Formale Spezifikation und Verifikation

Martin Hofmann

Wintersemester 2009-10

Was ist Spezifikation?

Beschreibung eines Teils des gewünschten Systemverhaltens.

- QuickSort berechnet sortierte Permutation
- Iterator bietet alle Elemente der Menge in einer beliebigen Reihenfolge an
- Wird equals() überschrieben, so muss auch hashcode() entsprechend angepasst werden
- Die vom Roboterarm beschriebene Trajektorie muss ineerhalb der erlaubten Toleranz mit der im Programm vorgegebenen übereinstimmen. Die Grenzgeschwindigkeiten und -beschleunigungen dürfen nicht überschritten werden
- Die Handy-Anwendung darf nicht mehr SMS verschicken als zuvor autorisiert.

Nicht: Zweck des Systems (“Ihre Software soll in der Buchhaltung 10% Personal einsparen.”)

Was ist Verifikation?

Nachweis, dass System eine gegebene Spezifikation erfüllt

- Durch Tests (unvollständig)
- Durch Argumentation
- Durch rigorosen Beweis
- Durch automatisches Verifikationswerkzeug
- Durch formalisierten Beweis

Was heißt "formal"?

Mit Mitteln der Logik ausgedrückt.

- Bedeutung einer formalen Spezifikation ist exakt festgelegt (könnte aber möglicherweise das Gewünschte nicht erfassen)
- Eine formale Verifikation liefert die Gültigkeit der zugehörigen Spezifikation mit 100%-iger Sicherheit, allerdings nur für das zugrundegelegte formale Modell des Systems.
  (Extrembeispiele: Kurzschluss, Überlastung, Compilerfehler)
- Formale Spezifikation und Verifikation kann die Systementwicklung sinnvoll unterstützen, ist aber kein Allheilmittel.

Inhalt

Kapitel I Propositionale Logik: Wdh. Syntax, Semantik. Reduktion auf SAT, SAT-Solver, Anwendung auf Modellierung. BDDs.

Kapitel II Temporallogik und Modelchecking: Temporallogik LTL, Syntax und Semantik, Büchi-Automaten, die Werkzeuge SMV und SPIN.

Kapitel III Erststufige Logik, Programmlogik: Wdh. Syntax, Semantik, Hoare-Logik, JML.

Kapitel IV Typsysteme und Programmanalyse: Operationelle Semantik, Datenflussanalyse, Java Bytecode Verifier, Typsysteme für Ressourcen.

Termine und Organisatorisches

Vorlesungstermine

26.10.,	29.10.,	09.11.,	12.11.,		
16.11.,	19.11.,	23.11.,	26.11,	30.11.,	03.12.,
07.12.,	10.12.,	14.12.,	17.12.,	11.01.,	14.01,
25.01.,	28.01.,	01.02.,	04.02.,	08.02.	

Übungen

Dr. Ulrich Schöpp und Julien Oster.
Fr. 10-12, Fr 12-14

Literatur

- Huth, Ryan: Logic in Computer Science.
- Clarke, Grunberg, Peled: Model checking.
- Nielson, Nielson, Hankin: Program analysis.
- JML Tutorials.
- Als Wdh. der Logik: Skripten von Till Tantau, Martin Lange und MH.

Das Folienskript sollte als Unterlage genügen; ersetzt aber nicht den Besuch der Vorlesung und Übungen.

Kapitel I

Propositionale Logik

Inhalt Kapitel I

Protokollchef

- Der Botschafter bittet Sie eine Einladungsliste für den Ball der Botschaft zusammenzustellen.
- Sie sollen Peru einladen oder Katar nicht einladen.
- Der Vizebotschafter möchte, dass Sie Katar, Rumänien, oder beide einladen.
- Aufgrund eines aktuellen Vorfalls können Sie nicht Rumänien und Peru zusammen einladen.

Wie stellen Sie die Einladungsliste zusammen?

Modellierung in propositionaler Logik

- Peru oder nicht Katar: $P \vee \neg K$
- Katar oder Rumänien: $K \vee R$
- Nicht Rumänien und Peru zusammen: $\neg(R \wedge P)$

Man muss die propositionalen Variablen P, K, R so mit Wahrheitswerten *true*, *false* belegen, dass alle drei Formeln wahr werden.
Ist das möglich, so ist die Formelmenge erfüllbar.
Ist hier der Fall, z.B. mit $P = \mathit{true}, K = \mathit{true}, R = \mathit{false}$.
Alternative: $P = \mathit{false}, R = \mathit{true}, K = \mathit{false}$.
Die Formelmenge $\{P \vee \neg K, K \vee R, \neg(R \wedge P)$ ist also erfüllbar.
Oben stehen zwei erfüllende Belegungen.

Syntax der Aussagenlogik

Sei eine Menge $\mathcal{A}$ von Aussagenvariablen $A, B, C, D \ldots$ gegeben. Die *aussagenlogischen Formeln* über $\mathcal{A}$ sind durch folgende BNF Grammatik definiert.

$$
\begin{array}{lcll}
\mathcal{F} & ::= & \mathcal{A} & \\
& | & \neg\mathcal{F} & \text{(Negation, Verneinung)} \\
& | & \mathcal{F} \wedge \mathcal{F} & \text{(Konjunktion, logisches Und)} \\
& | & \mathcal{F} \vee \mathcal{F} & \text{(Disjunktion, logisches Oder)} \\
& | & \mathcal{F} \Rightarrow \mathcal{F} & \text{(Implikation, Wenn-Dann-Beziehung)} \\
& | & \mathcal{F} \Leftrightarrow \mathcal{F} & \text{(Äquivalenz, Genau–Dann-Wenn-Beziehung, “iff”)} \\
& | & \top & \text{Verum, wahre Formel} \\
& | & \bot & \text{Falsum, falsche Formel}
\end{array}
$$

Die Symbole $\{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$ heißen Junktoren.

Syntaxbäume

Formal ist eine Formel also ein Syntaxbaum:

entspricht $(A \wedge B) \vee (C \Rightarrow D) \wedge (\neg A)$.
Manche Klammern darf man weglassen, da nach Konvention $\neg$ am stärksten bindet und dann $\wedge, \vee, \Rightarrow, \Leftrightarrow$.
Gleiche Junktoren werden von rechts geklammert, also
$A \Rightarrow B \Rightarrow C$ ist $A \Rightarrow (B \Rightarrow C)$.

Semantik der Aussagenlogik

- Eine Formel der Form $\phi \wedge \psi$ ist wahr, wenn ϕ und ψ beide wahr sind. Ist entweder ϕ oder ψ falsch, so ist $\phi \wedge \psi$ falsch.
- Eine Formel der Form $\phi \vee \psi$ ist wahr, wenn ϕ wahr ist oder wenn ψ wahr ist. Nur wenn ϕ und ψ beide falsch sind, ist die Formel $\phi \vee \psi$ falsch.
- $\neg\phi$ ist wahr, wenn ϕ falsch ist. Ist ϕ wahr, so ist $\neg\phi$ falsch.
- $\phi \Rightarrow \psi$ ist wahr, wenn entweder ϕ falsch ist, oder aber ϕ wahr ist und ψ dann auch wahr ist. Nur wenn ϕ wahr ist und zugleich ψ falsch ist, ist $\phi \Rightarrow \psi$ falsch.
- Eine Formel der Form $\phi \Leftrightarrow \psi$ ist wahr, wenn ϕ und ψ denselben Wahrheitsgehalt haben, also entweder beide wahr, oder beide falsch sind.
- Die Formel $\top$ ist wahr, die Formel $\bot$ ist nicht wahr; beides unabhängig vom Wahrheitsgehalt der Variablen.

Formale Semantik

Formal definiert man für jede Belegung η, die den Variablen Wahrheitswerte zuweist und Formel ϕ einen Wahrheitswert $[\![\phi]\!]\eta$.
Z.B.: $[\![\phi \wedge \psi \vee \rho]\!]\eta = ([\![\phi]\!]\eta \,\&\, [\![\psi]\!]\eta) \,||\, [\![\rho]\!]\eta$
wobei & und || durch folgende Wahrheitstafeln gegeben sind.

&	*false*	*true*
false	*false*	*false*
true	*false*	*true*

||	*false*	*true*
false	*false*	*true*
true	*true*	*true*

Tautologien

Definition

Eine Formel ϕ ist eine *Tautologie*, wenn sie unabhängig vom Wahrheitsgehalt der Variablen stets wahr ist. Formal also, wenn für alle Belegungen η ihrer Variablen gilt $[\![\phi]\!]\eta = \mathit{true}$.

Beispiele für Tautologien

- $A \vee \neg A$
- $(A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$
- $\neg\neg A \Rightarrow A$
- $A \wedge B \vee A \wedge \neg B \vee \neg A \wedge B \vee \neg A \wedge \neg B$

Erfüllbare Formeln

Definition

Eine Formel ϕ ist *erfüllbar* (satisfiable), wenn es eine Belegung ihrer Variablen gibt, die sie erfüllt (wahrmacht). Formal heißt das, dass eine Belegung η existiert, derart, dass $[\![\phi]\!]\eta = \mathit{true}$.
Eine Formel ϕ ist *unerfüllbar* (unsatisfiable), wenn sie nicht erfüllbar ist.

