfunktion $h(k) = k \; mod \; m$ festgelegt werden. Wenn nun die 12 eingefügt werden soll, so gilt $h(12) = 12 \; mod \; 7 = 5$, d. h. das Element wird an der Stelle 5 gespeichert. Sollte jedoch die 19 eingefügt werden, so würde diese – ohne Sondierungsfunktion – ebenfalls an der Stelle 5 gespeichert werden. Allerdings wird diese nun um mehrere Stellen nach hinten verschoben, bis eine freie Stelle gefunden wurde. In diesem Fall ist die Stelle 4 noch frei, d. h. es gilt $j = 1$, da lediglich ein Fehlversuch vorlag.
  Ein größeres Problem bei dem linearen Sondieren ist die primäre Häufung, was im Zusammenhang mit dem Belegungsfaktor a (siehe Kapitel 3.4.1 auf Seite 47) steht. Häufungspunkte senken die Effizienz und der Aufwand steigt erheblich, wenn $a \mapsto 1$. In diesem Fall würden nach dem Einfügen von $12, 53, 5$ weitere Schlüssel derartig abgelegt werden, dass Werte mit $h(k) = 1$ an die Stelle 1 sowie Werte mit $h(k) = 2 \ldots 5$ an der Stelle 2 platziert werden würden.

- **Quadratisches Sondieren**: Die Hashtabelle wird quadratisch nach Werten durchsucht, d. h. es gibt $s(j,k) = \lceil \frac{j}{2} \rceil^2 \cdot (-1)^j$ bzw. $h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4$ usw.
  Wenn also z. B. die 5 eingefügt werden soll, so würde das Element analog zum linearen Sondieren an die Stelle 5 geschoben werden, da es dort einen freien Platz gibt. Wenn wiederum die 19 eingefügt werden soll, so gilt $h(19) = 5$, d. h. es muss wieder ein Ausweichplatz gesucht werden. In diesem Fall besteht dann $s(1, 19) = \lceil \frac{1}{2}^2 \rceil \cdot (-1)^1 = -1$, womit die 19 an Stelle 4 geschoben wird. Würde diese Stelle nicht frei sein, so müsste j inkrementiert werden, womit $s(2, 19) = 1$ gelten würde. Danach würde dann $s(3, 19) = -4$ bzw. $s(4, 19) = 4$ usw. gelten, was ein quadratisches Verhalten beschreibt.
  Ein Problem hierbei ist die sekundäre Häufung, d. h. Synonyme behindern sich gegenseitig. Wenn davon ausgegangen wird, dass sich bei $m = 7$ bereits die Werte $15, 2, 53, 12, 5$ in der Hashtabelle befinden, so kann dieses Symptom bei 5 und 19 über die Suche nach einem freien Platz beobachtet werden. Insgesamt stören sich also auch hier Werte mit demselben Hashcode $h(k)$.

- **Doppeltes Hashing**: Es wird eine zweite Hashfunktion $h'(k)$ für das Sondieren $s(j,k) = j \cdot h'(k)$ verwendet, d. h. $h(k), h(k) - h'(k), h(k) - 2 \cdot h'(k), \ldots, h(k) - (m-1) \cdot h'(k)$. Die Anforderungen an $h'(k)$ sollten sein, dass $h'(k) \neq 0$ und dass diese kein Teiler von m sein darf, d. h. m sollte eine Primzahl sein (siehe Kapitel 3.4.1 auf Seite 47). Darüber hinaus sollte sie unabhängig von $h(k)$ sein, also sollte $p[h(k) = h(k') \wedge h'(k) = h'(k')] = p[h(k) = h(k')] \cdot p[h'(k) = h'(k')]$ gelten. Wenn m eine Primzahl ist, so gilt demnach $h'(k) = 1 + k \; mod \; (m-2)$.

Sollte also bspw. $m = 7$, $h(k) = k \; mod \; m$, $h'(k) = 1 + k \; mod \; (m - 2)$ und $s(j, k) = j \cdot h'(k)$ sein, so würde zunächst die 12 erneut an die Stelle 5 verschoben werden. Bei der Zahl 19 würde wieder eine Kollision passieren, wobei hier nun die zweite Hash- mit der Sondierungsfunktion greifen würde. Demnach besteht $h'(19) = 1 + 19 \; mod \; (7 - 2) = 0$, d. h. $s(1, 19) = 1 \cdot 0 = 0$, womit die 19 an der Stelle 0 platziert wird.
Durch diese Vorgehensweise entstehen – insgesamt gesehen – kaum Häufungen [Sed92, S.282-285].

Die Entfernung von Datensätzen bei offenen Hashverfahren stellt ein weiteres Problem dar. Sollte bspw. ein Wert v' eingefügt werden, wobei sich an der dafür ermittelten Position bereits ein Wert v befindet, so würde v' an einer anderen Position gespeichert werden. Wenn nun aber v gelöscht wird, so kann v' nicht wiedergefunden werden, da die leer gewordene Position bezüglich der Sondierungsfolge vor der aktuellen Position von v' auftritt. I. d. R. wird daher ein Datensatz v nicht wirklich entfernt, sondern nur als entfernt markiert. Wenn demnach ein neuer Datensatz eingefügt werden soll, so wird die Position von v als unbelegt angesehen. Wird hingegen ein Datensatz gesucht, so wird der Platz von v als belegt betrachtet.
Im Hinblick auf erfolglosen Suchen (terminieren erst bei einem freien Platz) ist eine derartige Vorgehensweise jedoch nicht besonders effektiv, da es häufig negativ beantwortete Resultate bezüglich Knotenexistenzprüfungen gibt. Dementsprechende Experimente sind in Kapitel 4.1 auf Seite 69 ersichtlich.

Hashverfahren mit Verkettung von Überläufern

Sollte bei solchen Verfahren eine Kollision auftreten, so wird der jeweilige Knoten in eine Liste eingehängt, d. h. außerhalb der jeweiligen Hashtabelle. Dies bringt den Vorteil mit sich, dass es deutlich weniger Platzverschwendung für unbelegte Einträge gibt. Zur Verdeutlichung dieses Sachverhaltes wird im Folgenden nachstehendes ROBDD in der Abbildung 18 betrachtet:

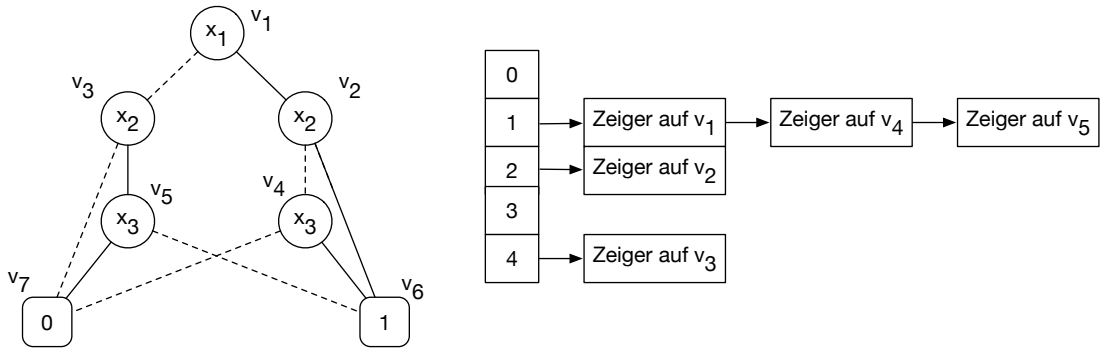

Abbildung 18: Verkettung von Überläufern

Die UT könnte insgesamt fünf Plätze umfassen, wobei eine Hashfunktion $h(x_i, v_j, v_k) = i + j + k \ (mod\ 5)$ vorliegt. Für den Knoten v_1 gilt somit $1 + 2 + 3 \ (mod\ 5) = 1$, was den Hashwert kennzeichnet. Somit ordnet sich der Wert an Stelle 1 im Feld ein. Allerdings hat v_4 bspw. ebenfalls diesen Hashwert. Aufgrund dieser Kollision wird der Zeiger auf diesen Knoten in eine Überläuferliste gespeichert. Unter der Annahme, dass das OBDD vor der Überprüfung reduziert war, gilt, dass eine Unterfunktion – die durch das Tripel (x_i, g, h) dargestellt wird – genau dann bereits in dem OBDD existiert, wenn es einen Knoten in der Liste gibt, der das gleiche Tripel repräsentiert [MT98, S.112-114]. Daraus folgt, dass es sehr leicht ist, den ROBDD auch während des Einfügens neuer Knoten aufrechtzuerhalten. Dementsprechend entfällt ein expliziter Aufruf von `reduce` (siehe Code 1 auf Seite 21).

Die Verkettung von Überläufern wurde bei diesem ROBDD-Paket dahingehend adaptiert, dass die Knoten v_i direkt in die Liste abgelegt wurden, anstelle der Referenzen darauf. Hierzu wurde eine Komponente in Gestalt eines Vektors hinsichtlich der Datenstruktur gewählt, die einen Verweis auf den Knoten enthält, der den Nachfolger in der dazugehörigen Kollisionsliste darstellt. Wie bereits in Kapitel 3.2 auf Seite 37 erwähnt, liegen die Knoten dadurch eng beieinander und durch ihre Position wird Zeiger-Arithmetik möglich, womit ein konstanter Zugriff auf alle Felder besteht. Die Annahme hierbei ist, dass das Finden eines Knotens in dem Vektor aufgrund des Lokalitätsprinzips schneller funktioniert, da der nächste Block darin gleichzeitig mit in den Cache geladen wird. Das Lokalitätsprinzip bezieht sich in diesem Zusammenhang auf die Zugriffswahrscheinlichkeit hinsichtlich einer Speicherzelle, d. h. dass die Wahrscheinlichkeit sehr hoch ist, dass diese auch in naher Zukunft wieder benötigt wird. Bei

einer ursprünglichen verketteten Liste ist dieser Vorgang wesentlich schwieriger abzuschätzen, was von Zeigern bedingt wird.
Eine dementsprechende Evaluation ist im Kapitel 4.1 auf Seite 69 ersichtlich.

Wird die Gesamtlaufzeit der UTs (siehe Code 6 auf Seite 42) betrachtet, so beträgt diese bei einer erfolglosen Suche (Prüfung auf Nicht-Existenz sowie anschließender Eintragung des ROBDD-Knotens v mit $l(v) = x_i$) durchschnittlich $O(\frac{n}{m})$ bezüglich der Schlüsselvergleiche. Die durchschnittliche Anzahl der Einträge der Überläuferliste entspricht $\frac{n}{m}$, wenn n Elemente auf m Listen verteilt werden. Diese entspricht dem Belegungsfaktor a und der durchschnittlichen Suchzeit, wenn kein Erfolg vorhanden ist. Der jeweilige Knoten wiederum kann in $O(1)$ am Ende der Liste eingetragen werden.
Bei einer erfolgreichen Suche hingegen, wird die Laufzeit durch $O(1 + \frac{a}{2})$ konkretisiert, weil lediglich von Bedeutung ist, dass beim Einfügen eines j-ten Knotens die durchschnittliche Listenlänge $\frac{j-1}{m}$ besteht [Het02, S.77].
In diesem Kapitel wurde beschrieben, wie eine UT funktioniert und wie eine damit zusammenhängende Hashfunktion gewählt werden muss, damit Knoten effizient gefunden werden können. Im nächsten Kapitel soll – darauf aufbauend – untersucht werden, inwieweit eine Kombination der UT und BDDs hinsichtlich der Datenstrukturen möglich ist.

3.4.3 Kombination von UT und BDD

Anstelle der Benutzung von separaten ADTs für die UT und ROBDD-Knoten, können beide als Kombination aufgefasst werden. Somit existiert allgemein für jeden Knoten ein zusätzliches Feld, welches einen Verweis auf die Kollisionskette hat [BRB07]. Dieser Zusammenhang wird von der Abbildung 19 wie folgt visualisiert:

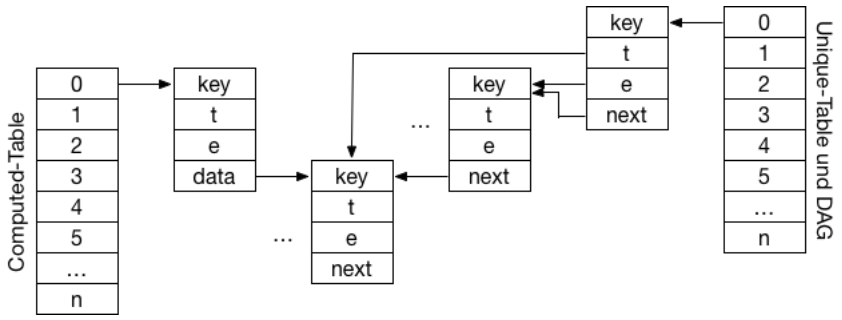

Abbildung 19: Kombination von UT und BDD

Hierbei steht DAG für einen gerichteten azyklischen Graphen (siehe Kapitel 2.1.6 auf Seite 16). Es werden zufällige Elemente im ROBDD in der Kollisionskette verbunden, was zur Folge hat, dass eine schnellere Ermittlung von Knoten im ROBDD stattfinden kann. Im Durchschnitt würde es hierbei nur noch vier Elemente pro Knoten geben.

Dies impliziert eine Verbesserung des Speicherverhaltens, da der Overhead für die dazugehörige Allokation vermindert wird [BRB07]. Wie bereits in Kapitel 3.4 auf Seite 47 erwähnt, ist es jedoch auch möglich, das Feld `next` direkt in Vektoren abzulegen. Dadurch werden die Nachfolger in gemeinsame Slots abgelegt, wodurch sie nebeneinander liegen und ein Zugriff schneller erfolgen kann.
Um das Speicherverhalten genauer zu untersuchen, soll angenommen werden, dass sowohl die Computed-Table (siehe Kapitel 3.5 auf Seite 55) als auch die Kombination der UT/Knoten dieselben Speicherplätze benutzen. Der Ladefaktor von 4 für die UT gewährleistet eine vertretbare gegenläufige Beziehung zwischen dem Auffinden eines Knotens in der Hashtabelle und dem dazugehörigen Overhead in Bezug auf den Speicher. Bei einem Ladefaktor von 4 liegt die Speicherbenutzung somit bei ungefähr 22 Bytes pro Knoten, wenn eine 64-Bit Maschine angenommen wird. Das Speicherverhalten geht aus den Experimenten in Kapitel 4.1 auf Seite 69 genauestens hervor. Es werden dabei vier Wörter für jeden Eintrag in der Kombination von UT/Knoten benutzt, wobei drei Wörter den Schlüssel für eine Operation ausmachen, d. h. f, g, h für $ite(f, g, h)$ (siehe Code 6 auf Seite 42) und ein Wort für das Resultat steht. So beschreiben (f, g, h) den Block $(key, t, e, data)$ und $ite(f, g, h)$ den Block $(key, t, e, next)$. Zusätzlich kommen zwei Bytes als Overhead bezüglich des Speichers dazu.
Neben des effizienten Speicherns und der Wiederauffindung von Knoten innerhalb der Synthese (siehe Kapitel 3.3.1 auf Seite 38) wird im Folgenden die Implementierung des Caches beschrieben, womit ebenfalls die Performanz der Synthese gesteigert wird.

