Begleitmaterial zur Vorlesung

Theorie des Logikentwurfs

Sommersemester 2000

Detlef Sieling

Universität Dortmund
FB Informatik, LS 2
44221 Dortmund

Von diesem Begleitmaterial dürfen einzelne Ausdrucke oder Kopien für private Zwecke hergestellt werden. Jede weitere Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne besondere Zustimmung des Autors unzulässig.

Inhaltsverzeichnis

1 Einleitung und Überblick **4**

2 Boolesche Funktionen, Schaltkreise, Kostenmaße **7**

2.1 Schaltkreise/Straight line programs, die Basen B_2 und U, Komplexitätsmaße von Schaltkreisen 7

2.2 Rechenregeln und Normalformen 7

3 Klassische zweistufige Logikminimierung **8**

3.1 Primimplikanten, Minimalpolynome, PLAs 8

3.2 Algorithmus von Quine/McCluskey 8

3.3 Baummethode und Methode des iterierten Konsensus 8

3.4 Überdeckungsprobleme 9

3.5 Unvollständig spezifizierte Funktionen 10

3.6 Funktionen mit mehreren Outputs 10

3.7 Monotone und Symmetrische Funktionen 10

3.8 Effizienz von Minimalpolynomen 13

4 Schaltkreise für die Grundrechenarten **15**

4.1 Schulmethode für die Addition 15

4.2 Carry-Look-Ahead Addierer 15

4.3 Conditional Sum Addierer 15

4.4 Optimale Präfixberechnung 15

4.5 Ladner/Fischer Addierer 15

4.6 Schulmethode für die Multiplikation 15

4.7 Multiplikation mit einem Divide-and-Conquer Verfahren 15

4.8 Schulmethode für die Division 15

4.9 Newtonmethode für die Division 15

5 Ordered Binary Decision Diagrams—Eine Datenstruktur für boolesche Funktionen **16**

5.1 Operationen auf booleschen Funktionen 16

5.2 Einführung in OBDDs 18

5.3 OBDD-Darstellungen von ausgewählten Funktionen 25

5.4 Das Variablenordnungsproblem 31
5.5 Algorithmen auf OBDDs 33
5.6 Schlussbemerkungen 41

6 OBDD-Anwendungen beim Schaltkreisentwurf, bei der Schaltkreisverifikation und bei der zweistufigen Logikminimierung 42
6.1 Der Pentium Dividierer 42
6.2 Anwendung von OBDDs bei der zweistufigen Logikminimierung . 47

7 Verifikation von sequentiellen Schaltkreisen 51
7.1 Flip-Flops und das Huffman-Modell für sequentielle Schaltkreise . 51
7.2 Die Erreichbarkeitsanalyse 55
7.3 Computation Tree Logic 58

1 Einleitung und Überblick

Die Vorlesung „Theorie des Logikentwurfs“ ist die theoretische Stammvorlesung über den Entwurf und die Verifikation von Hardware. Vieles, was heute mit Rechnern möglich ist, wurde nur durch Erfolge im Hardwarebereich möglich. Dieser Erfolg beruht wesentlich darauf, dass immer größere Schaltungen mit einer immer größer werdenden Anzahl von logischen Bausteinen auf einem Chip integriert werden können. Allerdings sind die Schaltungen so groß, dass ad-hoc Methoden für den Entwurf und die Verifikation solcher Schaltungen nicht mehr ausreichen. Für Fortschritte im Hardwarebereich werden daher auch effiziente Methoden für den Entwurf und die Verifikation von Hardware benötigt. Die Vorlesung beschäftigt sich mit dem Entwurf und der Analyse von Algorithmen für die beim Hardwareentwurf und bei der Hardwareverifikation anfallenden Probleme.

Beim Entwurf „guter“ Hardware hat man viele Probleme, die gleichzeitig gelöst werden müssen. Die Anzahl logischer Bausteine oder anderer Hardwarekomponenten soll minimiert werden, die Rechenzeit, d.h., die Tiefe bzw. Verzögerung von Schaltkreisen soll minimiert werden, die benötigte Chipfläche soll minimiert werden und zugleich soll der entworfene Schaltkreis leicht zu testen sein. In den meisten Fällen sind bereits die Optimierungsprobleme für jedes einzelne dieser Optimierungsziele NP-hart, so dass man sich mit nichtoptimalen Lösungen zufrieden geben muss. Häufig ist es nicht einmal möglich, mehrere dieser Ziele zugleich zu verfolgen, wenn z.B. ein Schaltkreis mit minimaler Rechenzeit viel größer ist als ein (langsamer) Schaltkreis mit einer minimalen Anzahl von Bausteinen.

Die Vorlesung ist folgendermaßen aufgebaut. Kapitel 2 liefert die formalen Grundlagen für die folgenden Kapitel. Zunächst werden verschiedene Varianten von Schaltkreisen definiert. Die betrachteten Komplexitätsmaße sind die Größe und die Tiefe (maximale Verzögerung) von Schaltkreisen. Es werden die wichtigsten Rechenregeln für boolesche Funktionen diskutiert. Eine Normalform für eine boolesche Funktion ist eine eindeutige Darstellung, d.h., für jede boolesche Funktion gibt es genau eine Darstellung in dieser Normalform. Dies erleichtert z.B. den Äquivalenztest von booleschen Funktionen.

Die Aufgabe der zweistufigen Logikminimierung besteht darin, für eine gegebene Funktion ein minimales PLA (programable logic array) zu berechnen. Wir behandeln die zweistufige Logikminimierung, da sie einen gut verstandenen Spezialfall der Logikminimierung darstellt. Es werden wichtige Techniken für den Umgang mit booleschen Funktionen vorgestellt. Anhand von Beispielen wird untersucht, in welchen Fällen die behandelten Techniken versagen (müssen). Weiterhin werden wichtige Spezialfälle von booleschen Funktionen wie monotone und symmetrische Funktionen betrachtet. Die Bezeichnung „klassisch“ bedeutet, dass die Funktionen „explizit“, also durch Wertetabellen oder boolesche Ausdrücke dargestellt werden. Im Gegensatz dazu gibt es auch sogenannte „implizite“ Darstellungen von booleschen Funktionen durch OBDDs (Ordered Binary Decision Diagrams), die in den Kapiteln 5–7 behandelt werden.

Die Methoden zur PLA-Minimierung sind im allgemeinen nicht sehr effizient. Daher sollte man für spezielle Funktionen, insbesondere für Funktionen mit viel Struktur, spezielle Schaltkreise entwerfen. Als Beispiel für gut strukturierte Funktionen werden die Grundrechenarten behandelt. Für die Addition, Multiplikation und Division werden verschiedene Schaltkreise entworfen und analysiert. Neben den interessanten Entwurfsideen für diese Schaltkreise werden Methoden für die Beschreibung und Analyse von Schaltkreisen behandelt. Es stellt sich heraus, dass die Schulmethoden für die Grundrechenarten nur bedingt zu effizienten Schaltkreisen führen. Ein aktuelles Beispiel für den Schaltkreisentwurf ist der Pentium-Dividierer. Da für die Verifikation der Korrektheit des Pentium-Dividierers OBDDs hilfreich sind, wird der Pentium-Dividierer (und der Fehler bei seiner Implementierung in den ersten Pentium Chips) erst in Kapitel 6 behandelt.

Kapitel 5 beschäftigt sich mit der Darstellung von booleschen Funktionen im Rechner. Wertetabellen sind nur für Funktionen mit relativ wenigen Inputs verwendbar. Darstellungen durch Schaltkreise oder boolesche Ausdrücke haben den Nachteil, dass Operationen wie z.B. der Äquivalenztest nicht besonders effizient möglich sind. (Der Äquivalenztest von Schaltkreisen ist ein NP-hartes Problem.) Ein guter Kompromiss zwischen der Effizienz der Darstellung und der Effizienz von Operationen scheinen OBDDs (Ordered Binary Decisions Diagrams) zu sein. In Kapitel 5 werden OBDDs zunächst definiert. Dann wird untersucht, wie OBDD-Darstellungen für wichtige Funktionen aussehen, wie Algorithmen für die Manipulation von OBDDs arbeiten und welche Probleme beim Einsatz von OBDDs auftreten.

Die letzten zwei Kapitel beschreiben Anwendungen von OBDDs. Zunächst behandeln wir den Pentium-Dividierer und OBDD-basierte Methoden für die zweistufige Logikminimierung. Im letzten Kapitel werden zunächst OBDD-Methoden für den Entwurf und die Verifikation von Finite State Machines (FSMs, endliche Automaten) behandelt. FSMs oder Simulationen von FSMs treten bei vielen Steuerungen auf. Ein typisches Beispiel ist eine Ampelsteuerung. Bei einer Ampelsteuerung stellt sich die Frage, ob man sicher sein kann, dass die Steuerung „sicher“ ist, also dass es z.B. nicht vorkommen kann, dass die Ampeln für alle Richtungen grün zeigen. Ein Lösungsansatz ist die Erreichbarkeitsanalyse für FSMs. Grob gesagt, dürfen vom Startzustand aus nur „sichere“ Zustände erreicht werden. Für die Beschreibung von komplexeren Anforderungen an FSMs benutzt man speziellen Logiken. Als Beispiel dafür behandeln wir CTL (Computation Tree Logic). Damit lassen sich auch Anforderungen an FSMs wie etwa: „Wenn ein Fußgänger die Taste drückt, schaltet die Fußgängerampel irgendwann auf grün.“ formulieren. Bei der Überprüfung, ob eine gegebene CTL-Formel für eine gegebene FSM wahr ist, sind wiederum OBDDs hilfreich.

Die meisten Inhalte der Vorlesung sind in den folgenden Büchern enthalten:

[MS] Molitor, P. und Scholl, C. (1999). Datenstrukturen und effiziente Algorithmen für die Logiksynthese kombinatorischer Schaltungen. Teubner.

[W1] Wegener, I. (1996). Effiziente Algorithmen für grundlegende Funktionen. Teubner. (Gegenüber der Auflage von 1989 wurden nur geringfügige Änderungen vorgenommen.)

[W2] Wegener, I. (2000). Branching Programs and Binary Decision Diagrams—Theory and Applications. Erscheint in der Reihe SIAM Monographs in Discrete Mathematics and Applications. Die für die Vorlesung relevanten Teile sind (noch) auf der Homepage der Vorlesung von 1999 (ls2-www.cs.uni-dortmund.de/lehre/sommer1999/tdl/) verfügbar.

Die Vorlesung folgt im wesentlichen dem Konzept der Vorlesung „Theorie des Logikentwurfs“ von Ingo Wegener im Sommersemester 1999. Aus urheberrechtlichen Gründen wird für einige Teile der Vorlesung auf die Literatur verwiesen. Korrekturen und Verbesserungsvorschläge für dieses Begleitmaterial sind stets willkommen.

2 Boolesche Funktionen, Schaltkreise, Kostenmaße

2.1 Schaltkreise/Straight line programs, die Basen B_2 und U, Komplexitätsmaße von Schaltkreisen

In der Vorlesung wurden die folgenden Punkte behandelt (siehe [W1] 1.3 bis S. 11): Definitionen von booleschen Funktionen, Straight line programs, Schaltkreisen, Basen B_2 und U, Größe und Tiefe von Schaltkreisen; Beispiel: Volladdierer; Motivation, dass effiziente boolesche Schaltkreise polynomielle Größe und logarithmische Tiefe haben sollten.

2.2 Rechenregeln und Normalformen

Es wurden die folgenden Punkte behandelt (siehe [W1] 2.1): Anzahl boolescher Funktionen; Rechenregeln für boolesche Ausdrücke, disjunktive und konjunktive Normalform, duale Funktion, Ring-Sum Expansion.

3 Klassische zweistufige Logikminimierung

3.1 Primimplikanten, Minimalpolynome, PLAs

Behandelt wurden: Definitionen von Monomen, Polynomen, Primimplikanten, Minimalpolynomen ([W1] 2.2 bis S. 32); Motivation durch PLAs ([MS] 2.2.1).

3.2 Algorithmus von Quine/McCluskey

Siehe [W1] 2.3.

3.3 Baummethode und Methode des iterierten Konsensus

Siehe [W1] 2.4 bis S. 46 oben (außer Beweis zu Satz 2.4.1). Für die Korrektheit der Methode des iterierten Konsensus wurde der folgende alternative Beweis behandelt:

Satz: Die Methode des iterierten Konsensus berechnet das Polynom aller Primimplikanten von f.

Beweis: Wir beobachten zunächst, dass ein Monom m nur dann aus dem Polynom entfernt wird, wenn es eine Verkürzung von m im Polynom gibt. Falls also ein Monom m einmal im Polynom enthalten ist, gibt es zu jedem späteren Zeitpunkt eine (nicht notwendigerweise echte) Verkürzung von m im Polynom. Da es nur endlich viele Monome gibt und jedes Monom höchstens einmal zum Polynom hinzugefügt wird, ist der Algorithmus insbesondere endlich.

Nach Lemma 2.4.10 [W1] stellt das Polynom zu jedem Zeitpunkt f dar, und es enthält nur Implikanten von f. Sei p^* das Polynom, das am Ende ausgegeben wird. Wir beweisen nun die folgende Aussage:

(*) Für jeden Implikanten m von f enthält das Polynom p^* eine (nicht notwendigerweise echte) Verkürzung von m.

Wegen der obigen Beobachtung genügt es zu zeigen, dass m oder eine Verkürzung von m entweder zu Beginn im Polynom vorhanden ist oder im Laufe des Algorithmus zum Polynom hinzugefügt wird. Wir beweisen die Behauptung (*) mit Induktion rückwärts über die Länge l von m.

Induktionsanfang: $l = n$. Die Implikanten der Länge n sind zugleich die Minterme von f. Da jeder Minterm m von f zu Beginn überdeckt wird, enthält das Polynom zu Beginn eine Verkürzung von m.

Induktionsschritt: $l \to l-1$. Sei m ein Implikant von f der Länge $l-1$, und sei x_i eine Variable, die nicht in m enthalten ist. Dann sind $x_i m$ und $\bar{x}_i m$ Implikanten von f mit der Länge l. Nach Induktionsannahme enthält p^* eine Verkürzung m_1 von $x_i m$ und eine Verkürzung m_2 von $\bar{x}_i m$.

1. Fall: m_1 enthält nicht x_i oder m_2 enthält nicht $\bar{x}_i$. Dann ist m_1 oder m_2 auch eine Verkürzung von m.

2. Fall: m_1 enthält x_i und m_2 enthält $\bar{x}_i$, d.h., $m_1 = x_i m_1'$ und $m_2 = \bar{x}_i m_2'$. Dann ist $m_1' m_2'$ ein Konsensus und zugleich eine Verkürzung von m. Also ist entweder $m_1' m_2'$ oder eine Verkürzung von $m_1' m_2'$ in p^* enthalten.

Damit folgt die Behauptung (*). Also enthält p^* insbesondere alle Primimplikanten. Da Verlängerungen von Monomen stets aus dem Polynom entfernt werden, enthält p^* keine Implikanten, die nicht zugleich Primimplikanten sind. □

3.4 Überdeckungsprobleme

Behandelt wurden: Primimplikantentafel, Vereinfachungsregeln, ein Branch-and-Bound Algorithmus für das Überdeckungsproblem, Überdeckung von Monomen durch Polynome, minimale Überdeckungen und ihre Berechnung ([W1] 2.5 bis S. 54 Mitte).

Ergänzend wurde die Zeilenreduktionsregel für Primimplikantentafeln behandelt.

Lemma: Seien m_i und m_j Primimplikanten von f. Falls die i-te Zeile der PI-Tafel an keiner Stelle größer ist als die j-te Zeile der PI-Tafel und die Kosten von m_i nicht kleiner als die von m_j sind, kann die i-te Zeile der PI-Tafel gelöscht werden.

Die Aussage folgt daraus, dass die Eingaben, die von m_i überdeckt werden, auch von m_j überdeckt werden und m_j nicht teurer als m_i ist. Also wird m_i nicht benötigt.

Anmerkung: Die Zeilenreduktionsregel ist auf eine PI-Tafel erst anwendbar, wenn zuvor die Kernimplikantenregel oder die Spaltenreduktionsregel angewendet worden ist. Anderenfalls wäre m_j eine Verkürzung von m_i, und damit wäre m_i kein Primimplikant.

Für die Berechnung der Mengen MU wurde der folgende Algorithmus behandelt. Das Lemma beschreibt ein Kriterium, das angibt, wann die Rekursion für die Berechnung von $\mathrm{MU}(p^*, m)$ abgebrochen werden kann.

Lemma: Sei $p^* = m_1 \vee \ldots \vee m_k$ das Polynom aller Primimplikanten von f und sei m ein Implikant von f. Sei $P_1 = \{i \mid m m_i = 0\}$ und sei
$P_2 = \{i \mid m \text{ ist Verlängerung von } m_i\}$. Falls $P_1 \cup P_2 = \{1, \ldots, k\}$, ist $\mathrm{MU}(p^*, m) = \{\{i\} \mid i \in P_2\}$.

Falls $m m_i = 0$, enthalten m und m_i widersprüchliche Literale, und damit ist $m^{-1}(1) \cap m_i^{-1}(1) = \emptyset$. Also kann nach Definition $\mathrm{U}(p^*, m)$ keine Menge enthalten, die i enthält. Falls m eine Verlängerung von m_i ist, folgt für alle Eingaben $a \in$

$m^{-1}(1)$, dass $m_i(a) = 1$, also dass m von m_i überdeckt wird. Damit ist $\{i\} \in \mathrm{U}(p^*, m)$ und auch $\{i\} \in \mathrm{MU}(p^*, m)$. Die Bedingung $P_1 \cup P_2 = \{1, \ldots, k\}$ stellt sicher, dass für jeden der Primimplikanten einer der beiden Fälle eingetreten ist, dass also m von einem Primimplikanten entweder vollständig oder gar nicht überdeckt wird.

Aus dem Lemma und aus Satz 2.5.17 [W1] ergibt sich folgender Algorithmus.

Eingabe: $p^* = m_1 \vee \ldots \vee m_k$ (das Polynom aller Primimplikanten von f), $m \in I(f)$.
Resultat: $\mathrm{MU}(p^*, m)$.

1) Berechne P_1 und P_2. Falls $P_1 \cup P_2 = \{1, \ldots, k\}$, ist $\mathrm{MU}(p^*, m) = \{\{i\} \mid i \in P_2\}$. Return.
2) Wähle (heuristisch) eine Variable x_j, die nicht in m vorkommt. Berechne rekursiv $\mathrm{MU}(p^*, mx_j)$ und $\mathrm{MU}(p^*, m\bar{x}_j)$.
3) Bilde alle Mengen $I = I_0 \cup I_1$ mit $I_0 \in \mathrm{MU}(p^*, m\bar{x}_j)$ und $I_1 \in \mathrm{MU}(p^*, mx_j)$. Streiche aus der Liste dieser Mengen alle Mengen, für die auch echte Teilmengen gebildet wurden. Die verbleibenden Mengen bilden $\mathrm{MU}(p^*, m)$. Return.

Satz 2.5.17 [W1] und das Lemma implizieren, dass auf diese Weise $\mathrm{MU}(p^*, m)$ korrekt berechnet wird. Formal kann man dies mit einer Induktion beweisen (Übungsaufgabe). Wir bemerken, dass bei der Ausführung des Algorithmus für die Berechnung von $\mathrm{MU}(p^*, m)$ und $\mathrm{MU}(p^*, m')$ möglicherweise dieselben rekursiven Aufrufe ausgeführt werden, dass es also sinnvoll sein kann, die rekursiven Aufrufe und ihre Ergebnisse zu speichern, um die Berechnungen nicht wiederholen zu müssen. Der Algorithmus hat im worst-case exponentielle Rechenzeit. Dies ist allerdings nicht unerwartet, da bereits der Test, ob ein gegebenes Polynom ein Monom überdeckt, NP-hart ist. Man kann allerdings wie bei der Baummethode hoffen, dass die Rekursion früh abbricht.

3.5 Unvollständig spezifizierte Funktionen

Siehe [W1] 2.7.

3.6 Funktionen mit mehreren Outputs

Siehe [W1] 2.8 bis S. 60, vorletzter Absatz.

3.7 Monotone und Symmetrische Funktionen

Siehe [W1] 2.9. Der Korrektheitsbeweis für den Algorithmus zur Minimalpolynomberechnung von Intervallfunktionen wurde folgendermaßen dargestellt.

Ein Minimalpolynom für die Intervallfunktion $I_{k,l}$ über n Variablen wird auf die folgende Weise berechnet: Für alle Vektoren $a \in \{0,1\}^n$ mit $\|a\| = k$ wird mit dem Algorithmus 2.9.7 [W1] ein Monom p_a berechnet. Wie im Folgenden gezeigt wird, ist dann das Polynom p^* aller dieser Monome ein Minimalpolynom für $I_{k,l}$. Es folgt direkt aus dem Algorithmus, dass p_a genau k positive und $n-l$ negierte Literale enthält. Also ist p_a in jedem Fall Primimplikant von $I_{k,l}$. Es genügt also zu zeigen, dass es für jede Eingabe $b \in I_{k,l}^{-1}(1)$ ein Monom p_a gibt, das b überdeckt, für das also $p_a(b) = 1$ ist. Wir geben einen Algorithmus an, der aus $b \in I_{k,l}^{-1}(1)$ einen Vektor a berechnet, für den gilt, dass $p_a(b) = 1$ ist. Wir betrachten also zwei Algorithmen, den Algorithmus 2.9.7, im Folgenden mit $A(a)$ bezeichnet, der aus a den Primimplikanten p_a berechnet, und den im Folgenden angegebenen Algorithmus $A(b)$, der aus $b \in I_{k,l}^{-1}(1)$ einen Vektor a berechnet.

Der Algorithmus $A(b)$ arbeitet folgendermaßen. Eingabe ist $b \in I_{k,l}^{-1}(1)$, also ein Vektor b mit $k \leq \|b\| \leq l$. Auf b werden die Schritte 1.) und 3.) des Algorithmus 2.9.7 angewendet. Die Ausgabe a ergibt sich auf die folgende Weise: Falls $b_i = 1$ und x_i eines der ersten k Literale ist, die in Schritt 3.) von $A(b)$ einen Pop auslösen, ist auch $a_i = 1$. Falls in Schritt 3.) von $A(b)$ nur $s < k$ Pops ausgelöst werden, ist zusätzlich $a_i = 1$ für die ersten $k - s$ positiven Literale x_i, die in $A(b)$ keinen Pop auslösen, weil der Stack leer ist. (Da x_i ein positives Literal ist, ist dann auch $b_i = 1$.)

Wir beachten die genaue Wortwahl: Positive Literal *lösen* einen Pop *aus*, wenn der Stack nicht leer ist; dabei werden negative Literale *gepoppt*.

Der Algorithmus $A(b)$ ist so konstruiert, dass für die Ausgabe a gilt, dass $\|a\| = k$. Weiterhin ist $a_i = 1$ nur möglich, wenn $b_i = 1$ ist; mit anderen Worten ist $a \leq b$.

Für eine Eingabe $b \in I_{k,l}^{-1}(1)$ wird also zunächst mit $A(b)$ der Vektor a berechnet, und dann mit $A(a)$ der Primimplikant p_a. Wir werden im Folgenden untersuchen, welche Gemeinsamkeiten und Unterschiede bei den Schritten 3.) der Algorithmen $A(a)$ und $A(b)$ auftreten. Sei also b mit $k \leq \|b\| \leq l$ gegeben.