Beispiele erfüllbarer Formeln

- A
- $A \wedge B$
- $(A \vee B) \wedge (\neg A \vee B) \wedge (A \vee \neg B)$

Äquivalenz

Äquivalenz

Zwei Formeln ϕ und ψ sind äquivalent, wenn für alle Belegungen η gilt: $[\![\phi]\!]\eta = [\![\psi]\!]\eta$

Beispiele äquivalenter Formeln

- $\phi \vee \psi$ ist äquivalent zu $\psi \vee \phi$
- $\phi \Rightarrow \psi \Rightarrow \rho$ ist äquivalent zu $\neg(\phi \wedge \psi) \vee \rho$

Erfüllbarkeitsäquivalenz

Erfüllbarkeitsäquivalenz

Zwei Formeln ϕ und ψ sind erfüllbarkeitsäquivalent, wenn gilt: ϕ erfüllbar gdw. ψ erfüllbar.

Beispiele erfüllbarkeitsäquivalenter Formeln

- $A \vee B$ ist erfüllbarkeitsäquivalent zu $\top$
- $\phi \vee \psi$ ist erfüllbarkeitsäquivalent zu $(\neg A \vee \phi) \wedge (A \vee \psi)$

Zusammenhänge

Satz

- ϕ ist Tautologie, gdw. $\neg\phi$ unerfüllbar ist.
- ϕ ist Tautologie, gdw. ϕ äquivalent zu $\top$ ist.
- ϕ ist erfüllbar, gdw. ϕ erfüllbarkeitsäquivalent zu $\top$ ist.

Mengennotation

Abkürzung $\bigwedge_{i \in I} \phi_i = \bigwedge \{\phi_i \mid i \in I\} = \phi_{i_1} \wedge \cdots \wedge \phi_{i_n}$, wenn $I = \{i_1, \ldots, i_n\}$.

Analog $\bigvee_{i \in I} \phi_i$.

Leere Konjunktion: $\bigwedge \emptyset = \top$.

Leere Disjunktion: $\bigvee \emptyset = \bot$.

Beispiel Sudoku

Für $i, j \in \{0, \ldots, 8\}$ und $k \in \{1, \ldots, 9\}$ führen wir eine Variable x_{ijk} ein. Bedeutung von x_{ijk}: In Zeile i, Spalte j steht die Zahl k. Folgende Formeln modellieren die Sudoku Spielregeln.

- $\bigwedge_i \bigwedge_j \bigvee_k x_{ijk}$ (alle Felder sind ausgefüllt).
- $\bigwedge_i \bigwedge_j \bigwedge_{k \neq k'} \neg(x_{ijk} \wedge x_{ijk'})$ (nur eine Zahl pro Feld).
- $\bigwedge_i \bigwedge_{j \neq j'} \bigwedge_k \neg(x_{ijk} \wedge x_{ij'k})$ (nicht dieselbe Zahl zweimal in einer Zeile)
- $\bigwedge_{i \neq i'} \bigwedge_j \bigwedge_k \neg(x_{ijk} \wedge x_{ij'k})$ (nicht dieselbe Zahl zweimal in einer Spalte)
- $\bigwedge_{z \in \{0,1,2\}} \bigwedge_{w \in \{0,1,2\}} \bigwedge_k \bigwedge_{u,u' \in \{0,1,2\}} \bigwedge_{v,v' \in \{0,1,2\}}$ *if* $u \neq u' \,||\, v \neq v'$ *then* $\neg(x_{3z+u,3w+v,k} \wedge x_{3z+u',3w+v'})$ *else* $\top$ (nicht dieselbe Zahl zweimal in einer 3x3 Box)

Lösung eines Sudoku

Gibt man zu den so formulierten Spielregeln die schon besetzten Felder dazu (per $\wedge$), so entsprechen die erfüllenden Belegungen gerade den Lösungen.
Beispiel: $\bigwedge_{i \in I} x_i$ wobei

$$
\begin{aligned}
I = \{&(0,0,6),(0,4,1),(0,5,7),(0,6,5),\\
&(1,1,8),(1,2,1),(1,3,2),(1,7,7),\\
&(2,5,5),\\
&(3,1,2),(3,2,9),(3,3,4),(3,8,1),\\
&(4,1,5),(4,2,4),(4,4,2),(4,7,3),\\
&(5,2,6),(5,4,7),(5,5,8),(5,7,5),(5,8,4),\\
&(6,5,9),(6,6,3),(6,8,7),\\
&(7,2,3),(7,3,8),(7,6,4),\\
&(8,2,5),(8,7,9)\}
\end{aligned}
$$

Aus der "Zeit" vom 22.10.2009 ("schwer").

Literale und Maxterme

Literal

Ein Literal ist eine negierte oder nichtnegierte Variable. Ist ℓ ein Literal, so definiert man $\neg\ell$ durch $\neg(A) = \neg A$ und $\neg(\neg A) = A$.

Minterm, Und-Block

Eine Konjunktion (Ver-undung) von Literalen heißt Minterm oder Und-Block.

Maxterm, Klausel

Eine Disjunktion (Ver-oderung) von Literalen heißt Maxterm oder Klausel oder Oder-Block.

Konjunktive Normalform, Disjunktive Normalform

Konjunktive Normalform, KNF

Eine Konjunktion von Klauseln (Oder-Blöcken) heißt Konjunktive Normalform (KNF).

Disjunktive Normalform, DNF

Eine Disjunktion von Mintermen (Und-Blöcken) heißt Disjunktive Normalform (DNF).

Satz von der DNF

Jede aussagenlogische Formel ist äquivalent zu einer DNF.

Beweis: Man führt je einen Minterm pro *true*-Zeile in der Wahrheitstafel ein.
Analoges gilt für die KNF (hier je ein Maxterm pro *false*-Zeile).

Berechnung einer erfüllbarkeitsäquivalenten KNF

Die zu einer Formel äquivalente KNF kann sehr (exponentiell) groß sein.
Besteht man nur auf Erfüllbarkeitsäquivalenz, so geht es effizienter:

Satz

Zu jeder Formel ϕ kann in polynomieller Zeit eine erfüllungsäquivalente KNF Γ linearer Größe (bezogen auf die Größe von ϕ) berechnet werden.

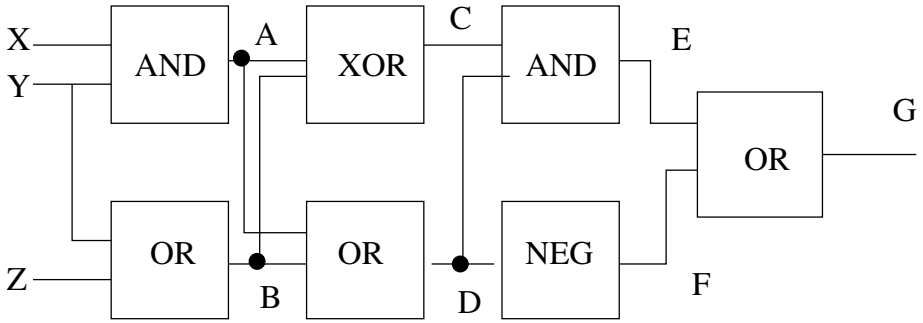
Man definiert rekursiv eine Funktion *KNF*, die zu jeder Formel ϕ eine KNF Γ und eine Variable A liefert, sodass Γ erfüllungsäquivalent mit $A \Leftrightarrow \phi$ ist.
Die gesuchte KNF ist dann $A \wedge \Gamma$.

Definition der Funktion *KNF* (Tseitin Übersetzung)

- $KNF(A) = (A, \top)$ (leere KNF)
- $KNF(\neg\phi) = let\ (B, \Gamma) = KNF(\phi)\ in$
  $(C, \Gamma \wedge (C \vee B) \wedge (\neg C \vee \neg B))$.
- $KNF(\phi_1 \vee \phi_2) = let\ (B_1, \Gamma_1) = KNF(\phi_1)\ in\ \ let\ (B_2, \Gamma_2) = KNF(\phi_2)\ in$
  $(C, \Gamma_1 \wedge \Gamma_2 \wedge (\neg C \vee B_1 \vee B_2) \wedge (\neg B_1 \vee C) \wedge (\neg B_2 \vee C))$
- Rest als Übung.

NB: Die Variable C ist jeweils frisch zu wählen.
NB: $\neg A_1 \vee \neg A_2 \vee \cdots \vee \neg A_m \vee B_1 \vee \cdots \vee B_n$ ist äquivalent zu $(\bigwedge_i A_i) \Rightarrow (\bigvee_j B_j)$.
NB: Es bietet sich an, mit dynamischer Programmierung dafür zu sorgen, dass identische Teilformeln nur einmal verarbeitet werden. Das ist insbesondere wichtig, wenn Teilformeln ge"shart" sind (Schaltkreise).

Beispiel



$$(A \vee \neg X \vee \neg Y) \wedge (\neg A \vee X) \wedge (\neg A \vee Y) \wedge$$
$$(\neg B \vee Y \vee Z) \wedge (B \vee \neg Y) \wedge (B \vee \neg Z) \wedge$$
$$(\neg C \vee A \vee B) \wedge (\neg C \vee \neg A \vee \neg B) \wedge (C \vee \neg A \vee B) \wedge (C \vee A \vee \neg B) \wedge$$
$$(\neg D \vee A \vee B) \wedge (D \vee \neg A) \wedge (D \vee \neg B) \wedge$$
$$(E \vee \neg C \vee \neg D) \wedge (\neg E \vee C) \wedge (\neg E \vee D) \wedge$$
$$(\neg F \vee \neg D) \wedge (F \vee D) \wedge$$
$$(\neg G \vee E \vee F) \wedge (G \vee \neg E) \wedge (G \vee \neg F) \wedge$$
$$G$$

Das SAT-Problem

Das SAT-Problem besteht darin, von einer gegebenen KNF zu entscheiden, ob sie erfüllbar ist.
Mit der Tseitin-Übersetzung lässt sich Erfüllbarkeit von Formeln und beliebigen Schaltkreisen auf SAT reduzieren.
Formeln (Schaltkreise) ϕ, ψ sind äquivalent, genau dann wenn $\neg(\phi \Leftrightarrow \psi)$ unerfüllbar ist.
Das SAT-Problem ist bekanntlich NP vollständig. Dennoch lässt es sich in vielen Fällen mittlerer Größe ($< 10^6$ Variablen und Klauseln) praktisch lösen.
Dazu gibt es als SAT-Solver bezeichnete Programme, z.B. zChaff, Minisat.