3.5 Computed-Table

Um den Rechenaufwand bzw. Speicherplatz des ITE-Algorithmus (siehe Code 6 auf Seite 42) zu verringern, wird eine zweite Hashtabelle benutzt, die sog. *Computed-Table* (CT). Diese Tabelle ist als Cache zu verstehen, d. h. bereits berechnete Ergebnisse werden dort abgespeichert, damit ein erneuter Zugriff in konstanter Zeit möglich bzw. der Gesamtaufwand bei den eben genannten Algorithmen polynomiell beschränkt ist, was im Folgenden diskutiert werden soll.
Bei der Beschreibung der Synthesen aus Kapitel 3.3 auf Seite 38 werden sukzessive Operationen auf OBDDs angewendet. Bezüglich des ITE-Algorithmus wird mittels `cT.hasNext` geprüft, ob es bereits eine Zusammenstellung von betrachteten Operanden gibt, wobei im positiven Fall eine vorhandene Berechnung zurückgeliefert wird, was durch den folgenden Code 10 verdeutlicht wird:

```
1 program hasNext()
2   input: key, node
3   output: Status, ob es den Knoten bereits in der CT gibt
4   pos := getKey(key)
5   items := Paar mit Abbildungen von Keys auf Knoten
```

```
6     if (key == items[pos].first) then
7       node := items[pos].second
8       return true
9     end if
10    return false
11  end program
```

Code 10: Implementierung von `hasNext`

Mithilfe von `cT.insert` wird wiederum eine solche Zusammenstellung sowie das berechnete Ergebnis gespeichert. Dementsprechend gilt hier `items[pos].first = key` und `items[pos].second = node` nach der Generierung des Schlüssels. Analog hierzu bildet der Cache drei Knoten f, g, h auf den resultierenden Knoten $ite(f, g, h)$ ab. Somit können beim ITE-Algorithmus Rekursionsaufrufe eingespart werden, da bei den betrachteten Wegen – von der Wurzel aus – zu den Blättern isomorphe Teilgraphen nicht öfter betrachtet werden müssen. Damit verbundene Tests sind in Kapitel 4.2 auf Seite 71 zu finden.
Hierbei ist ein ähnliches Problem zur UT erkennbar, das sich auf die Existenzprüfung sowie Speicherung von Ergebnissen in Form von Knoten bezieht, was im Best Case in $O(1)$ abläuft. Es ist daher unabdingbar, ebenfalls eine geeignete Wahl bezüglich der Hashfunktion sowie des Schlüssels (siehe Kapitel 3.4.1 auf Seite 47) zu treffen. Analog zur Funktion werden auch die darin benutzten Knoten f, g bzw. f, g, h benutzt. Hier kann dann ebenfalls entweder das Modulo- oder Multiplikationsverfahren angewendet werden.
Durch die Verwendung dieser Hashtabelle reicht somit bspw. bei `ite` eine Überprüfung aus, ob ein Tripel (v_1, v_2, v_3) bereits auf einen Knoten v zeigt, womit dieser dann wiederverwendet werden kann. Demzufolge muss lediglich erzielt werden, dass Rückschlüsse darauf gemacht werden können, für welche Operation ein Eintrag in der CT erzeugt wurde.
Im Gegensatz zur UT dürfen auch noch eventuell benötigte Knoten gelöscht werden, da hierdurch die Kanonizität nicht gefährdet wird. Es gibt also keine Notwendigkeit, Einträge bis zu ihrer Löschung zu speichern. Bezüglich der obigen Argumentation soll aufgrund der Effektivität auch kein Suchvorgang innerhalb der CT stattfinden. Um also Speicherplatz zu sparen, wird eine CT als sog. dynamischer *hashbasierter Cache* eingesetzt (anfangs limitiert), d. h. es wird keine Kollisionsstrategie eingesetzt, was die Abbildung 20 auf Seite 57 visualisiert:

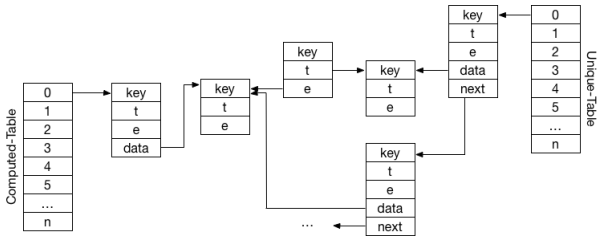

Abbildung 20: Zusammenhang zwischen Computed- und Unique-Table

Die zusammenhängenden Felder *key*, t und e umschreiben dabei die jeweiligen Knoten im BDD. Um das Speicherverhalten dahingehend zu optimieren, werden UT und Knoten in einen ADT zusammengefügt. Diese Technik wird in Kapitel 3.4.3 auf Seite 54 beschrieben.
Gibt es also bspw. bereits einen Eintrag für einen Knoten v an einer Position x, so wird dieser Knoten durch einen anderen Knoten v' überschrieben. Bezüglich des Ablaufs wird also zuerst die oberste Instanz einer Synthese abgefragt. Sollte hier seitens der CT keine positive Rückmeldung erfolgen, so besteht weiterhin die Chance, diese bei Subinstanzen zu bekommen. Empirische Studien zeigen, dass es häufig erfolglose Suchen gibt und daher die eben beschriebene Methode schneller als eine Suche hinsichtlich Überläuferlisten ist [Het02, S.86]. Im Gegensatz zu ROBDD-Knoten müsste dabei jeder Überläufer der CT zuerst den Speicherplatz der Stelle in der Liste allokieren, um anschließend Datensätze darin speichern zu können. Der größere Aufwand für eine solche Kollisionsstrategie wird also eingespart. Praktisch gesehen, kann es natürlich dennoch sein, dass die Synthese darunter leidet, da auch die Nachfolger für den überschriebenen Knoten nicht mehr in der CT enthalten sein müssen. So kann also bspw. die ITE-Operation nun im Worst Case eine exponentielle Laufzeit aufweisen, wenn alle Schlüssel auf denselben Wert bzw. Knoten zeigen. Dies ist jedoch in der Praxis kaum der Fall, sodass das Risiko praktisch eingegangen werden kann [Jan01]. Eine weitere Abhilfe schafft hierbei eine Approximation der Größe der CT. Empirische Resultate empfehlen hierzu insgesamt 500.000 Einträge [Het02, S.88]. Experimente zwischen einem hashbasiertem Cache bzw. einem Cache mit Kollisionsstrategie unterstützen diese These, die in Kapitel 4.2 auf Seite 71 ersichtlich sind.
Insgesamt sind also sowohl die Operation `hasNext` als auch `insert` offensichtlich in $O(1)$ möglich. Damit auch die Negation in konstanter Laufzeit ermöglicht wird, muss der Einsatz von komplementären Kanten erfolgen, die im nächsten Abschnitt erläutert werden.

3.6 Komplementäre Kanten

In Kapitel 3 auf Seite 33 wurde bereits angedeutet, dass die Verarbeitung der Negation nicht explizit durchgeführt werden muss, sondern mittels komplementärer Kanten in

konstanter Zeit ermöglicht wird, anstelle von $O(|f|)$. Hierzu sollen die ROBDD-Knoten $f = x_0 \wedge x_1 \vee \neg x_0 \wedge x_2$ und $f' = x_0 \wedge \neg x_1 \vee \neg x_0 \wedge \neg x_2$ betrachtet werden:

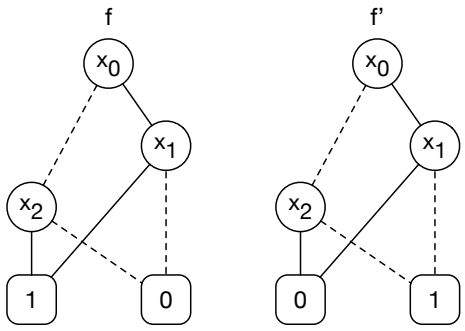

Abbildung 21: Negation einer Funktion

Bei einem Vergleich beider Darstellungen aus der Abbildung 21 fällt auf, dass sie sich sehr ähnlich sind, wobei lediglich die 0- und 1-Blatt vertauscht sind. Formal werden die ON- und OFF-Menge (siehe Kapitel 2.1.1 auf Seite 12) vertauscht. Diese Erkenntnis kann dazu genutzt werden, festzulegen, wie die Markierung der Blätter interpretiert wird [BRB07]. Der Knoten f' wird demnach nicht gespeichert, sondern eine spezielle Kante auf f, die ein *complement-Bit* gesetzt hat. Diese Kante wird als *komplementäre Kante* (CE) bezeichnet und sagt aus, dass die Funktion – in der diese Kante endet – als komplementär interpretiert werden muss. Konkret wird also das Komplement einer Kante berücksichtigt, indem ein Bit in der Speicheradresse eines Knotens reserviert wird. Sollte es gesetzt sein, so wird das dazugehörige ROBDD als negiert interpretiert. Wie bei herkömmlichen OBDDs (siehe Kapitel 2.3 auf Seite 19) ist die Auswertung derartig aufgebaut, dass der Berechnungspfad durchlaufen wird. Es wird jedoch zusätzlich gezählt, ob eine gerade oder ungerade Anzahl von Kanten mit der Markierung durchlaufen wird [Het02, S.42-44]. Sollte es eine ungerade Anzahl sein, so wird der Wert des jeweiligen erreichten Blattes negiert. Mit dieser beschriebenen Variante würde jedoch – ohne weitere Regeln – die Kanonizität verletzt werden, da sie nicht mehr bis auf Isomorphie eindeutig ist. So gibt es mehrere Fälle, wobei verschiedene Paare existieren, die jeweils funktional äquivalent zueinander sind [MT98, S.119-121]. Wenn für einen Knoten v das Tripel aus ausgehender high- und Low-Kante sowie eingehender Kante betrachtet wird, so gibt es genau $2^3 = 8$ Möglichkeiten, die Komplementbits zu setzen. Mithilfe der De Morganschen Gesetze (siehe Kapitel 2.1 auf Seite 11) kann diese funktionale Äquivalenz gezeigt werden:

$$\begin{aligned}
\overline{x_i f_{x_i} + \overline{x_i} f_{\overline{x_i}}} &= \overline{x_i f_{x_i}} \cdot \overline{\overline{x_i} f_{\overline{x_i}}} \\
&= (\overline{x_i} + \overline{f_{x_i}}) \cdot (x_i + \overline{f_{\overline{x_i}}}) \\
&= x_i \overline{f_{x_i}} + \overline{x_i} \overline{f_{\overline{x_i}}} + \overline{f_{x_i} f_{\overline{x_i}}} \\
&= x_i (\overline{f_{x_i}} + \overline{f_{x_i} f_{\overline{x_i}}}) + \overline{x_i} (\overline{f_{\overline{x_i}}} + \overline{f_{x_i} f_{\overline{x_i}}}) \\
&\overset{Absorption}{=} x_i \overline{f_{x_i}} + \overline{x_i} \overline{f_{\overline{x_i}}}
\end{aligned}$$

Diese Fälle bzw. vier Paare von Kombinationen zeigt die nachstehende Abbildung 22:

Abbildung 22: Äquivalente Paare bei komplementären Kanten

Hierbei stehen die schwarz gefärbten Kreise für sog. *Output-Inverter*, womit die jeweiligen Kantenmarkierungen gekennzeichnet werden. So sind bspw. eine Kante zu einem 1-Blatt und eine Kante mit Output-Inverter zu einem 0-Blatt äquivalent zueinander. Es bedarf also zusätzlicher Regeln für die Verwendung von Output-Invertern, um die Eindeutigkeit weiterhin sicherzustellen [Sie07, S.46-47]:

1. Es gibt nur noch ein 1-Blatt, wodurch ein Blatt also entfällt.
2. Output-Inverter sind nur auf Low-Kanten erlaubt.
3. Sollte ein OBDD eine Funktion f darstellen, so gibt es i. d. R. auch einen Zeiger zu f, der auch einen Output-Inverter beinhalten darf. Andererseits würde sich ansonsten die konstante 0-Funktion nicht als Komplement des 1-Blattes darstellen lassen.

Hinweis: Die ersten beiden Regeln können natürlich auch umgedreht werden. Wenn es also nur noch ein 0-Blatt gäbe, so würden Output-Inverter nur auf High-Kanten erlaubt sein.

Um also wieder für eine Eindeutigkeit zu sorgen, muss eine Eigenschaft ausgenutzt werden, die jeweils von den vier Paaren nur ein Bestimmtes erlaubt. Hier muss also bspw. die High-Kante immer regulär, d. h. nicht komplementär sein [Sch97, S.24-25]. Sämtliche Regeln – inklusive der Regeln aus Abbildung 6 auf Seite 20 – werden bei dem Aufruf von `findAdd` (siehe Code 6 auf Seite 42) angewendet, d. h. jedes OBDD wird weiterhin reduziert in der UT (siehe Kapitel 3.4 auf Seite 47) gespeichert:

1. Eliminierung: Es wird im Fall $t = e$ die Speicheradresse von t zurückgegeben.
2. Isomorphie: Folgt automatisch aus der Konstruktion, da zwei verschiedene Knoten mit der gleichen ITE-Operation nicht gespeichert werden können.
3. Negation: Gilt t', so wird `res = findAdd(v, t', e')` gespeichert und `res` zurückgegeben.

Alle OBDD-Operationen greifen lediglich nur mittels `findAdd` auf OBDD-Knoten zu. Somit erzeugen alle Operationen nur ROBDDs.
Die Abbildung 23 auf Seite 60 zeigt dementsprechend nun die Überführung zweier ROBDDs ohne Output-Inverter für Boolesche Funktionen f und f' zu ROBDDs mit Output-Invertern:

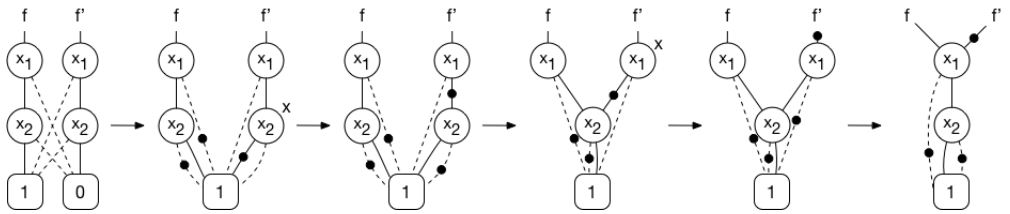

Abbildung 23: Überführung von ROBDDs zu ROBDDs mit Output-Invertern

Gemäß der Abbildung 23 wird also zunächst das 0-Blatt entfernt, wobei alle eingehenden Kanten davon umgelenkt und komplementiert werden. Anschließend wird mittels einem Bottom-up Verfahren für jeden Knoten eine eindeutige Entscheidung zur Normierung der Kanten getroffen. Der betrachtete Knoten wir hier mit einem x (siehe Abbildung) gekennzeichnet. Aufgrund der Kanonizität darf keine High-Kante einen Output-Inverter besitzen, weshalb x_2 normiert werden muss, d. h. die Marke wird entfernt und gemäß Abbildung 22 auf Seite 59 (4. Fall) reduziert. Falls es möglich ist, so müssen ebenfalls die Reduktionsregeln der Isomorphie bzw. Redundanz angewendet werden. Analog dazu wird dieses Vorgehen für jeden Knoten durchgeführt. In der Praxis wird bereits bei der Konstruktion auf die Normierung geachtet, sodass eine nachträgliche Überführung nicht notwendig ist [Het02, S.45].
Im Vergleich zu herkömmlichen OBDDs können letztendlich also mittels OBDDs mit CEs zu jedem Knoten v zwei Funktionen dargestellt werden, nämlich f_v und die dazugehörige Negation f'_v. Wenn v also über eine Kante ohne Output-Inverter erreicht wird, so gilt f_v, ansonsten f'_v. Dadurch entstehen auch in Bezug auf den ITE-Algorithmus ein neuer Terminalfall, nämlich $ite(f, 0, 1) = f'$. Die Ausführung von Booleschen Operationen kann also durch die Ausnutzung von Regeln wie $f \cdot f' = 0$ oder $f + f' = 1$ beschleunigt werden.
Durch CEs lässt sich die Komplexität des ITE-Algorithmus demnach quadratisch beschränken, gdw. zwei der Argumente in jedem Aufruf komplementär sind [MT98, S.119-121]. Für alle 16 binäre Operationen (siehe Tabelle 4 auf Seite 39) lässt sich dieses Kriterium anwenden, d. h. wenn es zwei OBDDs G_a, G_b $(a, b \in \mathbb{B}_2)$ mit CEs bezüglich einer Ordnung π gibt, so kann das ROBDD G' für $a \cdot b$ mit CEs mithilfe des ITE-Algorithmus in $O(|G_a||G_b|)$ bestimmt werden. Der zusätzliche Overhead, d. h. Verwaltungsaufwand von CEs bei ITE, ist als minimal zu bewerten [DFE05, S.19-22]. Darüber hinaus wird kein zusätzlicher Speicher benötigt, wenn das niederwertigste Bit (LSB) von jedem Zeiger zur Kodierung des complement-Bits verwendet wird, was über die Bitmaske `(DDNode*) (ddNode & ( ( ( (size_t) 0) >> 2) << 2) )` im Code abgefragt werden kann. Aufgrund dessen, dass alle gängigen Rechner nur gerade Adressen verwenden, entspricht dieses immer 0 und kann demnach dafür benutzt werden. Um das Komplement anzuzeigen, kann also der Fakt ausgenutzt werden, dass Zeiger-Adressen ein Multiplikator von 4 sind, weshalb die zwei niederwertigsten Bits genutzt werden, um Kanteninformationen zu erfassen. Die Not-Operation kann somit durch eine Kontra-

valenz mittels `(ddNode ^ complementEdge)` ausgedrückt werden. Das niederwertigste Bit davon wird wiederum für die Umkehrung benutzt. Im Best Case entspricht die Speicherplatzreduktion daher dem Faktor 2, wofür die Paritätsfunktion (siehe Kapitel 2.1.3 auf Seite 15) ein gutes Beispiel ist [BRB07].
Arbiträre ROBDDs benötigen durchschnittlich $2n - 1$ interne Knoten, wobei für ROBDDs mit CEs n interne Knoten genügen. Auch die Abbildung 23 auf Seite 60 zeigt darüber hinaus eine Knotenreduktion von 50 %. Neben der Speicherplatzersparnis kann so auch leichter erreicht werden, dass der Speicherplatz für einen Knoten ein Teiler der Größe einer Cache-Zeile (kleinste Einheit zur Verwaltung im Cache von CPUs) ist [Sie07, S.47]. Dieses Resultat kann konkret durch die eben angegebene Bitmanipulation erreicht werden, was eine verbesserte Performanz impliziert. Abschließend sei gesagt, wenn ein BDD mit CEs eingesetzt wird, so ist das ROBDD durchschnittlich 7 % kleiner als ein herkömmlicher ROBDD, was durch Experimente in Kapitel 4.3 auf Seite 72 beschrieben ist.
Neben den erläuterten CEs kann der ITE-Operator zudem dadurch verbessert werden, indem der Einsatz von Standard-Tripeln erfolgt. Im folgenden Kapitel soll daher deren Bedeutung erklärt werden.