1. Fall: Wir betrachten zunächst den einfacheren Fall, dass in Schritt 3.) von $A(b)$ mindestens k Pops ausgelöst werden. Wir betrachten den Ablauf von Schritt 3.) von $A(b)$ bis zu dem Zeitpunkt, zu dem das k-te Pop ausgelöst wird; die späteren Schritte des Algorithmus werden sich als irrelevant erweisen. Wir zerlegen den Ablauf von Schritt 3.) von $A(b)$ bis zum k-ten Pop in zwei Typen von Phasen. In Phasen vom Typ 1 werden positive Literale abgearbeitet, die keinen Pop auslösen, da der Stack leer ist. In Phasen vom Typ 2 werden positive Literale abgearbeitet, die einen Pop auslösen und negative Literale, die auf den Stack gelegt werden. Offensichtlich kann der Ablauf von Schritt 3.) von $A(b)$ so in diese Phasen zerlegt werden, dass sich diese Phasen abwechseln. (Eventuell gibt es auch nur eine Phase vom Typ 2.) Unter dem Zeitpunkt i verstehen wir den Zeitpunkt, zu dem das Literal x_i bzw. $\bar{x}_i$ abgearbeitet wird, also auf den Stack gelegt wird oder einen Pop auslöst oder keinen Pop auslöst, da der Stack leer ist. Für alle Zeitpunkte i in Phasen vom Typ 1 werden positive Literale behandelt, also ist $b_i = 1$. Wegen

der Voraussetzung für den 1. Fall ist $a_i = 0$.

Wir betrachten nun die Phasen vom Typ 2. Da es während dieser Phasen nicht vorkommt, dass Pops wegen leeren Stacks nicht möglich sind, gilt während jeder Phase von Typ 2, dass die Anzahl der ausgeführten Pops kleiner oder gleich der Anzahl der ausgeführten Pushs ist. Da wir nur den Anfang von $A(b)$ bis zu dem Zeitpunkt, wo das k-te Pop ausgelöst wird, betrachten, wird bei jedem durch x_i ausgelösten Pop der Wert a_i auf 1 gesetzt. Da x_i ein positives Literal ist, ist auch $b_i = 1$. Analog ist bei jedem Push $b_i = 0$ und damit auch $a_i = 0$. Insgesamt gilt für alle Zeitpunkte i in Phasen vom Typ 2, dass $a_i = b_i$ ist.

Auf das berechnete a wird nun der Algorithmus $A(a)$ angewendet. Für Zeitpunkte i in Phasen vom Typ 2 ist $a_i = b_i$. In jeder dieser Phasen (außer eventuell der letzten) werden gleichviele Pushs und Pops ausgeführt und zu jedem Zeitpunkt ist die Anzahl der ausgeführten Pops kleiner oder gleich der Anzahl der ausgeführten Pushs. Daher werden in $A(a)$ dieselben Literale gepusht und es lösen dieselben Literale Pops aus wie bei $A(b)$. Insbesondere werden Literale, die zu Beginn einer Phase vom Typ 2 auf dem Stack liegen, während dieser Phase nicht vom Stack entnommen. In Phasen vom Typ 1 ist in $A(b)$ der Stack leer, und es ist $a_i = 0$. Also werden in $A(a)$ die zugehörigen Literale auf den Stack gelegt, von diesem aber nicht mehr in Schritt 3.) entnommen. Die negativen Literale in p_a sind die ersten $n-l$ Literale $\bar{x}_i$, die während der Phasen vom Typ 2 vom Stack genommen werden, die also auch während der Phasen vom Typ 2 auf den Stack gelegt werden. (Hier nutzen wir aus, dass $n-l \leq k$ ist.) Dann aber ist $a_i = b_i = 0$. Die positiven Literale x_i in p_a sind die Literale mit $a_i = 1$. Wegen $a \leq b$ ist auch $b_i = 1$. Insgesamt folgt, dass $p_a(b) = 1$ ist.

2. Fall: Wir kommen nun zu dem Fall, dass in Schritt 3.) von $A(b)$ nur $s < k$ Pops ausgelöst werden. Wir beobachten den Ablauf von Schritt 3.) von $A(b)$ für das gesamte Wort b und zerlegen ihn wieder in die Phasen vom Typ 1 und 2. Der wesentliche Unterschied zum ersten Fall besteht nun darin, dass für die ersten $k - s$ positiven Literale x_i, die keinen Pop auslösen, weil der Stack leer ist, a_i auf 1 gesetzt wird. Ein Beispiel, bei dem die Abläufe der beiden Algorithmen $A(b)$ und $A(a)$ verglichen werden, ist in Abbildung 1 auf Seite 14 gezeigt.

Wir betrachten nun den Ablauf von $A(a)$. Mit denselben Argumenten wie beim 1. Fall folgt, dass während der Phasen vom Typ 2 nur die Literale, die während derselben Phase auf den Stack gelegt wurden, auch gepoppt werden. Insbesondere lösen in $A(a)$ und $A(b)$ dieselben Literale Pops aus, und es werden dieselben Literale gepoppt. Die s gepoppten Literale werden zu negativen Literalen in p_a. In Schritt 4.) von $A(a)$ werden solange Pops ausgeführt, bis der Stack leer ist, und die ersten $n - l - s$ gepoppten Literale werden ebenfalls zu negativen Literalen in p_a.

Sei x_i ein positives Literal in p_a. Dann ist $a_i = 1$ und wegen $a \leq b$ auch $b_i = 1$. Sei nun $\bar{x}_i$ ein negiertes Literal in p_a. Dann ist $a_i = 0$. Um zu zeigen, dass p_a die Eingabe b überdeckt, genügt es zu zeigen, dass $b_i = 0$ ist. Falls $\bar{x}_i$ ein Literal ist, das in einer Phase vom Typ 2 gepoppt wird, ist $a_i = b_i$ und damit $b_i = 0$.

Es bleibt also noch der Fall, dass $\bar{x}_i$ ein Literal ist, das in Schritt 4.) von $A(a)$ gepoppt wurde. Seien

$$\begin{aligned} I &= \{i \mid x_i \text{ löst in } A(b) \text{ einen Pop aus oder } \bar{x}_i \text{ wird in } A(b) \text{ gepoppt.}\} \text{ und} \\ J &= \{i \mid x_i \text{ löst in } A(b) \text{ keinen Pop aus bzw. } \bar{x}_i \text{ wird in } A(b) \text{ nicht gepoppt.}\}. \end{aligned}$$

Da nur $s < k$ Pops ausgelöst werden, ist $|I| = 2s$, und damit $|J| = n - 2s$. Die Anzahl der Indizes $i \in I$ mit $b_i = 0$ bzw. mit $b_i = 1$ ist jeweils gleich s. Die Anzahl der Indizes $i \in J$ mit $b_i = 1$ ist gleich $\|b\|$ minus der Anzahl der Indizes $i \in I$ mit $b_i = 1$, also gleich $\|b\| - s$. Also gibt es in J genau $|J| - (\|b\| - s) = n - \|b\| - s$ Indizes i mit $b_i = 0$. Da in $A(b)$ die x_i mit $b_i = 1$ und $i \in J$ keinen Pop auslösen, da der Stack leer ist, während die $\bar{x}_i$ mit $b_i = 0$ und $i \in J$ auf den Stack gelegt werden, folgt, dass alle Indizes $i \in J$ mit $b_i = 1$ kleiner als alle $i \in J$ mit $b_i = 0$ sind. Da in Schritt 4.) von $A(a)$ die $n-l-s$ zuerst gepoppten negativen Literale in p_a aufgenommen werden, handelt es sich um die $n-l-s$ Literale, deren Indizes die größten in der Menge J sind. Da in J für die $n - \|b\| - s \geq n - l - s$ größten Indizes i gilt, dass $b_i = 0$ ist, werden in Schritt 4.) nur Literale $\bar{x}_i$ mit $b_i = 0$ gepoppt. Also enthält p_a nur negative Literale $\bar{x}_i$, für die $b_i = 0$ ist und, wie oben gezeigt, nur positive Literale x_i, für die $b_i = 1$ ist. Insgesamt folgt, dass b von p_a überdeckt wird.

3.8 Effizienz von Minimalpolynomen

Siehe [W1] 2.11, S. 68 letzter Absatz bis S. 69 Mitte.

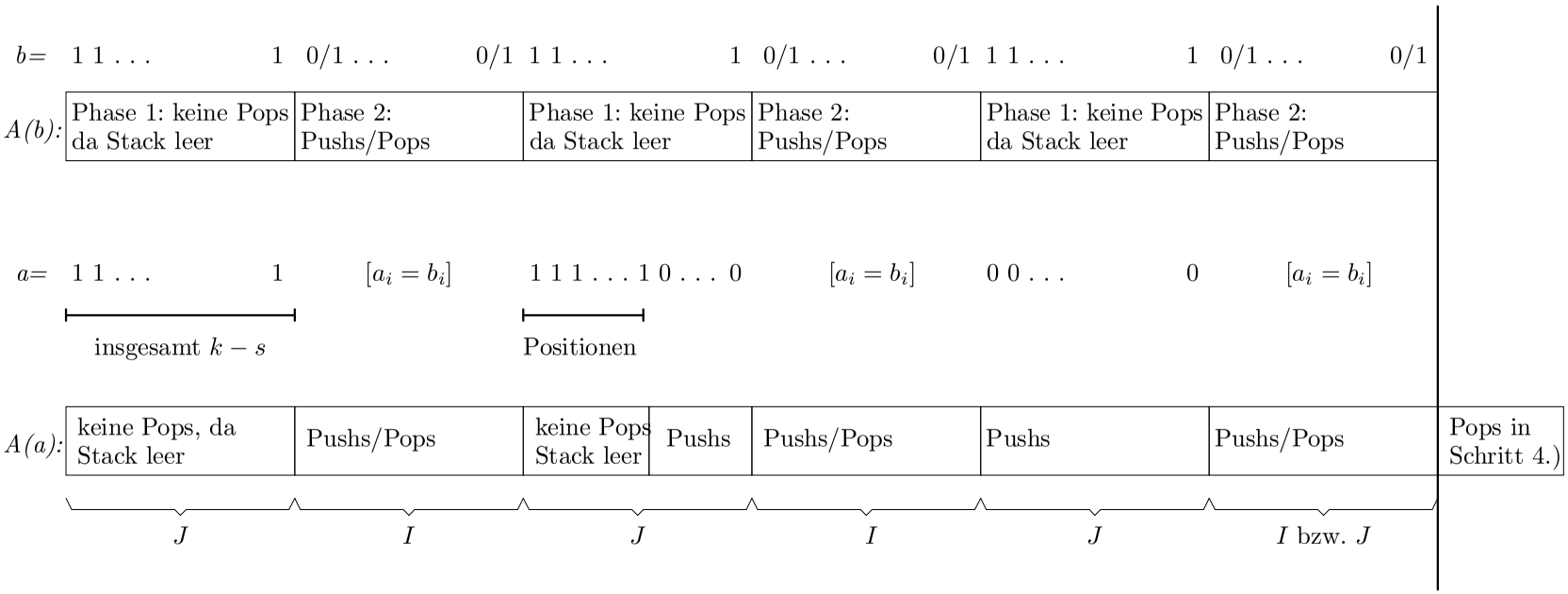


Abbildung 1: Der Ablauf von $A(b)$ und $A(a)$ im 2. Fall

4 Schaltkreise für die Grundrechenarten

4.1 Schulmethode für die Addition

Siehe [W1] 3.1.

4.2 Carry-Look-Ahead Addierer

Siehe [W1] 3.3.

4.3 Conditional Sum Addierer

Siehe [W1] 3.4.

4.4 Optimale Präfixberechnung

Siehe [W1] 3.5.

4.5 Ladner/Fischer Addierer

Siehe [W1] 3.6.

4.6 Schulmethode für die Multiplikation

Siehe [W1] 3.8.

4.7 Multiplikation mit einem Divide-and-Conquer Verfahren

Siehe [W1] 3.9.

4.8 Schulmethode für die Division

Siehe [W1] 3.15.

4.9 Newtonmethode für die Division

Siehe [W1] 3.16.

5 Ordered Binary Decision Diagrams—Eine Datenstruktur für boolesche Funktionen

5.1 Operationen auf booleschen Funktionen

Eine Datenstruktur für boolesche Funktionen soll die kompakte Darstellung von möglichst vielen (praktisch wichtigen) booleschen Funktionen und effiziente Algorithmen für die Manipulation der dargestellten Funktionen erlauben. Wir beginnen mit einer Liste von elementaren Operationen auf booleschen Funktionen. Anschließend diskutieren wir kurz, warum es sinnvoll ist, diese Operationen zu betrachten.

1. Auswertung: Gegeben sei eine Darstellung G für $f \in B_n$ und eine Eingabe $a \in \{0,1\}^n$. Berechne $f(a)$.
2. Erfüllbarkeit: Gegeben sei eine Darstellung G für $f \in B_n$. Gibt es eine Eingabe $a \in \{0,1\}^n$ mit $f(a) = 1$?
3. Erfüllbarkeit-Anzahl: Gegeben sei eine Darstellung G für $f \in B_n$. Berechne $|f^{-1}(1)|$.
4. Erfüllbarkeit-Alle: Gegeben sei eine Darstellung G für $f \in B_n$. Gib $f^{-1}(1)$ aus.
5. Reduktion/Minimierung: Gegeben sei eine Darstellung G für $f \in B_n$. Berechne für f eine Darstellung G' minimaler Größe. Falls G' für jede Funktion f eindeutig definiert ist, heißt diese Operation Reduktion.
6. Äquivalenztest: Gegeben seien Darstellungen G_f und G_g für $f, g \in B_n$. Teste, ob $f = g$.
7. Synthese: Gegeben seien Darstellungen G_f und G_g für $f, g \in B_n$ und eine Operation $\otimes \in B_2$. Berechne eine Darstellung für $f \otimes g$.
8. Ersetzung durch Konstanten: Gegeben sei eine Darstellung G für $f \in B_n$, eine Variable x_i und eine Konstante c. Berechne eine Darstellung für $f_{|x_i=c}$.
9. Ersetzung durch Funktionen: Gegeben seien Darstellungen G_f und G_g für $f, g \in B_n$ und eine Variable x_i. Berechne eine Darstellung für $f_{|x_i=g} = f(x_1, \ldots, x_{i-1}, g(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n)$.
10. Quantifizierung: Gegeben sei eine Darstellung G für $f \in B_n$ und eine Variable x_i. Berechne eine Darstellung für $(\forall x_i : f) := f_{|x_i=0} \wedge f_{|x_i=1}$ (bzw. für $(\exists x_i : f) := f_{|x_i=0} \vee f_{|x_i=1}$).
11. Redundanztest: Gegeben sei eine Darstellung G für $f \in B_n$ und eine Variable x_i. Teste, ob die Variable x_i redundant ist, d.h., ob $f_{|x_i=0} \equiv f_{|x_i=1}$ gilt.

Wir diskutieren nun Anwendungen für einige der Operationen.

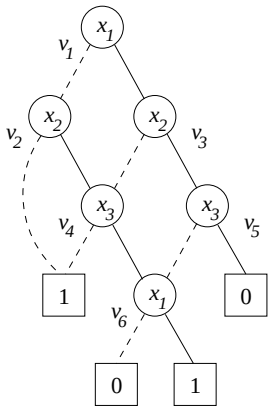
Bei der Hardwareverifikation soll für einen gegebenen Schaltkreis getestet werden, ob er die gewünschte Funktion realisiert. Die gewünschte Funktion ist durch eine Spezifikation oder durch einen anderen Schaltkreis gegeben. Für den Test, ob die gewünschte und die realisierte Funktion gleich sind, kann man für beide Funktionen Darstellungen (in unserem Fall eine bestimmte Variante von BDDs) berechnen und auf Gleichheit testen. Der Test, ob zwei Schaltkreise dieselbe Funktion berechnen, ist jedoch co-NP-vollständig. Das Beste, was man erwarten kann, ist also, dass die Umrechnung der Schaltkreise in die gewünschte Darstellung und der Äquivalenztest für praktisch relevante Funktionen effizient durchführbar sind.

Für die Umrechnung eines B_2-Schaltkreises in die gewählte Darstellung wird man den Schaltkreis in einer topologischen Ordnung durchlaufen (d.h., jeder Knoten/Baustein wird erst nach allen seinen Vorgängern abgearbeitet). Alternativ kann man das zugehörige Straight-line Programm durchlaufen. Zuerst werden Darstellungen der Funktionen $x_1, \ldots, x_n$ konstruiert. Wenn man einen Baustein C erreicht, der zwei Funktionen f_1 und f_2 mit einem binären Operator $\otimes$ verknüpft, muss man aus den Darstellungen für f_1 und f_2 eine Darstellung für $f_1 \otimes f_2$ berechnen, also die Operation Synthese anwenden. Auf diese Weise erhält man eine Darstellung für die vom Schaltkreis berechnete Funktion. Auch wenn die Berechnung eines OBDDs aus einem Schaltkreis ein NP-hartes Problem ist, haben sich dieser Ansatz bzw. Verfeinerungen dieses Ansatzes häufig als praktisch durchführbar erwiesen. Die Synthese wird bei Umrechnungen von Schaltkreisen in die gewählte Darstellung für jeden Baustein einmal ausgeführt, sollte also besonders effizient ausführbar sein. Wenn der Schaltkreis aus Modulen aufgebaut ist, kann man auch zuerst Darstellungen für die Funktionen, die durch die Module realisiert werden, berechnen und aus diesen hinterher eine Darstellung für die durch den Schaltkreis berechnete Funktion. Dazu benötigt man eine Operation, die aus Darstellungen für Funktionen f und g eine Darstellung für $f_{|x_i=g}$ berechnet. Da die Rechenzeit für die einzelnen Operationen von der Größe der Darstellung abhängt, sollte es außerdem effizient möglich sein, die Größe der Darstellung zu minimieren.

Wenn der untersuchte Schaltkreis nicht die gewünschte Funktion g, sondern eine fehlerhafte Funktion h realisiert, kann es für die Korrektur hilfreich sein, zu wissen, auf wievielen und auf welchen Eingaben der Schaltkreis falsch arbeitet. Es müssen also die Mächtigkeit und die Elemente der Menge $(g \oplus h)^{-1}(1)$ berechnet werden.

Die Operation Quantifizierung wird sich bei der Erreichbarkeitsanalyse von Finite State Machines und beim Entwurf des Pentium-Dividierers als nützlich erweisen.

Datenstrukturen für boolesche Funktionen sollten möglichst klein sein, um Speicherplatz und Rechenzeit zu sparen. Da es 2^{2^n} boolesche Funktionen über n Variablen gibt, muss jede Darstellung für fast alle booleschen Funktionen exponentielle Größe haben. Daher können wir nur erreichen, dass möglichst viele praktisch wichtige Funktionen kompakt darstellbar und effizient manipulierbar sind.



1: if x_1 then goto 3 else goto 2;
2: if x_2 then goto 4 else goto 8;
3: if x_2 then goto 5 else goto 4;
4: if x_3 then goto 6 else goto 8;
5: if x_3 then goto 7 else goto 6;
6: if x_1 then goto 8 else goto 7;
7: Ausgabe 0;
8: Ausgabe 1;

Abbildung 2: Beispiel für ein OBDD und das zugehörige if-then-else Programm

Die Größe von Wertetabellen ist für alle booleschen Funktionen exponentiell in der Anzahl der Variablen. Also können nur Funktionen über sehr wenigen Variablen dargestellt werden. Schaltkreise und Polynome sind für viele Funktionen viel kompakter als Wertetabellen. Allerdings ist für Schaltkreise und Polynome der Äquivalenztest co-NP-vollständig und das Erfüllbarkeitsproblem NP-vollständig.

5.2 Einführung in OBDDs

Basierend auf früheren Arbeiten von Lee (1959) und Akers (1978) hat Bryant 1985 OBDDs als Datenstruktur für boolesche Funktionen vorgeschlagen und effiziente Algorithmen für die o.g. Operationen entworfen. Wir definieren zunächst das allgemeinere Konzept von Binary Decision Diagrams (BDDs, in der Komplexitätstheorie auch Branchingprogramme genannt) und führen danach OBDDs als eingeschränkte BDDs ein. BDDs können als Graphen oder als spezielle Programme definiert werden, die nur Anweisungen der Form „if x_i then goto xxx else goto yyy“, wobei nur goto-Anweisung auf spätere Befehle erlaubt sind, und der Form „Ausgabe 0(1)“ enthalten. Abbildung 2 zeigt ein Beispiel für beide Darstellungsformen.

Ein BDD ist ein gerichteter azyklischer Graph mit einer Quelle (einem Knoten ohne eingehende Kanten). Der Graph enthält nur innere Knoten mit Ausgangsgrad 2 und Senken mit Ausgangsgrad 0. Die inneren Knoten sind mit einer Variablen markiert, und die beiden ausgehenden Kanten sind mit 0 bzw. 1 markiert. Die Senken sind ebenfalls mit 0 oder 1 markiert. In den Zeichnungen werden 0-Kanten durch gestrichelte Linien und 1-Kanten durch durchgezogene Linien dargestellt. Weiterhin werden die Pfeile an den Kanten weggelassen, wobei wir davon ausgehen, dass die Kanten immer nach unten gerichtet sind.

Die Berechnung der dargestellten Funktion für eine Eingabe $a = (a_1, \ldots, a_n)$ beginnt an der Quelle. Wenn ein erreichter Knoten mit der Variablen x_i markiert ist, wird entsprechend dem Wert von a_i über die mit 0 oder die mit 1 markierte

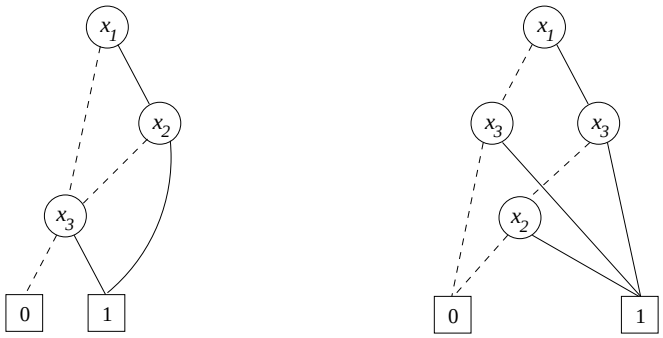


Abbildung 3: OBDDs für die Funktion $x_1x_2 \vee x_3$

Kante ein neuer Knoten erreicht. Dieses wird iteriert, bis man eine Senke erreicht. Die Markierung dieser Senke ist der gesuchte Funktionswert $f(a)$.

Die Beziehung zu dem if-then-else Programm sollte offensichtlich sein. Das wichtigste Komplexitätsmaß für BDDs ist die Anzahl der Knoten. Da jeder Knoten höchstens zwei ausgehende Kanten hat, ist dies zugleich ein Maß für den benötigten Speicherplatz. Wir bemerken, dass BDDs leicht für die Darstellung von mehreren Funktionen $f_1, \ldots, f_m$ erweitert werden können. Ein solches BDD hat insgesamt m mit $f_1, \ldots, f_m$ markierte Knoten (Quellen oder innere Knoten), an denen die Auswertung der jeweiligen Funktion gestartet wird. Zur Vereinfachung der Darstellung betrachten wir im Folgenden meistens BDDs für eine Funktion. Bei der Definition und Implementierung von OBDDs und den Algorithmen auf OBDDs gibt es zahlreiche weitere „Tricks“, die die OBDD-Größe kleiner und die Algorithmen (heuristisch) viel effizienter machen, auf die wir aber, um die Darstellung einfach zu halten, nicht eingehen.

Definition 5.1 Ein OBDD ist ein BDD, in dem auf jedem Pfad alle Variablen höchstens einmal und gemäß einer vorgegebenen Ordnung getestet werden. D.h., es gibt eine Permutation $\pi \in S_n$, und für jede Kante, die von einem mit x_i markierten Knoten zu einem mit x_j markierten Knoten führt, gilt $\pi(i) < \pi(j)$.