DIMACS-Format

Die SAT-Solver akzeptieren eine KNF in einem standardisierten Format ("DIMACS-Format").
Variablen werden durch Integer > 0 repräsentiert. Negierte Variablen durch entsprechende negative Zahlen. Klauseln werden durch 0-terminierte Zeilen, die ihre Literale enthalten, repräsentiert. Außerdem ist in einem Header die Zahl der Variablen und Klauseln anzugeben. Beispiel für $(\neg A \vee B \vee C) \wedge (A \vee \neg B \vee D \vee E)$:

```
c Kommentarzeile.
p cnf 5 2
-1 2 3 0
1 -2 4 50
```

Verwendet man einen SAT-Solver, so muss man eine Zuordnung der Variablen zu Zahlen fixieren und sich merken. Hier $A = 1, B = 2, C = 3, D = 4, E = 5$.

DPLL-Algorithmus

Alle SAT-Solver verwenden den DPLL-Algorithmus (Davis, Putnam, Logemann, Loveland). Prinzip: "Vorrechnen" und wenn das nicht hilft, verzweigen.
Formal kann DPLL als rekursive Prozedur $DPLL(\eta, \Gamma)$ beschrieben werden, die zu gegebener Belegung η und KNF Γ entweder eine erfüllende Belegung für Γ liefert, die η erweitert, oder "UNSAT" liefert, wenn es keine solche gibt.

DPLL genauer

$DPLL(\eta, \Gamma)$:

1. Vereinfache alle Klauseln gemäß η. D.h. entferne falsche Literale aus Klauseln, entferne Klauseln, die ein wahres Literal enthalten.
2. Falls Γ die leere Klausel enthält, so gib UNSAT zurück. Die leere Klausel entsteht insbesondere nach Schritt 1, wenn Γ Einerklauseln enthält, die η widersprechen.
3. Falls Γ noch Einerklauseln enthält, so füge die entsprechenden Setzungen zu η hinzu und wiederhole die Schritte 1 und 2. Das passiert z.B., wenn Γ eine Klausel $\{\neg A, B\}$ enthält und $\eta(A) = \mathit{true}$.
4. Wenn die Schritte 1,2,3 keinen Fortschritt mehr bringen, so wähle beliebige noch nicht von η belegte Variable X und rufe der Reihe nach $DPLL(\Gamma, \eta[X \mapsto \mathit{false}])$ und $DPLL(\Gamma, \eta[X \mapsto \mathit{true}])$ auf.

Die Schritte 1,2,3 heißen Unit Propagation

DPLL Algorithmus: Anmerkungen

Vorsicht: Man darf Γ nicht als globale Variable vorhalten, da sonst der Aufruf $DPLL(\Gamma, \eta[X \mapsto false])$ das Γ modifiziert ("zerschießt") und es dann nicht mehr für den Aufruf $DPLL(\Gamma, \eta[X \mapsto true])$ zur Verfügung steht.
Bei rekursiver Implementierung werden die Γs und ηs auf dem Keller abgelegt. In der Praxis implementiert man diesen Keller explizit und speichert auf diesem nur jeweils die Änderungen ab.

Optimierungen

- Anstatt die KNF Γ explizit zu verändern, merkt man sich nur für jede Klausel, welche Literale bereits gesetzt sind. Um neu gefundene Setzungen schnell zu propagieren, "beobachtet" jede Klausel die Variablen zweier ihrer unbesetzten Literale (watched literals). Wird eine von diesen gesetzt, so wird ein neues watched literal gesucht. Gibt es keines, so liegt eine Einerklausel vor ($\rightsquigarrow$ Unit Propagation).
- Bei einem Konflikt (UNSAT) wird ermittelt, welche Variablensetzungen für den Konflikt verantwortlich waren und als neue "gelernte" Klausel der KNF hinzugefügt. War z.B. $X = \mathit{true}, Y = \mathit{false}, Z = \mathit{false}$ verantwortlich, so "lernt" man $\neg X \vee Y \vee Z$.
  Man speichert hierzu bei jeder Setzung den entsprechend Grund mit ab.

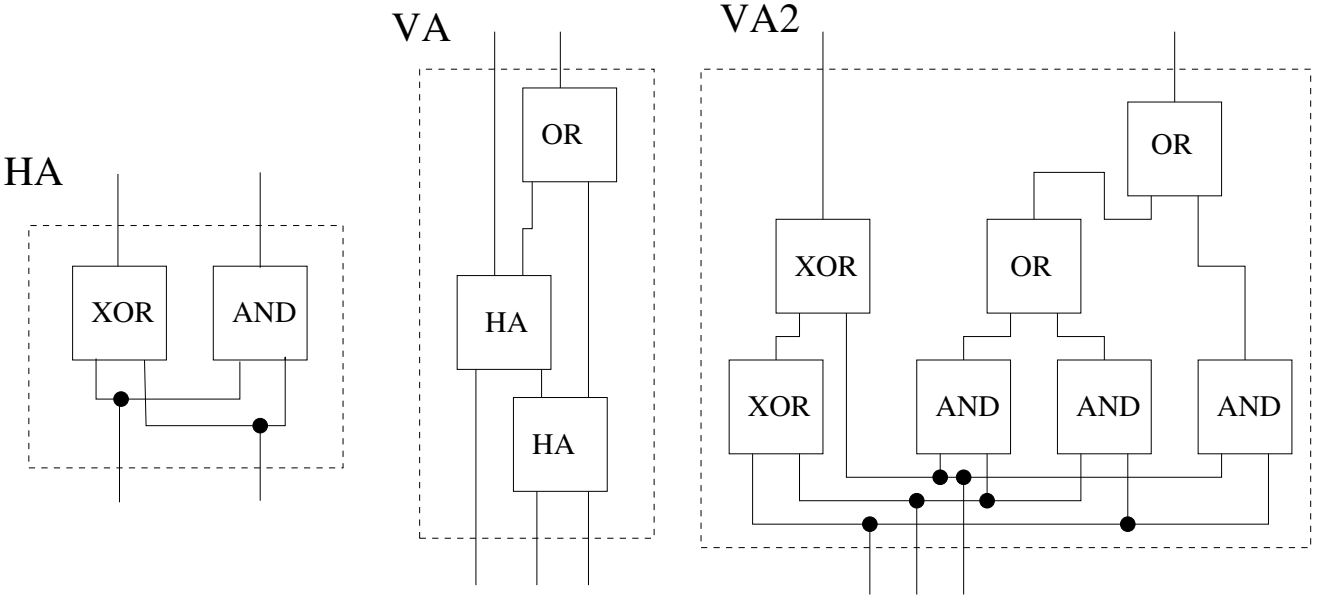
Beispiel

$$\Gamma = \bigwedge\{\bigvee\{A, B, C\}, \bigvee\{A, B, \neg C\}, \bigvee\{A, \neg B, C, X\}, \bigvee\{A, \neg B, \neg C, \neg X\}, \bigvee\{\neg A, \neg B, C, X\}\}$$

Wir beginnen mit $X = \mathit{false}$ und dann $A = \mathit{false}$ und dann $B = \mathit{false}$. Durch Unit Propagation erhalten wir aus den ersten beiden Klauseln einen Konflikt. Beteiligt waren nur die Setzungen $A = B = \mathit{false}$. Wir können daher die Klausel $\neg A \vee \neg B$ lernen. Sie hilft uns später bei $X = \mathit{true}$.

Schaltkreise

Ein Schaltkreis ist ein DAG (gerichteter azyklischer Graph) dessen Knoten mit Junktoren beschriftet sind.



Um $VA = VA2$ zu zeigen, beginnen wir mit drei Variablen X, Y, C und bilden die Formel

$$(VA(X,Y,C)_1 \Leftrightarrow VA2(X,Y,C)_1) \wedge (VA(X,Y,C)_2 \Leftrightarrow VA2(X,Y,C)_2)$$

wobei $VA(X,Y,C)_1$ den ersten Ausgang von $VA(X,Y,C)$ bezeichnet, etc.

Übersetzung in KNF

Tseitin-Transformation: KNF mit 37 Variablen, 58 Klauseln (UNSAT)
Ersetzt man das untere XOR von VA2 durch ein OR, so wird die entsprechende KNF erfüllbar. zChaff findet: $x = y = \mathit{true}, c = \mathit{false}$

Erzeugung von KNF mit OCAML

Das OCAML Modul cnf.ml auf der Homepage erlaubt die Erzeugung von KNF in DIMACS-Format ausgehend von logischen Formeln. Es bietet u.a. folgende Funktionen an:

```
val reset : unit -> unit (* resetting the variable generator *)
val newvar : unit -> int (* a fresh variable *)
val nvar : unit -> int   (* number of generated variables since

type formula (* abstract type of formulas *)
val var : int -> formula
val dtrue, dfalse : formula
val dor, dand : formula -> formula -> formula
val xor : formula -> formula -> formula
val iff : formula -> formula -> formula
val neg : formula -> formula
val dands : formula list -> formula
val dors : formula list -> formula
```

Ein- und Ausgabe

Tseitin-Transformation und Ausgabe als DIMACS Format:

```
val mk_dimacs : string -> formula list -> string -> unit
```

Aufruf (`mk_dimacs filename formulas header`)
Einlesen einer zChaff-Ausgabe:

```
val parse_zchaff : string -> int list option
```

Falls (`parse_zchaff filename`) = `None`, dann enthält `filename` die Inf UNSAT.
Falls (`parse_zchaff filename`) = `Some l`, dann ist `l` die in `filename` enthaltene erfüllende Belegung.