3.7 Standard-Tripel für ITE

In Kapitel 3.3.1 auf Seite 38 wurde der ITE-Operator diskutiert, indem unter anderem eine kubische bzw. quadratische Laufzeit bestimmt werden konnte. Darüber hinaus gibt es jedoch noch weitere Maßnahmen, um die Effizienz zu steigern, indem mehrdeutige ITE-Aufrufe hinsichtlich der Operanden für dieselbe Berechnung standardisiert werden.
Für den ITE-Operator gibt es Parameter f_1, f_2, f_3 und g_1, g_2, g_3, sodass $ite(f_1, f_2, f_3) = ite(g_1, g_2, g_3)$ gilt, wobei $\exists i : f_i \neq g_i$. Also kann auf der Menge von drei Funktionen f_1, f_2, f_3 bezüglich der Booleschen Funktion $ite(f_1, f_2, f_3)$ eine Äquivalenzrelation definiert werden, d. h. eine Relation, die reflexiv sowie symmetrisch und transitiv (siehe Kapitel 2.1.1 auf Seite 12) ist. Das Ziel dabei ist nun, Kombinationen von Operatoren – die zum gleichen Ergebnis führen – zu einer Äquivalenzklasse zusammenzufassen und einen Repräsentanten auszuwählen, der zur Durchführung einer ITE-Operation benutzt wird [Het02, S.40].
Dadurch kann die Berechnung beschleunigt bzw. wiederholte Berechnungen vermieden werden. Weiterhin kann aber auch die CT (siehe Kapitel 3.5 auf Seite 55) klein bzw. effizient gehalten werden [Sie07, S.48]. Wenn sich davon abstrahiert bspw. ein Eintrag (v_1, v_2) bezüglich $\vee$ darin befindet, so sollte dieser auch gefunden werden, wenn ein Knoten für (v_2, v_1) benötigt wird. Wie bereits in Tabelle 4 auf Seite 39 gezeigt, können alle praktisch eingesetzten binären Operatoren mithilfe des ITE-Operators dargestellt

werden. Demzufolge genügt in diesem Zusammenhang eine CT für diesen Operator.
Im Hinblick auf den ITE-Algorithmus (siehe Code 6 auf Seite 42) werden bei jedem Aufruf von $ite(f_1, f_2, f_3)$ zunächst die Standard-Argumente g_1, g_2, g_3 substituiert, bevor die Operationen `hasNext` bzw. `insert` bezüglich der CT aufgerufen werden. Die UT (siehe Kapitel 3.4 auf Seite 47) gewährleistet eine strenge Kanonizität, weiterhin werden CEs (siehe Kapitel 3.6 auf Seite 57) eingesetzt. Durch diese Methoden kann effizient bestimmt werden, ob zwei berechnete Boolesche Funktionen äquivalent sind [BRB07]. Somit gilt bspw. $f + g \Leftrightarrow ite(f, f, g) = ite(f, 1, g) = ite(g, 1, f) = ite(g, g, f)$. Um das jeweilige *Standard-Tripel* zu bestimmen, wird zunächst eine Umformung hinsichtlich der Parameter vorgenommen, wenn es möglich ist. Diese können wie folgt substituiert werden:

$$ite(f, f, g) \Rightarrow ite(f, 1, g)$$
$$ite(f, g, f) \Rightarrow ite(f, g, 0)$$
$$ite(f, g, f') \Rightarrow ite(f, g, 1)$$
$$ite(f, f', g) \Rightarrow ite(f, 0, g)$$

Hierbei kennzeichnet f' eine CE. Um dementsprechend eine Umformung wie $ite(f, g, f') \Rightarrow ite(f, g, 1)$ machen zu können, muss für zwei Funktionen effizient getestet werden, ob eine Funktion die Negation der anderen ist. Dies kann – wie bereits beschrieben – über CEs erfolgen.
Im Anschluss daran werden folgende Paare als Repräsentanten aus den Äquivalenzklassen benutzt:

$$ite(f, 1, g) \equiv ite(g, 1, f)$$
$$ite(f, g, 0) \equiv ite(g, f, 0)$$
$$ite(f, g, 1) \equiv ite(g', f', 1)$$
$$ite(f, 0, g) \equiv ite(g', 0, f')$$
$$ite(f, g, g') \equiv ite(g, f, f')$$

Durch die Aufzählung aller möglichen Tripel für Operanden der Menge $\{f, f', g, g', h, h', 0, 1\}$ lassen sich hierbei die Äquivalenzklassen bestimmen bzw. anschließend die eindeutigen Repräsentanten auswählen. Hinsichtlich der Implementierung werden beide Schritte in einer Funktion `standardize` als Code 11 zusammengefasst:

```
1 program standardize()
2   input: OBDD f, OBDD g, OBDD h, complementEdge
3   output: Standardisierung
4   // Identische Regeln
5   if (f = g) then
6     g := terminal(1)
```

```
7   else if (f = h) then
8     h := terminal(0)
9   else if (f = !h) then
10    terminal(1)
11  else if (f = !g) then
12    g := terminal(0)
13  end if
14  // Symmetrische Regeln
15  if ( g = terminal(1) ) then
16    if ( index(f) > index(h) ) then
17      swap(f, h)
18    end if
19  else if ( g = terminal(0) ) then
20    if ( index(f) > index(h) ) then
21      swap(f, h)
22      f = !f
23      h = !h
24    end if
25  else if (g = !h) then
26    if ( index(f) > index(g) ) then
27      swap(f, g)
28      h = !g
29    end if
30  else if ( h = terminal(1) ) then
31    if ( index(f) > index(g) ) then
32      swap(f, g)
33      f = !f
34      g = !g
35    end if
36  else if ( h = terminal(0) ) then
37    if ( index(f) > index(g) ) then
38      swap(f, g)
39    end if
40  end if
41  // CE-Regeln
42  if ( isComplementEdge(g) ) then
43    swap(g, h)
44    f = !f
45  end if
46  if ( isComplementEdge(g) ) then
47    g = !g
48    h = !h
49    complementEdge = !complementEdge
50  end if
51 end program
```

Code 11: Implementierung von `standardize`

Die Swap-Operation ist dafür zuständig, benachbarte Variablen zu vertauschen, wobei diese Prozedur eine lokale Operation kennzeichnet, in der nur die jeweiligen Knoten betrachtet werden, die mit diesen Variablen markiert sind. Um bspw. zwischen $ite(f, 1, g)$ und $ite(g, 1, f)$ den eindeutigen Repräsentanten zu bestimmen, wird das erste Argument mit der kleinsten top-Variable ausgewählt. Sollte es mehrere geben, so bestimmt der kleinste Adresszeiger der Knoten die Auswahl.
Wenn von CEs ausgegangen wird, so bestehen die Äquivalenzen
$ite(f, g, h) = ite(f', h, g) = ite(f, g', h')' = ite(f', h', g')'$, wobei f, g, h vereinfachte Parameter sind. Ein eindeutiger Repräsentant wird aus einem der vier Ausdrücke bestimmt, wobei – damit übereinstimmend – der ersten beiden Parameter von ITE keine CEs darstellen dürfen.
Diese Regeln können effektiv mithilfe der De Morganschen Gesetze (siehe Kapitel 2.1 auf Seite 11) bestimmt werden. Angenommen, l und h sind reguläre Kanten und es wird zunächst $l + h$ berechnet, so entsteht als Resultat $ite(l, 1, h)$. Wenn später $l' \cdot h'$, d. h. $ite(l', h', 0)$ berechnet wird, so resultiert $ite(l, 1, h)'$. Die CT hält also ein Resultat, was vor der Rückgabe lediglich komplementiert werden muss. Genauso können redundante Berechnungen wie z. B. $f + f' = ite(f, 1, f') = ite(f, 1, 1) = 1$ ermittelt werden.
Neben den bisher erläuterten Techniken der Hashtabellen, komplementären Kanten und Standardtripeln zur Verbesserung des ITE-Algorithmus, soll im nächsten Abschnitt die Betrachtung der Knoten als Speicherblöcke erfolgen bzw. wie mit Knoten umgegangen werden soll, die für die ITE-Synthese nicht mehr benötigt werden.

3.8 Die Speicherbereinigung

Konkret betrachtet, entsprechen Knoten bzw. Kanten Speicherblöcken, wobei diese über Referenzen angesprochen werden können. Allerdings finden Manipulationen darauf in zufälligen Zugriffsmustern statt, die bei der Implementierung einer Speicherbereinigung beachtet werden müssen [Het02, S.64]. Grundsätzlich wird Speicher während der ROBDD-Anwendung den Knoten zugewiesen, damit sie manipuliert werden können. Wenn Knoten nicht mehr benötigt werden, so muss die Freigabe des Speichers erfolgen, damit diese Speicherstellen neu besetzt werden können. In diesem Zusammenhang muss der jeweilige eingesetzte Rechner bezüglich der Speicherverwaltung betrachtet werden. Bei einer Computerarchitektur gibt es hinsichtlich der Speicherorganisation mehrere Komponenten, die im Hinblick auf die Größenordnung und Zugriffsgeschwindigkeit hierarchisch angeordnet sind:
Eine Speicherhierarchie trägt maßgeblich dazu bei, dass Daten effizient gelesen bzw. geschrieben werden können. Wie anhand der Abbildung 24 auf Seite 65 ersichtlich ist, wird die Performanz der Komponenten besser, je näher sich diese bei der CPU befinden, wobei auch dahingehend die Kosten steigen. Allerdings ist jedoch bspw. die

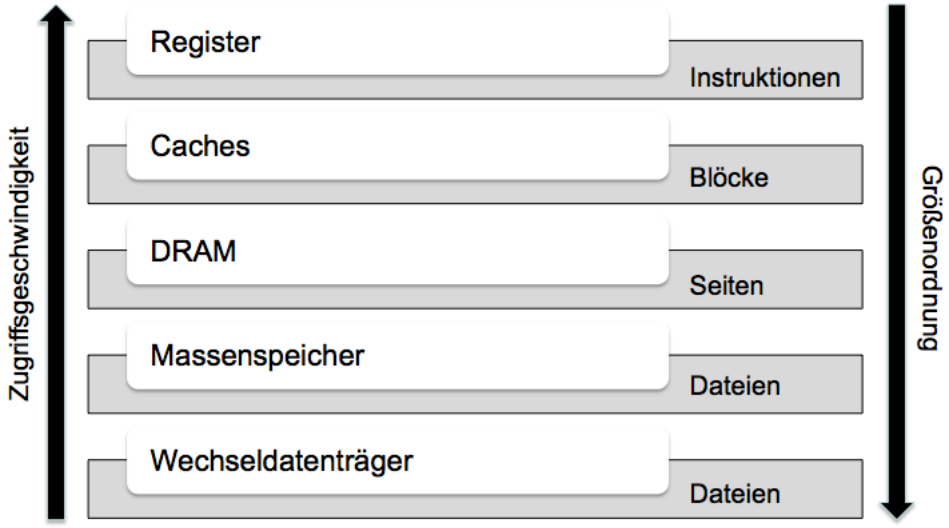

Abbildung 24: Die Speicherorganisation einer Computer-Architektur

Zugriffsgeschwindigkeit bei Caches wie dem Level-1-Cache (L1) oder Level-2-Cache (L2) schneller als beim Hauptspeicher. Die grundlegende Taktik sollte also sein, dafür Sorge zu tragen, dass wesentliche Inhalte als Kopien des DRAM in den Cache transferiert werden. Offensichtlich können somit auch Cache-Misses reduziert werden.
Die beschriebene Implementierung verfolgt den Ansatz einer automatischen Speicherbereinigung, wobei in den jeweiligen Konstruktoren (siehe Abbildung 14 auf Seite 35) mittels `new` Speicher allokiert bzw. mithilfe von `delete` in den Destruktoren dieser wieder freigegeben wird.
Wenn OBDDs konstruiert werden, so wird es häufiger vorkommen, dass berechnete Zwischenergebnisse nicht mehr benötigt werden, d. h. dass diese Knoten gelöscht werden können, womit auch die Nachfolger nicht mehr benötigt werden (insofern es nicht noch andere Zeiger darauf gibt). Um effizient zu ermitteln, ob ein Knoten noch benötigt wird, kann ein Zähler `id` für Referenzen realisiert werden. Jeder Knoten v des Typs `DDNode` (siehe Kapitel 3 auf Seite 33) hat einen Referenzzähler als Indikator für die Anzahl der Referenzen auf v. Bei dessen Initialisierung gilt der Wert 1. Um einen Overflow – d. h. eine Überschreitung des darstellbaren Zahlenbereiches – zu verhindern, gilt ein Maximum von 65535 (umfasst zwei Bytes). Sollte dieses erreicht werden, so wird der Zähler beim späteren Löschen von Verweisen nicht wieder verringert. Der damit zusammenhängende Knoten würde also nicht mehr gelöscht werden, was einen Kompromiss zwischen der kompakten Darstellung und dem Speicherverbrauch ist. Sollte eine Formel bezüglich mehrerer aufgebauter Knoten freigegeben werden, so wird der Referenzzähler bei dem jeweiligen Knoten $f = (v, g, h)$ dekrementiert. Wenn der Zähler von f dann 0 entspricht, so wird dieser Knoten als *tot* markiert, andernfalls gilt er als *lebendig*. Die Referenzzähler bei den Knoten g, h werden diesbezüglich dann ebenfalls rekursiv verringert. Damit diese Prozedur automatisch erfolgen kann, wird der `DDNode` in den `BDDNode` eingehüllt. Somit erfolgt nicht nur eine automatische Inkrementierung

der `id`, sondern auch eine automatische Dekrementierung. Der `BDDNode` hält also einen Zeiger auf den `DDNode` und es existiert eine bidirektionale Assoziation. In diesem Zusammenhang gibt es zwei Arten von Referenzzählern [Sch97, S.31]:

- **Interne Referenzzähler**: Sollte ein Wurzelknoten r verwendet werden, so wird der interne Zähler von r inkrementiert, um sicherzustellen, dass dieser nicht gelöscht werden kann. Wird hingegen r an einer Stelle im SBDD (siehe Kapitel 3.1 auf Seite 36) nicht mehr benötigt, so erfolgt dementsprechend eine Inkrementierung.
- **Externe Referenzzähler**: Wenn eine Operation dem Anwender einen ROBDD zurückgibt, dann wird der externe Zähler inkrementiert. Dieser ist nun solange im Speicher vorhanden, bis der Anwender ihn explizit freigibt.