Das BDD in Abb. 2 ist also kein OBDD. Abbildung 3 zeigt zwei OBDDs für die Funktion $f(x_1, x_2, x_3) = x_1x_2 \vee x_3$ mit den Variablenordnungen x_1, x_2, x_3 und x_1, x_3, x_2. Obwohl beide OBDDs dieselbe Funktion darstellen, hat das eine OBDD fünf, das andere aber sechs Knoten. Wir werden später noch sehen, dass die Variablenordnung entscheidenden Einfluss auf die Größe eines OBDDs haben kann.

Wir zeigen nun, dass jede boolesche Funktion $f(x_1, \ldots, x_n)$ durch ein OBDD dargestellt werden kann. Wir zerlegen f mit der Shannon-Zerlegung in
$f = \bar{x}_1 f_{|x_1=0} \vee x_1 f_{|x_1=1}$ und erzeugen einen mit x_1 markierten Knoten v. Am 0-Nachfolger dieses Knotens muss die Funktion $f_{|x_1=0}$ dargestellt werden; dazu berechnen wir rekursiv ein OBDD für diese Funktion und wählen dessen Quelle als 0-Nachfolger von v. Analog berechnen wir für den 1-Nachfolger ein OBDD für

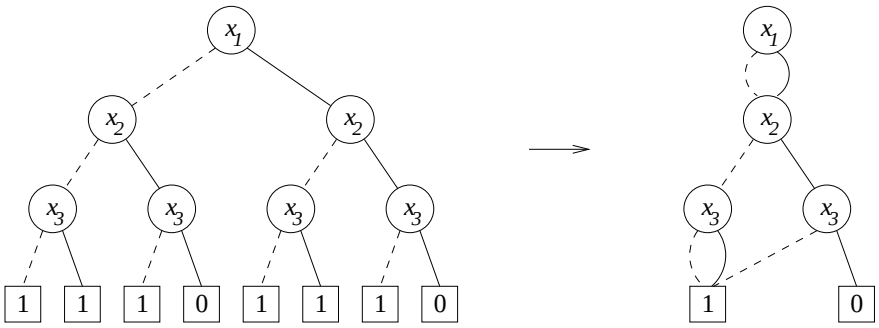


Abbildung 4: Ein Entscheidungsbaum und ein OBDD für die Funktion $\bar{x}_2 \vee \bar{x}_3$

$f_{|x_1=1}$. Abb. 4 zeigt ein OBDD, das auf diese Weise für $f = \bar{x}_2 \vee \bar{x}_3$ berechnet wird. Mit dieser Prozedur erhalten wir immer ein BDD, in dem alle Knoten den Eingangsgrad 1 haben. Derartige BDDs heißen Entscheidungsbäume, weil der zugrundeliegende Graph ein Baum ist.

Wenn wir auf diese Weise einen Entscheidungsbaum für eine Funktion $f : \{0,1\}^n \to \{0,1\}$ konstruieren, hat dieser Entscheidungsbaum $2^n - 1$ innere Knoten. In BDDs dürfen Knoten einen größeren Eingangsgrad als 1 haben; daher dürfen wir isomorphe Teilgraphen verschmelzen (siehe Abbildung 4). Für die Durchführung dieser Verschmelzungen von isomorphen Teilgraphen gibt es zwei sogenannte Reduktionsregeln.

Für die erste Reduktionsregel betrachten wir die Situation in Abb. 5. Sei v ein Knoten in einem BDD, der mit x_i markiert ist und für den die 0-Kante und die 1-Kante auf denselben Knoten w zeigen. Wenn wir den Funktionswert für eine Eingabe berechnen und dabei den Knoten v erreichen, gehen wir unabhängig vom Wert von x_i zum Knoten w weiter. Der Test von x_i ist also überflüssig, und wir können das BDD vereinfachen, indem wir den Knoten v löschen und die Kanten, die zu v führen, direkt zu w zeigen lassen. Diese Reduktionsregel heißt *Deletion Rule*.

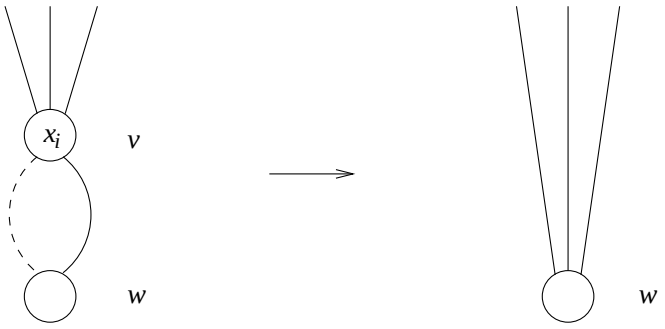


Abbildung 5: Deletion Rule

Die zweite Reduktionsregel, die *Merging Rule*, ist anwendbar, wenn es zwei Knoten v und w gibt, die mit der gleichen Variablen markiert sind und denselben 0-Nachfolger und denselben 1-Nachfolger haben (s. Abb. 6). Es ist klar, dass wir

denselben Funktionswert erhalten unabhängig davon, ob wir am Knoten v oder am Knoten w starten. Wir können also die Knoten v und w verschmelzen, d.h., wir können den Knoten v löschen und alle zu v führenden Kanten auf w zeigen lassen. Analog können wir Senken, die mit der gleichen Konstanten markiert sind, verschmelzen. Durch jede Anwendung einer Reduktionsregel wird die Zahl der Knoten im BDD um 1 verringert.

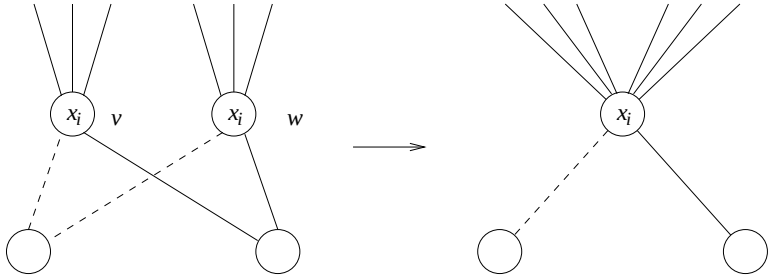


Abbildung 6: Merging Rule

Abb. 7 zeigt ein Beispiel für eine Reduktion in mehreren Schritten. Zunächst wird der mit x_4 markierte Knoten mit der Deletion Rule entfernt. Dann können die mit x_3 markierten Knoten verschmolzen werden, und dann kann die Deletion Rule auf den mit x_2 markierten Knoten angewandt werden. Man sieht an diesem Beispiel Abhängigkeiten zwischen den Reduktionsschritten, d.h., manchmal ist eine Regel erst anwendbar, nachdem andere Reduktionsschritte durchgeführt worden sind.

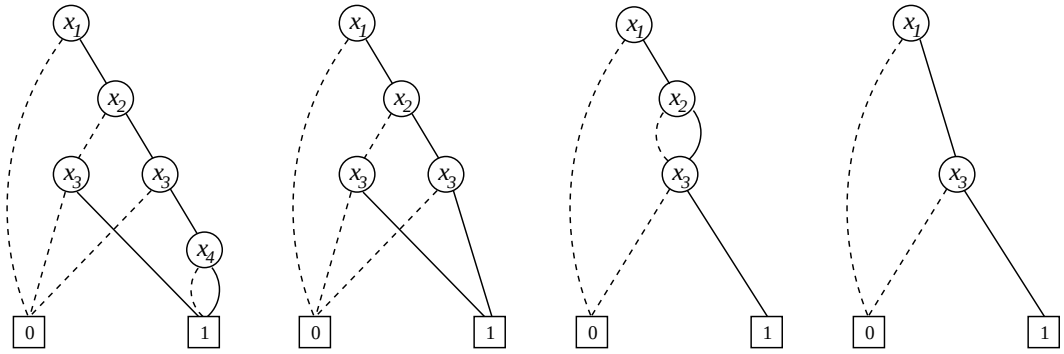


Abbildung 7: Beispiel für eine Reduktion

Wir wollen nun zeigen, dass es sinnvoll ist, von *dem* reduzierten OBDD für eine Funktion f und eine Variablenordnung π zu reden. Zuerst zeigen wir, dass die Funktionen, die an den Knoten eines OBDDs für f berechnet werden, bestimmte Subfunktionen von f sind. Subfunktionen von f (manchmal auch Kofaktoren von f genannt) sind Funktionen, die aus f durch das Konstantsetzen von einigen Variablen entstehen. Über diese Subfunktionen können OBDDs minimaler Größe für f beschrieben werden. Wir können dann beweisen, dass alle minimalen OBDDs für f isomorph sind. Der Einfachheit halber sei im Rest dieses Abschnitts die Variablenordnung $x_1, \ldots, x_n$. Dieses ist keine Einschränkung, denn wir können diese Ordnung aus jeder Ordnung erhalten, indem wir die Variablen umbenennen.

Sei v ein Knoten in einem OBDD für die Funktion f. Die Funktion, die wir erhalten, wenn wir v als Quelle eines OBDDs interpretieren, wird mit f_v bezeichnet. Der Knoten v sei mit x_i markiert. Sei P ein Pfad, der von der Quelle zu v führt. Auf P dürfen nur die Variablen $x_1, \ldots, x_{i-1}$ getestet werden. Der Pfad P wird durchlaufen, wenn die Variablen, die auf P getestet werden, passende Werte haben, wenn also $x_1 = c_1, \ldots, x_{i-1} = c_{i-1}$ für geeignete Konstanten $c_1, \ldots, c_{i-1}$ gilt. Am Knoten v und an Knoten unterhalb von v dürfen nur die Variablen $x_i, \ldots, x_n$ getestet werden. Daher wird an v die Funktion $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}}$ berechnet. Sollte es außer P noch einen Pfad Q geben, der von der Quelle zu v führt und der für $x_1 = d_1, \ldots, x_{i-1} = d_{i-1}$ durchlaufen wird, wird an v zugleich die Funktion $f_{|x_1=d_1,\ldots,x_{i-1}=d_{i-1}}$ berechnet. Da im OBDD an v und an Knoten unterhalb von v nur die Variablen $x_i, \ldots, x_n$ getestet werden dürfen, gilt $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}} \equiv f_{|x_1=d_1,\ldots,x_{i-1}=d_{i-1}}$. An jedem mit x_i markierten Knoten wird also genau eine Subfunktion von f berechnet, die wir aus f erhalten können, indem wir $x_1, \ldots, x_{i-1}$ auf geeignete Weise konstantsetzen.

Wir können nun minimale OBDDs für jede boolesche Funktion und jede Variablenordnung beschreiben.

Satz 5.2 Sei S_i die Menge der Subfunktionen von f, die wir erhalten, indem wir $x_1, \ldots, x_{i-1}$ konstantsetzen, und die essentiell von x_i abhängen. Es gibt bis auf Isomorphie genau ein OBDD minimaler Größe für f und die Variablenordnung $x_1, \ldots, x_n$. Dieses OBDD enthält genau $|S_i|$ Knoten, die mit x_i markiert sind.

Beweis Wir konstruieren ein OBDD für f, das für jedes i genau $|S_i|$ Knoten enthält, die mit x_i markiert sind. Wenn f eine konstante Funktion ist, besteht das OBDD aus einer Senke, die mit der entsprechenden Konstanten markiert ist. Anderenfalls enthält das OBDD eine 0-Senke, eine 1-Senke und für jede Subfunktion $g \in S_i$ genau einen Knoten v, der mit x_i markiert ist. Seien $g_0 := g_{|x_i=0}$ und $g_1 := g_{|x_i=1}$. Für diese Funktionen enthält das OBDD die Knoten v_0 und v_1. Jeder dieser Knoten ist entweder eine Senke, wenn g_0 bzw. g_1 eine konstante Funktion ist, oder ein mit x_j markierter innerer Knoten, wobei $j > i$ gilt. Wir wählen als 0-Nachfolger von v den Knoten v_0 und als 1-Nachfolger v_1.

Das konstruierte OBDD berechnet die Funktion f. Dazu genügt es zu zeigen, dass an jedem Knoten v die zugehörige Subfunktion berechnet wird. Wir beweisen diese Aussage mit Induktion, wobei wir die Knoten in einer umgekehrten topologischen Ordnung durchlaufen (d.h., jeder Knoten wird erst bearbeitet, nachdem seine Nachfolger abgearbeitet wurden). Die Aussage gilt für die Senken, weil an den Senken die konstanten Funktionen berechnet werden. Für einen inneren Knoten v, der mit x_i markiert ist und an dem die Funktion g berechnet werden soll, gilt nach Induktionsannahme, dass am 0-Nachfolger die Funktion $g_{|x_i=0}$ und am 1-Nachfolger die Funktion $g_{|x_i=1}$ berechnet wird. Aus der Shannon-Zerlegung folgt, dass an v die Funktion g berechnet wird.

Wenn das konstruierte OBDD nicht minimale Größe hat, gibt es ein OBDD für f,

das für ein i weniger als $|S_i|$ Knoten enthält, die mit x_i markiert sind. Da es aber $|S_i|$ verschiedene Subfunktionen von f der Form $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}}$ gibt, die essentiell von x_i abhängen, gibt es eine Zuweisung $c = (c_1, \ldots, c_{i-1})$ zu $x_1, \ldots, x_{i-1}$, für die $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}} \in S_i$ gilt, so dass der zugehörige Pfad entweder zu einer Senke führt oder zu einem mit x_j markierten Knoten, wobei $j > i$, oder zu einem mit x_i markierten Knoten, an dem eine andere Subfunktion als $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}}$ berechnet wird. Die ersten beiden Fälle führen zum Widerspruch, weil $f_{|x_1=c_1,\ldots,x_{i-1}=c_{i-1}}$ essentiell von x_i abhängt, der letzte Fall, weil in einem OBDD an jedem Knoten nur eine Subfunktion berechnet werden kann.

Jedes minimale OBDD für f und die Variablenordnung $x_1, \ldots, x_n$ muss also für jede Subfunktion $g \in S_i$ einen mit x_i markierten Knoten enthalten, dessen Nachfolger Knoten sind, an denen $g_{|x_i=0}$ bzw. $g_{|x_i=1}$ berechnet wird. Ein OBDD für f, das zu dem oben konstruierten nicht isomorph ist, muss daher zusätzliche Knoten enthalten, ist also nicht minimal. □

Die Eigenschaft, dass es für jede Funktion f und jede Variablenordnung π nur ein minimales OBDD gibt, kann nur dann praktisch genutzt werden, wenn das minimale OBDD effizient aus jedem anderen OBDD für f und π berechnet werden kann. Wir zeigen zunächst, dass wir das minimale OBDD erhalten, wenn wir ein gegebenes OBDD solange reduzieren, bis keine der Reduktionsregeln mehr anwendbar ist; dies ist eine weitere Motivation, OBDDs minimaler Größe bezüglich einer festen Variablenordnung als reduzierte OBDDs zu bezeichnen.

Lemma 5.3 Sei G ein OBDD für die Funktion f und die Variablenordnung $x_1, \ldots, x_n$. Das OBDD G ist genau dann isomorph zum minimalen OBDD G^* für f und die Variablenordnung $x_1, \ldots, x_n$, wenn auf G keine der Reduktionsregeln anwendbar ist.

Beweis Auf ein minimales OBDD ist offensichtlich keine Reduktionsregel anwendbar; daher genügt es zu zeigen, dass auf einem OBDD G, das nicht minimal ist, eine der Regeln anwendbar ist. Sei f o.B.d.A. keine konstante Funktion. Wir gehen davon aus, dass G nicht mehrere 0- oder 1-Senken enthält, da diese sonst verschmolzen werden können.

Aus dem Beweis von Satz 5.2 folgt, dass jedes OBDD für f für jede Subfunktion in S_i mindestens einen Knoten enthält, der mit x_i markiert ist. Wenn G zum minimalen OBDD G^* nicht isomorph ist, gibt es mindestens ein i, so dass G mehr als $|S_i|$ Knoten enthält, die mit x_i markiert sind. Sei i^* das größte derartige i. Dann werden in G alle Subfunktionen in S_j mit $j > i^*$ und die beiden konstanten Funktionen durch genau einen Knoten dargestellt. Da es mehr als $|S_i|$ mit x_i markierte Knoten gibt, enthält G entweder mindestens einen Knoten u, der mit x_i markiert ist und an dem eine Subfunktion g berechnet wird, die von x_i nicht essentiell abhängt, oder es gibt zwei Knoten v und w, die mit x_i markiert sind und an denen die gleiche Subfunktion $h \in S_i$ berechnet wird. Im ersten Fall gilt

$g = g_{|x_i=0} = g_{|x_i=1}$. Da die Funktionen $g_{|x_i=0}$ und $g_{|x_i=1}$ am gleichen Knoten dargestellt werden, kann der Knoten u mit der Deletion Rule entfernt werden. Im zweiten Fall werden an den Nachfolgern von v und w die Funktionen $h_{|x_i=0}$ bzw. $h_{|x_i=1}$ berechnet. Für jede dieser Funktionen gibt es in G nur einen Knoten, daher können v und w aufgrund der Merging Rule verschmolzen werden. □

Wir wollen jetzt zeigen, wie aus einem OBDD für die Funktion f und die Variablenordnung π, o.B.d.A. $x_1, \ldots, x_n$, das reduzierte OBDD berechnet werden kann. Es genügt zwar, solange Reduktionsregeln anzuwenden, bis keine Regel mehr anwendbar ist, wir sollten aber verhindern, dass für einen Knoten häufig geprüft werden muss, ob eine der Regeln anwendbar ist. Der Beweis von Lemma 5.3 legt nahe, die Reduktion bottom-up durchzuführen, weil für Knoten endgültig entschieden werden kann, ob sie gelöscht oder mit anderen verschmolzen werden können, wenn auf ihre Nachfolger keine Regel mehr angewandt werden kann.

Jeder innere Knoten v des OBDDs G wird als Record gespeichert, das eine Knotennummer $v.id$, den Index $v.var$ der Variablen, mit der v markiert ist, und Zeiger $v.null$ und $v.eins$ auf den 0- und den 1-Nachfolger von v enthält. Da wir keine rückwärts gerichteten Zeiger haben, können Knoten des OBDDs nicht direkt gelöscht werden. Wenn wir einen Knoten v mit der Deletion Rule löschen wollen, müssen die Zeiger auf v auf den Nachfolger von v umgesetzt werden; um einen Knoten v mit einem anderen Knoten w zu verschmelzen, müssen die Zeiger auf v auf den Knoten w umgesetzt werden. Wir realisieren das Löschen, indem wir v die Nummer seines Nachfolgers bzw. die Nummer von w geben. Da es jetzt mehrere Knoten mit derselben Nummer gibt, speichern wir in einem zusätzlichen Array Z zu jeder Nummer, welcher Knoten mit dieser Nummer die übrigen repräsentiert. Wir erhalten den folgenden Algorithmus (Bryant (1986)).

Algorithmus 5.4
Eingabe: Ein OBDD G für f mit der Ordnung $x_1, \ldots, x_n$.
Ausgabe: Das reduzierte OBDD G^* für f und die Ordnung $x_1, \ldots, x_n$.

- Durchlaufe G von der Quelle mit einem depth first search Ansatz. Dabei
  - verschmelze alle 0-Senken zu einer 0-Senke und alle 1-Senken zu einer 1-Senke, und setze

    $$\text{0-Senke}.id := 0;\ Z[\text{0-Senke}.id] := \text{0-Senke};$$
    $$\text{1-Senke}.id := 1;\ Z[\text{1-Senke}.id] := \text{1-Senke};$$

  - ordne alle mit x_i markierten Knoten in die Liste S_i ein $(i = 1, \ldots, n)$.
- $id := 1$.

- FOR $i := n$ DOWNTO 1 DO
  - *Anwendung der Deletion Rule:* Suche in S_i alle Knoten v, für die $v.null.id = v.eins.id$ gilt. Lösche v aus dem OBDD durch die Zuweisung $v.id := v.null.id$, und entferne v aus der Liste S_i.
  - Sortiere die Liste S_i, wobei jeder Knoten v als Schlüssel das Paar $(v.null.id, v.eins.id)$ erhält. Als Ordnung benutzen wir z.B. die lexikographische Ordnung auf den Paaren.
  - *Anwendung der Merging Rule:* Knoten, die verschmolzen werden können, stehen in der sortierten Liste hintereinander; daher muss die Liste nur einmal durchlaufen werden, um diese Knoten zu erkennen. Ein Knoten v_1, der im reduzierten OBDD enthalten sein soll, bekommt die nächste freie Knotennummer, d.h.

    $$id := id + 1; v_1.id := id; Z[v_1.id] := v_1;$$

    Der 0- und der 1-Nachfolger von v_1 können gelöscht worden sein. Damit die Zeiger auf die Nachfolger von v_1 nicht auf gelöschte Knoten, sondern auf die nicht gelöschten Repräsentanten zeigen, setzen wir

    $$v_1.null := Z[v_1.null.id]; v_1.eins := Z[v_1.eins.id].$$

    Der Knoten v_1 kann nun mit anderen Knoten $v_2, \ldots, v_l$ durch die Zuweisungen

    $$v_2.id := v_1.id, \ldots, v_l.id := v_1.id$$

    verschmolzen werden.
- Entferne die gelöschten Knoten.

Alle Operationen außer dem Sortieren benötigen pro Knoten nur konstante Zeit, insgesamt also $O(|G|)$. Bryant (1986) benutzt für das Sortieren ein allgemeines Sortierverfahren, die Rechenzeit beträgt daher $O(\sum_{i=1}^{n} |S_i| \log |S_i|) = O(|G| \log |G|)$. Für jeden Knoten v müssen $v.id$, $v.var$, $v.null$ und $v.eins$ gespeichert werden, außerdem benötigen wir das Array Z und die Zeiger auf die Nachfolger in den Listen S_i. Der Speicherbedarf beträgt dann $O(|G|+n)$. Die Zahl n der Variablen ist typischerweise viel kleiner als die Zahl der Knoten im OBDD G. Wir bemerken, dass die Reduktion durch einen modifizierten Bucketsort-Algorithmus sogar in linearer Zeit möglich ist (Sieling und Wegener (1993)).

5.3 OBDD-Darstellungen von ausgewählten Funktionen

In diesem Abschnitt wollen wir reduzierte OBDDs für einige spezielle Funktionen angeben, um ein Gefühl dafür zu erhalten, wie OBDDs arbeiten und in welchen Fällen ihr Einsatz sinnvoll ist. Weiterhin wird deutlich, dass die Wahl der Variablenordnung von OBDDs ein wichtiges Problem ist.

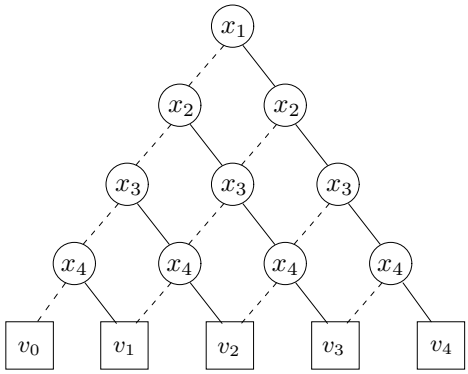


Abbildung 8: Ein OBDD für symmetrische Funktionen auf 4 Variablen

Wir beginnen mit OBDDs für symmetrische Funktionen. Wir haben schon gesehen, dass jede symmetrische Funktion in B_n durch einen Wertevektor $(v_0, \ldots, v_n)$ dargestellt werden kann. Abbildung 8 zeigt ein (nicht reduziertes) OBDD für die symmetrische Funktion mit dem Wertevektor $(v_0, \ldots, v_4)$. Man verifiziert leicht, dass die mit v_i markierte Senke genau für die Eingaben mit i Einsen erreicht wird. Die Verallgemeinerung für symmetrische Funktionen über n Variablen ist einfach. Dabei entsteht ein OBDD quadratischer Größe. (Auf der i-ten Ebene gibt es i Knoten, so dass die Gesamtzahl der Knoten gleich $\sum_{i=1}^{n+1} i = O(n^2)$ ist.) Je nach Wertevektor sind noch Verschmelzungen von Knoten möglich, so dass das OBDD auch viel kleiner werden kann, z.B. linear für die Parity-Funktion auf n Variablen. Da sich symmetrische Funktionen nicht verändern, wenn Variablen vertauscht werden, hängt die OBDD-Größe von symmetrischen Funktionen nicht von der Variablenordnung ab.