Gleichheit von Addierern

```
let ha x y = (xor x y,dand x y)

let va x y c =
     let (yc,d) = ha y c in
     let (s,e) = ha x yc in
     let cr = dor e d in (s,cr)

let va2 x y c =
     let s = xor (xor x y) c in
     let cr = dor (dor (dand c y) (dand y x)) (dand c x)) in
     (s,cr)
```

Gleichheit von Addierern

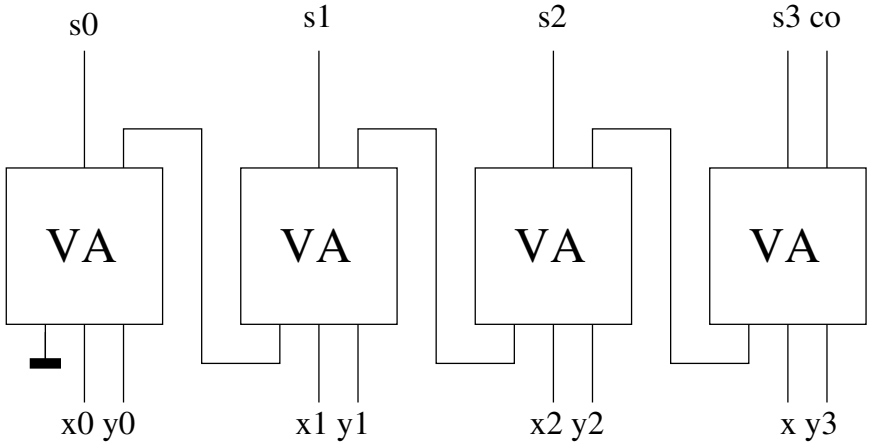
```
let test1 =
    let xx = var (newvar()) in
    let yy = var (newvar()) in
    let cc = var (newvar()) in
    let (s1,cr) = va xx yy cc in
    let (s2,cr2) = va2 xx yy cc in
    neg(dand (iff s1 s2) (iff cr cr2))

let _ = mk_dimacs "adder_test1.cnf" [test1] ""
```

Compilierung und Ausführung

```
ocamlopt -c adder.ml
ocamlopt -c cnf.ml
ocamlopt -o adder str.cmxa cnf.cmx adder.cmx
./adder
./zchaff adder_test1.cnf
```

Ripple-Addierer

Um Binärzahlen zu addieren, kann man Volladdierer hintereinanderschalten (Ripple-Adder):



Die Durchlaufzeit ist hier proportional zur Zahl n der addierten Bits (schlecht bei $n = 64$).
Grund: Übertrag muss durch alle n Volladdierer laufen ("ripple")

Carry-Lookahead Addierer

Günstiger ist es, den Übertrag c_i am i-ten Volladdierer als logische Funktion der Inputs direkt zu berechnen.

carry, gen und prop

$$
\begin{array}{lcl}
c_i & = & \mathit{if}\ i{=}0\ \mathit{then}\ \bot\ \mathit{else}\ \mathit{gen}_{0i} \\
\mathit{gen}_{ij} & = & \mathit{if}\ i{=}j\ \mathit{then}\ x_i \wedge y_i\ \mathit{else}\ \mathit{gen}_{kj} \vee (\mathit{gen}_{i(k-1)} \wedge \mathit{prop}_{kj}) \\
\mathit{prop}_{ij} & = & \mathit{if}\ i{=}j\ \mathit{then}\ x_i \vee y_i\ \mathit{else}\ \mathit{prop}_{i(k-1)} \wedge \mathit{prop}_{kj}
\end{array}
$$

wobei jeweils $k = \lfloor (j-i+1)/2 \rfloor + i$ der "Mittelpunkt" von i und j.

Es bedeutet gen_{ij}, dass am Ende des Blocks $i \ldots j$ ein Übertrag auf jeden Fall entsteht.
Und prop_{ij} bedeutet, dass der Block $i \ldots j$ einen möglicherweise einlaufenden Übertrag propagiert.
Da sich das Intervall jeweils halbiert ist die Rekursionstiefe und damit die Tiefe der entstehenden Schaltkreise $O(\log(n))$.

Implementierung in OCAML (1)

Wir verwenden das Funktional `fordo` aus `cnf.mli`:

$$fordo\ i\ j\ a\ f = f\ (j-1)\ (f\ (j-2)\ \ldots(f\ (i+1)\ (f\ ia))\ldots)$$

um zwei Assoziationslisten ("maps") für die Eingabevariablen zu erzeugen:

```
let size = 30

let xs = fordo 0 size [] (fun i l -> (i,newvar())::l)
let x i = var(List.assoc i xs)
let ys = fordo 0 size [] (fun i l -> (i,newvar())::l)
let y i = var(List.assoc i ys)
```

Jetzt ist zum Beispiel `x 5` die Eingabevariable x_5.

Implementierung in OCAML (2)

Den Ripple-Addierer implementiert man so:

```
let rec rippleadd i = if i=0 then (dfalse,[]) else
          let (c,ss) = rippleadd (i-1) in
          let (s,d) = va (x (i-1)) (y (i-1)) c in
          (d,ss@[s])
```

Es ist insbesondere `rippleadd size`$= \mathit{cout}, [s_0; \ldots; s_{\mathit{size}-1}]$.

Implementierung in OCAML (3)

Die gen- und prop-Schaltkreise:

```
let rec carry i =
     if i = 0  then dfalse else gen 0 (i-1)
and gen i j =
     if i = j
        then dand (x i) (y i)
        else let k = (j+1-i)/2 + i in
             dor (gen k j) (dand (gen i (k-1)) (prop k j))
and prop i j =
     if i = j
        then dor (x i) (y i)
        else let k = (j+1-i)/2 + i in
             dand (prop k j) (prop i (k-1))
```

Implementierung in OCAML (4)

Die Äquivalenz der Summenbits:

```
let lookaheadaddbit i =
                    va2 (x i) (y i) (carry i)
let carryout,sumbits = rippleadd size

let test2= neg
       (dand (iff carryout (snd (lookaheadaddbit (size-1))))
             (dands (fordo 0 size [] (fun i l ->
                    let (s,c) = lookaheadaddbit i in
                        iff s (List.nth sumbits i) :: l))))

let _ = mk_dimacs "adder2.cnf" [test2] ""
```

Modellierung nebenläufiger Systeme

Man kann SAT-Solver dazu verwenden, um zu zeigen, dass ein System nach n Schritten keinen verbotenen Zustand erreicht:
Z: Systemzustände; $I \subseteq S$: Startzustände;
$V \subseteq Z$: verbotene Zustände
Variablen x_{zt} für $z \in Z$ und $0 \leq t < n$.
Klauseln:

- Für alle t genau ein x_{zt} gesetzt (vgl. Sudoku)
- $\bigvee_{z \in I} x_{z0}$
- $\bigwedge_{t=0}^{n-2} \bigvee_{z \mapsto z'} x_{zt} \wedge x_{z'(t+1)}$
- $\bigvee_{t=0}^{n-1} \bigvee_{z \in V} x_{zt}$ (verbotener Zustand erreichbar)

Das $\bigvee_{z \mapsto z'}$ erstreckt sich über alle Paare (z, z') von Zuständen, sodass z' Folgezustand von z ist.
Hat der Zustand mehrere Komponenten $z = (z_1, \ldots, z_\ell)$, so kann man auch Variablen $x_{z_i t}$ einführen ("$10 + 10 + 10 + 10$ statt $10 \cdot 10 \cdot 10 \cdot 10$" bei vier Komponenten à 10 Zuständen).

Gegenseitiger Ausschluss mit Semaphor

Es gibt P Prozesse, jeder Prozess ist in einem von drei Zuständen ($\{sleep, wait, work\}$).
Dazu gibt es einen Semaphor mit zwei Zuständen ($\{free, occ\}$).
Zu jedem Zeitpunkt macht genau ein Prozess und möglicherweise der Semaphor eine Aktion. Möglich sind:

- $sleep \mapsto wait$ (Semaphor unverändert)
- $wait \mapsto work$ (nur wenn Semaphor *free*, anschl. *occ*)
- $work \mapsto sleep$ (Semaphor anschl. *free*)

```
sleep: goto wait;
wait:  if (sem==free) {sem=occ;goto work}
work: sem=free;goto sleep
```

Modellierung mit Aussagenlogik

Variablen: Für $p < P$ und $z \in \{sleep, wait, work\}$ und $t < n$ eine Variable q_{pzt}. Außerdem für $w \in \{free, occ\}$ und $t < n$ eine Variable s_{wt}.
Klauseln:

- Für $t < n$, $p < P$, genau ein q_{pzt} und genau ein s_{wt} gesetzt.
- $\bigwedge_p x_{psleep0} \wedge s_{free0}$
- $\bigvee_t \bigvee_{p_1 \neq p_2} q_{p_1 work t} \wedge q_{p_1 work t}$
- $\bigwedge_{t<n-1} \bigvee_p (\bigwedge_{p_1 \neq p} \bigwedge_z q_{p_1 zt} \Leftrightarrow q_{p_1 z(t+1)}) \wedge \bigvee_{(z,w,z',w') \in R} q_{pzt} \wedge q_{pz'(t+1)} \wedge s_{wt} \wedge s_{w'(t+1)}$

Hier ist $R = \{(z, w, z', w') \mid z{=}sleep \& z'{=}wait \& w{=}w' \;||\; z{=}wait \& z'{=}work \& w{=}free \& w'{=}occ \;||\; z{=}work \& z'{=}sleep \& w'{=}occ\}$.
Siehe `sema.ml` für OCAML-Modellierung.