Die lebendigen Knoten werden dabei von der UT (siehe Kapitel 3.4 auf Seite 47) verwaltet. Wenn ein Knoten stirbt, so wird er demnach aus der Tabelle entfernt, indem der dazugehörige Speicher freigegeben wird. Die Effektivität der Speicherverwaltung wird im nächsten Kapitel 4 gezeigt.

4 Experimentelle Ergebnisse

In diesem Kapitel werden die experimentellen Resultate dieses Softwarepakets präsentiert. Um das Speicherverhalten bzw. die Laufzeit zu bewerten, werden komplexere kombinatorische Schaltungen als Benchmarks von ISCAS'85 in Form von sog. Traces hinzugezogen [Bry88]. Ein *Trace* kennzeichnet dabei eine Menge von Aufrufen in dem Paket, die durch einen *Trace-Treiber* hinsichtlich der aufgerufenen Operationen bei anderen Paketen wiederholt werden können. Somit ist also auch ein Vergleich zwischen verschiedenen Paketen möglich. Insgesamt umfassen die Schaltungen ALUs, Addierer, Komparatoren und Multiplizierer, wobei diese durch verschiedene Nummern charakterisiert werden. Bei den jeweiligen Paket-internen Tests wurden Multiplizierer betrachtet, wobei diese durch „C6288-x" bezeichnet sind. Dabei steht „x" für die Bits, womit bspw. „C6288-16" für einen 16-bit Multiplizierer steht, der 32 Ein- und Ausgänge sowie 2406 Gatter besitzt. Konkret wurde $x \in \{8, 9, 10, 11, 12, 13, 14\}$ betrachtet, da bezüglich $x > 14$ immer die festgelegte Zeit überschritten wurde. Bei einem späteren Vergleich mit dem Softwarepaket *CUDD* (siehe Kapitel 4.5 auf Seite 74) wurden weiterhin ALUs, Addierer sowie Komparatoren hinzugezogen, die in Tabelle 6 ersichtlich sind:

Tabelle 6: ISCAS'85 Schaltkreise

Bezeichnung	**Funktion**
C432	27-Kanal Interrupt Controller
C499/C1355	32-bit SEC
C880	8-bit ALU
C1908	16-bit SEC/DED
C2670	12-bit ALU und Controller
C3540	8-bit ALU
C5315	9-bit ALU
C6288-x	x-bit Multiplizierer
C7552	32-bit Addierer/Komparator

Eine genaue Beschreibung dieser einzelnen Schaltkreise ist im Anhang ab Seite 86 ersichtlich.

Für die durchgeführten internen Tests hinsichtlich der Multiplizierer galt die Variablenordnung $x_{n-1} < x_{n-2} < \cdots < y_{n-1} < y_{n-2} < \cdots < y_0$, wenn von den dazugehörigen Funktionen $X = \sum_{i=0}^{n-1} 2^i x_i$ und $Y = \sum_{i=0}^{n-1} 2^i y_i$ ausgegangen wird. Hinsichtlich dem Vergleich mit CUDD galt für die restlichen Schaltkreise für die Variablenordnung die jeweilige Reihenfolge, in der die Eingaben stehen.

Die Implementierung ist im Wesentlichen schneller als eine einfache Umsetzung der originalen Algorithmen [Bry86], die von Bryant vorgestellt wurden, was in Kapitel 3 auf Seite 33 im Hinblick auf die Analyse der Laufzeiten begründet wurde. Um eventuelle Bugs hinsichtlich Speicherüberläufe, Nullzeiger-Dereferenzierungen, „Out of bounds" usw. festzustellen, wurde außerdem das Tool „CPPCheck" verwendet. Es zeigte bei kei-

nem Bestandteil diesbezüglich Auffälligkeiten. Darüber hinaus wurden Unittests mit „Catch“ hinsichtlich der Klassen durchgeführt, um Fehler der berechneten Ergebnisse auszuschließen. Damit zusammenhängend wurde explizit das Verhalten des Referenzzählers sowie die Korrektheit der Tabellen und Synthesen überprüft. Die Tests wurden dabei in Szenarios gruppiert und folgen dem Stil von Kausalketten. Ein Auszug des daraus resultierenden Reports ist im Anhang auf Seite 83 ersichtlich.
Neben dem Test der implementierten Techniken des Paketes erfolgte – wie bereits erwähnt – außerdem ein Vergleich hinsichtlich der Performanz mit dem sequenziellen Paket CUDD. Dieses C-Paket wurde von Fabio Somenzi im Jahre 1996 entwickelt und wird in verschiedenen Model-Checkern verwendet sowie stetig überarbeitet. Die bemerkenswerte Eigenschaft ist dabei eine große Sammlung von Algorithmen zur Verbesserung der Variablenordnung. Es basiert – im Gegensatz zu diesem Softwarepaket – auf Zeiger und ist eines der effizientesten und wohl meist eingesetzten Pakete weltweit [Jan03].
Die Bibliothek CUDD ist weiterhin im Gegensatz zu anderen Paketen wie *JDD* oder *BuDDy* stabiler im Umgang mit der Synthese, was beim Model Checking herausgefunden werden konnte [Kra17]. Zudem nutzt diese Bibliothek ähnliche Umsetzungen der Techniken, welche in dieser Arbeit beschrieben worden sind und eignet sich – wie bereits erwähnt – auch für das Model Checking, da bspw. das relationale Produkt bezüglich BDDs bestimmt werden kann [McM92].
Die Arbeitsumgebung für die Experimente umfasste eine PC-Workstation (macOS Sierra, Intel i5 2,4 GHz CPU, 3 MB Cache und 16 GB RAM). Die initiale Größe für den Cache und die Hashtabelle betrug gemäß Kapitel 3.5 auf Seite 55 approximiert 500.009 Knoten. Somit handelt es sich um eine Primzahl, was für die durchgeführten Tests im Hinblick auf die Hashverfahren (siehe Kapitel 3.4.1 auf Seite 47) wichtig war. Für die Variablenordnung wurde immer dieselbe Ordnung für die Ausgaben genutzt. Die CPU-Zeit wird in Sekunden angegeben, der benutzte Speicher hingegen in MB. Zudem wurden immer zehn Versuche gemacht, wovon der durchschnittliche Wert (bis auf eine Stelle nach dem Komma gerundet) ermittelt wurde. Das dazugehörige Verhältnis wird in „TR“ (Time Ratio) bzw. „MR“ (Memory Ratio) angegeben, wobei diese Resultate bis auf zwei Stellen nach dem Komma gerundet sind. Die maximale Ausführungszeit wurde auf 30 Minuten, d. h. 1800 Sekunden eingestellt. Sollte eine Zeitüberschreitung vorliegen, so wird dies durch „TO“ gekennzeichnet, was für „Time Over“ steht. Eine Speicherüberschreitung wird hingegen durch „MO“ beschrieben, was für „Memory Over“ steht. Hinsichtlich der Gesamtbewertung wird dies als festgelegter Maximalwert gewertet. Wenn es nicht anders erwähnt ist, so gilt hinsichtlich des Hashings das Modulo-Verfahren (siehe Kapitel 3.4.1 auf Seite 49) mit der Funktion $h(f, g, h) = ((g + h) >> f) \; mod \; m$ sowie als Kollisionsstrategie die Verkettung mit Überläufern (siehe Kapitel 3.4.2 auf Seite 53). Hierbei wurden die Knoten in einer Kollisionskette bzw. Vektor zufällig miteinander verbunden. Bezüglich der CT wurde

keine Kollisionsstrategie verwendet. Darüber hinaus wurde generell ein ROBDD mit komplementären Kanten (siehe Kapitel 3.6 auf Seite 57) aufgebaut, die Standardisierung (siehe Kapitel 3.7 auf Seite 61) war eingeschaltet und es galt eine auf Indizes basierende Arbeitsweise.

4.1 Management der UT

Die Wahl der jeweiligen Hashfunktion (siehe Kapitel 3.4 auf Seite 47) ist ein entscheidender Faktor, wie groß der Aufwand der Synthesen ist. In Kapitel 3.4.1 auf Seite 47 wurden zwei mögliche Varianten vorgestellt, nämlich das Modulo- und Multiplikations-Verfahren, die untersucht worden sind. Die Hashfunktion für das Multiplikations-Verfahren war gemäß Kapitel 3.4.1 auf Seite 49 durch $h(g,h) = ((g + h \cdot 12.582.917) \cdot 4.256.249) >> (64 - log_2(500.009))$ gekennzeichnet.
Weiterhin wurden in Kapitel 3.4.2 auf Seite 50 Kollisionsstrategien bezüglich der UT vorgestellt, die im Folgenden ebenfalls gegenübergestellt werden. Als offenes Hashing wurde die Variante des doppelten Hashings für die Sondierung verwendet, d. h.
$h'(f,g,h) = (1 + ((g + h) >> f))\ mod\ m$ bzw. $h'(g,h) = (1 + ((g + h \cdot 12.582.917) \cdot 4.256.249)) >> (64 - log_2(500.009))$ und $s(j,k) = j \cdot h'(k)$, womit sich beide Hashfunktionen voneinander unterschieden haben. Folgende Tabellen zeigen die daraus resultierenden Ergebnisse:

Tabelle 7: Modulo-Verfahren mit Kollisionsstrategien

Schaltung	**Offenes Hashing**		**Verkettung von Überläufern**			
	Speicher	**CPU-Zeit**	**Speicher**	**CPU-Zeit**	**MR**	**TR**
C6288-8	31,5	0,9	23,1	0,6	1,36	1,50
C6288-9	43,5	3,4	34,1	2,6	1,28	1,31
C6288-10	75,9	11,5	63,9	8,3	1,19	1,39
C6288-11	201,3	40,7	139,2	32,9	1,45	1,24
C6288-12	588,2	187,5	465,9	151,3	1,26	1,24
C6288-13	1.304,9	701,7	1.175,2	563,6	1,11	1,25
C6288-14	3.984,1	TO	3.573,3	1.775,1	1,11	> 1,01
Gesamt	6.229,4	> 2.745,7	5.474,7	2.534,4	1,14	> 1,08

Tabelle 8: Multiplikations-Verfahren mit Kollisionsstrategien

Schaltung	**Offenes Hashing**		**Verkettung von Überläufern**			
	Speicher	**CPU-Zeit**	**Speicher**	**CPU-Zeit**	**MR**	**TR**
C6288-8	33,4	1,0	22,3	0,5	1,50	2,0
C6288-9	42,8	3,4	37,3	2,8	1,15	1,21
C6288-10	81,3	12,1	67,0	8,9	1,21	1,36
C6288-11	210,1	42,2	145,1	33,5	1,45	1,26
C6288-12	599,6	193,5	458,9	148,1	1,31	1,31
C6288-13	1.295,3	698,2	1.183,5	568,2	1,01	1,23
C6288-14	4.040,8	TO	3.580,9	1.778,7	1,13	> 1.01
Gesamt	6.303,3	$> 2.750,4$	5.495,0	2.540,7	1,15	$> 1,08$

Aus der Tabelle 8 mit dem Modulo-Verfahren geht hervor, dass das Speicherverhalten als auch die Laufzeit zum Aufbau des ROBDDs mit der Verkettung von Überläufern als Kollisionsstrategie besser ist. So wurden hierbei mehr als 211,3 Sekunden weniger benötigt, was einer Zeitersparnis von ≈ 14 % entspricht. Bezüglich der Speichernutzung wurden 757,7 MB weniger beansprucht, was wiederum eine Ersparnis von ≈ 8 % kennzeichnet. Typisch für ROBDD-Anwendungen ist, dass es häufig negative Resultate bezüglich Knotenexistenzprüfungen gibt. Erfolglose Suchen terminieren darüber hinaus erst, sobald eine freie Stelle gefunden werden konnte. Damit ist die längere Laufzeit bzw. das größere Speicheraufkommen beim offenen Hashing zu erklären. Dagegen gibt es bei der Verkettung mit Überläufern nur vier Elemente pro Knoten (siehe Kapitel 3.4.3 auf Seite 54), womit der Overhead zur Allokation vermindert wird. Dadurch, dass Knoten bei Kollisionen außerhalb der jeweiligen Hashtabelle gespeichert werden, gibt es zudem weniger Platzverschwendung für unbelegte Einträge.
Weiterhin wurde – bezogen auf das Multiplikations-Verfahren gemäß Tabelle 7 auf Seite 69 – bei der Variante mit Verkettung von Überläufern 808,3 MB weniger an Speicher benötigt, was einer Einsparung von ≈ 15 % im Vergleich zum offenen Hashing entspricht. Zudem benötigte diese Variante 209,7 Sekunden weniger, um den ROBDD aufzubauen. Dies entspricht einer Zeiteinsparung von ≈ 8 %. Die Begründung ist analog zum Modulo-Verfahren zu betrachten. Bezüglich einer ROBDD-Anwendung ist also eine Verkettung von Überläufern dem offenen Hashing vorzuziehen.
Ein direkter Vergleich – hinsichtlich der Verkettung von Überläufern – zwischen dem Modulo- und Multiplikations-Verfahren ergibt, dass das Modulo-Verfahren insgesamt gesehen 20,3 MB weniger Speicher benötigte, um die ROBDDs aufzubauen. Zudem gab es eine Zeitersparnis von 6,3 Sekunden. Der Divisionsblock der ALU ist eine zeitintensive Operation. Allerdings entspricht bspw. ein Rechts-Shift um i Stellen einem Divisor 2^i für ein $i \in \mathbb{N}$, wobei beim Modulo-Verfahren auch die höheren Bits zur Adressberechnung benutzt werden können, wenn für m eine Primzahl gewählt wird. Bei der Multiplikation hingegen wird innerhalb der ALU eine Fließkommazahl eingesetzt, um irrationale Zahlen zu verarbeiten. Ganze Zahlen im Rechner können als Bruchzahlen mit einem Dezimalpunkt vor der höchstwertigen Ziffer betrachtet werden, wobei für m dann eine Zweierpotenz gewählt wird. Wird hierbei für x der goldene Schnitt be-

stimmt, so ergibt sich ebenfalls eine Gleichverteilung der Knoten, womit insgesamt der Unterschied marginal ausfällt.