Das nächste Beispiel ist die Funktion DQF_n (disjunkte quadratische Form), die über den $2n$ Variablen $x_1, \ldots, x_{2n}$ durch

$$\mathrm{DQF}_n(x_1, \ldots, x_{2n}) = x_1x_2 \vee \ldots \vee x_{2n-1}x_{2n}$$

definiert ist. Wir betrachten die zwei Variablenordnungen $x_1, x_2, x_3, \ldots, x_{2n}$ und $x_1, x_3, x_5, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n}$. OBDDs für beide Variablenordnungen und DQF_6 sind in Abbildung 9 gezeigt. Man sieht sofort, dass die erste Variablenordnung zu einem viel kleineren OBDD als die zweite Variablenordnung führt. Wir wollen nun die OBDD-Größen berechnen und beginnen mit der ersten Variablenordnung. Offensichtlich hat das OBDD einen Knoten, der mit x_1 markiert ist und an dem DQF_n berechnet wird. Am 0-Nachfolger v_0 dieses Knotens wird die Funktion

$$f^0 = \mathrm{DQF}_{n|x_1=0} = x_3x_4 \vee \ldots \vee x_{2n-1}x_{2n}$$

und am 1-Nachfolger v_1 die Funktion

$$f^1 = \mathrm{DQF}_{n|x_1=1} = x_2 \vee x_3x_4 \vee \ldots \vee x_{2n-1}x_{2n}$$

berechnet. Offensichtlich hängt die an v_0 berechnet Funktion nicht essentiell von x_2 ab, wohl aber von x_3, so dass v_0 mit x_3 markiert ist. Dagegen ist v_1 mit x_2

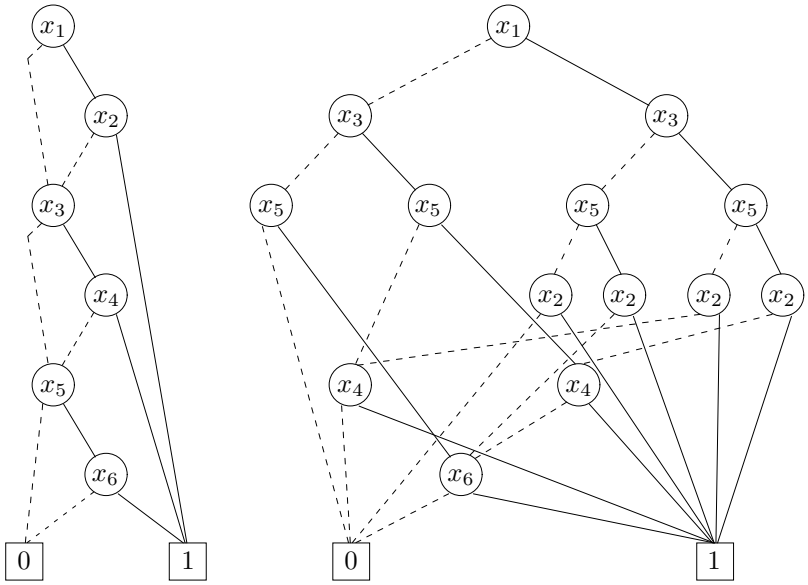


Abbildung 9: OBDDs für die Funktion DQF

markiert. Am 0-Nachfolger von v_1 wird die Funktion $f^1_{|x_2=0} = f^0$ berechnet, und am 1-Nachfolger von v_1 die Funktion $f^1_{|x_2=1} = 1$. Die 1-Kante von v_1 zeigt also zur 1-Senke. Insgesamt folgt, dass es jeweils einen mit x_1 und mit x_2 markierten Knoten gibt. Eine der ausgehenden Kanten zeigt zur 1-Senke und die andere zu einem Knoten, an dem $f^0 = \mathrm{DQF}_{n-1}(x_3, \ldots, x_{2n})$ berechnet wird. Also ist die OBDD-Größe $S(n)$ von DQF_n gleich der Summe von 2 und der OBDD-Größe $S(n-1)$ von DQF_{n-1}. Wegen $S(1) = 4$ folgt, dass reduzierte OBDDs von DQF_n für die erste Variablenordnung genau $2n + 2$ Knoten haben.

Der Einfachheit halber wollen wir die OBDD-Größe von DQF_n für die zweite Variablenordnung nur abschätzen. Wir zählen nur die mit x_2 markierten Knoten. Die gesamte OBDD-Größe ist dann auf jeden Fall größer als die berechnete Anzahl von x_2-Knoten. Nach Satz 5.2 ist die Anzahl von x_2-Knoten gleich der Anzahl der Subfunktionen von DQF_n, die durch Konstantsetzen der Variablen $x_1, x_3, x_5, \ldots, x_{2n-1}$ entstehen und die von x_2 essentiell abhängen. Sei

$$F[c_1, c_3, \ldots, c_{2n-1}](x_2, x_4, \ldots, x_{2n})$$

die Subfunktion von DQF_n, die entsteht, wenn $x_1, x_3, x_5, \ldots, x_{2n-1}$ durch $c_1, c_3, c_5, \ldots, c_{2n-1}$ ersetzt wurden. Sei $J(c_1, c_3, \ldots, c_{2n-1})$ die Menge der ungeraden Zahlen j mit $c_j = 1$. Offensichtlich ist

$$F[c_1, c_3, \ldots, c_{2n-1}](x_2, x_4, \ldots, x_{2n}) = \bigvee_{j \in J(c_1, c_3, \ldots, c_{2n-1})} x_{j+1}.$$

Die Funktion hängt essentiell von x_2 ab, wenn $1 \in J$, also wenn $c_1 = 1$. Offensichtlich sind für verschiedene Belegungen von $c_1, c_3, \ldots, c_{2n-1}$ die Funktionen $F[c_1, c_3, \ldots, c_{2n-1}]$ verschieden. Es gibt 2^{n-1} Belegungen von $c_1, c_3, \ldots, c_{2n-1}$ mit

$c_1 = 1$. Also hat das OBDD 2^{n-1} mit x_2 markierte Knoten. Wir bemerken, dass man die OBDD-Größe von DQF_n für die zweite Variablenordnung auch genau ausrechnen kann; sie beträgt 2^{n+1}. Wir erhalten dieselbe OBDD-Größe für jede Variablenordnung, für die für jedes Paar (x_i, x_{i+1}), i ungerade, eine der beiden Variablen in der ersten Hälfte der Variablenordnung und die andere in der zweiten Hälfte der Variablenordnung angeordnet ist.

Die Wahl der Variablenordnung kann also zwischen polynomieller und exponentieller OBDD-Größe entscheiden. Aus praktischer Sicht kann dies bedeuten, dass eine Anwendung in dem einen Fall erfolgreich beendet werden kann, während sie in dem anderen Fall wegen Speichermangel abgebrochen werden muss (man vergleiche die OBDD-Größen von DQF_n und die beiden o.g. Variablenordnungen für $n = 100$). Bei der Funktion DQF ist intuitiv klar, dass die Variablen x_i und x_{i+1} für ungerades i „zusammengehören“ und daher auch in der Variablenordnung zusammenstehen sollten.

Leider haben auch sehr wichtige Funktionen wie die Addition die Eigenschaft, dass die Variablenordnung zwischen polynomieller und exponentieller OBDD-Größe entscheiden kann. Wir bezeichnen im Folgenden das $(i-1)$-te Bit der Summe von zwei i-Bit Zahlen $x_{i-1}, \ldots, x_0$ und $y_{i-1}, \ldots, y_0$ mit ADD_i. Mit ähnlichen Methoden wie bei der Funktion DQF kann gezeigt werden, dass bei einer Variablenordnung, bei der alle x-Variablen vor allen y-Variablen getestet werden, die OBDD-Größe für ADD_i mindestens $2^{\Omega(i)}$ beträgt (Übungsaufgabe). Wir betrachten hier nur die (natürlicheren) Variablenordnungen $x_0, y_0, \ldots, x_{i-1}, y_{i-1}$ und $x_{i-1}, y_{i-1}, \ldots, x_0, y_0$. OBDDs für die beiden Variablenordnungen sind in Abb. 10 gezeigt. Es ist leicht, analog zur Funktion DQF zu verifizieren, dass diese OBDDs wirklich die Funktion ADD_i darstellen. Bei beiden Variablenordnungen haben die OBDDs lineare Größe. Dennoch gibt es einen Unterschied. Bei der zweiten Variablenordnung stimmen die unteren Teile der OBDDs für ADD_i und ADD_j für $i \neq j$ überein. Hieraus folgt, dass alle Bits der Summe zweier n-Bit Zahlen, also die Funktionen ADD_i, $i \in \{0, \ldots, n-1\}$ (und auch das Übertragsbit), zusammen in einem OBDD linearer Größe dargestellt werden können. Man überzeugt sich leicht, dass bei der ersten Variablenordnung quadratische Größe erforderlich ist.

Schließlich gibt es auch viele Funktionen, für die OBDDs für jede Variablenordnung exponentielle Größe haben. Hierzu gehört z.B. die Multiplikation (genauer gesagt, die Funktion MUL_n, die das n-te Bit des Produktes zweier n-Bit Zahlen beschreibt). Wir wollen hier eine „künstliche“ Funktion mit dieser Eigenschaft vorstellen, um zu motivieren, warum es überhaupt passieren kann, dass Funktionen nur exponentielle OBDDs haben. Wir werden eine Methode vorstellen, mit der man beweisen kann, dass eine Funktion für alle Variablenordnung exponentielle OBDDs hat. Mit ähnlichen Beweismethoden kann man zeigen, dass MUL_n nur exponentielle OBDDs hat; allerdings erfordert dieser Beweis mehr Tricks und Techniken.

Wir definieren nun die Funktion ACH, eine Variante der sogenannten Achillesfersenfunktion von Ashar, Ghosh und Devadas (1991). Sei $n = 2^k$. Für die Variablen

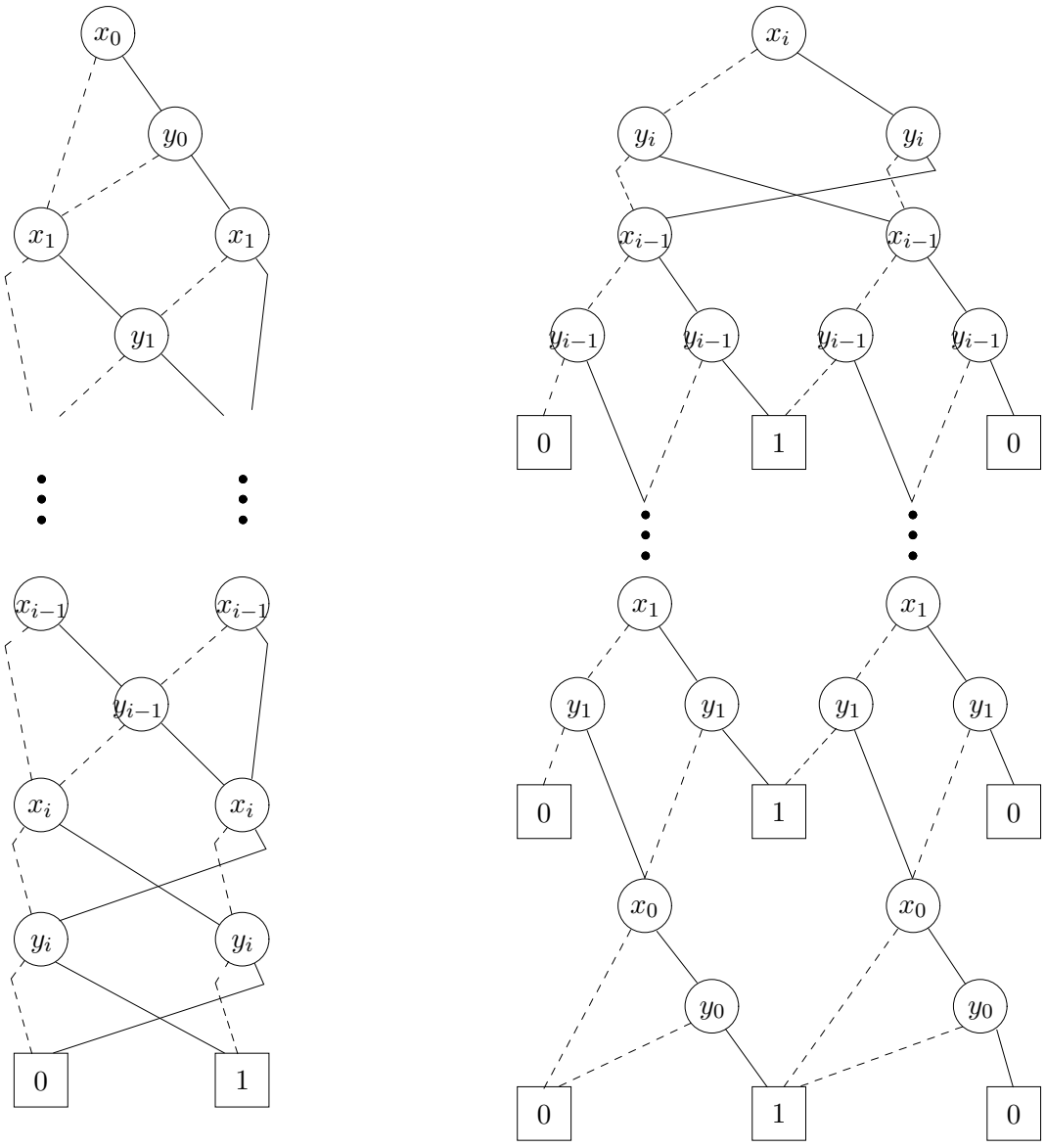


Abbildung 10: OBDDs für das i-te Bit der Addition

$s_{k-1}, \ldots, s_0$ bezeichne $|s|$ den Wert von $(s_{k-1}, \ldots, s_0)$ als Binärzahl interpretiert. Die Funktion ACH_n ist auf $2n+k$ Variablen $x_0, \ldots, x_{n-1}, y_0, \ldots, y_{n-1}, s_0, \ldots, s_{k-1}$ definiert:

$$\mathrm{ACH}_n(x, y, s) = x_0 y_{|s|} \vee x_1 y_{(|s|+1) \bmod n} \vee \ldots \vee x_{n-1} y_{(|s|+n-1) \bmod n}.$$

Die Funktion wird also folgendermaßen ausgewertet: Zuerst wird $y_0, \ldots, y_{n-1}$ um $|s|$ Positionen nach links rotiert. Anschließend wird die Funktion DQF angewendet, wobei jedes Paar aus einer x-Variablen und einer y-Variablen besteht. Die Idee für den Beweis, dass ACH für jede Variablenordnung exponentielle Größe hat, besteht darin, dass man für jede Variablenordnung einen Wert für $|s|$ finden kann, für den viele Paare von x- und y-Variablen, deren Konjunktion zu berechnen ist, in der Variablenordnung getrennt werden. Dies wollen wir nun formalisieren.

Sei ein OBDD G für ACH gegeben. Wir betrachten nur die x- und y-Variablen, die gemäß der Variablenordnung von G in einer Folge angeordnet sind. Sei L die Menge der vorderen n Variablen dieser Folge, und sei R die Menge der hinteren n Variablen dieser Folge. O.B.d.A. enthält L eine Menge L' von mindestens $n/2$ x-Variablen (und damit R eine Menge R' von mindestens $n/2$ y-Variablen). Anderenfalls können wir den folgenden Beweis ebenfalls anwenden, müssen aber die Rollen der x- und y-Variablen vertauschen.

Sei $t \in \{0, \ldots, n-1\}$. Sei $J(t)$ die Menge der Paare $(x_l, y_{(l+t) \bmod n})$, für die $x_l \in L'$ und $y_{(l+t) \bmod n} \in R'$. Die Menge $J(0) \cup \ldots \cup J(n-1)$ enthält mindestens $n^2/4$ solche Paare (jedes Paar einer x-Variablen aus R' und einer y-Variablen aus L' kommt in dieser Menge vor). Dann gibt es ein t, für das $|J(t)| \geq n/4$. (Wären alle Mengen $J(\cdot)$ kleiner, hätte die Vereinigung dieser Mengen nicht $n^2/4$ Elemente.) Wir belegen nun die s-Variablen so, dass $|s| = t$, und berechnen das OBDD für die entsprechende Subfunktion von ACH. Wie oben erwähnt, ist dies die Funktion DQF. Sei $J(t) \supseteq \{(x_{i(1)}, y_{j(1)}), \ldots, (x_{i(n/4)}, y_{j(n/4)})\}$. Wir setzen alle x-Variablen, die nicht in $\{x_{i(1)}, \ldots, x_{i(n/4)}\}$ enthalten sind, und alle y-Variablen, die nicht in $\{y_{j(1)}, \ldots, y_{j(n/4)}\}$ enthalten sind, auf 0. Nach Konstruktion entsteht die Subfunktion

$$x_{i(1)} y_{i(1)} \vee \ldots \vee x_{i(n/4)} y_{i(n/4)},$$

wobei alle x-Variablen vor allen y-Variablen angeordnet sind. Nach den Betrachtungen zur Funktion DQF folgt, dass das OBDD G' für diese Subfunktion und eine solche Variablenordnung $2^{n/4+1}$ Knoten hat. Es ist leicht einzusehen, dass sich OBDDs beim Konstantsetzen von Variablen nicht vergrößern (siehe auch Abschnitt 5.5). Da das OBDD G' aus G durch Konstantsetzen von Variablen entstanden ist, folgt, dass auch G mindestens $2^{n/4+1}$ Knoten hat.

Wir haben schon bei der Funktion DQF gesehen, dass zusammengehörende Variablen zusammen in der Variablenordnung angeordnet sein sollten. Bei der Funktion ACH entscheidet die Belegung der s-Variablen, welche x- und y-Variablen zusammengehören, so dass es nicht möglich ist, eine Variablenordnung zu finden, bei der alle zusammengehörenden Paare dicht beieinander angeordnet sind. Als

Faustregel kann man sagen, dass Funktionen, bei denen jede Variable mit vielen anderen zusammengehört, häufig nur exponentielle OBDDs haben.

5.4 Das Variablenordnungsproblem

An den Beispielen im letzten Abschnitt haben wir gesehen, dass die Wahl der Variablenordnung zwischen polynomieller und exponentieller OBDD-Größe entscheiden kann. Daher ist es wichtig, gute Algorithmen für das Variablenordnungsproblem zu haben. Man unterscheidet zwei verschiedene Varianten des Variablenordnungsproblems. Bei der ersten ist die darzustellende Funktion f durch einen Schaltkreis gegeben. Die Aufgabe besteht darin, ein möglichst kleines OBDD für f zu konstruieren. Wir bemerken nur, dass dies ein NP-hartes Problem ist und dass nur Heuristiken für dieses Problem bekannt sind. Die Heuristiken versuchen, aus der Schaltkreisbeschreibung Zusammenhänge zwischen der Variablen zu extrahieren und zusammengehörende Variablen zusammen anzuordnen. Ein Beispiel für eine solche Heuristik besteht darin, den Schaltkreis von den Ausgängen mit einem DFS-Durchlauf zu durchlaufen und die Variablen in der Reihenfolge anzuordnen, wie sie beim DFS-Durchlauf gefunden werden. Man überlegt leicht, dass bei der Funktion DQF z.B. aus der Polynomdarstellung in jedem Fall eine optimale Variablenordnung gefunden wird.

Bei der zweiten Variante des Variablenordnungsproblem geht man davon aus, dass die darzustellende Funktion bereits durch ein OBDD gegeben ist. Formal ist dieses Problem folgendermaßen definiert:

MinOBDD
Eingabe: Ein OBDD für f.
Ausgabe: Ein OBDD minimaler Größe für f.

Wir weisen noch einmal darauf hin, dass hier das Ziel ist, eine gute Variablenordnung zu berechnen, während bei der Reduktion die Variablenordnung fest ist. Die Motivation für das Problem MinOBDD ist die folgende: In der Einleitung des Kapitels haben wir beschrieben, wie ein Schaltkreis in ein OBDD umgerechnet werden kann. Im Laufe dieser Rechnung verändert sich die Menge der dargestellten Funktionen. Damit verändert sich auch die Menge der Variablenordnungen, bezüglich derer die Darstellung kompakt ist. Also ist es sinnvoll, im Laufe der Umformung die Variablenordnung zu verbessern. Dies ist genau das Problem MinOBDD. Die Idee, die Variablenordnung im Laufe von Berechnungen auf OBDDs zu ändern, wird in der Literatur als dynamische Variablenordnung bezeichnet (Rudell (1993)).

Leider gibt es auch für MinOBDD vermutlich keine effizienten Algorithmen. Bollig und Wegener (1996) haben bewiesen, dass das Problem NP-hart ist. Sieling (1998) hat bewiesen, dass auch die Existenz von polynomiellen Approximationsalgorithmen mit jeder konstanten Güte $c > 1$ impliziert, dass P=NP ist. (Ein Approximationsalgorithmus mit Güte c für MinOBDD berechnet für alle Eingaben ein

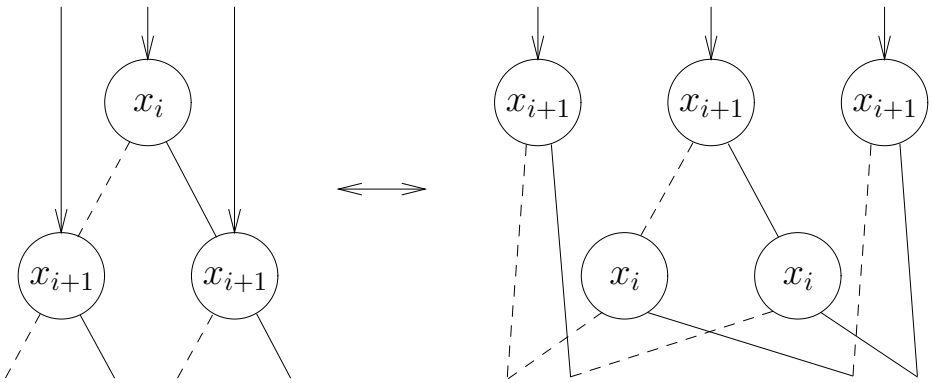


Abbildung 11: Ausführung der Operation SWAP(x_i, x_{i+1})

OBDD, das um höchstens den Faktor c größer als ein minimales OBDD ist.) Also müssen wir uns auch beim Problem MinOBDD mit Heuristiken zufrieden geben.

Eine elementare Operation beim Verändern der Variablenordnung von OBDDs ist das Vertauschen benachbarter Variablen. Sei $x_1, \ldots, x_n$ die Variablenordnung des gegebenen OBDDs. SWAP(x_i, x_{i+1}) verändert die Variablenordnung in $x_1, \ldots, x_{i-1}, x_{i+1}, x_i, x_{i+2}, \ldots, x_n$. Es folgt direkt aus Satz 5.2, dass sich nur die Ebenen mit den Variablen x_i und x_{i+1} verändern. In Abbildung 11 ist gezeigt, in welcher Weise die Kanten verändert werden müssen.