Peterson Algorithmus

Semaphor erfordert atomares Abprüfen und Setzen.
Ohne solche Unterstützung funktioniert der Peterson Algorithmus.

Zwei Prozesse 0,1. Jeder hat ein Flag $flag[i]$ und es gibt eine Variable $turn : \{0, 1\}$, die angibt, wer dran ist.

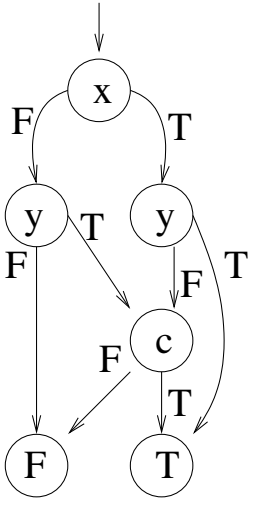
Möchte ein Prozess den kritischen Bereich betreten, so signalisiert er das durch Setzen seines Flags.

Sodann setzt er *turn* auf den jeweils anderen Prozess und wartet dann ("busy wait"), bis er an der Reihe ist oder das Flag des anderen nicht gesetzt ist.

```
p:   0 flag[p] := 1; goto 1
     1 turn := other(p); goto 2
     2 if flag[other(p)] goto 3 else goto 4
     3 if turn = p goto 4 else goto 2
     4 flag[p] := 0; goto 0 (* kritischer Bereich hier *)
```

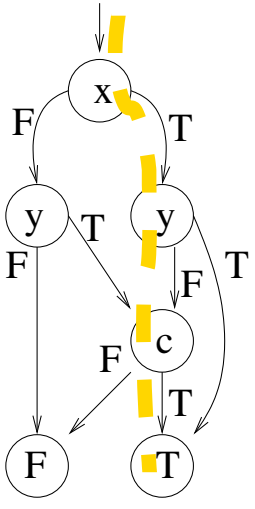
BDDs Grundidee

Man kann Boole'sche Funktionen durch Entscheidungsdiagramme (binary decision diagrams, BDDs) repräsentieren:



Ein Entscheidungsdiagramm für die Boole'sche Funktion $x \wedge y \vee x \wedge c \vee y \wedge c$, welche den Übertrag in einem Volladdierer berechnet.

Um zu gegebener Belegung den Funktionswert auszurechnen, verfolgt man im Entscheidungsdiagramm den entsprechenden Pfad. Man endet dann im Knoten, der dem Funktionswert entspricht. Z.B.: $x = z = \mathit{true}$ (im Diagramm "T") und $y = \mathit{false}$ (im Diagramm "F"):



Definition BDD

Es wird eine Ordnung auf den Variablen fixiert.

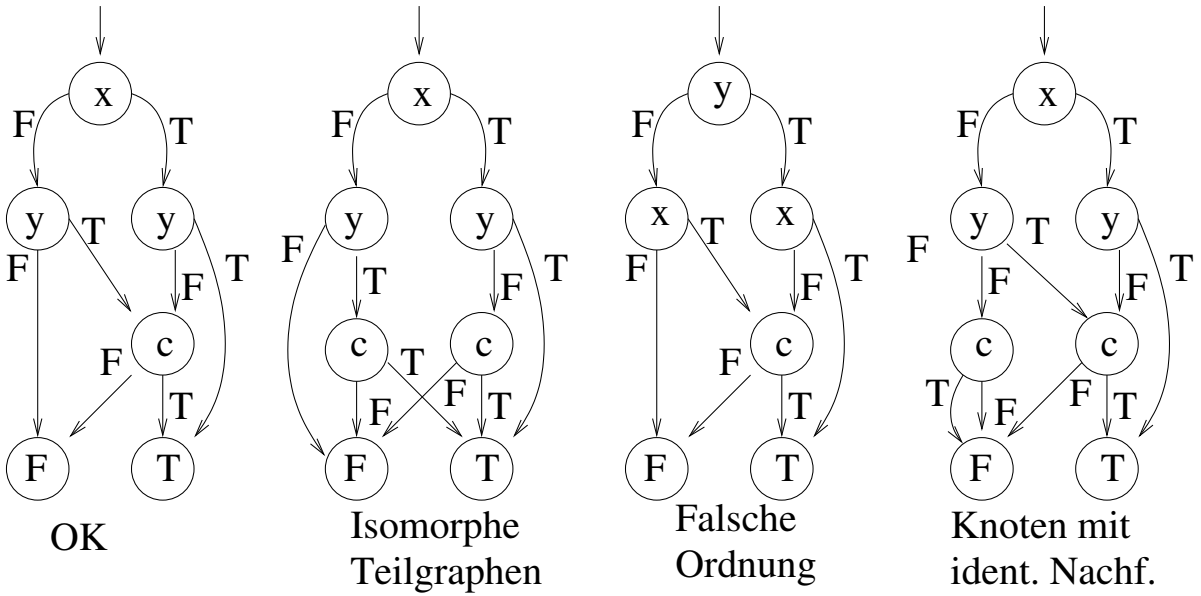
Definition BDD

Ein BDD ist ein gerichteter azyklischer Graph mit Wurzel, so dass:

- Jeder Knoten ist entweder ein Blatt oder ein innerer Knoten.
- Bltter besitzen keine ausgehenden Kanten und sind mit *true* oder *false* beschriftet.
- Jeder innere Knoten ist mit einer Variablen beschriftet und besitzt genau zwei ausgehende Kanten, die mit *false*, bzw. *true* beschriftet sind.
- Auf allen Pfaden von der Wurzel zu einem Blatt treten die Variablen gemäß der vorab fixierten Ordnung auf.
- Es gibt keine zwei verschiedene isomorphe Teilgraphen, insbesondere gibt es höchstens zwei Blätter.
- Kein Knoten hat zwei identische Nachfolger.

Beispiele

Ordnung der Variablen: $x < y < c$



NB: "Isomorph" heißt: "bis auf Herumschieben von Knoten".
Formal: bijektive Abbildung der Knoten, die mit den Kanten und den Beschriftungen verträglich ist. Im 2. Beispiel sind die Teilgraphen mit Wurzel c isomorph.

Erzeugung von BDDs

Ein *nichtreduziertes BDD* ist ein BDD, welches isomorphe verschiedene Teilgraphen und identische Nachfolger enthalten kann. Etwa das 2.Beispiel der letzten Folie.
Mit folgenden Regeln kann ein nichtreduziertes BDD in ein BDD überführt werden.

Reduktion

- Knoten mit identischen Nachfolgern entfernen.
- Knoten mit derselben Beschriftung und gleichen Nachfolgern (*true* und *false* Kanten) verschmelzen.
- Alle *true*-Blätter verschmelzen; alle *false*-Blätter verschmelzen.

Satz: Kann keine dieser Regeln mehr angewendet werden, so liegt ein BDD vor.

Bemerkung: Oft werden unsere BDDs als OBDDs (ordered BDD) oder ROBDDs (reduced OBDDs) bezeichnet.

Eindeutigkeit der BDDs

Satz

Definieren zwei BDDs über derselben Ordnung dieselbe Boole'sche Funktion, so sind sie gleich (genau: isomorph).

Beweis durch Induktion über die Größe der beiden BDDs p_1 und p_2.

Daraus ergibt sich ein Algorithmus für Äquivalenz Boole'scher Formeln: Für jede der beiden BDD aufstellen; prüfen, ob isomorph.

Shannon Zerlegung und sub-BDDs

Shannon-Zerlegung

Sei f eine Boole'sche Funktion und x eine Variable. Mit $f[x{:=}\mathit{true}]$ und $f[x{:=}\mathit{false}]$ bezeichnet man die Funktionen, die man erhält, wenn x entsprechend vorbesetzt wird.
Es gilt (Shannon-Zerlegung):

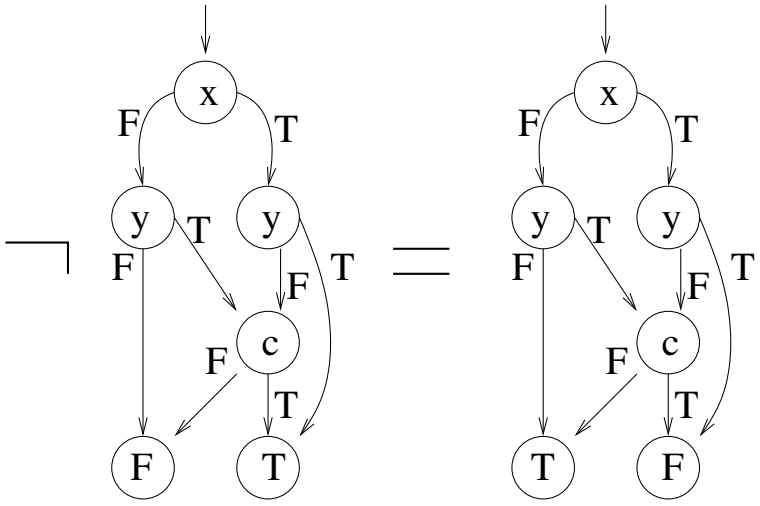
$$\begin{aligned} f &= x \wedge f[x{:=}\mathit{true}] \vee \neg x \wedge f[x{:=}\mathit{false}] \\ &= (\neg x \vee f[x{:=}\mathit{true}]) \wedge (x \vee f[x{:=}\mathit{false}]) \end{aligned}$$

Beweis durch Fallunterscheidung nach x.