4.2 Management der CT

In Kapitel 3.5 auf Seite 55 wurde die CT diskutiert, nämlich als sog. hashbasierter Cache bzw. Cache mit Kollisionsstrategie, wodurch der Speicherplatz/Rechenaufwand der Synthese verringert werden soll. Zunächst einmal wurde untersucht, inwieweit der Einsatz einer CT die Synthese bezüglich der Laufzeit und dem Speicherverhalten verbessert, was anhand folgender Tabelle 9 ersichtlich ist:

Tabelle 9: Inaktive und aktive CT

Schaltung	**Inaktive CT**		**Aktive CT**			
	Speicher	**CPU-Zeit**	**Speicher**	**CPU-Zeit**	**MR**	**TR**
C6288-8	1118,9	29,7	23,9	0,7	46,82	42,43
C6288-9	1499,7	125,9	32,9	2,5	45,58	50,36
C6288-10	3.107,1	413,5	63,6	8,3	48,85	49,82
C6288-11	7.209,8	TO	138,3	32,3	52,13	55,95
C6288-12	MO	TO	469,1	152,8	$> 34{,}11$	$> 11{,}78$
C6288-13	MO	TO	1.179,1	568,9	$> 13{,}57$	$> 3{,}16$
C6288-14	MO	TO	3.580,1	1.783,5	$> 4{,}47$	$> 1{,}01$
Gesamt	$> 60.935{,}5$	$> 7.769{,}1$	5487,0	2549,0	$> 11{,}11$	$> 3{,}05$

Aus dem direkten Vergleich geht hervor, dass der Einsatz eines Caches alle ROBDDs aufstellen konnte. Wird kein Cache eingesetzt, so war zu erkennen, dass der Speicher bei den Schaltkreisen „C6288-12“, „C6288-13“ und „C6288-14“ voll ausgelastet wurde bzw. die Berechnung nicht fertiggestellt werden konnte. Bei „C6288-11“ hingegen konnte nur die Berechnung nicht in der gemessenen Zeit fertiggestellt werden. Wenn z. B. der Schaltkreis „C6288-10“ betrachtet wird, so kann eine Zeitersparnis – durch den Einsatz eines Caches – von 405,2 Sekunden festgestellt werden. Der Einsatz eines Cache ist hier somit rund 50 Mal schneller gewesen. Demnach ist der Einsatz eines Caches insgesamt mehr als lohnenswert.
Für den weiteren Vergleich bezüglich der Cache-Arten wurde die Kollisionsstrategie der Verkettung von Überläufern eingesetzt, die analog zur UT (siehe Kapitel 3.4.2 auf Seite 53) betrachtet werden kann. Hierbei werden also Knoten nicht direkt überschrieben, sondern in einer Liste als Nachfolger gespeichert. Die jeweiligen Ergebnisse sind in der nachfolgenden Tabelle 10 auf Seite 72 ersichtlich:

Tabelle 10: Hashbasierter Cache und Cache mit Kollisionsstrategie

Schaltung	**Cache mit Kollisionsstrategie**		**Hashbasierter Cache**			
	Speicher	**CPU-Zeit**	**Speicher**	**CPU-Zeit**	**MR**	**TR**
C6288-8	26,5	0,6	22,9	0,6	1,16	1,0
C6288-9	38,1	2,3	38,8	2,9	0,98	0,79
C6288-10	71,9	8,0	64,9	8,8	1,11	0,91
C6288-11	153,8	30,5	143,9	33,2	1,07	0,92
C6288-12	513,4	145,2	452,9	149,9	1,13	0,97
C6288-13	1.512,6	542,8	1.183,5	568,2	1,28	0,96
C6288-14	3.998,7	1.728,0	3.580,9	1.778,7	1,12	0,97
Gesamt	6.315,0	2.457,4	5.487,8	2.542,3	1,15	0,97

Wenn demzufolge ein Cache mit Kollisionsstrategie eingesetzt wird, so werden im Gegensatz zum hashbasierten Cache 84,9 Sekunden, d. h. $\approx$ 3 % weniger benötigt, die ROBDDs aufzubauen. Dies liegt vor allem daran, dass es mehr Cache-Misses gibt, weshalb neue Berechnungen stattfinden müssen. Konkret betrachtet gab es $\approx$ 14,2 % mehr ITE-Aufrufe. Jedoch benötigte der hashbasierte Cache 827,2 MB bzw. 15 % weniger Speicher zum Aufbau. Hierbei brauchte der Cache mit Kollisionsstrategie insbesondere mehr Verarbeitungszeit für die Operation `hasNext` (siehe Code 6 auf Seite 42) sowie die damit zusammenhängende Speicherbereinigung. Gemäß Kapitel 3 auf Seite 33 muss der Speicher besonders optimiert werden, da dieser ein Hauptproblem im Umgang mit ROBDDs darstellt. Weiterhin wird durch die Approximation der Größe des Caches dem Fall entgegengewirkt, dass der ITE-Algorithmus im Worst Case einen exponentiellen Aufwand hat. Aufgrund dem wesentlich geringeren Overhead bezüglich dem Speicher sollte daher der hashbasierte Cache benutzt werden.

4.3 Nutzung von komplementären Kanten

In Kapitel 3.6 auf Seite 57 wurden komplementäre Kanten eingeführt, womit zusätzlich die Negation direkt in einem ROBDD gespeichert wird. Damit hierbei die Kanonizität als Eigenschaft nicht verloren geht, mussten zusätzliche Regeln bzw. Abfragen innerhalb des ITE-Operators (siehe Kapitel 3.3.1 auf Seite 38) eingefügt werden. Daher wurde weiterhin untersucht, inwieweit sich diese Technik auf das Speicherverhalten bzw. die Laufzeit auswirkt. Die Ergebnisse dazu sind in der folgenden Tabelle 11 auf Seite 73 ersichtlich:

Tabelle 11: Nutzung von komplementären Kanten

Schaltung	**Ohne CEs**		**Mit CEs**			
	Speicher	**CPU-Zeit**	**Speicher**	**CPU-Zeit**	**MR**	**TR**
C6288-8	33,9	1,3	24,1	0,7	1,41	1,86
C6288-9	43,0	4,9	36,6	2,7	1,17	1,81
C6288-10	76,3	14,9	64,4	8,7	1,18	1,71
C6288-11	163,8	54,2	145,5	33,9	1,13	1,60
C6288-12	495,2	298,2	463,4	156,1	1,07	1,91
C6288-13	1.412,6	1.021,7	1.193,9	570,5	1,18	1,79
C6288-14	4.050,3	3.378,1	3.569,7	1.772,3	1,13	1,91
Gesamt	6.275,1	4.773,3	5.497,6	2.544,9	1,14	1,88

Aus der Tabelle 11 geht hervor, dass das Speicheraufkommen bei ROBDD mit CEs 777,5 MB weniger benötigte, prozentual ist die Verbesserung demnach auf $\approx$ 14 % beziffert. Dies liegt vor allem daran, dass die ROBDDs mit CEs kleiner sind bzw. weniger Knoten beim damit zusammenhängenden Aufbau benötigen. Weiterhin wird kein zusätzlicher Speicher beansprucht, da der LSB von Zeigern zur Kodierung des complement-Bits benutzt wird. Zudem bestand aber auch ein Zeitgewinn von 2.228,4 Sekunden, d. h. $\approx$ 88 %. Dies liegt hauptsächlich daran, dass komplementäre Operationen in $O(1)$ durchgeführt werden können, da die Negation der jeweiligen Funktion bei der Synthese direkt aufgebaut wird.

4.4 Nutzung von Standard-Tripeln

In Kapitel 3.7 auf Seite 61 wurde die Standardisierung von ITE-Aufrufe vorgestellt, was zu einer Verbesserung der Performanz führen soll. So können Kombinationen bezüglich der Parameter – die zum gleichen Ergebnis führen – in Äquivalenzklassen zusammengefasst werden, wobei dann ein Repräsentant daraus gewählt wird, der das jeweilige Resultat repräsentiert. Die folgende Tabelle 12 zeigt die jeweiligen Ergebnisse, die sich infolge der Experimente ergeben haben:

Tabelle 12: Nutzung von Standard-Tripeln

Schaltung	**Ohne Standardisierung**		**Mit Standardisierung**			
	Speicher	**CPU-Zeit**	**Speicher**	**CPU-Zeit**	**MR**	**TR**
C6288-8	25,1	0,7	23,2	0,7	1,08	1,0
C6288-9	40,5	3,2	35,3	2,7	1,15	1,19
C6288-10	69,4	9,4	62,9	8,6	1,10	1,09
C6288-11	160,9	38,8	149,1	34,5	1,08	1,12
C6288-12	510,3	166,9	470,5	158,5	1,08	1,05
C6288-13	1.261,7	586,0	1.175,2	567,9	1,07	1,03
C6288-14	3.715,8	1.827,7	3.598,4	1.780,1	1,08	1,03
Gesamt	5.783,7	2.632,7	5.514,6	2.553,0	1,05	1,03

Insgesamt geht aus der Tabelle hervor, dass die Berechnung der jeweiligen ROBDDs mit Standardisierung 79,7 Sekunden weniger gedauert hat, was eine Zeiteinsparung

von 3 % bedeutet. Außerdem wurden 269,1 MB weniger verbraucht, d. h. es gab eine Einsparung von 5 %. Die bessere Performanz ist dadurch zu erklären, dass es weniger redundante Berechnungen gibt bzw. die CT kleiner gehalten wird.

4.5 Vergleich mit CUDD

Bei diesem Test wurde die in Kapitel 4 auf Seite 67 vorgestellte CUDD-Bibliothek mit diesem ROBDD-Paket *iBDD* anhand der erwähnten Schaltkreise unter den gleichen beschriebenen Bedingungen verglichen. Die entsprechenden Ergebnisse sind in der nachfolgenden Tabelle 13 zu finden:

Tabelle 13: Vergleich mit CUDD

Schaltung	**iBDD**		**CUDD**			
	Speicher	**CPU-Zeit**	**Speicher**	**CPU-Zeit**	**MR**	**TR**
C6288-8	23,8	0,6	6,28	0,1	3,79	6,0
C6288-9	33,3	2,5	19,8	0,1	13,32	25,0
C6288-10	60,7	8,1	32,6	0,2	1,86	40,5
C6288-11	142,6	34,2	80,6	0,7	1,77	48,86
C6288-12	455,8	148,9	182,6	2,8	2,50	53,18
C6288-13	1.162,8	558,9	499,0	11,0	2,33	50,81
C6288-14	3.580,1	1.779,8	1.563,7	41,3	2,29	43,09
C6288-15	> 11.176,8	TO	4.652,0	118,9	> 2,40	> 15,14
C6288-16	MO	TO	MO	TO	-	-
C17	16,3	0,1	3,2	0,1	5,09	1,0
C432	481,1	449,8	10,9	0,1	44,14	4.498,0
C499	> 31,2	TO	17,0	0,1	> 1,84	> 18.000,0
C880	49,6	1369,4	83,6	0,8	0,59	1.711,75
C1355	> 31,9	TO	17,0	0,1	> 1,88	> 18.000,0
C1908	43,8	478,5	16,3	0,1	2,69	4.785,0
C2670	MO	TO	MO	TO	-	-
C3540	> 1.000,1	TO	170,5	2,1	> 476,22	> 857,14
C5315	MO	TO	MO	TO	-	-
C7552	MO	TO	MO	TO	-	-
Gesamt	> 82.289,9	> 19.230,8	> 23.168,3	> 1.976,3	> 3,55	> 9,73

Insgesamt fällt bei den Schaltkreisen auf, dass iBDD im Vergleich zu CUDD mehr als 59.121,6 MB benötigt hat, um die dazugehörigen ROBDDs aufzubauen, womit CUDD ca. nur $\frac{1}{3}$ von diesem Speicheraufkommen benötigte. Bei dem Schaltkreis „C880“ brauchte iBDD jedoch nur rund 50 % des Speicherplatzes, den CUDD für den ROBDD benutzt hat. Weiterhin fällt auf, dass mit zunehmender Größe der Schaltung das eben genannte Speicherverhältnis von $\frac{1}{3}$ ansonsten gleichbleibend ist. Anders verhält es sich bei der CPU-Zeit, wobei CUDD insgesamt mehr als 17.254,5 Sekunden weniger zum Aufbau der ROBDDs benötigte. Die Zeit, die iBDD dementsprechend – im Vergleich zu CUDD – brauchte, um einen ROBDD zu erstellen, entspricht somit dem Faktor ≈ 10. Diesem Aspekt ist hinzuzufügen, dass sowohl CUDD als auch iBDD mit verschiedenen Schaltkreisen wie z. B. „C7552“ Probleme hatten, die dazugehörigen ROBDDs in der

gemessenen Zeit überhaupt aufzustellen. Zudem war zu erkennen, dass der Speicher in diesem Kontext vollständig ausgelastet wurde. Je komplexer der Schaltkreis war, desto größer wurde der Abstand zu CUDD hinsichtlich der CPU-Zeit.
Der Grund hierfür liegt wahrscheinlich in der Speicherverwaltung bzw. an der Speicherart von Nachfolgern in der Kollisionskette der UT. Sollte ein Knoten v (siehe Kapitel 3.2 auf Seite 37) während der Synthese erstellt werden, so gilt der Referenzzähler 1 und der jeweilige Knoten wird in der UT für einen späteren schnelleren Zugriff gespeichert. Sollte eine Kollision stattfinden, so wird der abzuspeichernde Knoten in einem Vektor des schon vorhandenen Knotens gespeichert. Wird dieser Knoten an anderen Stellen wie z. B. $a.low$ verwendet, so wird der Referenzzähler inkrementiert. Wenn eine Formel freigegeben wird, so wird der Referenzzähler bei den damit zusammenhängenden aufgebauten Knoten dekrementiert. Die Frage besteht darin, wann ein Zähler inkrementiert werden muss, was bei CUDD manuell geschieht und anfällig für Fehler ist. Wie bereits erwähnt, funktioniert ein Überladen des Zeigers nicht, da sowohl $a.low$ und v Zeiger sind. Der Referenzzähler kann jedoch dadurch automatisch verwaltet werden, dass der Zeiger in eine Klasse eingehüllt wird und Operationen der Knoten darüber abgefragt werden. Diese Variante ist weniger fehleranfällig, wobei ebenfalls die Annahme getroffen wurde, dass das Finden eines Knotens in dem Vektor aufgrund des Lokalitätsprinzips schneller funktioniert, da der nächste Block darin gleichzeitig mit in den Cache geladen wird.
Allerdings sind die Operationen *Einfügen* und *Löschen* beim Vektor problematisch, weil diese – im Gegensatz zu einer verketteten Liste – einen linearen Aufwand erfordern, was während des Prozesses langsamer ist. Bezüglich dem Prozess muss zusätzlich der interne Speicher betrachtet werden. Ein Prozess verwaltet seinen Speicher selber. Sollte nicht mehr genug prozessinterner freier Speicher verfügbar sein, so wird dieser vom Betriebssystem (OS) explizit angefordert und ein Verweis darauf zurückgegeben. Ein OS wie bspw. Linux bietet mithilfe des Buddy-Algorithmus nur Blöcke der Größe 2^k an [Spi08]. Dies wird vor allem deshalb derartig realisiert, um die externe Fragmentierung zu reduzieren und das Verschmelzen von Speicherblöcken zu vereinfachen. Wenn ein Knoten 22 Bytes benötigt und es ist kein prozessinterner freier Speicher mehr verfügbar, muss die Anwendung daher einen Speicherblock allokieren. In diesem Fall würden dann $2^5 = 32$ Bytes reserviert werden, wobei eine interne Fragmentierung von 10 Bytes besteht. Wenn die Anwendung nun mit Millionen von Knoten arbeitet, so würde nicht nur die interne Fragmentierung weiter ansteigen, sondern es gibt auch eine Erhöhung der Cache-Misses, da aufgrund des zufälligen Zugriffsmusters Knoten verteilt auf dem Hauptspeicher (DRAM) liegen.