Eine naheliegende Heuristik ist eine lokale Suche. Wir nennen zwei Variablenordnungen benachbart, wenn sie sich durch eine Swap-Operation ineinander überführen lassen. Bei einer lokalen Suche probiert man alle möglichen Swap-Operationen für die aktuelle Variablenordnung aus. Wenn man eine Swap-Operation findet, bei der sich die OBDD-Größe verkleinert, wird die neue Variablenordnung zur aktuellen Variablenordnung. Dies wird solange iteriert, bis keine Verbesserung der OBDD-Größe mehr möglich ist. Dieser Algorithmus (und Verfeinerungen davon) wurde von Ishiura, Sawada und Yajima (1991) untersucht.

Es ist leicht einzusehen, dass die lokale Suche häufig in lokalen Optima „steckenbleibt“. Dasselbe gilt für den sogenannten Sifting-Algorithmus (Rudell (1993)), der eine lokale Suche mit einer anderen Nachbarschaftsbeziehung ist.

Beim Sifting-Algorithmus wird eine Variable x_i ausgewählt, und es wird nach einer optimalen Position für x_i gesucht. Dazu wird x_i durch iteriertes Anwenden der Swap-Operation probeweise an alle möglichen Stellen in der Variablenordnung geschoben, und es wird die OBDD-Größe bestimmt. Schließlich wird x_i an die Stelle geschoben, die zu minimaler OBDD-Größe führt. Dies wird (für andere Variablen) iteriert, bis keine Verbesserung der OBDD-Größe mehr möglich ist. (Eventuell werden Variablen auch mehrfach verschoben.) Auch der Sifting-Algorithmus wird häufig in lokalen Optima enden. Allerdings wird berichtet, dass der Sifting-Algorithmus sehr schnell ist und gute Ergebnisse liefert.

5.5 Algorithmen auf OBDDs

Auswertung

Eingabe: Ein OBDD G für $f \in B_n$, $a \in \{0,1\}^n$.
Ausgabe: $f(a)$.
Rechenzeit: $O(n)$. **Speicherplatz:** $O(|G|)$.

Wir durchlaufen das OBDD wie oben beschrieben von der Quelle zu einer Senke und geben die Markierung der erreichten Senke aus. Weil jede Variable höchstens einmal getestet werden darf, ist die Rechenzeit durch $O(n)$ beschränkt.

Erfüllbarkeit

Eingabe: Ein OBDD G für $f \in B_n$.
Ausgabe: 1, falls f ungleich der Nullfunktion ist, anderenfalls 0.
Rechenzeit: $O(|G|)$. **Speicherplatz:** $O(|G|)$.

Durchlaufe das OBDD von der Quelle mit einem depth first search Ansatz, bis eine 1-Senke erreicht wird. Der gefundene Pfad von der Quelle zu der 1-Senke gibt eine Eingabe a mit $f(a) = 1$ an; wenn keine 1-Senke erreicht wird, gibt es auch keine derartige Eingabe.

Wenn G reduziert ist, ist der Erfüllbarkeitstest der Test, ob die Quelle von G ungleich der 0-Senke ist. Dieses ist sogar in konstanter Zeit möglich.

Erfüllbarkeit-Anzahl

Eingabe: Ein OBDD G für $f \in B_n$.
Ausgabe: $|f^{-1}(1)|$.
Rechenzeit: $O(|G|)$. **Speicherplatz:** $O(|G|)$.

Wir wollen für jede Kante und jeden Knoten von G berechnen, für wieviele Eingaben diese Kante oder dieser Knoten durchlaufen wird. Die Quelle wird offensichtlich für alle 2^n Eingaben durchlaufen, und die 1-Senken werden für insgesamt $|f^{-1}(1)|$ Eingaben erreicht. Wir betrachten nun einen inneren Knoten v, der mit x_i markiert ist und der für c Eingaben durchlaufen wird. Wenn der Knoten v für die Eingabe $a = (a_1, \ldots, a_n)$ erreicht wird, wird er auch für $a' = (a_1, \ldots, a_{i-1}, \bar{a}_i, a_{i+1}, \ldots, a_n)$ erreicht, weil die Variable x_i nicht vor v getestet werden darf. Für a wird die a_i-Kante, die v verlässt, durchlaufen, für a' die $\bar{a}_i$-Kante. Die c Eingaben, für die v erreicht wird, werden also auf die 0-Kante und die 1-Kante gleichmäßig verteilt.

Wenn ein Knoten v die eingehenden Kanten $e_1, \ldots, e_k$ hat, müssen wir die Anzahl der Eingaben, für die $e_1, \ldots, e_k$ durchlaufen werden, addieren, um die Zahl der Eingaben zu erhalten, für die v erreicht wird. Im folgenden Algorithmus vermei-

den wir, auch für die Kanten die Zahl der Eingaben, für die sie erreicht werden, zu speichern.

- Markiere die Quelle mit 2^n und die übrigen Knoten mit 0.
- Durchlaufe G in einer topologischen Ordnung. Wenn ein Knoten v erreicht wird, der mit c markiert ist, addiere zu den Markierungen beider Nachfolger von v den Wert $c/2$.
- Das Ergebnis ist die Markierung der 1-Senke bzw. die Summe der Markierungen der 1-Senken.

Erfüllbarkeit-Alle

Eingabe: Ein OBDD G für $f \in B_n$.
Ausgabe: $f^{-1}(1)$.
Rechenzeit: $O(|G| + n|f^{-1}(1)|)$. **Speicherplatz:** $O(|G|)$.

Wir suchen alle Pfade, die von der Quelle zu einer 1-Senke führen. Wir erhalten alle erfüllenden Belegungen für die Eingabevariablen von f, wenn wir für jeden derartigen Pfad die getesteten Variablen so belegen, dass der Pfad durchlaufen wird, und die nicht getesteten Variablen auf alle möglichen Weisen belegen.

Die Pfade von der Quelle zur 1-Senke können wir mit einem einfachen rekursiven Algorithmus berechnen, der an der Quelle beginnt. Für jeden erreichten inneren Knoten v rufen wir dieselbe Prozedur nacheinander für den 0-Nachfolger von v und den 1-Nachfolger von v rekursiv auf. Wenn wir eine 1-Senke erreichen, rekonstruieren wir den Pfad, auf dem wir zu dieser Senke gelangt sind, belegen die getesteten Variablen so, dass dieser Pfad durchlaufen wird, und die nicht getesteten Variablen auf alle möglichen Weisen. Da die Größe der Ausgabe $n|f^{-1}(1)|$ ist, ist die Rechenzeit $O(|G| + n|f^{-1}(1)|)$ optimal.

Äquivalenztest

Eingabe: OBDDs G_f und G_g für $f, g \in B_n$.
Ausgabe: 1, wenn $f = g$, anderenfalls 0.
Rechenzeit: $O(|G_f| + |G_g|)$. **Speicherplatz:** $O(|G_f| + |G_g|)$.

Da reduzierte OBDDs bei gleicher Variablenordnung bis auf Isomorphie eindeutig sind, genügt es, G_f und G_g zu reduzieren und die reduzierten OBDDs G_f^* und G_g^* auf Isomorphie zu testen. Die Reduktion ist mit dem oben erwähnten Algorithmus in linearer Zeit auf linearem Platz möglich. Der Isomorphietest ist ebenfalls in linearer Zeit möglich, weil OBDDs Graphen mit einer Quelle und markierten Knoten und Kanten sind. Wir durchlaufen G_f^* und G_g^* simultan mit depth first search beginnend bei den Quellen und testen, ob die jeweils erreichten Knoten die gleiche Markierung haben. Wenn f und g gemeinsam in einem reduzierten OBDD

dargestellt werden, ist der Äquivalenztest sogar in konstanter Zeit möglich, da es genügt zu testen, ob die Zeiger für f und g auf denselben Knoten zeigen.

Synthese

Eingabe: OBDDs G_f und G_g für $f, g \in B_n$, ein Operator $\otimes \in B_2$.
Ausgabe: Ein OBDD G für $f \otimes g$.
Rechenzeit: $O(|G_f||G_g|)$. **Speicherplatz:** $O(|G_f||G_g|)$.

Wir wollen zunächst zeigen, wie wir für eine Eingabe $a \in \{0,1\}^n$ mit einem simultanen Durchlauf durch G_f und G_g die Funktionswerte $f(a)$ und $g(a)$ berechnen können. Bei dieser Berechnung soll jedes Eingabebit x_i nur einmal abgefragt werden und dann gleichzeitig für die Rechnung in G_f und G_g benutzt werden. Wir benötigen dafür, dass beide OBDDs dieselbe Variablenordnung, o.B.d.A. $x_1, \ldots, x_n$, haben. Wir beginnen in beiden OBDDs die Rechnung an den Quellen. Wenn wir in G_f einen Knoten v_f und in G_g einen Knoten v_g erreicht haben, sind die folgenden Situationen möglich.

1. Fall: v_f und v_g sind mit x_i markiert.
Wir gehen in G_f und G_g gemäß dem Wert von a_i zum 0- oder 1-Nachfolger von v_f bzw. v_g.

2. Fall: v_f ist mit x_i markiert, v_g ist mit x_j markiert, und $i > j$.
Die Variable x_j steht in der Variablenordnung vor x_i, daher warten wir in G_f am Knoten v_f, gehen aber in G_g gemäß dem Wert von a_j zum 0- oder 1-Nachfolger von v_g.

3. Fall: v_f ist eine Senke, v_g ist mit x_j markiert.
Wie im 2. Fall warten wir an v_f und gehen in G_g gemäß dem Wert von a_j zum 0- oder 1-Nachfolger.

4. Fall: v_f ist mit x_i markiert, v_g ist mit x_j markiert, und $i < j$.
Analog zum 2. Fall warten wir in G_g am Knoten v_g und gehen in G_f gemäß dem Wert von a_i zum 0- oder 1-Nachfolger von v_f.

5. Fall: v_f ist mit x_i markiert, v_g ist eine Senke.
Wir gehen gemäß dem Wert von a_i zum 0- oder 1-Nachfolger von v_f und warten an v_g.

6. Fall: v_f und v_g sind Senken.
Die Markierungen von v_f und v_g sind die gesuchten Funktionswerte $f(a)$ und $g(a)$. Wir können $(f \otimes g)(a)$ berechnen, indem wir die Markierungen der Senken mit $\otimes$ verknüpfen.

Wir konstruieren nun ein OBDD G, in dem jeder Pfad einer derartigen simultanen Rechnung entspricht. G enthält für jedes Paar von einem Knoten v_f aus G_f und

v_g aus G_g einen Knoten (v_f, v_g). Die neue Quelle ist das Paar der Quellen von G_f und G_g. Die Markierung jedes Knotens (v_f, v_g) und seine Nachfolger erhalten wir durch dieselbe Fallunterscheidung wie oben.

1. Fall: v_f und v_g sind mit x_i markiert.
Der Knoten (v_f, v_g) wird mit x_i markiert, sein 0-Nachfolger ist der Knoten $(v_{f,0}, v_{g,0})$, sein 1-Nachfolger $(v_{f,1}, v_{g,1})$. Dabei bezeichnen $v_{f,c}$ bzw. $v_{g,c}$ den c-Nachfolger von v_f bzw. v_g.

2. Fall: v_f ist mit x_i markiert, v_g ist mit x_j markiert, und $i > j$.
Der Knoten (v_f, v_g) wird mit x_j markiert, seine Nachfolger sind $(v_f, v_{g,0})$ bzw. $(v_f, v_{g,1})$, da am Knoten v_f gewartet wird.

3. Fall: v_f ist eine Senke, v_g ist mit x_j markiert.
Der Knoten (v_f, v_g) wird mit x_j markiert, seine Nachfolger sind $(v_f, v_{g,0})$ bzw. $(v_f, v_{g,1})$.

4. Fall: v_f ist mit x_i markiert, v_g ist mit x_j markiert, und $i < j$.
Der Knoten (v_f, v_g) wird mit x_i markiert, seine Nachfolger sind $(v_{f,0}, v_g)$ bzw. $(v_{f,1}, v_g)$.

5. Fall: v_f ist mit x_i markiert, v_g ist eine Senke.
Der Knoten (v_f, v_g) wird mit x_i markiert, seine Nachfolger sind $(v_{f,0}, v_g)$ bzw. $(v_{f,1}, v_g)$.

6. Fall: v_f und v_g sind Senken.
Der Knoten (v_f, v_g) ist eine Senke; die Markierung dieser Senke erhalten wir, indem wir die Markierungen von v_f und v_g mit $\otimes$ verknüpfen.

Das konstruierte OBDD wird auch Produktgraph von G_f und G_g genannt. Es simuliert die simultane Rechnung in G_f und G_g und ist daher ein OBDD für die Funktion $f \otimes g$. Der Produktgraph enthält in der Regel Knoten, die von der Quelle aus nicht erreichbar sind. Im Produktgraphen in Abb. 12 sind nur die fett gezeichneten Knoten und Kanten von der Quelle (die dem Paar der Quellen von G_f und G_g entspricht) erreichbar. Nachdem die nicht erreichbaren Knoten und Kanten entfernt sind, ist der entstandene Graph nicht notwendigerweise ein reduziertes OBDD, auch wenn G_f und G_g reduziert sind. Nachdem in dem Produktgraphen in Abb. 12 die nicht erreichbaren Knoten entfernt worden sind, können noch die Knoten (v_3, v_6) und (v_4, v_7) verschmolzen werden.

Der Algorithmus von Bryant (1986) vermeidet die Erzeugung von nicht erreichbaren Knoten, indem er die Berechnung des OBDDs für $f \otimes g$ an der Quelle beginnt und nur Nachfolger von bereits konstruierten Knoten erzeugt. Das OBDD wird in einer depth first search Ordnung aufgebaut. Der rekursive Algorithmus wird mit dem Paar der Quellen von G_f und G_g aufgerufen und gibt einen Zeiger auf das erzeugte OBDD zurück. Auf einem Paar (v_f, v_g) werden folgende Schritte ausgeführt:

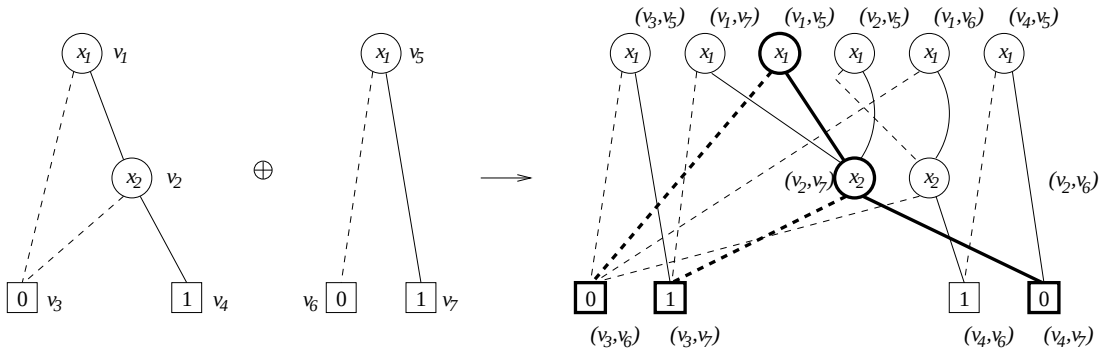


Abbildung 12: Beispiel für die Berechnung eines Produktgraphen

- Erzeuge einen Knoten für (v_f, v_g).
- Bestimme mit der Fallunterscheidung von oben die Markierung von (v_f, v_g).
- Wenn (v_f, v_g) ein innerer Knoten ist, bestimme mit der Fallunterscheidung von oben die Paare, die die Nachfolger von (v_f, v_g) sind. Teste nacheinander für jedes der beiden Paare, ob bereits ein Knoten erzeugt wurde.
  - Wenn ja, setze den Zeiger für den entsprechenden Nachfolger von (v_f, v_g) auf den erzeugten Knoten.
  - Wenn nein, erzeuge den entsprechenden Nachfolger mit einem rekursiven Aufruf, und setze den Zeiger für den Nachfolger von (v_f, v_g) auf den erzeugten Knoten.
- Gib an die aufrufende Prozedur einen Zeiger auf (v_f, v_g) zurück.

Der Test, ob für ein Paar bereits ein Knoten erzeugt wurde, verhindert, dass Teile des konstruierten OBDDs mehrfach erzeugt werden. Wir können dafür ein spezielles Array (die sog. Computed Table) der Größe $|G_f||G_g|$ benutzen, das für jedes Paar von Knoten aus G_f und G_g einen Zeiger auf den eventuell erzeugten Knoten oder ein *nil*-Zeiger, wenn noch kein Knoten erzeugt wurde, enthält. Daher ist die Größenordnung von Rechenzeit und Speicherplatz bei der Berechnung des Produktgraphen wie auch beim Synthesealgorithmus von Bryant $O(|G_f||G_g|)$. Aber auch wenn das erzeugte OBDD viel kleiner ist, ist die Größenordnung der Rechenzeit $|G_f||G_g|$, da das Array initialisiert werden muss. Man kann Speicherplatz sparen, wenn man statt des Arrays eine Hashtabelle benutzt, die kleiner als $|G_f||G_g|$ ist. Nun kann aber nicht mehr garantiert werden, dass der Test, ob für ein Paar bereits ein Knoten erzeugt wurde, in konstanter Zeit möglich ist, und damit auch nicht die Rechenzeit $O(|G_f||G_g|)$.

Eine weitere Verfeinerung des Algorithmus von Bryant testet im 3. Fall, wenn v_f eine Senke ist, (und analog im 5. Fall,) ob die Konstante c, mit der v_f markiert ist, für den Operator $\otimes$ dominierend ist, d.h., ob $c \otimes 0 = c \otimes 1$ gilt. In diesem

Fall kann die Rekursion abgebrochen werden, und (v_f, v_g) ist eine Senke mit der Markierung $c \otimes 0$.

Auch beim Synthesealgorithmus von Bryant ist das erzeugte OBDD in der Regel nicht reduziert. Die Reduktion kann aber in die Synthese integriert werden (Brace, Rudell, Bryant (1991)). Sobald für einen Knoten (v_f, v_g) beide Nachfolger berechnet worden sind, kann getestet werden, ob beide Nachfolger identisch sind, also ob die Deletion Rule anwendbar ist. In diesem Fall wird der Knoten (v_f, v_g) wieder entfernt und ein Zeiger auf seinen Nachfolger (v_f^*, v_g^*) an die aufrufende Prozedur zurückgegeben. Zudem wird in der Computed Table an der Stelle (v_f, v_g) gespeichert, dass der Knoten (v_f, v_g) durch (v_f^*, v_g^*) repräsentiert wird. Wenn dieses nicht gespeichert würde, müssten, wenn der Algorithmus später noch einmal für das Paar (v_f, v_g) aufgerufen wird, die Nachfolger von (v_f, v_g) noch einmal berechnet werden. Für die Merging Rule benötigen wir eine weitere Hashtabelle, die sog. Unique-Table. Wenn für einen Knoten (v_f, v_g) beide Nachfolger berechnet worden sind und die Deletion Rule nicht anwendbar ist, wird mit Hilfe der Unique Table getestet, ob es bereits einen Knoten $(\hat{v}_f, \hat{v}_g)$ gibt, der mit derselben Variablen markiert ist und denselben 0- und denselben 1-Nachfolger hat. Wenn dieses der Fall ist, können die beiden Knoten verschmolzen werden, d.h., der Algorithmus gibt einen Zeiger auf $(\hat{v}_f, \hat{v}_g)$ zurück und speichert wie eben in der Computed Table, dass (v_f, v_g) durch $(\hat{v}_f, \hat{v}_g)$ repräsentiert wird. Anderenfalls ist der Knoten (v_f, v_g) notwendig, und wir speichern in der Unique Table für den erzeugten Knoten das Tripel aus der Markierung des Knotens (v_f, v_g), seinem 0-Nachfolger und seinem 1-Nachfolger.

Wenn wir die Reduktion in die Synthese integrieren, ist es wichtig, dass das synthetisierte OBDD in einer depth first search Ordnung aufgebaut wird; bei der Synthese ohne Reduktion ist z.B. auch eine breadth first search Ordnung möglich. Der Vorteil der depth first search Ordnung liegt darin, dass es im erzeugten OBDD nur einen Pfad gibt, auf dem Knoten liegen, für die noch nicht beide Nachfolger berechnet worden sind. Es werden daher gleichzeitig höchstens n Knoten gespeichert, die eventuell später aufgrund einer Reduktionsregel wieder gelöscht werden können. Die Anzahl der Knoten, die insgesamt gespeichert werden müssen, ist daher durch $|G^*| + n$ beschränkt, wobei G^* das reduzierte OBDD für $f \otimes g$ ist. Bei einem Aufbau mit breadth first search ist es dagegen möglich, dass viel mehr Knoten gespeichert werden müssen, für die noch nicht beide Nachfolger berechnet worden sind und die damit zu einem späteren Zeitpunkt eventuell wieder gelöscht werden.

m-äre Synthese

Die Synthese kann auch auf m-äre Operatoren erweitert werden:

Eingabe: OBDDs $G_1, \ldots, G_m$ für die Funktionen $f_1, \ldots, f_m \in B_n$, eine Funktion $h \in B_m$.
Ausgabe: Ein OBDD für $h(f_1, \ldots, f_m)$.
Rechenzeit: $O(|G_1||G_2| \ldots |G_m|)$. **Speicherplatz:** $O(|G_1||G_2| \ldots |G_m|)$.

Der simultane Durchlauf durch m OBDDs ist genauso möglich wie der simultane Durchlauf durch zwei OBDDs. Dieser Durchlauf kann wiederum durch einen Produktgraphen von $G_1, \ldots, G_m$ simuliert werden. Dieser Produktgraph enthält für jedes m-Tupel $(v_1, \ldots, v_m)$ von Knoten aus $G_1, \ldots, G_m$ einen Knoten. Sei x_i die Variable, die von den Variablen, mit denen $v_1, \ldots, v_m$ markiert sind, als erste in der Variablenordnung steht. Dann wird am Knoten $(v_1, \ldots, v_m)$ die Variable x_i getestet. Die Nachfolger von $(v_1, \ldots, v_m)$ sind $(v_1^0, \ldots, v_m^0)$ bzw. $(v_1^1, \ldots, v_m^1)$, wobei v_j^c der c-Nachfolger von v_j im Graphen G_j ist, falls an v_j die Variable x_i getestet wird. Wenn an v_j nicht x_i getestet wird, müssen wir beim simultanen Durchlauf am Knoten v_j warten, daher setzen wir v_j^c auf v_j. Wenn alle Knoten $v_1, \ldots, v_m$ Senken sind, die mit $c_1, \ldots, c_m$ markiert sind, ist auch $(v_1, \ldots, v_m)$ eine Senke, die mit $h(c_1, \ldots, c_m)$ markiert ist.

Wie bei der binären Synthese kann mit dem DFS-Ansatz von Bryant die Konstruktion nicht erreichbarer Knoten verhindert werden, und es kann auch die Reduktion integriert werden.

Ersetzung durch Konstanten

Eingabe: Ein OBDD G für $f \in B_n$, eine Variable x_i, eine Konstante c.
Ausgabe: Ein OBDD G' für $f_{|x_i=c}$.
Rechenzeit: $O(|G|)$. **Speicherplatz:** $O(|G|)$.