Ist p ein BDD und x die erste Variable, die in p abgefragt wird, so sind die beiden Nachfolger des ersten Knotens BDDs für $f_p[x{:=}\mathit{false}]$ und $f_p[x{:=}\mathit{true}]$ (wobei f_p die durch p definierte Boole'sche Funktion ist).

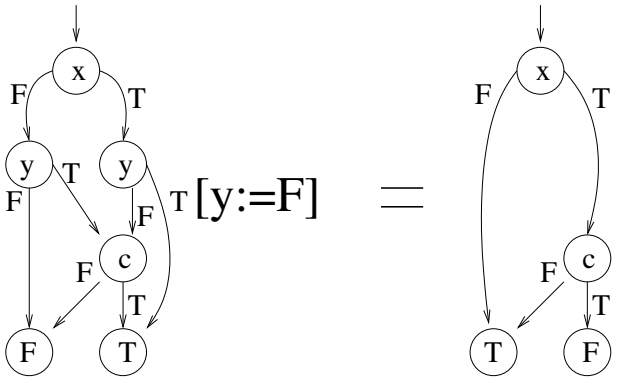
Negation und einstellige Operationen

Um ein BDD zu negieren, vertauscht man einfach die beiden Blätter. Damit ist die Anwendung einstelliger Boole'scher Funktionen auf ein BDD bereits erledigt, denn es gibt nur die Negation und die Identitätsfunktion.



Setzen einer Variablen

Um eine Variable x in einem BDD auf einen Boole'schen Wert b zu setzen, entfernt man alle Knoten für diese Variable und lenkt die einlaufende Kante zum b-Nachfolger um. Anschließend ggf. reduzieren!



Es gilt also $f_{p[x:=b]} = (f_p)[x:=b]$ (hier bezeichnet $p[x:=b]$ das BDD p nach der Einsetzoperation).

Spezialfälle:

- x kommt in p nicht vor: $p[x:=b] = p$
- x ist p's Kopfvariable: $p[x:=b]$ ist der entsprechende Nachfolger der Wurzel; insbes. ist keine Reduktion erford.

Zweistellige Boole'sche Operationen

Sei $\oplus$ eine zweistellige Boole'sche Funktion und seien p_1, p_2 BDDs, die die Boole'schen Funktionen f_1 und f_2 berechnen.

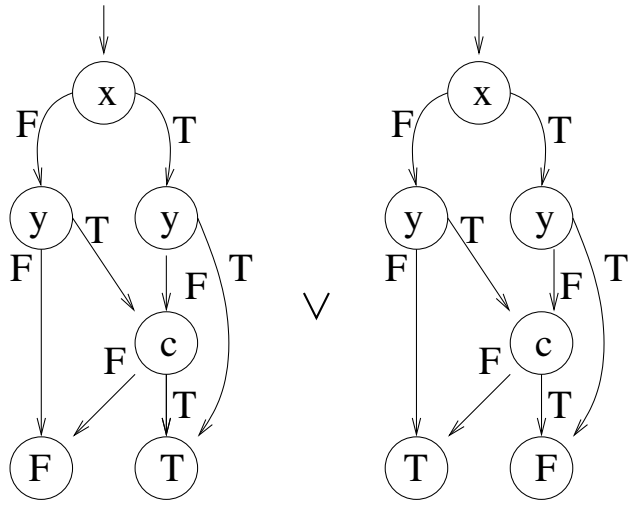
Algorithmus zur Bestimmung von $p_1 \oplus p_2$ für $f_1 \oplus f_2$

Wir berechnen mit dynamischer Programmierung für jedes Paar (q_1, q_2), wobei q_1, q_2 sub-BDDs von p_1, p_2 sind, ein (das BDD) für die Boole'sche Funktion $g_1 \oplus g_2$, wobei g_1, g_2 die von q_1, q_2 berechneten Funktionen sind.

- Ist q_1 ein Blatt b, so wende man die einstellige Funktion $b \oplus -$ auf q_2 an. Fall "q_2 ein Blatt" ist analog.
- Seien x_1 und x_2 die Kopfvariablen von q_1 und q_2 und sei o.B.d.A. $x_1 < x_2$. Dann ist entweder $x_1 = x_2$ oder x_1 kommt in q_2 gar nicht vor. Damit sind $q_1[x_1{:=}b]$ und $q_2[x_1{:=}b]$ sub-BDDs von q_1 bzw. q_2. Das ges. BDD hat Kopfvariable x_1 und b-Nachfolger $q_1[x{:=}b] \oplus q_2[x{:=}b]$ für $b = \mathit{false}, \mathit{true}$.

Beispiel

Man berechne:

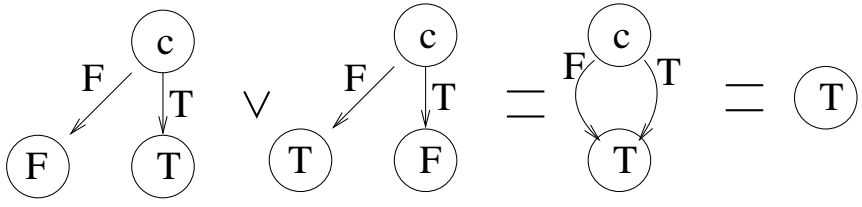


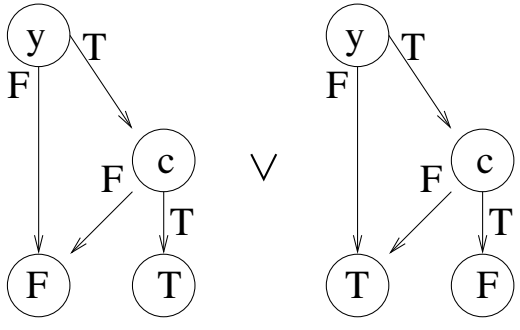
$\vee$

Nebenrechnungen:

$F \vee T = T$

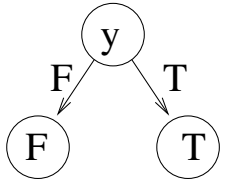
$T \vee F = T$





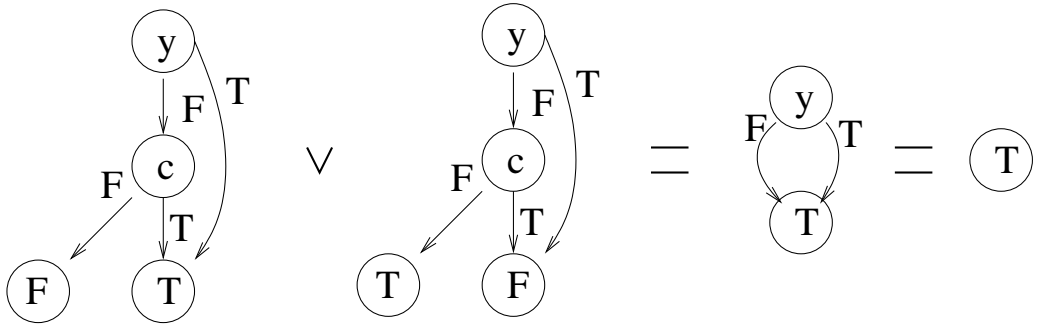
$\vee$

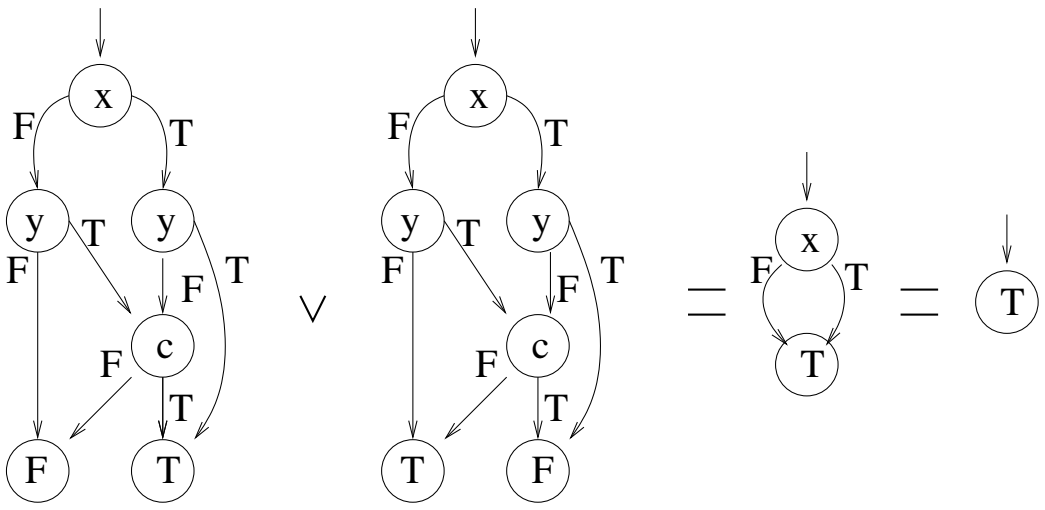
$=$



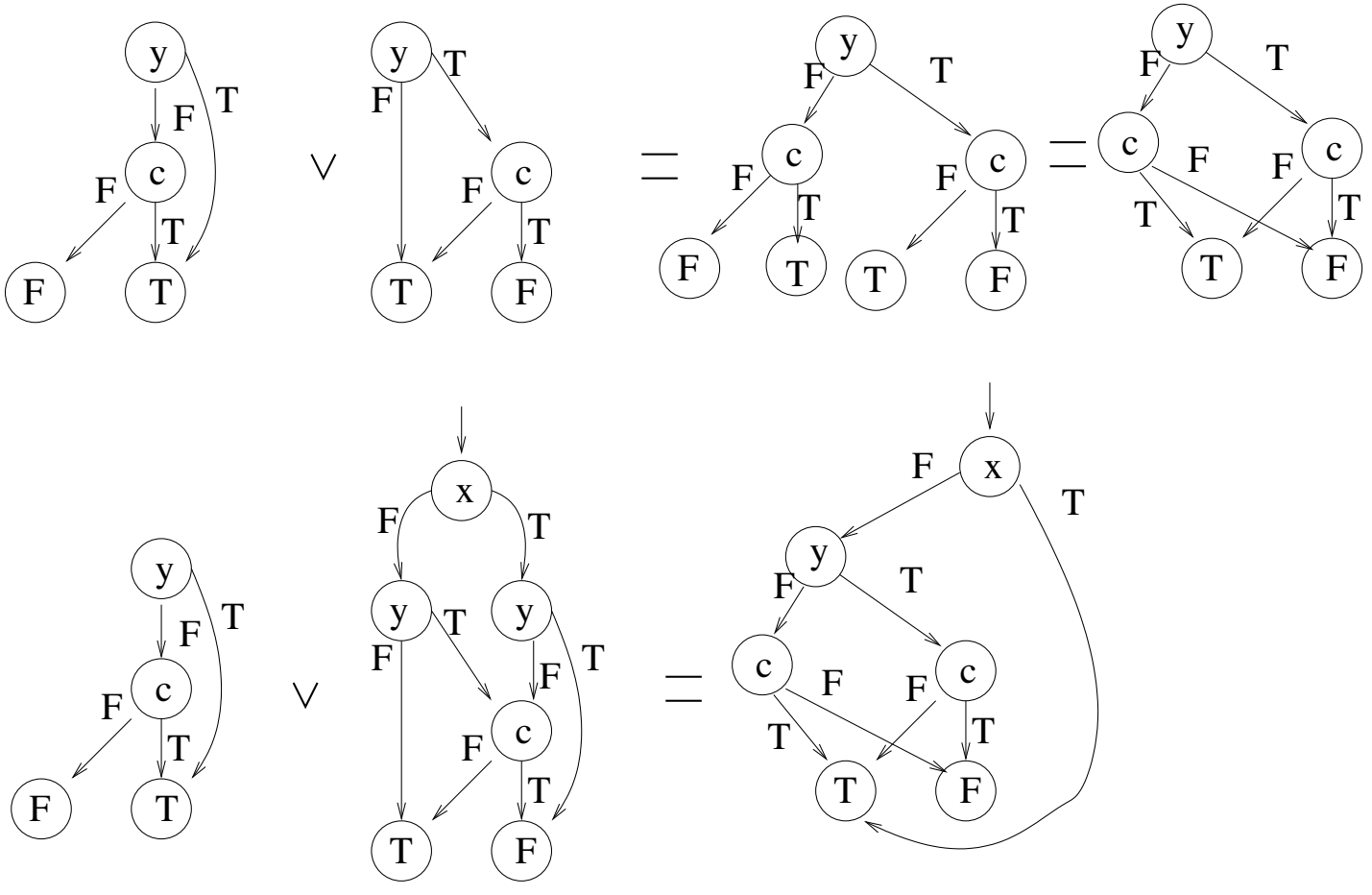
$=$ T

Beispiel Forts.





Beispiel Forts.



Diese Nebenrechnung ist nicht wirklich erforderlich, illustriert aber die Vorgehensweise zusätzlich.

Existentielle Quantifizierung

In den Anwendungen ist es häufig nützlich, eine oder mehrere Variablen existentiell zu quantifizieren.
Man führt das auf Einsetzen und Disjunktion zurück:

$$\exists x.p := p[x{:=}\mathit{false}] \vee p[x{:=}\mathit{true}]$$

Im allgemeinen wird sich die Größe hier verdoppeln und also bei der Quantifizierung von n Variablen ver-2^n-fachen. Man hofft natürlich, dass die jeweils anfallenden Reduktionen die Größe wieder verringern.

Größe der BDDS

Im allgemeinen haben BDDs eine (in der Zahl der Variablen) exponentielle Größe.
In den praktisch vorkommenden Beispielen sind die BDDs aber häufig beherrschbar klein.

Allerdings hängt die Größe oft von der gewählten Ordnung ab.

Es gibt viel Literatur über die richtige Wahl der Variablenordnung

...

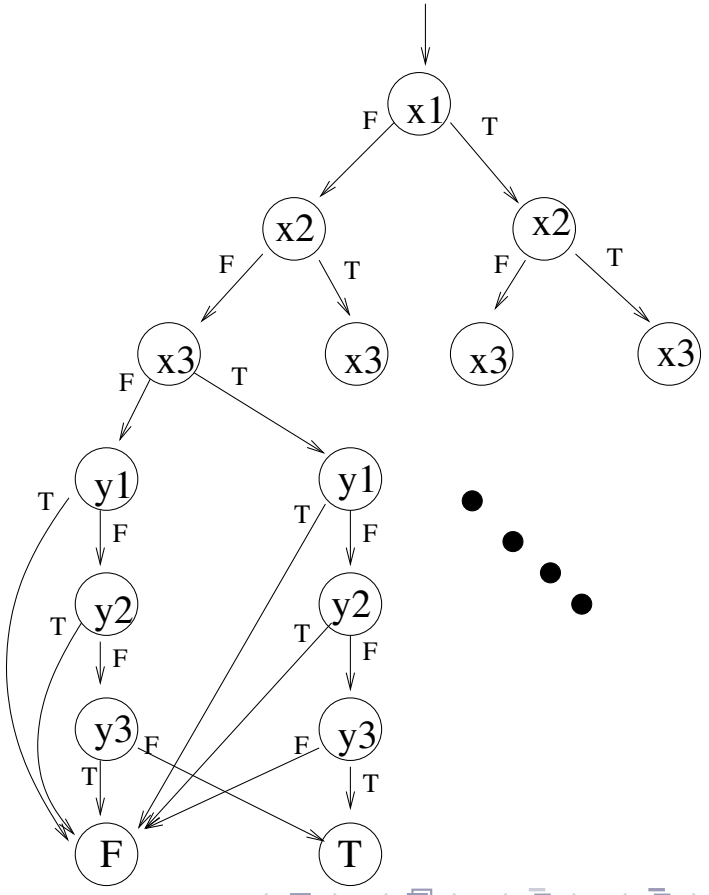
Beispiel einer Familie von BDDs exponentieller Größe

Wir betrachten die Variablenordnung

$$x_1 < x_2 < \cdots < x_n < y_1 < y_2 < \cdots < y_n$$

Das BDD für die Boole'sche Funktion $\bigwedge_{i=1}^{n} x_i \Leftrightarrow y_i$ beginnt mit einem vollständigen binären Baum der Tiefe n, in dem jede der 2^n Belegungen der x_i separat behandelt wird.

Danach sind die y_i leicht zu behandeln.

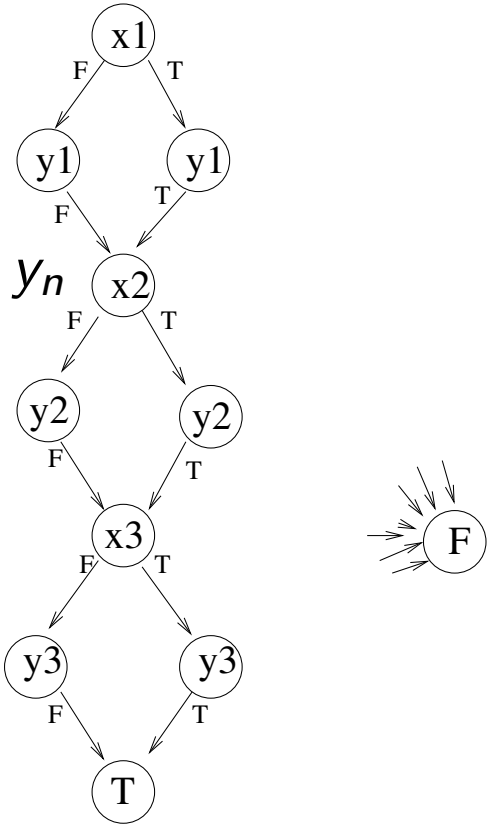


Selbe Funktion, andere Ordnung

Mit der Variablenordnung

$$x_1 < y_1 < x_2 < y_2 < x_3 < y_3 < \cdots < x_n < y_n$$

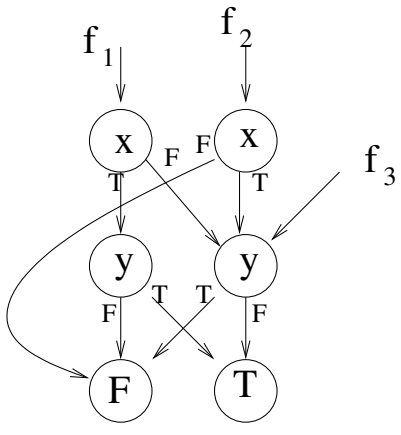
haben die BDDs lineare Größe.
Alle fehlenden Kanten führen in das *false*-Blatt.



Implementierung I

In der Praxis repräsentiert man alle während einer Sitzung auftretenden BDDs nur einmal (sharing):

Treten zum Beispiel die Funktionen $f_1 = x \Leftrightarrow y$, $f_2 = \neg y$, $f_3 = x \wedge \neg y$ auf, so entsteht nebenstehendes Diagramm:



Gleichheit von BDDs ist dann einfach Gleichheit von Zeigern.