5 Zusammenfassung und Fazit

In dieser Arbeit wurde eine effiziente Implementierung eines Softwarepaketes vorgestellt, um Boolesche Funktionen - in Form von ROBDDs - zu manipulieren, was für viele Applikationen bezüglich symbolischer Simulationen oder Model Checking wichtig ist. ROBDDs eignen sich hierzu besser als andere Darstellungen wie KNFs/DNFs, da sie weniger Speicher benötigen und praktische Funktionen wie die Paritätsfunktion effizient darstellen können (siehe Kapitel 2.1.3 auf Seite 15). Darüber hinaus sind Anforderungen wie der Äquivalenztest oder SAT in konstanter Zeit zu bewältigen, die in Kapitel 2.4 auf Seite 24 diskutiert worden sind.
Die Implementierung ist im Wesentlichen schneller als eine einfache Implementierung der originalen Algorithmen [Bry86], die von Bryant vorgestellt wurden, was in Kapitel 3 auf Seite 33 durch algorithmische Analysen gezeigt werden konnte. Damit zusammenhängend ist besonders das Speicherverhalten optimiert worden, da bereits kleinere Optimierungen darüber entscheiden können, ob ein ROBDD überhaupt aufgebaut werden kann. Unittests und statische Code-Analysen haben zudem gezeigt, dass die implementierten Techniken fehlerfrei sind.
Dementsprechend führt der Einsatz von SBDDs (siehe Kapitel 3.1 auf Seite 36) dazu, dass Berechnungen schneller stattfinden können bzw. das Speicheraufkommen geringer ist. Durch die Nutzung einer dynamischen CT als Cache (siehe Kapitel 3.5 auf Seite 55) wird weiterhin der Rechenaufwand/Speicherplatz der Synthese (siehe Kapitel 3.3 auf Seite 38) verringert, da isomorphe Teilgraphen nicht öfter betrachtet werden müssen, was in Kapitel 4 auf Seite 67 gezeigt werden konnte. Mit dem Einsatz von dynamischen UTs (siehe Kapitel 3.4 auf Seite 47) wird hingegen die Kanonizität gesichert. Durch die richtige Wahl der Hashfunktion für die CT bzw. Kollisionsstrategie für die UTs kann die Synthese polynomiell beschränkt, infolge der Tautologieprüfung können wiederum weitere Aufrufe diesbezüglich gespart werden.
Weitere Verbesserungen wie komplementäre Kanten (siehe Kapitel 3.6 auf Seite 57), Standard-Tripel (siehe Kapitel 3.7 auf Seite 61) sowie die Zusammenfassung von Knoten/UTs in eine Datenstruktur (siehe Kapitel 3.4.3 auf Seite 54) führen außerdem dazu, dass das Speicherverhalten sowie die Performanz im Vergleich effizient sind, was durch kleinere BDDs bzw. keine redundanten Berechnungen bedingt wird. Für komplementäre Kanten kann dabei besonders der Umstand von genutzten geraden Adressen in Rechnern ausgenutzt werden, d. h. das LSB von Zeigern wird zur Kodierung dieser Kanten benutzt. Durch derartige und weitere Implementierungstricks kommen demnach auch einige maschinennahe Ansätze zum Einsatz, wodurch CPU-Sekunden nochmals vermindert werden konnten.
Insgesamt bleibt jedoch zu erwähnen, dass das BDD-Paket CUDD wesentlich schneller und auch schonender im Umgang mit dem verwendeten Speicher ist (siehe Kapitel 4.5 auf Seite 74). Der Grund hierfür liegt wahrscheinlich in der Speicherverwaltung

des hier beschriebenen Paketes. Zwar wird der Referenzzähler eines Knotens über eine eingehüllte Klasse automatisch in- und dekrementiert, jedoch dauern – im Hinblick auf die Kollisionskette in Form eines Vektors – die Operationen *Einfügen* und *Löschen* merkbar länger, weil sie eine höhere Komplexitätsklasse besitzen. Hier könnte dazu übergegangen werden, dass die Nachfolger direkt im Knoten, anstelle in der UT abgelegt werden (siehe Abbildung 19 auf Seite 54) und eine verkettete Liste benutzt wird. Darüber hinaus besteht eine insgesamt zu hohe interne Fragmentierung von 10 Bytes pro Knoten. Hier könnte eine Abhilfe sein, dass immer größere Blöcke für Knoten angefordert werden, d. h. dass die Speicherverwaltung innerhalb der Blöcke selbst organisiert wird, sodass eine gezielte Partitionierung der Blöcke für die Knoten erfolgt. Zudem könnte in diesem Zusammenhang realisiert werden, dass die UT nicht sofort tote Knoten löscht, sondern bspw. darauf wartet, bis in der CT ein relativer Anteil von 10 % tot ist. Hierdurch würde zwar ein größeres Speicheraufkommen bestehen, jedoch bestünde eine Entlastung seitens der UT. Außerdem könnte hierbei eine sog. Wiederbelebung in der CT implementiert werden, sodass tote Knoten reaktiviert und somit eine weitere direkte Berechnung damit gemacht werden kann. Auch eine automatische Erweiterung der UT – insofern die Kollisionsrate zu hoch wird – macht in dem Kontext Sinn, wodurch bspw. ein Problem des Rehashings gelöst werden kann [Het02, S.78-80]. Weiterhin konnte festgestellt werden, dass während der Synthese oftmals ein wiederholtes Interesse von demselben Knoten in der Liste bestand. Wenn demzufolge ein solcher Knoten an den Anfang der Liste gestellt wird, kann eine Suche dementsprechend schneller erfolgen.
Abschließend sei hierzu erwähnt, dass das in dieser Arbeit beschriebene Paket natürlich nicht den Funktionsumfang vom langjährig entwickelten CUDD bzw. deren Mechanismen zur Bestimmung einer optimalen Variablenordnung besitzt, kann jedoch dahingehend weiterentwickelt und optimiert werden.

Literaturverzeichnis

[Art10] Artale, A.: *BINARY DECISION DIAGRAMS*. https://www.inf.unibz.it/~artale/FM/slide7.pdf, 2010. – Zuletzt zugegriffen am: 2018-02-21

[Bau08] Baumgartner, J.: *Moore's Law v. Verification Complexy*. http://www.research.ibm.com/haifa/conferences/hvc2008/present/Moores_Law_Verification_Complexity.pdf, 2008. – Zuletzt zugegriffen am: 2018-02-25

[BMH84] Brayton, R. K. ; McMullen, C. ; Hachtel, A. G. D. und Sangiovanni-Vincentelli: Logic Minimization Algorithms for VLSI Synthesis. (1984)

[BMM01] Becker, B. ; M., Fujita ; Meinel, F. C. und S. C. und Somenzi: *Computer Aided Design and Test: BDDs vs. SAT*. https://www.dagstuhl.de/Reports/01/01051.pdf, 2001. – Zuletzt zugegriffen am: 2018-02-19

[BRB07] Brace, K. S. ; Rudell, R. L. ; Bryant, R. E.: *Efficient Implementation of a BDD Package*. http://www.ccs.neu.edu/home/pete/courses/Decision-Procedures/2007-Fall/readings/Brace_Rudell_Bryant_Efficient_Implementation_BDD_Package.pdf, 2007. – Zuletzt zugegriffen am: 2018-02-16

[Bry86] Bryant, R. E.: Graph-Based Algorithms for Boolean Function Manipulation. (1986)

[Bry88] Bryan, D.: *The ISCAS'85 benchmark circuits and netlist format*. https://ddd.fit.cvut.cz/prj/Benchmarks/iscas85.pdf. Version: 1988. – Zuletzt zugegriffen am: 2018-02-23

[BW99] Bollig, B. ; Wegener, I.: Complexity Theoretical Results on Partitioned Binary Decision Diagrams. 32 (1999), Nr. 4, S. 487–503

[Cas94] Casar, R. A. und M. A. und Meolic: *Representation of Boolean Functions with ROBDDs*. http://research.meolic.com/papers/robdd.pdf, 1994. – Zuletzt zugegriffen am: 2018-02-17

[Cho88] Cho, K.: Test Pattern Generation for Combinational and Sequential MOS Circuits by Symbolic Fault Simulation. (1988)

[CLRS01] Cormen, T. H. ; Leiserson, C. E. ; Rivest, R. L. ; Stein, C.: *Introduction to Algorithms*. Cambridge : The Mit Press, 2001

[DFE05] Drechsler, R. ; Fey, G. ; Ebendt, R.: *Advanced BDD Optimization*. Berlin : Springer, 2005

[EKN15] Emden, R. G. ; Koutsofios, E. ; North, S.: Drawing graphs with dot. (2015). http://www.graphviz.org/pdf/dotguide.pdf. – Zuletzt zugegriffen am: 2018-03-03

[FH78] Fortune, S. ; Hopcroft, E. J. und S. J. und Schmidt: *The Complexity of Equivalence and Containment for Free Single Variable Program Schemes*. Springer, 1978

[Gar15] Garg, L.: *Why Do We Need size_t?* https://prateekvjoshi.com/2015/01/03/why-do-we-need-size_t/. Version: 2015. – Zuletzt zugegriffen am: 2018-03-02

[Gün02] Günther, W.: *Binary Decision Diagrams.* https://ira.informatik.uni-freiburg.de/teaching/verif-2002/Folien/2bdd.pdf, 2002. – Zuletzt zugegriffen am: 2018-02-10

[Hay99] Hayes, J. P.: *ISCAS High-Level Models.* http://web.eecs.umich.edu/~jhayes/iscas.restore/. Version: 1999. – Zuletzt zugegriffen am: 2018-03-02

[Hei14] Hein, J. L.: *Discrete Structures, Logic, And Computability.* Burlington : Jones & Bartlett Learning, 2014

[Het02] Hett, A.: *Binäre Expression-Diagramme.* http://d-nb.info/96453326X/34, 2002. – Zuletzt zugegriffen am: 2018-02-13

[Hof11] Hofferek, G.: *Binary Decision Diagrams (BDDs).* https://www.iaik.tugraz.at/content/teaching/bachelor_courses/logik_und_berechenbarkeit/download/BDDs_handout_LuB.pdf, 2011. – Zuletzt zugegriffen am: 2018-02-20

[IKRR10] Igler, D. ; Krümmel, N. ; Ried, M. ; Renz, B.: *Kurzanleitung UML.* https://homepages.thm.de/~hg11260/mat/uml.pdf. Version: 2010. – Zuletzt zugegriffen am: 2018-03-03

[Jan01] Janssen, G.: *Design of a Pointerless BDD Package.* https://www.research.ibm.com/haifa/projects/verification/SixthSense/papers/bdd_iwls_01.pdf, 2001. – Zuletzt zugegriffen am: 2018-02-22

[Jan03] Janssen, G.: A Consumer Report on BDD Packages. (2003)

[KM07] Kong, S. ; Malec, D.: *Advanced Algorithms.* http://pages.cs.wisc.edu/~shuchi/courses/787-F07/scribe-notes/lecture09.pdf, 2007. – Zuletzt zugegriffen am: 2018-02-20

[Knu11] Knuth, D. E.: *The Art of Computer Programming.* Boston : Addison Wesley, 2011

[Kra17] Krauss, R.: *Performance von BDD-basiertem Model Checking.* https://runekrauss.com/pdf/model_checking.pdf, 2017. – Zuletzt zugegriffen am: 2018-02-24

[LL92] Liaw, H. T. ; Lin, C. S.: On the OBDD-representation of general Boolean functions. 41 (1992), Nr. 6, S. 661–664

[Loo07] Loogen, R.: *Schaltfunktionen und ihre Darstellung.* http://www.mathematik.uni-marburg.de/~loogen/Lehre/ws07/TechInf1/Folien/TechInf1Lo02Folien.pdf, 2007. – Zuletzt zugegriffen am: 2018-02-17

[LYD71] Lum, V. Y. ; Yuen, P. S. T. ; Dodd, M.: Key-to-address transform techniques: Performance study on large existing formatted files. 14 (1971), Nr. 4, S. 228–239

[McM92] McMillan, K. L.: *Symbolic Model Checking*. http://www.kenmcmil.com/pubs/thesis.pdf, 1992. – Zuletzt zugegriffen am: 2018-02-12

[MIY90] Minato, S. ; Ishiura, N. ; Yajima, S.: Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In: *Design Automation Conference* (1990), S. 52–57

[Mol07] Molitor, P.: *Darstellung Boolescher Funktionen*. http://nirvana.informatik.uni-halle.de/~molitor/pearson/7092/vorlesung/kapitel_06/kapitel6.pdf, 2007. – Zuletzt zugegriffen am: 2018-02-19

[MS99] Molitor, P. ; Scholl, C.: *Datenstrukturen und effiziente Algorithmen für die Logiksynthese kombinatorischer Schaltungen*. Leipzig : Teubner, 1999

[MT98] Meinel, C. ; Theobald, T.: *Algorithmen und Datenstrukturen im VLSI-Design*. Berlin : Springer, 1998

[MWB88] Malik, S. ; Wang, A. ; Brayton, A. R. und Sangiovanni-Vincentelli: Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. In: *Proc. Int. Conf. CAD (ICCAD-88)* (1988), S. 6–9

[OW90] Ottmann, T. ; Widmayer, P.: *Algorithmen und Datenstrukturen*. Berlin : BI-Wissenschaftsverlag, 1990

[Pet57] Peterson, W. W.: Addressing for random-access storage. 1 (1957), Nr. 2, S. 130–146

[Pfe10] Pfenning, F.: *Lecture Notes on Binary Decision Diagrams*. https://www.cs.cmu.edu/~fp/courses/15122-f10/lectures/19-bdds.pdf, 2010. – Zuletzt zugegriffen am: 2018-02-15

[PKK91] P., Ashar ; Keutzer K., Devadas S.: Gate-delay-fault testability properties of multiplexer-based networks. (1991)

[Sch97] Schneider, C.: *Implementierung von Automaten-Algorithmen mit Hilfe von binären Entscheidungsdiagrammen*. http://www.risc.jku.at/publications/download/risc_182/OBDDAutomaton.ps, 1997. – Zuletzt zugegriffen am: 2018-02-25

[Sch10] Schneider, T.: *Aussagenlogik*. http://www.informatik.uni-bremen.de/tdki/lehre/ws10/logik/Teil1.pdf, 2010. – Zuletzt zugegriffen am: 2018-02-11

[Sed92] Sedgewick, R.: *Algorithmen in C*. Boston : Addison-Wesley, 1992

[Sie07] Sieling, D.: *Binary Decision Diagrams*. http://ls2-www.cs.uni-dortmund.de/lehre/winter200607/bdd/skript.pdf, 2007. – Zuletzt zugegriffen am: 2018-02-27

[Spi08] Spinczyk, O.: *Speicherverwaltung*. https://ess.cs.tu-dortmund.de/Teaching/SS2008/BSRvS1/Downloads/U4-Speicherverwaltung-1x2.pdf. Version: 2008. – Zuletzt zugegriffen am: 2018-02-16

[YBO+98] Yang, B. ; Bryant, R. E. ; O'Hallaron, D. ; Biere, A. ; Coudert, O. ; Janssen, G. ; Ranjan, R. K. ; Somenzi, F.: A performance study of BDD-based model checking. (1998), S. 255–289

Beispiel für eine Programmausführung

```cpp
#include <iostream>
#include <fstream>
#include "Manager.hpp"

int main()
{
  /*
   * Create variables and load UT as well as CT
   * It applies the following order: 4 < 3 < 2 < 1
   */
  Manager manager(4, 521, 521);
  BDDNode a( manager.createVariable(1) );
  BDDNode b( manager.createVariable(2) );
  BDDNode c( manager.createVariable(3) );
  BDDNode d( manager.createVariable(4) );
  // Create BDD by combinations (synthesis)
  BDDNode g = (a * b) ^ (!c + d);
  // Compute the high child of the first variable
  BDDNode f = g.getCofactor( 1, BDDNode::getHighFactor() );
  /**
   * Show information to the BDD
   * Compiling with DEBUG also displays
   * information about references
   */
  std::cout << f;
  // Visualize the BDD
  std::ofstream file("f.dot");
  manager.printNode(f, "f", file);
  system("dot -o f.png -T png f.dot");
  // Perform a manual garbage collection
  manager.clear();
  return 0;
}
```

Code 12: Implementierung von `main`

```
0x1006346e0 [4, ~, 2] ( 0x1006346c0 [3, -, 2] ( 0x100634ba0 [2, -, 4]
( 0x1006348a0 [0, -, 16] 0x1006348a0 [0, +, 16, X] ) 0x100634ba0
[2, +, 4, X] ) 0x100634ba0 [2, +, 4, X] )
#Knoten: 4
```