Wenn die Quelle mit x_i markiert ist, definieren wir den c-Nachfolger der Quelle als neue Quelle. Anderenfalls durchlaufen wir G mit depth first search und testen für jeden Knoten, ob einer seiner Nachfolger w mit x_i markiert ist. In diesem Fall wird der Zeiger auf w auf den c-Nachfolger von w umgesetzt. Anschließend können die mit x_i markierten Knoten gelöscht werden. Nun sind möglicherweise nicht mehr alle Knoten des OBDDs von der Quelle aus erreichbar, z.B. ist im OBDD in Abb. 13 nach der Ersetzung von x_2 durch 0 die 1-Senke nicht mehr erreichbar. Die nicht erreichbaren Knoten werden ebenfalls gelöscht.

Auch wenn das OBDD G reduziert ist, ist das neue OBDD G' nicht notwendigerweise reduziert (s. Abb. 13). Es genügt, die Reduktion bottom-up auf den Schichten des OBDDs durchzuführen, die oberhalb von der Schicht der mit x_i markierten Knoten liegen. Unterhalb dieser Schicht wurden nur nicht erreichbare Knoten entfernt, aber keine Zeiger auf Nachfolger verändert. Daher gibt es dort keine Knoten, auf die die Deletion Rule oder die Merging Rule anwendbar ist.

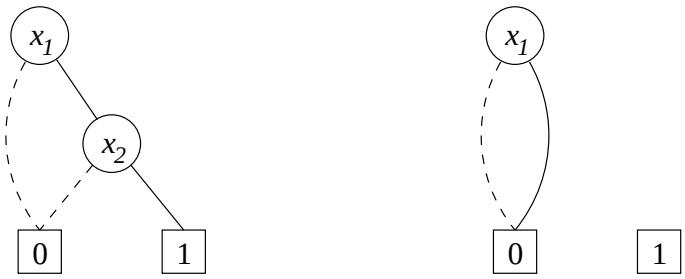


Abbildung 13: OBDDs für $f(x_1, x_2) = x_1 x_2$ und $f_{|x_2=0}$

Ersetzung durch Funktionen

Eingabe: OBDDs G_f und G_g für die Funktionen $f, g \in B_n$, eine Variable x_i.
Ausgabe: Ein OBDD G für $f_{|x_i=g}$.
Rechenzeit: $O(|G_f|^2|G_g|)$. **Speicherplatz:** $O(|G_f|^2|G_g|)$.

Die Funktion $f_{|x_i=g}$ kann mit der Shannon-Zerlegung geschrieben werden als

$$f_{|x_i=g} = \bar{g} f_{|x_i=0} \vee g f_{|x_i=1} = ite(g, f_{|x_i=1}, f_{|x_i=0}).$$

Dabei ist der ternäre Operator *ite* (if-then-else) definiert durch

$$ite(a, b, c) := \bar{a}c \vee ab.$$

Wir wenden daher den Synthesealgorithmus auf den ternären Operator *ite* und die OBDDs für die Funktionen g, $f_{|x_i=1}$ und $f_{|x_i=0}$ an und erhalten in der Zeit $O(|G_f|^2|G_g|)$ ein OBDD für $f_{|x_i=g}$. Die OBDDs für $f_{|x_i=1}$ und $f_{|x_i=0}$ müssen nicht explizit berechnet werden, wenn wir den Synthesealgorithmus so modifizieren, dass an mit x_i markierten Knoten direkt zum 1- bzw. 0-Nachfolger weitergegangen wird.

Quantifizierung

Eingabe: Ein OBDD G für $f \in B_n$, eine Variable x_i.
Ausgabe: Ein OBDD für $(\exists x_i : f) = f_{|x_i=0} \vee f_{|x_i=1}$ bzw. $(\forall x_i : f) = f_{|x_i=0} \wedge f_{|x_i=1}$.
Rechenzeit: $O(|G|^2)$. **Speicherplatz:** $O(|G|^2)$.

Wir wenden den Synthesealgorithmus auf OBDDs für $f_{|x_i=0}$ und $f_{|x_i=1}$ und $\vee$ bzw. $\wedge$ an und erhalten in der Zeit $O(|G|^2)$ ein OBDD für die gewünschte Funktion. Wie bei der Ersetzung durch Funktionen können wir vermeiden, dass die OBDDs für $f_{|x_i=0}$ und $f_{|x_i=1}$ berechnet und gespeichert werden, indem wir die Konstantsetzung von x_i in die Synthese integrieren.

Redundanztest

Eingabe: Ein OBDD G für $f \in B_n$, eine Variable x_i.
Ausgabe: 1, falls $f_{|x_i=0} \equiv f_{|x_i=1}$, anderenfalls 0.
Rechenzeit: $O(|G|)$. **Speicherplatz:** $O(|G|)$.

Ein reduziertes OBDD für f enthält genau dann mit x_i markierte Knoten, wenn f essentiell von x_i abhängt, also $f_{|x_i=0} \not\equiv f_{|x_i=1}$ gilt: Wenn ein OBDD für f keinen mit x_i markierten Knoten enthält, wird für $x_i = 0$ und $x_i = 1$ derselbe Funktionswert berechnet, die Funktion hängt also nicht essentiell von x_i ab. Wenn f nicht essentiell von x_i abhängt, gilt dieses auch für alle Subfunktionen von f. Daher ist die Menge S_i leer, und nach Satz 5.2 enthält das reduzierte OBDD für f keinen mit x_i markierten Knoten.

Es genügt also, mit einem Graphdurchlauf nach mit x_i markierten Knoten zu suchen, nachdem G reduziert worden ist. Dazu genügen lineare Rechenzeit und linearer Speicherplatz.

5.6 Schlussbemerkungen

OBDDs erlauben die kompakte Darstellung von vielen interessanten booleschen Funktionen, und es gibt effiziente Algorithmen für die Manipulation von Funktionen, die durch OBDDs dargestellt sind. Hierbei sollten wir allerdings nicht vergessen, dass z.B. bei der Umformung eines Schaltkreises in ein OBDD solche Operationen, insbesondere die Synthese, häufig hintereinander ausgeführt werden, so dass exponentielle Vergrößerungen auftreten können. Ebenso gibt es auch Funktionen, die nur exponentielle OBDDs haben. OBDDs sind also nicht *die* Datenstruktur für boolesche Funktionen, mit der sich alle Probleme lösen lassen, sondern nur ein heuristischer Ansatz, mit dem man manche Probleme lösen kann, die mit konventionellen Ansätzen (z.B. Darstellung von Funktionen durch Wertetabellen oder Schaltkreise) nicht praktisch lösbar sind. Manchmal können Probleme auch nur dadurch gelöst werden, dass OBDDs zusammen mit weiteren „Tricks“ eingesetzt werden. Ein weiterer Lösungsansatz besteht in der Entwicklung von Erweiterungen von OBDDs, die die kompakte Darstellung von einer größeren Klasse von Funktionen als OBDDs erlauben. Die Entwicklungen in diese Richtung sind allerdings zu umfangreich für die Darstellung in einer Stammvorlesung.

6 OBDD-Anwendungen beim Schaltkreisentwurf, bei der Schaltkreisverifikation und bei der zweistufigen Logikminimierung

6.1 Der Pentium Dividierer

Die Radix-4 Darstellung ist in [W1] Kap. 3.10 definiert, und es werden einige Eigenschaften gezeigt ([W1] 3.10 bis zum Beweis von Satz 3.10.2).

Der Dividierer im Pentium ist ein sog. SRT-Dividierer, der von Atkins (1969) beschrieben wurde. Wir behandeln den Dividierer nicht nur, um ein Beispiel eines Dividierers aus der Praxis zu sehen, sondern auch, um zu zeigen, dass OBDD-Methoden auch für den Schaltkreisentwurf hilfreich sind. Darüberhinaus zeigt der Dividierer, wie die Radix-4 Darstellung trickreich eingesetzt werden kann.

Die Schulmethode für die Division benötigt für die Berechnung eines Quotienten mit n Bits n Iterationen. Jede Iteration besteht aus einem Vergleich, ob der noch vorhandene Rest kleiner ist als der Divisor. In diesem Fall ist das berechnete Bit 0 und anderenfalls 1. Im letzteren Fall wird der Divisor vom noch vorhandenen Rest subtrahiert. Der Rest wird mit 2 multipliziert. Aus dieser Darstellung folgt, dass jede Iteration logarithmische Tiefe benötigt, da sie einen Vergleich und eine Subtraktion enthält. Der SRT-Dividierer variiert die Schulmethode für die Division, so dass konstante Tiefe für jede Iteration genügt. Dazu wird eine redundante Zahlendarstellung, nämlich die Radix-4 Darstellung, für den Quotienten benutzt.

Für die Beschreibung des Dividierers gehen wir von folgenden Voraussetzungen aus. Der Divisor D liegt im Intervall $[1, 2)$ und der (Divisions-)Rest R im Intervall $[-2, 2)$. Der Rest ist zu Beginn mit dem Dividenden initialisiert. Falls die zu dividierenden Zahlen nicht in diesen Bereichen liegen, ist ein Preprocessing und ein Postprocessing nötig und möglich, um sicherzustellen, dass Dividend und Divisor in den angegebenen Bereichen liegen. Der Divisor wird als Fixkommazahl dargestellt, wobei die Stelle vor dem Komma gleich 1 ist. Für den (Divisions-)Rest R benutzen wir eine redundante Zahlendarstellung, nämlich die Summe von zwei Fixkommazahlen S und C, die beide im Bereich $[-8, 8)$ liegen. Diese Zahlen bestehen aus dem Vorzeichenbit, drei Vorkommastellen und den Nachkommastellen.

Wir beschreiben die i-te Stufe des SRT-Dividierers ($i \geq 0$). Inputs sind der Divisionsrest, gegeben als $S_i + C_i$, sowie der Divisor D. Ausgabe ist die Stelle $Q_{i+1} \in \{-2, \ldots, 2\}$ des Quotienten, sowie ein neuer Rest $S_{i+1} + C_{i+1}$. Wir beachten, dass Q_{i+1} eine Stelle einer Radix-4 Zahl ist, deren einzige Vorkommastelle Q_1 und deren Nachkommastellen $Q_2 Q_3 \ldots$ sind. Hierbei genügt eine eingeschränkte Radix-4 Darstelllung mit Ziffern aus dem Bereich $\{-2, \ldots, 2\}$. Alle anderen Zahlen sind Binärzahlen (Zweierkomplement).

Die i-te Stufe arbeiten folgendermaßen (siehe auch Abbildung 14): Seien S'_i und C'_i die Zahlen, die aus den 4 Vorkommabits und den ersten 3 Nachkommabits

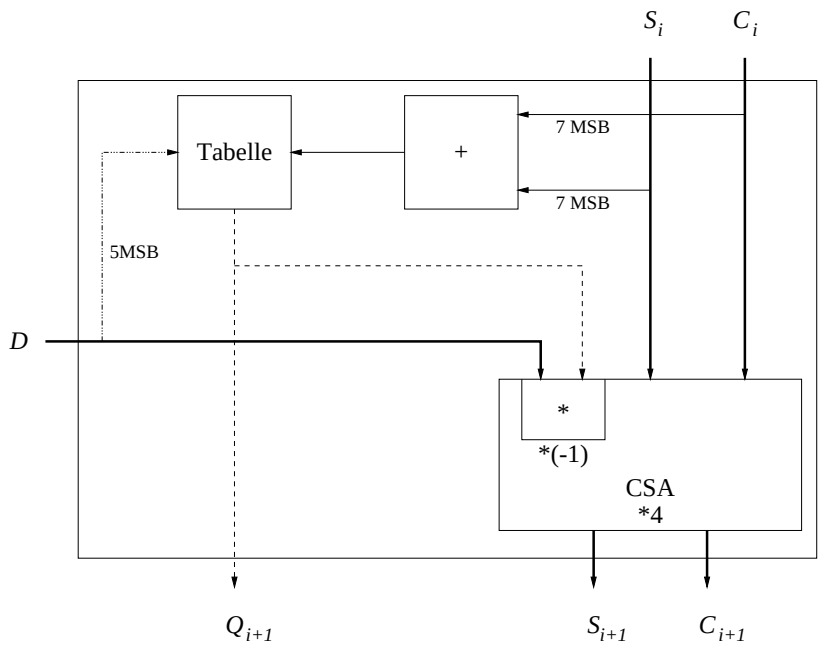


Abbildung 14: Schaltkreis für die Realisierung einer Stufe des SRT-Dividierers

von S_i und C_i bestehen. Sei D' die Zahl, die aus den 5 höchstwertigen Bits von D besteht. Die Stufe enthält eine Tabelle, die aus $S'_i + C'_i$ und aus D' eine geeignete Zahl $Q_{i+1} \in \{-2, -1, 0, 1, 2\}$ bestimmt. Von dem Divisionsrest $S_i + C_i$ wird $D \cdot Q_{i+1}$ subtrahiert und das Ergebnis wird mit 4 multipliziert, so dass der ein korrekter Divisionsrest entsteht. Der Divisionsrest wird wieder als Summe von zwei Zahlen S_{i+1} und C_{i+1} dargestellt.

Bevor wir diskutieren, warum dieser Schaltkreis korrekt implementiert werden kann (dies ist nichttrivial), überlegen wir, warum eine Stufe in konstanter Tiefe realisiert werden kann. Die Addition von S'_i und C'_i ist in konstanter Tiefe möglich, da beide Zahlen nur aus 7 Bits bestehen. Da die Tabelle konstante Größe hat, kann auch sie in konstanter Tiefe realisiert werden. Die Multiplikation von D mit 0, 1 oder 2 ist einfach. Eine eventuelle Multiplikation mit -1 kann mit Hilfe des nachgeschalteten Carry-Save-Adders durchgeführt werden, da bei der Zweierkomplementdarstellung eine Multiplikation mit -1 durch das Negieren aller Bits und Addition der Zahl mit einer 1 im niederwertigsten Bit realisiert werden kann. Schließlich ist die Addition von S_i, C_i und $-Q_{i+1} \cdot D$ mit einem Carry-Save-Adder in konstanter Tiefe möglich, da das Ergebnis als Summe von zwei Zahlen dargestellt wird.

Wir beobachten, dass die Stelle Q_{i+1} des Quotienten nur aus konstant vielen Bits von S_i, C_i und D bestimmt wird. Es ist a priori nicht klar, dass dies überhaupt möglich ist, also dass es eine geeignete Tabelle gibt, die ein korrektes Q_{i+1} bestimmt. Damit dies möglich ist, benötigen wir noch eine weitere Voraussetzung. Wir beschreiben im Folgenden auch, was es überhaupt heißt, dass Q_{i+1} korrekt

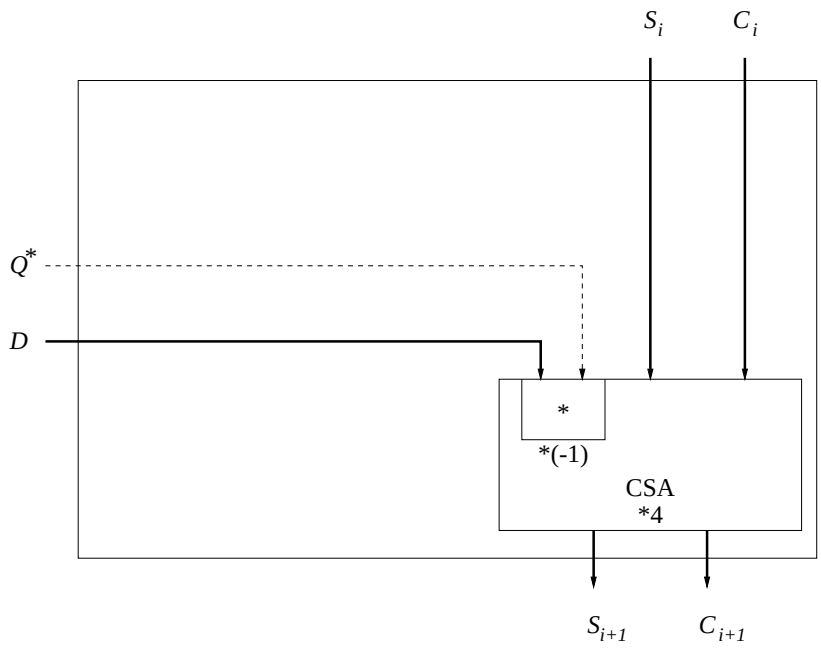


Abbildung 15: Hilfsschaltkreis zur Berechnung der zulässigen Tabelleneinträge

ist.

Die weitere Voraussetzung, die wir benötigen, besteht darin, dass

$$-8D \leq 3(S_i + C_i) \leq 8D$$

gilt. Da vor der ersten Iteration $S_0 + C_0$ gleich dem Dividenden ist und damit im Intervall $[-2, 2)$ liegt und nach Voraussetzung $D \in [1, 2)$ ist, ist diese Voraussetzung zu Beginn erfüllt. Die i-te Stufe arbeitet korrekt, wenn gilt: Falls $-8D \leq 3(S_i + C_i) \leq 8D$ gilt, dann folgt

$$\begin{array}{ll} -8D \leq 3(S_{i+1} + C_{i+1}) \leq 8D & \text{und} \\ (S_{i+1} + C_{i+1}) = 4 * (S_i + C_i - Q_{i+1}D) & \text{wobei } S_{i+1}, C_{i+1} \in [-8, 8). \end{array}$$

Die erste Bedingung stellt sicher, dass die Voraussetzung auch für die nachfolgende Stufe gilt, und die zweite Bedingung, dass korrekt dividiert wurde und kein Überlauf eintritt.

Aus den Bedingungen folgt zusammen mit $D \in [1, 2)$, dass $-16/3 \leq S_i + C_i \leq 16/3$. Es ist dann naheliegend, dass dann auch $S'_i + C'_i$ nicht viel kleiner aus $-16/3$ und nicht viel größer als $16/3$ sein kann. Daher genügt eine Tabelle mit 2^7 Zeilen (für $S'_i + C'_i$) und 2^4 Spalten (für D', hierbei beachten wir, dass das höchstwertige Bit von D' gleich 1 ist).

Zur Berechnung der korrekten Tabelleneinträge konstruieren wir zunächst den in Abbildung 15 gezeigten Teilschaltkreis, der beschreibt, wie aus dem in der Tabelle gelesenen Ergebnis Q^*, sowie aus den Eingaben S_i, C_i und D die Outputs

S_{i+1} und C_{i+1} der i-ten Stufe berechnet werden. Wir interpretieren die Outputs als boolesche Funktionen $S_{i+1}(Q^*, S_i, C_i, D)$ und $C_{i+1}(Q^*, S_i, C_i, D)$. Für diese Funktionen können OBDDs berechnet werden.

Was ist ein korrekter Tabelleneintrag an der Stelle (R^*, D^*)? Der Eintrag Q^* ist zulässig, falls für alle Zahlen S_i, C_i und D, aus $R^* = S_i' + C_i'$, aus $D^* = D'$ und aus $-8D \leq 3(S_i + C_i) \leq 8D$ folgt, dass $-8D \leq 3(S_{i+1}(Q^*, S_i, C_i, D) + C_{i+1}(Q^*, S_i, C_i, D)) \leq 8D$ und $(S_{i+1}(Q^*, S_i, C_i, D) + C_{i+1}(Q^*, S_i, C_i, D)) = 4 * (S_i + C_i - Q^* D)$, wobei $S_{i+1}(Q^*, S_i, C_i, D), C_{i+1}(Q^*, S_i, C_i, D) \in [-8, 8)$ gilt. Dies ist offensichtlich eine Bedingung, die nicht leicht von Hand getestet werden kann. Im Folgenden bezeichne [Bedingung] den Wert 1, wenn die Bedingung wahr ist, und 0 anderenfalls. Die Funktionen

$$
\begin{aligned}
f_1 &= \Big[R^* = S_i' + C_i'\Big] \\
f_2 &= \Big[D^* = D'\Big] \\
f_3 &= \Big[-8D \leq 3(S_i + C_i) \leq 8D\Big] \\
f_4 &= \Big[-8D \leq 3(S_{i+1}(Q^*, S_i, C_i, D) + C_{i+1}(Q^*, S_i, C_i, D)) \leq 8D\Big] \\
f_5 &= \Big[(S_{i+1}(Q^*, S_i, C_i, D) + C_{i+1}(Q^*, S_i, C_i, D)) = 4 * (S_i + C_i - Q^* D)\Big] \\
&\quad \wedge \Big[S_{i+1}(Q^*, S_i, C_i, D), C_{i+1}(Q^*, S_i, C_i, D) \in [-8, 8)\Big]
\end{aligned}
$$

können mit OBDDs dargestellt werden. Dann wird ein OBDD für die folgende Funktion konstruiert.

$$(\forall S_i, C_i, D) : f_1 \wedge f_2 \wedge f_3 \Rightarrow f_4 \wedge f_5$$

Der Ausdruck rechts vom Allquantor hat die folgende Interpretation: Wenn die vorliegende Eingabe S_i, C_i, D nicht korrekt ist oder für diese Eingabe nicht auf den Tabelleneintrag (R^*, D^*) zugegriffen wird, nehmen f_1, f_2 oder f_3 den Wert 0 an, und damit nimmt der Ausdruck den Wert 1 an (dieser Fall interessiert nicht). Wenn die Eingabe S_i, C_i, D korrekt ist und dafür auf den Tabelleneintrag (R^*, D^*) zugegriffen wird, nimmt der Ausdruck den Wert 1 an, wenn die i-te Stufe für den Tabelleneintrag Q^* korrekt arbeitet. Durch die Quantifizierung erhält man eine Funktion, die nur noch von R^*, D^* und Q^* abhängt und den Wert 1 annimmt, wenn für alle Eingaben S_i, C_i und D_i der Tabelleneintrag Q^* ein korrekter Eintrag für die Position (R^*, D^*) ist. Aus dem berechneten OBDD können also die korrekten Tabelleneinträge entnommen werden.

Bryant war in der Lage, alle erlaubten Tabelleneinträge zu bestimmen. Da kein Tabelleneintrag leer blieb, folgt daraus, dass die konstant vielen Bits von S_i, C_i und D ausreichen, um einen zulässigen Quotienten Q_{i+1} zu bestimmen.

Wir wollen hier nur „theoretisch“ überlegen, wie die Tabelle für die Berechnung des höchstwertigen Bits Q_1 aussieht[1]. Die größte Zahl Q, die in der Radix-4

[1]Die Tabelle befindet sich in *R. E. Bryant, „Bit-Level Analysis of an SRT Divider Circuit“, 33rd Design Automation Conference, 1996, 661–665, www.cs.cmu.edu/∼bryant/pubdir/dac96b.ps*

Darstellung mit Ziffern aus dem Bereich $\{-2, \ldots, 2\}$ mit einer Stelle vor dem Dezimalpunkt darstellbar ist, ist kleiner als

$$\sum_{-\infty<i\leq 0} 2 \cdot 4^i = 8/3.$$

Dies ist der Grund, warum $-8D \leq 3(S_0 + C_0) \leq 8D$ vorausgesetzt wird. Falls $Q_1 = 2$, ist der Quotient kleiner als $8/3$ und größer als

$$2 + \sum_{\infty<i\leq -1} (-2) \cdot 4^i = 4/3.$$

Daher ist $Q_1 = 2$ nur erlaubt, wenn $4D \leq 3(S_0 + C_0) \leq 8D$. Ähnliche Berechnungen führen zu

- $Q_1 = 1$ ist nur erlaubt, wenn $D \leq 3(S_0 + C_0) \leq 5D$ gilt.
- $Q_1 = 0$ ist nur erlaubt, wenn $-2D \leq 3(S_0 + C_0) \leq 2D$ gilt.
- $Q_1 = -1$ ist nur erlaubt, wenn $-5D \leq 3(S_0 + C_0) \leq -D$ gilt.
- $Q_1 = -2$ ist nur erlaubt, wenn $-8D \leq 3(S_0 + C_0) \leq -4D$ gilt.