Implementierung II

- Um festzustellen, ob ein gerade benötigter BDD schon vorhanden ist, verwendet man eine Hashtabelle.
- Den Hashwert eines BDDs berechnet man aus der Kopfvariablen und (rekursiv) aus den Hashwerten der beiden Nachfolger.
- Vgl. Behandlung von Stringliteralen in Java.

Implementierung III

- Anstatt einen BDD durch Vertauschen der Blätter zu komplementieren, kann man auch in jedem Knoten ein Komplementbit vorsehen, welches besagt, dass sich der darunterliegende BDD komplementiert versteht.
- Man kann für jeden Knoten einen Referenzzähler einführen, der die Zahl der Verweise auf den Knoten mitzählt. Sind keine vorhanden, so kann er freigegeben werden. ("garbage collection").
- Es gibt viele BDD-Bibliotheken mit Frontends für verschiedene Programmiersprachen. Wir verwenden die C-Bibliothek CUDD mit dem OCAML-Interface MLCUDDIDL. Es gibt auch Java, C++, Perl Interfaces.

Das cudd Interface mlcuddidl

- Verwendet die C-Bibliothek CUDD (F. Somenzi)
- Installation und Compilierung siehe Doku, bzw. Makefile
- Standalone Programme können auf das Modul `Cudd` zugreifen, welches die Untermodule `Man` (Sitzungsmanager) und `Bdd` (BDDs) enthält. Typischerweise öffnet man alle drei Module.
- Zum Experimentieren empfiehlt sich der OCAML-Toplevel `cuddtop`. In diesem ist `Cudd` bereits geöffnet.

Beispielsitzung mit mlcuddidl

```
# open Cudd;;
# open Bdd;;
# open Man;;
# let man = make_d();;
val man : Cudd.Man.d Cudd.Man.t = <abstr>
# let x = newvar man;;
val x : Cudd.Man.d Cudd.Bdd.t = <abstr>
# let y = newvar man;;
val y : Cudd.Man.d Cudd.Bdd.t = <abstr>
# let c = newvar man;;
val c : Cudd.Man.d Cudd.Bdd.t = <abstr>
# let carry = dor (dand x y) (dor (dand x c) (dand y c));;
val carry : Cudd.Man.d Cudd.Bdd.t = <abstr>
# is_equal carry (dor x (dor y c));;
- : bool = false
# is_equal carry (dor (dand x (dor y c)) (dand y c));;
- : bool = true
```

Lookahead Addierer mit CUDD

Der Programmtext ist ähnlich wie bei der Lösung mit SAT-Solver.
Siehe `adder.ml`.
Unterschiede:

- Variablen werden mit `newvar man` erzeugt (`man` ist der zuvor erzeugte Manager).
- `dors dands fordo` müssen selbst definiert werden.
- Einige Funktionen heißen anders: Z.B. `neg` heißt `dnot`, siehe Doku.
- Test auf Gleichheit erfolgt mit `is_equal`:

```
let rec test i = if i=0 then true else
    let s,c = lookaheadaddbit (i-1) in
    is_equal s (List.nth sumbits (i-1)) &
    (if i = size then is_equal carryout c else true)&
    test (i-1)
let _ = if test size then print_string "OK\n"
        else print_string "Error!\n"
```

Nebenläufige Systeme mit BDDs

- BDDs erlauben eine Modellierung nebenläufiger Systeme ohne Zeitschranke ("tmax").
- Wir führen zunächst nur Variablen ein, die den Zustand zu einem festen aber beliebigen Zeitpunkt beschreiben.
- Im Beispiel "Semaphor" wären dies also: Für $p < P$ und $z \in \{sleep, wait, work\}$ je eine Variable q_{pz}. Außerdem für $w \in \{free, occ\}$ je eine Variable s_w
- Jeder Zustand zu einem gewissen Zeitpunkt entspricht also einer Belegung dieser Variablen.
- Mengen von Zuständen entsprechen demnach Boole'schen Funktionen (besser: Ausdrücken) über diesen Variablen.
- Ziel ist nun, ein BDD über diesen Variablen zu finden, welches gerade die Menge der von einem (hier: dem) Anfangszustand aus erreichbaren Zustände repräsentiert.
- Man kann dann prüfen, ob diese Menge verbotene Zustände enthält.

Semaphor mit BDDs

- Sinnvolle Zustände:

$$sanity = \bigwedge_p \bigvee_z q_{pz} \wedge \bigwedge_{z \neq z'} \neg q_{pz'} \wedge (s_{free} \vee s_{occ}) \wedge \neg(s_{free} \wedge s_{occ})$$

- Anfangszustand(e):

$$initial = sanity \wedge \bigwedge_p q_{psleep} \wedge s_{free}$$

- Verbotene Zustände:

$$undesired = sanity \wedge \bigvee_{p<p'} q_{pwork} \wedge q_{p'work}$$

- Erreichbare Zustände:

$$reachable = \text{“siehe nächste Folien”}$$

- Man prüfe, ob $reachable \wedge undesired = \bot$. Wenn ja: OK; wenn nein: System fehlerhaft.

Zuständsübergangsrelation

- Führe einen zweiten Satz Variablen für Folgezustand ein: q'_{pz} und s'_w.
- Wähle Variablenordnung so, dass eine gestrichene Variable immer unmittelbar auf ihr ungestrichenes Pendant folgt, also nicht: erst alle ungestrichenen, dann alle gestrichenen.
- Definiere nun BDD *next* für die Relation "ist möglicher Folgezustand von". Im Beispiel:

$$next = \bigvee_{p} \Big(\bigwedge_{p_1 \neq p} \bigwedge_{z} q_{p_1 z} \Leftrightarrow q'_{p_1 z t} \Big) \wedge \bigvee_{(z,w,z',w') \in R} q_{pz} \wedge q'_{pz'} \wedge s_w \wedge s'_{w'}$$

wobei $R = \{(z, w, z', w') \mid z{=}sleep \& z'{=}wait \& w{=}w' \;||\; z{=}wait \& z'{=}work \& w{=}free \& w'{=}occ \;||\; z{=}work \& z'{=}sleep \& w'{=}occ\}$.

Iteration

Für $i = 0, 1, 2, 3, \ldots$ berechne nun die Menge $reach_i$ der in höchsten i Schritten erreichbaren Zustände. Es ist:

- $reach_0 = initial$,
- Falls $reach_i$ bereits definiert ist, so ist

$$reach_{i+1} = reach_i \vee sanity \wedge subst(exists(reach_i \wedge next))$$

Hier bezeichnet:

- *exists* die existentielle Quantifizierung der ungestrichenen Variablen. Die Formel (BDD) $exists(reach_i \wedge next)$ enthält also nur noch die gestrichenen Variablen;
- *subst* die Ersetzung der gestrichenen Variablen durch die ungestrichenen.

Bezeichnen wir mit $s, s', \check{s}$ Kopien der Variablensätze und machen wir die Abhängigkeiten durch Klammern explizit, so haben wir intuitiv: $reach_{i+1}(s) = reach_i(s) \vee \exists \check{s}.reach_i(\check{s}) \wedge next(\check{s}, s)$.

Erreichbare Zustände

Es ist klar, dass $reach_i \Rightarrow reach_{i+1}$ Tautologie ist.

"In höchstens i Schritten" impliziert ja "in höchstens $i+1$ Schritten. Da es nur endlich viele Zustände gibt, muss es also ein i_0 geben, sodass

$$reach_{i_0} = reach_{i_0+1}$$

Ab diesem Zeitpunkt i_0 ändert sich also nichts mehr und $reach_{i_0}$ beschreibt somit die Menge der überhaupt erreichbaren Zustände.

Wie gesagt, ist i_0 höchstens so groß wie die Zahl der Zustände. In der Praxis ist i_0 aber meist viel kleiner. Es ist ja die maximale Zeit, nach der jeder überhaupt erreichbare Zustand erreicht wird. Im Semaphorbeispiel ist z.B. $i_0 = |P| + 1$.

Algorithmus zur Berechnung der erreichbaren Zustände

Mit folgendem Algorithmus kann man also die erreichbaren Zustände als BDD berechnen.

$$
\begin{aligned}
&\mathit{new} := \bot \\
&\text{repeat} \\
&\quad \mathit{old} := \mathit{new} \\
&\quad \mathit{new} := \mathit{old} \vee \mathit{sanity} \wedge \mathit{subst}(\mathit{exists}(\mathit{old} \wedge \mathit{next})) \\
&\text{until } \mathit{new} = \mathit{old} \\
&\mathit{reach} := \mathit{new}
\end{aligned}
$$

Die Abbruchbedingung in der repeat-until Schleife kann bei BDD-Repräsentation in konstanter Zeit geprüft werden.

Siehe `bdd.tgz` für entsprechende Implementierung für Semaphor und Petersons Algorithmus.

Zusammenfassung Kapitel 1

- Wdh. Aussagenlogik
- Erfüllungsäquivalente KNF mit Tseitin Übersetzung
- DPLL Algorithmus, SAT Solver
- Äquivalenz von Schaltkreisen, Bsp. Lookahead Addierer
- Modellierung nebenläufiger Systeme durch exhaustive Simulation in Aussagenlogik
- Binäre Entscheidungsdiagramme BDDs, Definition und Grundalgorithmen
- Implementierung von BDDs
- Modellierung nebenläufiger Systeme mit BDDs, Berechnung *aller* erreichbarer Zustände durch Fixpunktiteration.