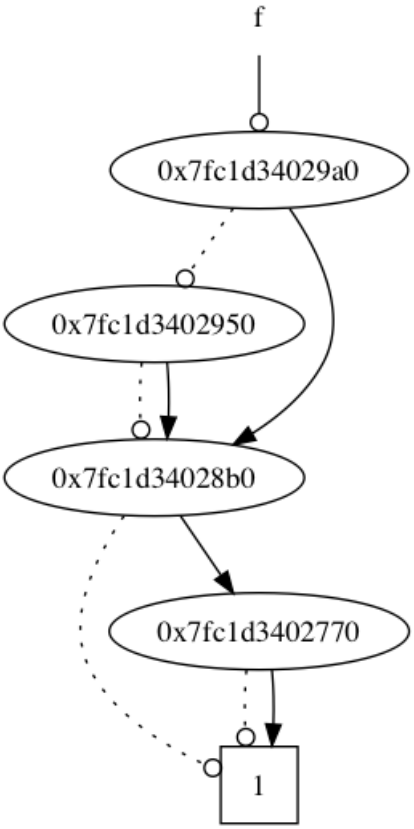

Abbildung 25: Visualisierung des BDDs nach der Programmausführung

```
digraph {
  node [shape=plaintext];
  terminal [label="1", shape=square];
  { rank=source; "f"; }
  node [shape=oval];
  "f" -> "0x7fc29b600d00" [arrowhead=odot]
  { rank=same; "0x7fc29b600d00"; }
  "0x7fc29b600d00" -> "0x7fc29b600ce0" [style=dotted];
  "0x7fc29b600d00" -> "0x7fc29b600c70";
  { rank=same; "0x7fc29b600ce0"; }
  "0x7fc29b600ce0" -> "0x7fc29b600c70" [style=dotted];
  "0x7fc29b600ce0" -> "0x7fc29b600c70";
  { rank=same; "0x7fc29b600c70"; }
  "0x7fc29b600c70" -> "terminal" [style=dotted];
  "0x7fc29b600c70" -> "terminal";
  { rank=same; "terminal"; }
}
```

Auszug des Reports der Unittests


```
<Catch name="ibdd_test">
<Group name="ibdd_test">
<TestCase name="Scenario: Working method of the unique table" tags="[unique-table]" filename="agrabddTest.cpp" line="6">
<Section name="Given: Three nodes, one key and slots in the unique table" filename="agrabddTest.cpp" line="8">
<Section name="When: a node is added" filename="agrabddTest.cpp" line="21">
<Section name="Then: it should also be found" filename="agrabddTest.cpp" line="23">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="5" failures="0" expectedFailures="0"/>
</Section>
<Section name="Given: Three nodes, one key and slots in the unique table" filename="agrabddTest.cpp" line="8">
<Section name="When: a garbage collection is performed" filename="agrabddTest.cpp" line="27">
<Section name="Then: the unique table should be empty" filename="agrabddTest.cpp" line="29">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="5" failures="0" expectedFailures="0"/>
</Section>
<OverallResult success="true"/>
</TestCase>
<TestCase name="Scenario: Working method of the computed table" tags="[computed-table]" filename="agrabddTest.cpp" line="35">
<Section name="Given: Three nodes, one key and slots in the computed table" filename="agrabddTest.cpp" line="37">
<Section name="When: a node is added" filename="agrabddTest.cpp" line="50">
<Section name="Then: it should also be found" filename="agrabddTest.cpp" line="52">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="5" failures="0" expectedFailures="0"/>
</Section>
<Section name="Given: Three nodes, one key and slots in the computed table" filename="agrabddTest.cpp" line="37">
<Section name="When: a garbage collection is performed" filename="agrabddTest.cpp" line="56">
<Section name="Then: the computed table should be empty" filename="agrabddTest.cpp" line="58">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="5" failures="0" expectedFailures="0"/>
</Section>
<OverallResult success="true"/>
</TestCase>
<TestCase name="Scenario: Application of operations to BDDs" tags="[operator]" filename="agrabddTest.cpp" line="64">
<Section name="Given: Two BDDs und slots in the unique as well as computed table" filename="agrabddTest.cpp" line="66">
<Section name="When: the conjunction is applied" filename="agrabddTest.cpp" line="73">
<Section name="Then: three nodes should exist" filename="agrabddTest.cpp" line="75">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="3" failures="0" expectedFailures="0"/>
</Section>
<Section name="Given: Two BDDs und slots in the unique as well as computed table" filename="agrabddTest.cpp" line="66">
<Section name="When: the disjunction is applied" filename="agrabddTest.cpp" line="79">
<Section name="Then: three nodes should exist" filename="agrabddTest.cpp" line="81">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="3" failures="0" expectedFailures="0"/>
</Section>
<Section name="Given: Two BDDs und slots in the unique as well as computed table" filename="agrabddTest.cpp" line="66">
<Section name="When: the adjunction is applied" filename="agrabddTest.cpp" line="85">
<Section name="Then: three nodes should exist" filename="agrabddTest.cpp" line="87">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="3" failures="0" expectedFailures="0"/>
</Section>
<Section name="Given: Two BDDs und slots in the unique as well as computed table" filename="agrabddTest.cpp" line="66">
<Section name="When: the &amp;quot;More than&amp;quot; operator is applied" filename="agrabddTest.cpp" line="91">
<Section name="Then: three nodes should exist" filename="agrabddTest.cpp" line="93">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
```

```
</Section>
<OverallResults successes="3" failures="0" expectedFailures="0"/>
</Section>
<Section name="Given: Two BDDs und slots in the unique as well as computed table" filename="agrabddTest.cpp" line="66">
<Section name="When: the &quot;Less than&quot; operator is applied" filename="agrabddTest.cpp" line="97">
<Section name="Then: three nodes should exist" filename="agrabddTest.cpp" line="99">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="3" failures="0" expectedFailures="0"/>
</Section>
<Section name="Given: Two BDDs und slots in the unique as well as computed table" filename="agrabddTest.cpp" line="66">
<Section name="When: the XNOR operator is applied" filename="agrabddTest.cpp" line="103">
<Section name="Then: three nodes should exist" filename="agrabddTest.cpp" line="105">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="3" failures="0" expectedFailures="0"/>
</Section>
<Section name="Given: Two BDDs und slots in the unique as well as computed table" filename="agrabddTest.cpp" line="66">
<Section name="When: the NAND operator is applied" filename="agrabddTest.cpp" line="109">
<Section name="Then: three nodes should exist" filename="agrabddTest.cpp" line="111">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="3" failures="0" expectedFailures="0"/>
</Section>
<Section name="Given: Two BDDs und slots in the unique as well as computed table" filename="agrabddTest.cpp" line="66">
<Section name="When: the NOR operator is applied" filename="agrabddTest.cpp" line="115">
<Section name="Then: three nodes should exist" filename="agrabddTest.cpp" line="117">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="3" failures="0" expectedFailures="0"/>
</Section>
<Section name="Given: Two BDDs und slots in the unique as well as computed table" filename="agrabddTest.cpp" line="66">
<Section name="When: the NOT operator is applied" filename="agrabddTest.cpp" line="121">
<Section name="Then: two nodes should exist" filename="agrabddTest.cpp" line="123">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="3" failures="0" expectedFailures="0"/>
</Section>
<Section name="Given: Two BDDs und slots in the unique as well as computed table" filename="agrabddTest.cpp" line="66">
<Section name="When: the high child is computed" filename="agrabddTest.cpp" line="127">
<Section name="Then: one node should exist" filename="agrabddTest.cpp" line="129">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="3" failures="0" expectedFailures="0"/>
</Section>
<Section name="Given: Two BDDs und slots in the unique as well as computed table" filename="agrabddTest.cpp" line="66">
<Section name="When: the low child is computed" filename="agrabddTest.cpp" line="133">
<Section name="Then: one node should exist" filename="agrabddTest.cpp" line="135">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="3" failures="0" expectedFailures="0"/>
</Section>
<OverallResult success="true"/>
</TestCase>
<TestCase name="Scenario: BDDs with complement edges" tags="[complementEdges]" filename="agrabddTest.cpp" line="141">
<Section name="Given: Two BDDs und slots in the unique as well as computed table" filename="agrabddTest.cpp" line="143">
<Section name="When: the function f=!(a*b) exists" filename="agrabddTest.cpp" line="150">
<Section name="Then: the root node should have a complement edge" filename="agrabddTest.cpp" line="152">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="3" failures="0" expectedFailures="0"/>
</Section>
<Section name="Given: Two BDDs und slots in the unique as well as computed table" filename="agrabddTest.cpp" line="143">
<Section name="When: the function f=a*b exists" filename="agrabddTest.cpp" line="156">
<Section name="Then: the root node should not have a complement edge" filename="agrabddTest.cpp" line="158">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
```

```
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="3" failures="0" expectedFailures="0"/>
</Section>
<OverallResult success="true"/>
</TestCase>
<TestCase name="Scenario: Nodes and their references" tags="[referenceCounter]" filename="agrabddTest.cpp" line="164">
<Section name="Given: Four BDDs und slots in the unique as well as computed table" filename="agrabddTest.cpp" line="166">
<Section name="When: a node is created for a variable" filename="agrabddTest.cpp" line="177">
<Section name="Then: the cofactor should be a leaf" filename="agrabddTest.cpp" line="178">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="5" failures="0" expectedFailures="0"/>
</Section>
<Section name="Given: Four BDDs und slots in the unique as well as computed table" filename="agrabddTest.cpp" line="166">
<Section name="When: a combination of BDDs exists" filename="agrabddTest.cpp" line="182">
<Section name="Then: the root node should have been reused once" filename="agrabddTest.cpp" line="185">
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="1" failures="0" expectedFailures="0"/>
</Section>
<OverallResults successes="5" failures="0" expectedFailures="0"/>
</Section>
<OverallResult success="true"/>
</TestCase>
<OverallResults successes="69" failures="0" expectedFailures="0"/>
</Group>
<OverallResults successes="69" failures="0" expectedFailures="0"/>
</Catch>
```

ISCAS'85 Schaltkreise

C432 27-Channel Interrupt Controller

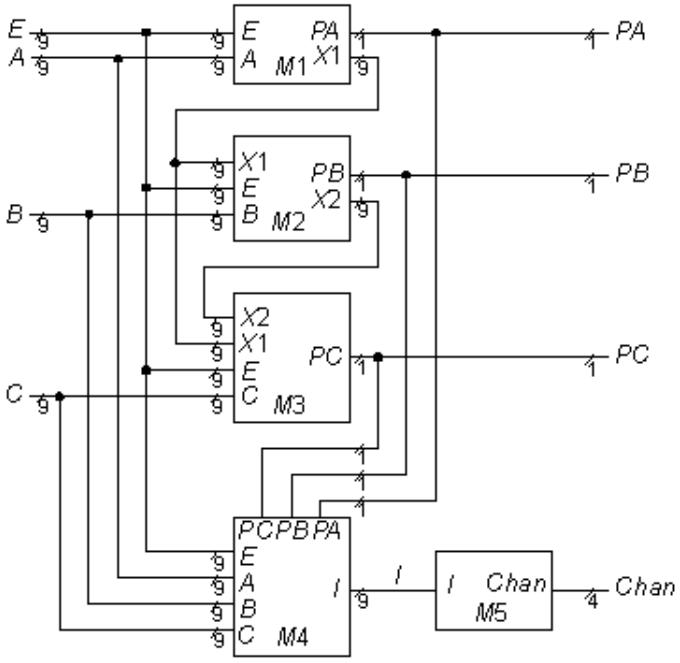

Abbildung 26: C432 27-Channel Interrupt Controller [Hay99]

Tabelle 14: C432 27-Channel Interrupt Controller

Statistik	Es gibt 36 Eingänge, 7 Ausgänge und 160 Gatter.
Funktion	Die Eingänge A, B und C werden in Busse gruppiert, die 9 Bit umfassen, wobei jedes Bit darin über die Anfrage-Priorität für einen Interrupt entscheidet. So haben Anfragen über A immer eine höhere Priorität als Anfragen über B bzw. C. Sollten bspw. zwei Anfragen über A vorliegen, so entscheidet der jeweilige Index über die priorisierte Behandlung. Ein höherer Index hat hierbei eine höhere Priorität. Der Bus E wiederum aktiviert (und deaktiviert) die Interrupts hinsichtlich der jeweiligen Bitpositionen. Die Ausgänge PA, PB, PC und $Chan$ spezifizieren dabei, welche Channels einen bestätigten Interrupt-Requests besitzen, wobei $Chan$ diese dekodiert.

C499/C1355 32-Bit Single-Error-Correcting Circuit

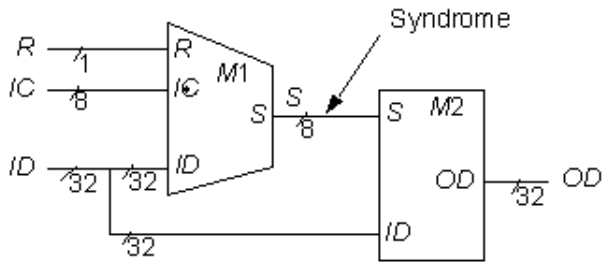

Abbildung 27: C499/C1355 32-Bit Single-Error-Correcting Circuit [Hay99]

Tabelle 15: C499/C1355 32-Bit Single-Error-Correcting Circuit

Statistik	Es gibt 41 Eingänge, 32 Ausgänge und 202/546 Gatter.
Funktion	Der C499-Schaltkreis hat insgesamt gesehen die Aufgabe, Einzelfehler zu korrigieren. Die Eingänge werden hierzu kombiniert, um einen Bus S zu erstellen, der 8 Bit besitzt. Anschließend erfolgt wiederum eine Kombination mit den 32 primären Eingängen, um die 32 primären Ausgänge zu bilden. Konkret betrachtet stellt S eine Matrix für einen (40, 32)-Hamming-Code dar. Das Modul *M1* erkennt hierbei einen Fehler, das Modul *M2* hingegen kann diesen korrigieren. Der C1355-Schaltkreis hat dieselbe Funktionalität, jedoch sind hierbei alle XOR-Gatter durch die äquivalenten vier NAND-Gatter ersetzt worden.

C880 8-Bit ALU

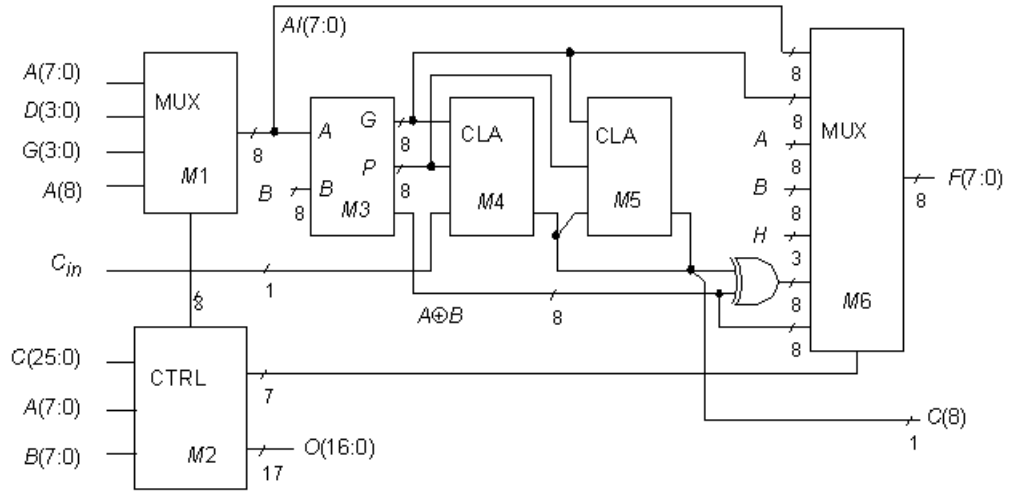

Abbildung 28: C880 8-Bit ALU [Hay99]

Tabelle 16: C880 8-Bit ALU

Statistik	Es gibt 60 Eingänge, 26 Ausgänge und 383 Gatter.
Funktion	Diese ALU stellt ein Rechenwerk dar, um Operationen zu berechnen. Welche konkrete Operation für Operanden eingesetzt wird, entscheiden die jeweiligen Multiplexer *M1* und *M6*, die wiederum von einem Mikrocode *M2* kontrolliert werden.