Anschaulich bedeutet die Bedingung für $Q_1 = 2$, dass man in die Tabelle Geraden mit den Steigungen $4/3$ und $8/3$ einzeichnen kann, und der Wert $Q_1 = 2$ nur in den Tabelleneinträgen zwischen diesen Geraden erlaubt ist. (Hierbei gehen wir davon aus, dass auf der x-Achse D und auf der y-Achse $S_0 + C_0$ aufgetragen ist.) Analoges gilt für die anderen Werte von Q_1. Die Werte oberhalb der Geraden mit der Steigung $8/3$ und unterhalb der Geraden mit der Steigung $-8/3$ sind unbedeutend, da diese Situation nach der Voraussetzung nicht vorkommen kann.

Wir beachten allerdings, dass tatsächlich $Q_1 = 2$ nicht in allen Tabelleneinträgen zwischen den Geraden mit den Steigungen $4/3$ und $8/3$ erlaubt sein muss, da wir nur mit Näherungen für den Divisionsrest (eben den höchstwertigen 7 Bits von S_0 und C_0) arbeiten. Aus demselben Grund gibt es auch Tabelleneinträge oberhalb der Geraden mit der Steigung $8/3$, die den Wert 2 haben müssen und nicht, wie oben angedeutet, unbedeutend sind. Es wird berichtet, dass der Fehler im Pentium-Dividierer darin bestand, dass einige Tabelleneinträge oberhalb der Geraden mit der Steigung $8/3$, die den Wert 2 haben müssen, fälschlicherweise den Wert 0 hatten.

6.2 Anwendung von OBDDs bei der zweistufigen Logikminimierung

Bei den in Kapitel 3 behandelten Ansätzen zur zweistufigen Logikminimierung wurden die Funktionen und die Mengen von Implikanten und Primimplikanten *explizit* gespeichert, d.h. durch Speichern der Wertetabelle oder durch Aufzählen der Monome. Eine andere Darstellungsform ist das Speichern der charakteristischen Funktionen der jeweiligen Mengen mit OBDDs. Diese Darstellung wird häufig als *implizite* Darstellung bezeichnet.

Sei U eine Menge von Objekten (das Universum). Sei M eine Teilmenge von U. Die charakteristische Funktion $\chi_M(x) : U \to \{0,1\}$ ist definiert durch

$$\chi_M(x) = \begin{cases} 0, & \text{falls } x \notin M, \\ 1, & \text{falls } x \in M. \end{cases}$$

Häufig hat man den Spezialfall $U = \{0,1\}^n$. Dann kann χ_M als boolesche Funktion über n Variablen aufgefasst werden (siehe Kapitel 7). Im Folgenden wollen wir jedoch Mengen von Monomen über den Variablen $x_1, \ldots, x_n$ darstellen. Offensichtlich gibt es 3^n solche Monome. Eine Menge M von Monomen wird dargestellt durch eine Funktion (zur Vereinfachung der Notation auch mit M statt χ_M bezeichnet) $M : \{0,1\}^{2n} \to \{0,1\}$ mit $M(y_1, \ldots, y_n, z_1, \ldots, z_n) = 1$ genau dann, wenn (die Menge) M das Monom

$$m = \bigwedge_{i|y_i=1,z_i=1} x_i \wedge \bigwedge_{i|y_i=1,z_i=0} \bar{x}_i$$

enthält. D.h., y_i gibt an, ob x_i (oder $\bar{x}_i$) im Monom m enthalten ist oder nicht, und im Fall $y_i = 1$ gibt z_i an, ob x_i negiert oder nicht negiert in m vorkommt. Die Darstellung ist redundant, da $(y_i, z_i) = (0,0)$ und $(y_i, z_i) = (0,1)$ dieselbe Bedeutung haben. Durch die Redundanz verändert sich eventuell auch die Bedeutung der Operationen, z.B. liefert SAT-Count auf die charakteristische Funktion von M angewandt nicht unbedingt die Anzahl der Monome in M.

Wenn wir M mit OBDDs darstellen, müssen wir eine Variablenordnung wählen. Die Heuristik, dass zusammengehörende Variablen zusammen angeordnet sein sollen, impliziert offensichtlich, dass y_i und z_i zusammen getestet werden sollen. Wir wählen daher die Ordnung $y_1, z_1, \ldots, y_n, z_n$. Diese Variablenordnung wird in der Literatur als *interleaved variable ordering* bezeichnet.

Wir beschreiben nun einige Funktionen, mit denen Eigenschaften von Monomen getestet werden können. Sei also m ein Monom, das durch (y, z) beschrieben wird. Die Funktion $g_1(m, x)$ testet, ob m die Eingabe x überdeckt, d.h., ob $m(x) = 1$ ist. Man verifiziert leicht, dass

$$g_1(m, x) = \bigwedge_{i=1}^{n} [y_i = 1 \Rightarrow z_i = x_i]$$

gilt und dass für die Variablenordnung $x_1, y_1, z_1, \ldots, x_n, y_n, z_n$ OBDDs für g_1 lineare Größe haben.

Die zweite Funktion testet zwei Monome m und m' auf Gleichheit. Diese Funktion ist wegen der Redundanz der Darstellung etwas aufwändiger als im ersten Moment erwartet. Hier wie auch im Folgenden sei m durch die Variablen y, z und m' durch die Variablen y', z' beschrieben. Es ist

$$g_2(m, m') = \bigwedge_{i=1}^{n} [y_i = y_i' \wedge (y_i = 1 \Rightarrow z_i = z_i')].$$

Für die Variablenordnung $y_1, y_1', z_1, z_1', \ldots, y_n, y_n', z_n, z_n'$ ist die OBDD-Größe wieder linear.

Die dritte Funktion testet, ob m eine (nicht notwendigerweise echte) Verkürzung von m' ist. Dann ist

$$g_3(m, m') = \bigwedge_{i=1}^{n} [y_i = 1 \Rightarrow (y_i' = 1 \wedge z_i = z_i')],$$

und für die o.g. Variablenordnung ist die OBDD-Größe wieder linear.

Wir beschreiben nun zwei Ansätze für die Berechnung der Menge $\mathrm{PI}(f)$. Der erste Ansatz führt die Berechnung der Menge direkt auf Mengenoperationen bzw. Operationen auf den charakteristischen Funktionen zurück, während der zweite Ansatz eine Übertragung der Baummethode auf die implizite Darstellung ist.

Mit $\mathrm{I}(f)$ bezeichnen wir die Menge der Implikanten von f, und es ist wieder $\mathrm{I}(f)(m) = 1$ genau dann, wenn $m \in \mathrm{I}(f)$. Analog bezeichnen $\mathrm{PI}(f)$ und $\mathrm{EPI}(f)$ die Menge der Primimplikanten bzw. der Kernimplikanten (essential prime implicants) von f. Dann ist:

$$\begin{aligned}
\mathrm{I}(f)(m) &= \forall x : g_1(m, x) \Rightarrow f(x) \\
\mathrm{PI}(f)(m) &= \mathrm{I}(f)(m) \wedge \forall m' : [(\mathrm{I}(f)(m') \wedge g_3(m', m)) \Rightarrow g_2(m, m')] \\
\mathrm{EPI}(f)(m) &= \mathrm{PI}(f)(m) \\
&\quad \wedge \exists x : [f(x) \wedge \forall m' : [(\mathrm{PI}(f)(m') \wedge g_1(m', x)) \Rightarrow g_2(m, m')]]
\end{aligned}$$

Die erste Formel lautet in Worten: m ist genau dann Implikant von f, wenn für alle Inputs x gilt, dass $m(x) = 1$ impliziert, dass $f(x) = 1$ ist. (Wir beachten hierbei, dass bei der gewählten Codierung der Monome widersprüchliche Monome nicht vorkommen können, so dass hier der Test, ob das Monom überhaupt mindestens einen Input überdeckt, entfallen kann.) Also ist die Formel genau eine Umsetzung der Definition des Begriffs Implikant. Dasselbe gilt für die anderen Formeln: Das Monom m ist Primimplikant, wenn m Implikant ist und für alle Monome m' gilt, dass die Tatsache, dass m' Implikant und Verkürzung von m ist, impliziert, dass m und m' übereinstimmen. Schließlich ist m Kernimplikant,

wenn m Primiplikant ist und es eine Eingabe x mit $f(x) = 1$ gibt, so dass für alle Primimplikanten m', die x überdecken, gilt, dass $m = m'$ ist.

Es ist wichtig, diese Formeln und auch die weiter unten behandelten Formeln zu „lesen“, d.h., wie im letzten Abschnitt ihre Bedeutung in eigenen Worten zu beschreiben. Die Leserinnen und Leser sollten sich (nur) die Bedeutung der Formeln merken und üben, die Bedeutung in Formeln zurück zu übersetzen. Dies ist viel wichtiger als die Formeln zu lernen. Wir sehen, dass man z.B. die Definitionen von Implikanten und Primimplikanten direkt in Formeln übersetzen kann, die nur Operationen enthalten, für die wir Algorithmen auf OBDDs kennen.

In ähnlicher Weise können wir die Baummethode in OBDD-Operationen umsetzen. Das folgende Lemma haben wir bereits teilweise im Zusammenhang mit der Baummethode bewiesen. Es ist eine gute Übungsaufgabe, die anderen Aussagen des Lemmas zu beweisen.

Lemma: Sei $f : \{0,1\}^n \to \{0,1\}$ eine boolesche Funktion, sei $h_0 = f_{|x_1=0}$, $h_1 = f_{|x_1=1}$ und $h = f_{|x_1=0} \wedge f_{|x_1=1}$. Dann gilt:

1. $m \in \mathrm{PI}(f); x_1, \bar{x}_1 \notin m \Leftrightarrow \exists m_0, m_1 : m = m_0 m_1, m_0 \in \mathrm{PI}(h_0), m_1 \in \mathrm{PI}(h_1) \Leftrightarrow m \in \mathrm{PI}(h)$.
2. $x_1 m \in \mathrm{PI}(f); x_1 \notin m \Leftrightarrow m \in \mathrm{PI}(h_1) - \mathrm{I}(h_0) \Leftrightarrow m \in \mathrm{PI}(h_1) - \mathrm{I}(h)$.
3. $\bar{x}_1 m \in \mathrm{PI}(f); \bar{x}_1 \notin m \Leftrightarrow m \in \mathrm{PI}(h_0) - \mathrm{I}(h_1) \Leftrightarrow m \in \mathrm{PI}(h_0) - \mathrm{I}(h)$.

Wir definieren $\mathrm{PI}(h_0) \otimes \mathrm{PI}(h_1)$ als die Menge aller Monome m, die sich in $m = m_0 m_1$ mit $m_0 \in \mathrm{PI}(h_0)$ und $m_1 \in \mathrm{PI}(h_1)$ zerlegen lassen. Diese Operation erinnert an die OBDD-Synthese; man überlegt sich aber leicht, dass es keine OBDD-Synthese ist, sondern aufwändiger realisiert werden muss. Es ist:

$$\begin{aligned}\mathrm{PI}(h_0) \otimes \mathrm{PI}(h_1) = \quad \exists m' \exists m'' : \quad & g_3(m', m) \wedge \mathrm{PI}(h_0)(m') \\ & \wedge g_3(m'', m) \wedge \mathrm{PI}(h_1)(m'') \\ & \wedge \bigwedge_{i=1}^{n} [y_i = 1 \Rightarrow (y'_i = 1 \vee y''_i = 1)]\end{aligned}$$

Es wird berichtet, dass die Realisierung dieser Operation zeitaufwändig ist, so dass man versuchen sollte, diese Operation zu vermeiden. Das Lemma impliziert, dass die folgenden Darstellungen der Menge der Primimplikanten von f korrekt sind.

$$\begin{aligned}\mathrm{PI}(f)(m) &= \bar{y}_1[\mathrm{PI}(h_0)(m) \otimes \mathrm{PI}(h_1)(m)] \\ &\quad \vee y_1 \bar{z}_1 \mathrm{PI}(h_0)(m) \overline{\mathrm{I}(h_1)(m)} \\ &\quad \vee y_1 z_1 \mathrm{PI}(h_1)(m) \overline{\mathrm{I}(h_0)(m)} \\ &= \bar{y}_1 \mathrm{PI}(h)(m) \\ &\quad \vee y_1 \bar{z}_1 \mathrm{PI}(h_0)(m) \overline{\mathrm{I}(h)(m)} \\ &\quad \vee y_1 z_1 \mathrm{PI}(h_1)(m) \overline{\mathrm{I}(h)(m)}\end{aligned}$$

Diese Darstellung legt zwei verschiedene rekursive Algorithmen zur Berechnung von PI(f) nahe. Man beachte bei der Implementierung, dass es eventuell mehrere rekursive Aufrufe zur Berechnung derselben Menge gibt, so dass man alle rekursiven Aufrufe und ihre Ergebnisse speichern sollte.

7 Verifikation von sequentiellen Schaltkreisen

7.1 Flip-Flops und das Huffman-Modell für sequentielle Schaltkreise

Sequentielle Schaltkreise (manchmal auch Schaltwerke genannt) unterscheiden sich von den bisher betrachteten Schaltkreisen (den sog. kombinatorischen Schaltkreisen) dadurch, dass sie auch Rückkopplungen enthalten dürfen, d.h., der Ausgang jedes Bausteins G darf auch wieder die Eingänge von G beeinflussen. Bei dieser allgemeinen Variante von sequentiellen Schaltkreisen handelt man sich viele Probleme ein, z.B. kann es passieren, dass ein solcher Schaltkreis keine stabilen Zustände hat: Man betrachte nur einen Negationsbaustein, dessen Ausgang mit seinem Eingang verbunden ist. Daher beschränkt man sich in der Regel auf sequentielle Schaltkreise, deren Rückkopplungen in speziellen Speicherbausteinen, den Flip-Flops realisiert werden. Wir werden im Folgenden zunächst zwei einfache Flip-Flop Varianten diskutieren und dann das Huffman-Modell für sequentielle Schaltkreise. Am Ende dieses Abschnitts geben wir auch ein Beispiel für einen einfachen sequentiellen Schaltkreis an.

Schon aus dieser Beschreibung von sequentiellen Schaltkreisen folgt, dass es bei ihrer Verifikation andere und weitere Problem als bei der Verifikation von gewöhnlichen Schaltkreisen gibt. In Abschnitt 7.2 behandeln wir die Erreichbarkeitsanalyse von sequentiellen Schaltkreisen und in Abschnitt 7.3 einen allgemeineren Ansatz zur Analyse und Verifikation von sequentiellen Schaltkreisen, das sog. Model Checking mit CTL-Formeln (CTL=Computation Tree Logic).

Die linke Seite von Abbildung 16 zeigt ein sogenanntes D-Flip-Flop. Man erarbeitet sich leicht die Funktionsweise. Nehmen wir zunächst an, dass der Input T auf 1 liegt. Dann ist $a = D$ und $b = \bar{D}$. Durch Unterscheidung der Fälle $D = 0$ und $D = 1$ verifiziert man leicht, dass $Q = D$ und $\bar{Q} = \bar{D}$ gilt. Nehmen wir nun an, dass der Input T auf 0 wechselt. Offensichtlich hat der Input D nun keinen Einfluss mehr auf die Outputs. Die Rückkopplungen von den Ausgängen Q und $\bar{Q}$ sorgen nun dafür, dass sich bei dem Wechsel von $T = 1$ nach $T = 0$ die Outputs nicht ändern. Insgesamt ergibt sich, dass bei $T = 1$ der Eingang D zum Ausgang Q „durchgeschaltet“ wird. Bei einem Wechsel von $T = 1$ nach $T = 0$ wird der Zustand der Ausgänge erhalten, und solange $T = 0$ ist, hat der Eingang D keinen Einfluss mehr auf die Ausgänge. Also ist ein D-Flip-Flop ein Speicherbaustein, der den Eingang D zu dem Zeitpunkt des Wechsels von T von 1 auf 0 speichert.

Eine Zusammenschaltung von zwei D-Flip-Flops ist in Abbildung 16 rechts gezeigt. Diese Anordnung heißt auch „Master-Slave Flip-Flop“. Wir beachten, dass die T-Eingänge der beiden Flip-Flops immer verschieden sind. Dies führt dazu, dass in Abhängigkeit vom Input T immer eines der beiden Flip-Flops den Eingang „durchschaltet“, während das andere seinen jeweiligen Zustand speichert. Bei festem T ändert sich der Ausgang nicht, wenn sich der Eingang D verändert, da immer eines der beiden Flip-Flops „speichert“. Zu dem Zeitpunkt, wo T von

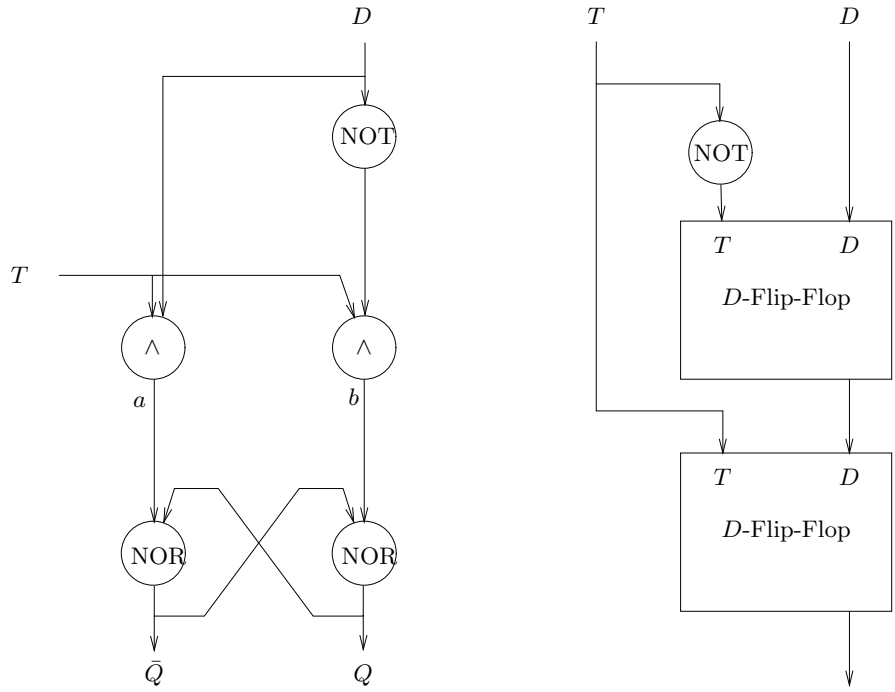


Abbildung 16: Ein D-Flip-Flop und ein Master-Slave Flip-Flop

0 auf 1 wechselt, speichert das erste Flip-Flop den Wert der Leitung D, und das zweite Flip-Flop schaltet diesen Wert zum Ausgang durch. Wenn dann T von 1 auf 0 wechselt, speichert das zweite Flip-Flop den Wert. Insgesamt erhalten wir einen Speicherbaustein, der das Speichern bei einem „0-1-Takt“ an T ausführt und ansonsten Veränderungen am Eingang D nicht weiterleitet. Dies ist die Eigenschaft, die D-Flip-Flops nicht haben und die wir im Folgenden benötigen.

Im Huffman-Modell für sequentielle Schaltkreise betrachtet man Schaltkreise, deren Rückkopplung nur über Master-Slave Flip-Flops erfolgt (siehe auch Abbildung 17). Der Schaltkreis besteht also aus zwei Teilen, nämlich aus l Flip-Flops in der Rückkopplung und einem kombinatorischem Schaltkreis G. Alle Flip-Flops werden mit demselben Takt T angesteuert. Der Inhalt der Flip-Flops wird auch als Zustand des Schaltkreises bezeichnet. Der kombinatorische Schaltkreis G berechnet aus den Ausgängen der Flip-Flops (also dem Zustand) und der Eingabe die Ausgabe und den neuen Input für die Flip-Flops, also den neuen Zustand s'. Zu dem Zeitpunkt, wo T von 0 auf 1 wechselt, wird dieser Nachfolgezustand in den Flip-Flops gespeichert, d.h., s' wird zum aktuellen Zustand. Die Wechsel von T von 0 auf 1 passieren in genügend großen Zeitabständen, so dass G jeweils den Nachfolgezustand des aktuellen Zustands berechnen kann.

Formal kann der sequentielle Schaltkreis beschrieben werden durch sein Eingabealphabet $I = \{0,1\}^n$, sein Ausgabealphabet $O = \{0,1\}^m$, seine Zustandsmenge $S = \{0,1\}^l$, seine Übergangsfunktion $\delta : I \times S \to S$, seine Ausgabefunktion $\lambda : I \times S \to O$ und seinen Startzustand. Offensichtlich hat der Schaltkreis 2^l

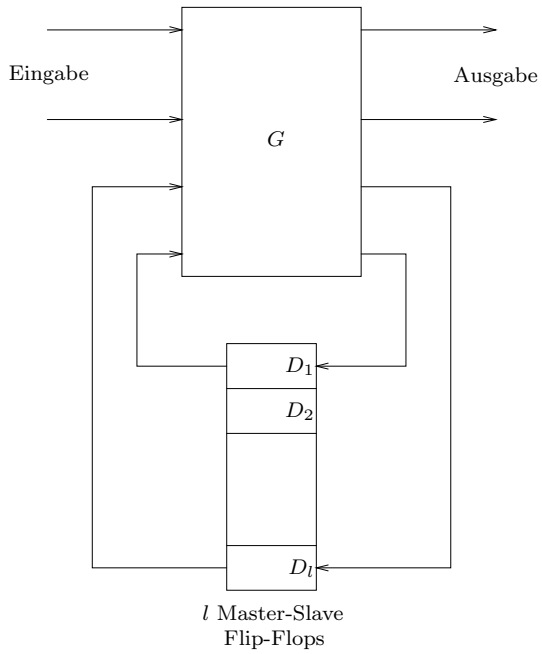


Abbildung 17:

Zustände.

Zur Illustration der sequentiellen Schaltkreise behandeln wir das einfaches Beispiel einer Ampelschaltung für eine Fußgängerampel. Die Ampel für die Straße kann Rot (SR∼Straße-rot), Gelb (SY), Grün (SG) und Rot-Gelb (SRY) anzeigen, die Fußgängerampel Rot (FR) und Grün (FG). Die Ampel steht in der Regel auf SG und FR. Es gibt einen Drucktaster für die Fußgänger, und wenn diese Taste gedrückt wird, durchläuft die Ampel die folgende Folge: 1 Zeittakt SY, FR, 1 Zeittakt SR, FR, 7 Zeittakte SR, FG, 1 Zeittakt SR, FR, 1 Zeittakt SRY, FR und geht anschließend wieder in den Zustand SG, FR.