C1908 Single-Error-Correcting/Double-Error-Detecting Circuit

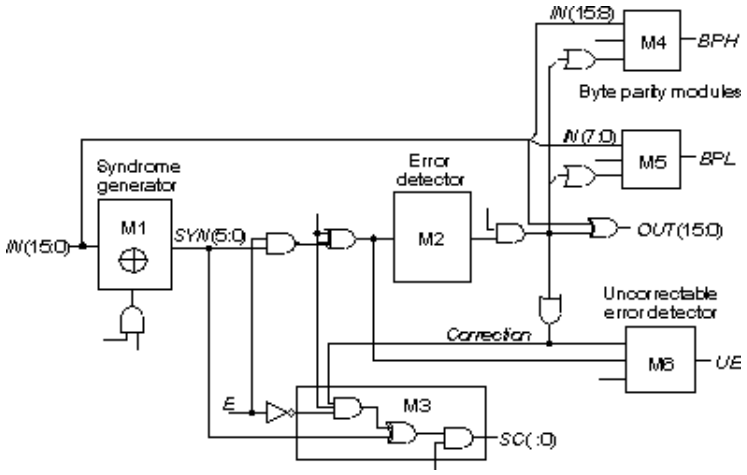

Abbildung 29: C1908 Single-Error-Correcting/Double-Error-Detecting Circuit [Hay99]

Tabelle 17: C1908 Single-Error-Correcting/Double-Error-Detecting Circuit

Statistik	Es gibt 33 Eingänge, 25 Ausgänge und 880 Gatter.
Funktion	Dieser Schaltkreis kann Einzelfehler beheben und Doppelfehler erkennen. Er generiert 6-bit Syndrome aus einem 16-bit Eingang *IN*, wobei dieser speziell dafür dekodiert wurde, um Bitfehler zu finden, insofern welche existieren. Wenn ein Fehler gefunden werden konnte und die Kontrolleingänge gesetzt sind, so wird dieser korrigiert. Zudem gibt es einen Ausgang, der anzeigt, ob es mehrere Fehler gibt, wobei diese dann nicht korrigiert werden können.

C2670 12-bit ALU und Controller

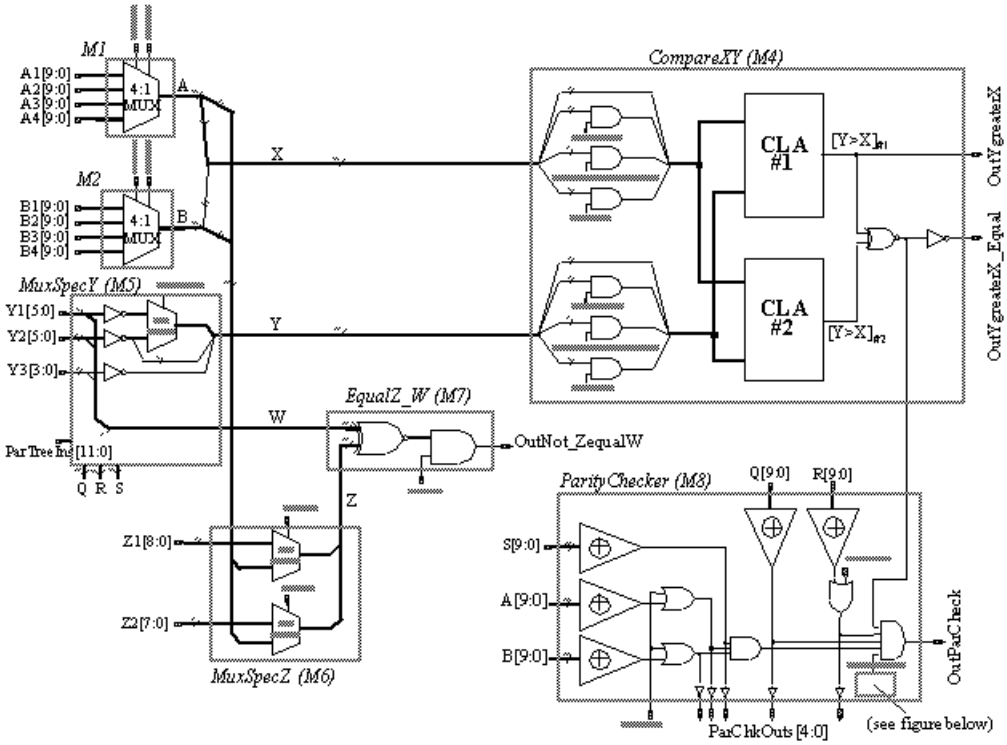

Abbildung 30: C2670 12-bit ALU und Controller [Hay99]

Tabelle 18: C2670 12-bit ALU und Controller

Statistik	Es gibt 233 Eingänge, 140 Ausgänge und 1193 Gatter.
Funktion	Dieser Benchmark beinhaltet eine ALU mit einem Komparator, Äquivalenz-Checker und verschiedene Paritätsbäume. Der Komparator hat zwei 12-bit Eingänge X sowie Y und berechnet $Y > X$ durch die Benutzung von Carry-Lookahead Addierern. Gilt am Ausgang *OutYgreaterX_Equal* eine 1, so waren die Ausgänge der Addierer identisch, ansonsten ist das Ergebnis eine konstante 0. Das Modul *M7* führt wiederum einen Äquivalenztest durch und das Modul *M8* kennzeichnet den Paritätschecker, der Paritätsbäume enthält, die mithilfe einer Konjunktion kombiniert werden.

C3540 8-bit ALU mit BCD-Arithmetik

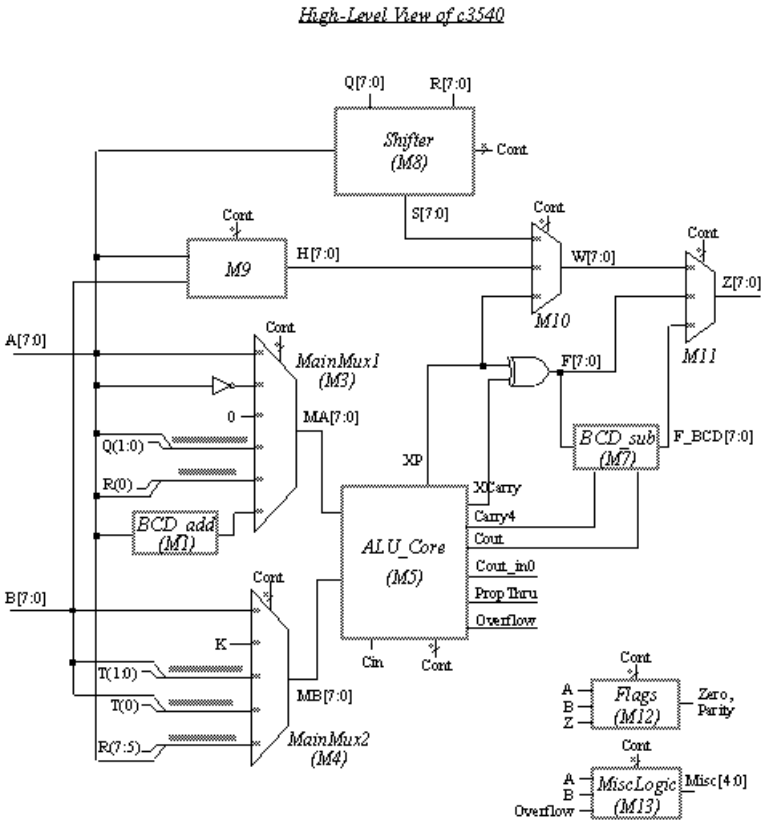

Abbildung 31: C3540 8-bit ALU mit BCD-Arithmetik [Hay99]

Tabelle 19: C3540 8-bit ALU mit BCD-Arithmetik

Statistik	Es gibt 50 Eingänge, 22 Ausgänge und 1669 Gatter.
Funktion	Dieser Benchmark enthält eine 8-bit ALU, die arithmetische und logische Operationen auf binär kodierte Dezimalzahlen ausführen kann. Die Addition wird dabei z. B. durch einen Zweierkomplement-Addierer erledigt, der eine 6 zu beiden Ziffern des ersten Operanden addiert und dann eine 6 von den Ziffern des Ergebnisses subtrahiert, sollten diese keinen Carry produzieren. Das hierbei größte Modul ist *M5*, das über zwei 4-bit Carry-Lookahead Addierern verfügt.

C5315 9-bit ALU mit Paritätsberechnung

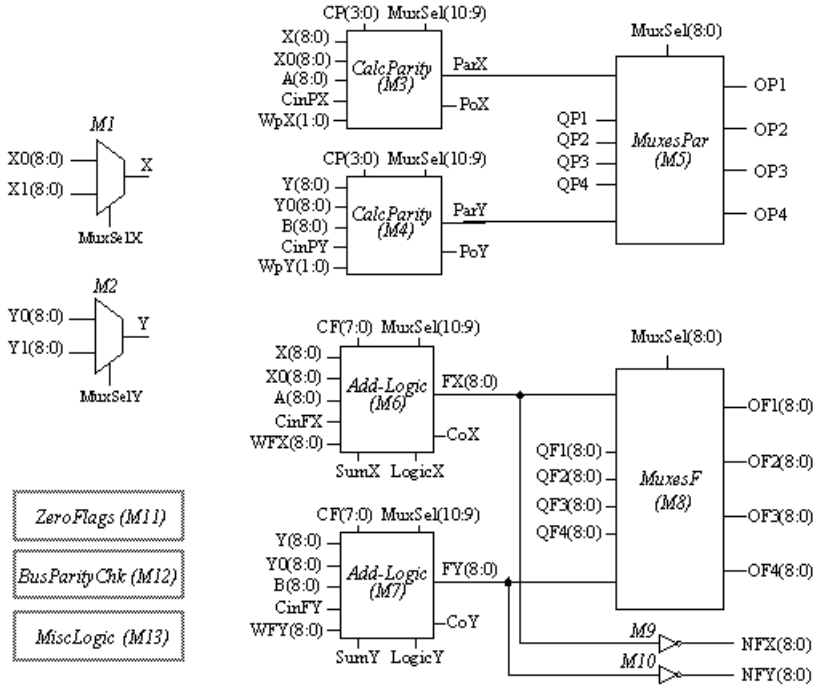

Abbildung 32: C5315 9-bit ALU mit Paritätsberechnung [Hay99]

Tabelle 20: C5315 9-bit ALU mit Paritätsberechnung

Statistik	Es gibt 178 Eingänge, 123 Ausgänge und 2403 Gatter.
Funktion	Bei diesem Schaltkreis handelt es sich um eine ALU, die simultan arithmetische und logische Operationen auf zwei 9-bit Eingängen ausführt und dazu die jeweilige Parität dieser Resultate berechnet. Die Module *M6* und *M7* berechnen hierbei die arithmetische oder logische Operation, die durch das Bauteil *CF* spezifiziert wird. Das Modul *M5* beinhaltet wiederum Multiplexer, die die jeweiligen Resultate weiterleitet. Die Module *M3* und *M4* sind hingegen dafür zuständig, die Parität des Resultates zu berechnen.

C6288 16-bit Multiplizierer

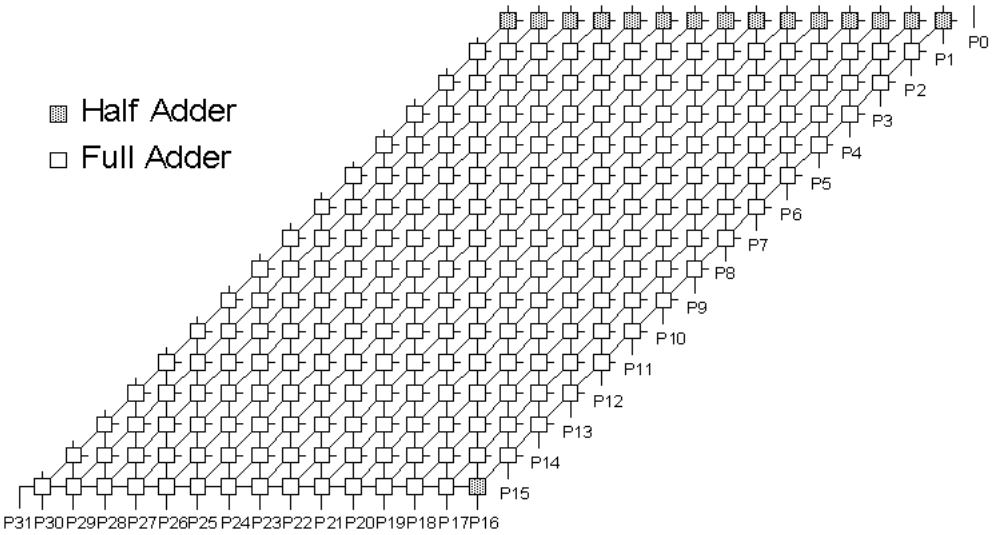

Abbildung 33: C6288 16-bit Multiplizierer [Hay99]

Tabelle 21: C6288 16-bit Multiplizierer

Statistik	Es gibt 32 Eingänge, 32 Ausgänge und 2406 Gatter.
Funktion	Dieser Schaltkreis dient zur Multiplikation von zwei 16-bit Eingängen. Dabei kennzeichnen die 2406 Gatter 240 Voll- und Halbaddierer in Form von Zellen hinsichtlich einer 15x16 Matrix.

C7552 32-bit Addierer/Komparator

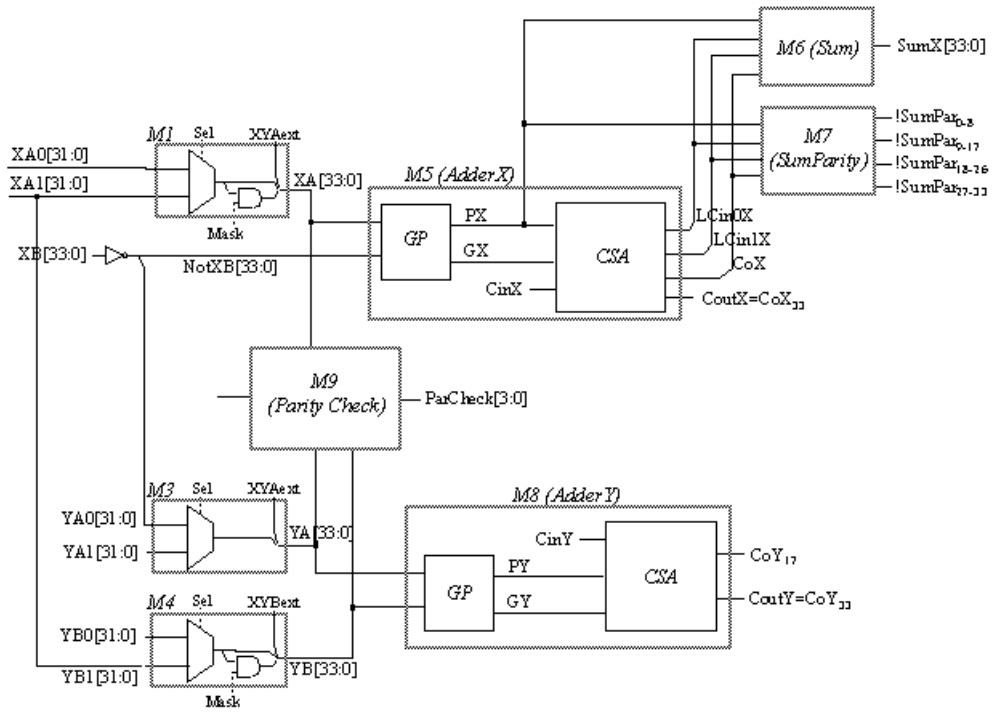

Abbildung 34: C7552 32-bit Addierer/Komparator [Hay99]

Tabelle 22: C7552 32-bit Addierer/Komparator

Statistik	Es gibt 207 Eingänge, 108 Ausgänge und 3512 Gatter.
Funktion	Dieser Benchmark hat einen 34-bit Addierer (*M5*), Komparator (*M8*) und Paritätschecker (*M9*). Jeder der Eingänge *XA*, *YA* und *YB* wird von einer Menge von 2:1 Multiplexern bzw. deren SEL-Eingang versorgt. Der Komparator (*M8*) verhält sich ähnlich zu dem Komparator von dem Schaltkreis *C2670*, wobei die Addierer *M5* und *M8* sich identisch zu den Addierern des Schaltkreises *C5315* verhalten.

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur Arbeit

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

$Unterschrift:$ $Ort, Datum:$