Zum Entwurf eines sequentiellen Schaltkreises legen wir zunächst I und O fest. Wir wählen $I = \{0,1\}$ (gibt an, ob die Taste gedrückt wurde), und $O = \{0,1\}^5$ (gibt für die 5 Lampen in der Ampel an, ob sie leuchten oder nicht). Für die Zustandsmenge wählen wir $S = \{0,1\}^6$ mit der folgenden Interpretation: Das erste Flip-Flop s_1 repräsentiert FR, das zweite Flip-Flop s_2 SR, das dritte Flip-Flop s_3 SY und die übrigen drei Flip-Flops s_4, s_5, s_6 bilden einen 3-Bit-Zähler, um die Zeit zu messen, in der die Fußgängerampel auf Grün steht. Die Ausgabefunktion λ ergibt sich offensichtlich auf die folgende Weise:

$$\begin{aligned}
\lambda_{FR}(t, s_1, \ldots, s_6) &= s_1 \\
\lambda_{FG}(t, s_1, \ldots, s_6) &= \bar{s}_1 \\
\lambda_{SR}(t, s_1, \ldots, s_6) &= s_2 \\
\lambda_{SY}(t, s_1, \ldots, s_6) &= s_3 \\
\lambda_{SG}(t, s_1, \ldots, s_6) &= \bar{s}_2 \wedge \bar{s}_3
\end{aligned}$$

Von den 64 möglichen Zuständen kommen nur 12 vor. (Man könnte also auch mit 4 Flip-Flops auskommen.) Nach der obigen Beschreibung genügt es, eine Übergangsfunktion zu benutzen, für die folgendes gilt:

$$\begin{aligned}
\delta(0;1,0,0,0,0,0) &= (1,0,0,0,0,0)\\
\delta(1;1,0,0,0,0,0) &= (1,0,1,0,0,0)\\
\delta(X;1,0,1,0,0,0) &= (1,1,0,0,0,0)\\
\delta(X;1,1,0,0,0,0) &= (0,1,0,1,1,0)\\
\delta(X;0,1,0,1,1,0) &= (0,1,0,1,0,1)\\
&\vdots\\
\delta(X;0,1,0,0,0,0) &= (1,1,0,0,0,1)\\
\delta(X;1,1,0,0,0,1) &= (1,1,1,0,0,0)\\
\delta(X;1,1,1,0,0,0) &= (1,0,0,0,0,0)
\end{aligned}$$

Hierbei steht X als Abkürzung für beide Belegungen des jeweiligen Inputs. Der Startzustand ist $(1,1,0,0,0,1)$. Die Funktionsweise ist leicht zu verstehen: Sobald der Taster gedrückt wurde, durchläuft die Ampel den oben genannten Zyklus. Sobald die Fußgänger Grün haben, zählt der Zähler rückwärts von 6 bis 0. Der Zustand SR, FR kommt zweimal in dem Zyklus vor. Um diese beiden Zustände zu unterscheiden, wird der zweite dieser Zustände mit einer 1 in s_6 codiert. Es ist nun einfach, mit den aus der Vorlesung bekannten Methoden Schaltkreise (z.B. PLAs) für die Übergangs- und Ausgabefunktion zu konstruieren.

Bei diesem kleinen Beispiel ist die Verifikation der Korrektheit einfach, da es möglich ist, alle Zustände zu betrachten. Allerdings wächst die Anzahl der (theoretisch möglichen) Zustände exponentiell in der Anzahl der Flip-Flops, so dass es schon für ca. 40 bis 50 Flip-Flops unmöglich wird, alle Zustände zu betrachten. Wir diskutieren zwei Probleme, für die es hilfreich ist, alle Zustände zu betrachten.

Das erste ist der Äquivalenztest von zwei sequentiellen Schaltkreisen
$C_1 = (I, O, S_1, \delta_1, \lambda_1, s^1)$ und $C_2 = (I, O, S_2, \delta_2, \lambda_2, s^2)$. Diese Schaltkreise haben also dasselbe Eingabealphabet und dasselbe Ausgabealphabet, dürfen aber verschiedene Zustandsmengen (und verschiedene Anzahlen von Flip-Flops), verschiedene Ausgabefunktionen und verschiedene Startzustände haben. Aus GTI ist bekannt, dass man zu zwei endlichen Automaten einen Produktautomaten konstruieren kann (dies ist ähnlich zur OBDD-Synthese), und auch hier können wir einen Produktautomaten

$$C = (I, \{0,1\}, S_1 \times S_2, \delta, \lambda, (s^1, s^2))$$

konstruieren, der einen simultanen Lauf von C_1 und C_2 simuliert, d.h.

$$\delta(\sigma, (s,t)) = (\delta_1(\sigma, s), \delta_2(\sigma, t)),$$

und genau dann eine 1 ausgibt, wenn C_1 und C_2 verschiedene Ausgaben berechnen, d.h.

$$\lambda(\sigma, (s,t)) = 1 \Leftrightarrow [\lambda_1(\sigma, s) \neq \lambda_2(\sigma, t)].$$

Offensichtlich sind die Schaltkreise C_1 und C_2 genau dann äquivalent, wenn in C kein Zustand (s,t) erreichbar ist, für den es einen Input $\sigma \in I$ mit $\lambda(\sigma,(s,t)) = 1$ gibt.

Als zweite Anwendung der Erreichbarkeitsanalyse betrachten wir das Erkennen von „unsicheren“ Zuständen. Es ist leicht einzusehen, dass in dem Ampelbeispiel der Zustand $(0,0,0,0,0,0)$ ein unsicherer Zustand (beide Ampeln zeigen grün) ist, der unter allen Umständen vermieden werden muss. Wir haben verifiziert, dass dieser Zustand nicht auftritt, indem wir alle vom Startzustand erreichbaren Zustände aufgezählt haben. Wenn dies bei 2^{50} Zuständen nicht mehr in akzeptabler Zeit möglich ist, benötigen wir andere Methoden, um sicherzustellen, dass unsichere Zustände nicht erreichbar sind. Dies ist offensichtlich eine Anwendung der Erreichbarkeitsanalyse.

7.2 Die Erreichbarkeitsanalyse

Wir verwenden das folgende Programm, um die Menge der erreichbaren Zuständen zu berechnen:

$i := 0$; $S_{-1} = \emptyset$;
$S_0 = \{\text{Startzustand}\}$;
Solange $S_{i-1} \neq S_i$
    $i := i + 1$;
    $S_i := S_{i-1} \cup \bigcup_{\sigma \in I} \delta(\sigma, S_{i-1})$;
Ausgabe: Die Menge der erreichbaren Zustände ist S_i.

Es ist leicht zu sehen, dass S_i jeweils die Menge der Zustände ist, die in höchstens i Schritten vom Startzustand aus erreichbar sind (Induktion). Die Schleife realisiert eine Fixpunktberechnung, d.h., es wird solange das i erhöht, bis keine neuen Zustände mehr dazukommen.

Dieses Programm ist natürlich nicht praktisch ausführbar, wenn die betrachteten Mengen sehr groß werden und das Programm sie *explizit*, d.h. durch Listen, Bäume o.Ä. speichert. Eine in vielen Fällen kompaktere Darstellung sind OBDDs oder Erweiterungen von OBDDs. Wir untersuchen nun, wie die Mengen und die Übergangsfunktion durch OBDDs dargestellt werden können.

Die Menge aller Zustände ist eine Teilmenge von $\{0,1\}^l$. Wir können Mengen M von Zuständen durch ihre charakteristische Funktion $\chi_M : \{0,1\}^l \to \{0,1\}$ darstellen, d.h., $s \in M$ genau dann, wenn $\chi_M(s) = 1$. Zur Vereinfachung der Notation identifizieren wir im folgenden Mengen und ihre charakteristischen Funktionen, d.h., es ist genau dann $M(s) = 1$, wenn $s \in M$. Eine Mengenvereinigung entspricht dann der Synthese der charakteristischen Funktionen mit dem Operator $\vee$, der Mengendurchschnitt der Synthese mit $\wedge$, der Test, ob $s \in M$, ist die Auswertung der charakteristischen Funktion $M(s)$, und die Berechnung von $|M|$ ist die Operation Erfüllbarkeit-Anzahl.

In ähnlicher Weise können wir Übergangsfunktionen oder (allgemeiner) Übergangsrelationen durch ihre charakteristischen Funktionen darstellen. Eine Übergangsrelation δ ist eine Teilmenge von $I \times S \times S$, und $(\sigma, s, t) \in \delta$ genau dann, wenn t ein Nachfolgezustand von s bei Eingabe σ ist. Die charakteristische Funktion von δ ist dann eine Funktion $\delta : I \times S \times S \to \{0, 1\}$.

Im ersten Schritt beseitigen wir die Abhängigkeit von der Eingabe des Schaltkreises. Die Relation $\delta' \subseteq S \times S$ ist definiert durch $(s, t) \in \delta'$ genau dann, wenn es eine Eingabe $\sigma \in I$ mit $(\sigma, s, t) \in \delta$ gibt. Die Relation gibt also an, ob es überhaupt möglich ist, in einem Schritt von s nach t zu kommen. Die charakteristische Funktion von δ' kann mit der Operation Quantifizierung auf die folgende Weise berechnet werden:

$$\delta'(s, t) = (\exists\sigma_1) \cdots (\exists\sigma_n) : \delta((\sigma_1, \ldots, \sigma_n), s, s').$$

Hierbei ist $\sigma = (\sigma_1, \ldots, \sigma_n)$ die Zerlegung der Eingabe in ihre Bits. Man erhält S_i aus S_{i-1} durch die folgende Beschreibung der Menge S_i:

$$S_i = S_{i-1} \cup \{t | (\exists s \in S_{i-1}) : (s, t) \in \delta'\}.$$

Auch dies ist durch Quantifizierung realisierbar. Sei $s = (s_1, \ldots, s_l)$ und $t = (t_1, \ldots, t_l)$ die Beschreibung der Zustände durch Bitfolgen. Die folgende Berechnung von S_i aus S_{i-1} wird als Image-Computation bezeichnet:

$$S_i(t_1, \ldots, t_l) = \\ S_{i-1}(t_1, \ldots, t_l) \vee (\exists s_1) \cdots (\exists s_l) : S_{i-1}(s_1, \ldots, s_l) \wedge \delta'(s_1, \ldots, s_l, t_1, \ldots, t_l).$$

Also haben wir die Berechnungen im oben aufgeführten Programm auf die Ausführung von Operationen auf booleschen Funktionen zurückgeführt. Wenn wir die booleschen Funktionen durch OBDDs darstellen, haben wir außerdem eine Variablenordnung zu wählen. Es ist naheliegend, *interleaved variable orderings* zu benutzen, also Variablenordnungen, wo s_i und t_i zusammen getestet werden. Natürlich sind durch die längeren Folgen von Quantifizierungen exponentielle blow-ups der OBDD-Größe möglich. Daher verwendet man weitere Verfeinerungen, z.B. kompaktere OBDD-Varianten, oder man verwendet Heuristiken, die „geschickte“ Reihenfolgen für die Ausführung der Quantifizierungen bestimmen.

Wenn schließlich die Menge der erreichbaren Zustände berechnet worden ist, kann mit einer Durchschnittsberechnung getestet werden, ob ein Zustand mit der Ausgabe 1 oder ein unsicherer Zustand erreichbar ist. (Es ist natürlich auch möglich und naheliegender, in jeder Iteration des oben aufgeführten Programms zu testen, ob S_i einen solchen Zustand enthält.)

Eine andere Variante besteht darin, mit der Menge der unerwünschten Zustände (der Zustände mit Ausgabe 1 oder der unsicheren Zustände) zu beginnen und die Menge der Zustände zu berechnen, von denen aus ein solcher Zustand erreichbar ist. Am Ende (oder während jeder Iteration) kann dann getestet werden,

ob der Startzustand in dieser Menge enthalten ist. Wir benutzen also folgendes Programm:

> Sei T_0 die Menge der unsicheren Zustände; $i := 0$; $T_{-1} = \emptyset$.
> Solange $T_i \neq T_{i-1}$
> $i := i + 1$;
> $T_i = T_{i-1} \cup \{s | \exists t \in T_{i-1} : (s, t) \in \delta'\}$;
> Ausgabe: T_i ist die Menge der Zustände, von denen ein unsicherer Zustand aus erreichbar ist

In diesem Programm ist T_i die Menge der Zustände, von denen aus in höchstens i Schritten ein unsicherer Zustand erreichbar ist. Die Berechnung der Menge der möglichen Vorgänger von Zuständen in diesem Programm wird auch als Pre-Image Computation bezeichnet. Die Darstellung der Mengen und Relationen durch ihre charakteristischen Funktionen wird wie bei dem ersten Programm realisiert.

Wieviele Iterationen brauchen die Programme? Wenn der sequentielle Schaltkreis lange Zyklen enthält, was zum Beispiel bei Zählern der Fall ist, brauchen diese Programme auch viele Iterationen, bis sie terminieren. Als Ausweg wurde ein Verfahren vorgeschlagen, das man als iteriertes Quadrieren bezeichnet. Wir bezeichnen mit $P(R, S, k)$ die Funktion, die den Wert 1 annimmt, wenn ein Zustand in der Menge R von einem Zustand in der Menge S in höchstens 2^k Schritten erreichbar ist. Es ist leicht einzusehen, dass

$$P(R, S, k) = (\exists t) : P(R, \{t\}, k - 1) \wedge P(\{t\}, S, k - 1),$$

was einen rekursiven Algorithmus zur Berechnung von $P(R, S, k)$ nahelegt. Die Tiefe der Rekursion dieses Algorithmus ist nur logarithmisch in der Länge des längsten Zyklus des Schaltkreises. Es wird berichtet, dass die OBDD-Darstellungen der Funktionen $P(\cdot)$ viel größer als die OBDD-Darstellungen bei den vorher behandelten Algorithmen sind, so dass der Ansatz des iterierten Quadrierens nur bei einfachen Schaltkreisen wie Zählern erfolgreich angewendet werden kann.

Wir haben an den Darstellungen der Programme für die Erreichbarkeitsanalyse gesehen, dass OBDDs nur eine Möglichkeit für die Darstellung der betrachteten Mengen und Relationen sind, da die Programme unabhängig von der Darstellungsform der charakteristischen Funktionen sind. Weiterhin sind OBDDs nur eine Heuristik, die in manchen Fällen anwendbar ist, wo konventionelle Ansätze nicht mehr durchführbar sind. Man sollte also nicht erwarten, dass sich jedes Problem mit OBDDs lösen lässt. In vielen Fällen ist eine Lösung auch nur mit vielen Implementierungstricks und weiteren Verfeinerungen möglich. Da bei den vielen Quantifizierungen die OBDDs sehr groß werden können, ist es schwierig, theoretisch zu untersuchen, in welchen Fällen OBDDs erfolgreich sein können.

7.3 Computation Tree Logic

Die Computation Tree Logic (CTL) ist eine Sprache, mit der man Anforderungen an sequentielle Schaltkreise formulieren kann. Zu einem sequentiellen Schaltkreise kann man dann automatisch überprüfen lassen, ob er diese Anforderungen erfüllt.

Wie im letzten Abschnitt wird von den tatsächlich anliegenden Inputs abstrahiert, d.h., es wird nur betrachtet, welche Übergänge überhaupt möglich sind. Im Folgenden bezeichnet f eine boolesche Formel, die wir als charakteristische Funktion einer Zustandsmenge interpretieren. Eine CTL-Formel kann auf die folgende Weise gebildet werden:

1. Ein boolesche Formel f ist eine CTL-Formel.
2. Wenn g eine CTL-Formel ist, ist auch QTg eine CTL-Formel, wobei $Q \in \{E, A\}$ ein Quantor (Existenz- oder Allquantor) ist und $T \in \{X, G, F\}$ ein temporaler Operator ist.
3. Wenn f und g CTL-Formeln sind, ist auch $QfUg$ eine CTL-Formel, wobei $Q \in \{E, A\}$ und U der temporale Operator Until ist. Weiterhin sind auch $\neg f$, $f \wedge g$, $f \vee g$ und $f \Rightarrow g$ CTL-Formeln.

Wir wollen nun zunächst die temporalen Operatoren erklären. Jeder temporale Operator ist für eine Folge $z = (z_0, z_1, z_2, \ldots)$ von Zuständen definiert. Seien f und g boolesche Formeln, die charakteristische Funktionen von Mengen von Zuständen darstellen.

1. X ist der *Nexttime Operator*, d.h. Xf ist für den Zustand z_0 erfüllt, wenn f für z_1 erfüllt ist ($(Xf)(z_0) = 1 \Leftrightarrow f(z_1) = 1$).
2. G ist der *Global Operator*, d.h. Gf ist für z_0 erfüllt, wenn f für alle Zustände $z_0, z_1, z_2, \ldots$ erfüllt ist.
3. F ist der *Future Operator*, d.h. Ff ist für z_0 erfüllt, wenn f für einen Zustand in $z_0, z_1, z_2, \ldots$ erfüllt ist.
4. U ist der *Until Operator*, d.h. fUg ist für z_0 erfüllt, wenn es ein $i \geq 0$ gibt, so dass g für z_i erfüllt ist und f für $z_0, \ldots, z_{i-1}$ erfüllt ist.

Nun spezifiziert ein sequentieller Schaltkreis bzw. die Übergangsrelation δ' nicht nur eine, sondern viele Folgen von Zuständen. Um Formeln zu erhalten, die auf solche Übergangsrelationen angewendet werden können, wird für CTL-Formeln daher verlangt, dass jeder temporale Operator von einem davorstehenden Quantor bestimmt wird. Der Allquantor A bedeutet dann, dass alle Zustandsfolgen die Eigenschaft haben, die der folgende temporale Operator beschreibt. D.h., ATg ist die Menge aller Zustände z_0, für die für alle für den sequentiellen Schaltkreis

möglichen Zustandsfolgen $z_0, z_1, \ldots$ gilt, dass $(Tg)(z_0) = 1$ ist. Der Existenzquantor E bedeutet, dass es eine Zustandsfolge mit der Eigenschaft gibt, die durch den folgenden temporalen Operator beschrieben wird, d.h., ETg ist die Menge aller Zustände z_0, für die es eine für den sequentiellen Schaltkreis mögliche Zustandsfolge $z_0, z_1, \ldots$ mit $(Tg)(z_0) = 1$ gibt.

Nach dieser recht abstrakten Definition von CTL sollen die folgenden Beispiele illustrieren, welche Arten von Aussagen über sequentielle Schaltkreise mit CTL beschrieben werden können. Die ersten Beispiele beziehen sich auf die Ampelschaltung von oben.

1. $EF(\bar{s}_1 \wedge \bar{s}_2 \wedge \bar{s}_3)$. Diese Formel beschreibt die Menge aller Zustände, von denen aus ein Zustand mit $s_1 = s_2 = s_3 = 0$, also ein unsicherer Zustand erreichbar ist. Wenn der Startzustand dabei ist, ist es möglich, vom Startzustand zu einem unsicheren Zustand zu kommen.

2. $AG(EF\bar{s}_1)$. Die Formel beschreibt die Menge aller Zustände z mit der folgenden Eigenschaft: Von jedem von z aus erreichbaren Zustand gibt es eine Möglichkeit, einen Zustand mit $s_1 = 0$ zu erreichen, also einen Zustand, wo die Fußgängerampel Grün anzeigt. Wenn der Startzustand diese Eigenschaft nicht hat, hat die Ampelschaltung einen Deadlock, d.h., einen Zustand, so dass danach die Fußgänger nie wieder grün bekommen.

Weitere, etwas abstraktere Beispiele sind:

1. $AG(req \Rightarrow AFack)$. Für alle erreichbaren Zustände gilt: Wenn ein Request req eintritt, wird dieser Request irgendwann danach bestätigt, d.h., ein Zustand mit der Eigenschaft ack (acknowledgment) erreicht.

2. $AG(req \Rightarrow A(reqUack))$. Für alle erreichbaren Zustände gilt: Ein Request wird gespeichert, bis er bestätigt wird.

Man kann sich leicht vorstellen, dass man mit CTL-Formeln komplexe Eigenschaften von sequentiellen Schaltkreisen beschreiben kann, insbesondere wenn die CTL-Formeln mehrere Quantoren/temporale Operatoren enthalten. Es stellt sich die Frage, wie man die Menge der Zustände berechnen kann, für die eine CTL-Formel wahr ist. Um CTL-Formeln auszuwerten, beginnen wir mit einigen Vereinfachungen, d.h., mit einer Verringerung der Anzahl der verschiedenen Operatoren.

Man überlegt sich leicht, dass der temporale Operator Ff durch $trueUf$ ersetzt werden kann. Es genügt also, nur die Operatoren X, G und U zu behandeln. Weiterhin können wir Allquantoren durch Existenzquantoren ersetzen. Für das Verständnis der folgenden Formeln ist es wieder wichtig, sie zu „lesen“, also für jeden Formelteil mit eigenen Worten zu erklären, was er bedeutet.

1. $AXf = \neg EX(\neg f)$. Eine Eigenschaft f gilt für alle möglichen Nachfolgezustände, wenn es keinen Nachfolgezustand gibt, für den f nicht gilt.

2. $AGf = \neg EF(\neg f) = \neg E(trueU\neg f)$. Eine Eigenschaft f gilt für alle erreichbaren Zustände, wenn kein Zustand erreichbar ist, für den sie nicht gilt.

3. $A(fUg) = \neg EG(\neg g) \wedge \neg E(\neg gU(\neg f \wedge \neg g))$. Auf allen Zustandsfolgen gilt genau dann f bis g gilt, wenn es keine Zustandsfolge gibt, auf der g nirgendwo gilt und wenn es keine Zustandsfolge gibt, bei der zu einem Zeitpunkt gleichzeitig f und g nicht gelten, und vorher g nicht galt.

Es genügt also im Folgenden, nur Formeln mit Existenzquantoren und mit den temporalen Operatoren X, G und U zu betrachten.

Wir beginnen mit der Formel EXf, wobei wir davon ausgehen, dass wir für f bereits ein OBDD vorliegen haben. Offensichtlich gilt:

$$(EXf)(s) = (\exists t) : f(t) \wedge \delta'(s,t).$$

In dieser Formel kommen nur die bekannten Operationen vor, und daher kann ein OBDD für EXf berechnet werden.

Als zweiten Operator untersuchen wir EGf. Im Gegensatz zum eben behandelten EX-Operator haben wir das Problem, dass wir nicht nur Informationen über den direkten Nachfolgezustand, sondern auch über spätere Nachfolgezustände benötigen. Daher können wir den Operator EG nicht direkt, sondern nur über eine Fixpunktberechnung realisieren. Da die Theorie von Fixpunkten und Fixpunktberechnungen nicht Gegenstand der Vorlesung Theorie des Logikentwurfs ist, beschreiben wir nur, auf welche Weise die Fixpunktberechnung auf Operationen auf OBDDs zurückgeführt werden kann, und verweisen für weitere Einzelheiten auf die Vorlesung Theorie der Programmierung.

Zur Berechnung von $h = EGf$ nutzen wir aus, dass h der größte Fixpunkt der Gleichung

$$g = f \wedge EXg$$

ist. Es ist bekannt, dass man den größten Fixpunkt dieser Gleichung erhält, indem man mit $g = f$ startet und solange g durch $f \wedge EXg$ ersetzt, bis sich g nicht mehr verändert. Wir erhalten den folgenden Algorithmus.

$g_0 := f; g_{-1} := 0; i := 0;$
Solange $g_i \not\equiv g_{i-1}$
    $i := i + 1;$
    $g_i := f \wedge (\exists t) : \delta'(s,t) \wedge g_{i-1}(t);$
Ausgabe: g_i;

Als letztes bleibt die Operation $EfUg$. Man kann zeigen, dass $h = EfUg$ der kleinste Fixpunkt der Gleichung

$$h'(s) = g(s) \vee (f(s) \wedge (\exists t) : (\delta'(s,t) \wedge h'(t))$$

ist und dass der kleinste Fixpunkt mit analogen Methoden berechnet werden kann: Man startet mit $h := g$ und wendet die Gleichung solange iteriert an, bis man einen Fixpunkt erreicht.

Insgesamt kann man CTL-Formeln „von innen nach außen“ auswerten, d.h., zu der Formel und einem sequentiellen Schaltkreis die Mengen der Zustände berechnen, für die diese Formel gilt. Algorithmen für die benötigten Operationen stehen zur Verfügung, wenn die Mengen/charakteristischen Funktionen durch OBDDs dargestellt werden. Allerdings kann man wieder erwarten, dass die OBDDs häufig sehr groß werden.