Formale Spezifikation und Verifikation

Ulrich Schöpp

Sommersemester 2016

Was ist Spezifikation?

Beschreibung eines Teils des gewünschten Systemverhaltens.

- QuickSort berechnet sortierte Permutation.
- Iterator bietet alle Elemente der Menge in einer beliebigen Reihenfolge an.
- Haben zwei Objekte den gleichen hashcode(), so sind sie gleich bzgl. equals().
- Die vom Roboterarm beschriebene Trajektorie muss innerhalb 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 informelle Argumentation
- Durch systematische Tests
- 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

- I **Aussagenlogik:** Wdh. Syntax, Semantik, Reduktion auf SAT, SAT-Solver, Anwendung auf Modellierung, BDDs.
- II **Temporallogik und Model Checking:** Temporallogik CTL, Syntax und Semantik, Model Checking Algorithmen, Temporallogik LTL und Büchi-Automaten, das Werkzeug SMV.
- III **Programmanalyse und Typsysteme:** Operationelle Semantik, Datenflussanalyse, Typsysteme.
- IV **Programmlogik:** Hoare-Logik, JML.

Organisatorisches, Literatur

Vorlesungs- und Klausurtermine, siehe Homepage der Veranstaltung.

Literatur

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

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

Kapitel I

Aussagenlogik

Inhalt Kapitel I

- Überblick über die Vorlesung
- Motivation
- Syntax und Semantik der Aussagenlogik
- Grundbegriffe
  - Mengennotation und indizierte Variablen
- Normalformen
- SAT-Solver
- Anwendungen
  - Sudoku
  - Schaltkreisverifikation
  - Verifikation nebenläufiger Systeme
- BDDs (binäre Entscheidungsdiagramme)
  - Logische Operationen auf BDDs
  - Implementierung von BDDs
  - Nebenläufige Systeme mit BDDs

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 Aussagenlogik

- 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 *aussagenlogischen Variablen* P, K, R so mit Wahrheitswerten *true*, *false* belegen, dass alle drei *Formeln* wahr werden. Ist das möglich, so ist die Formelmenge *erfüllbar*.

Die Formelmenge $\{P \vee \neg K, K \vee R, \neg(R \wedge P)\}$ ist erfüllbar.
Eine *erfüllende Belegung* ist $P = \textit{true}$, $K = \textit{true}$, $R = \textit{false}$.
Alternative: $P = \textit{false}$, $R = \textit{true}$, $K = \textit{false}$.

Syntax der Aussagenlogik

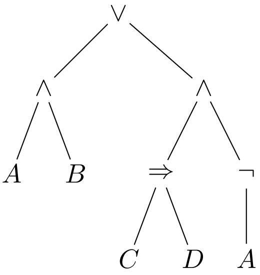
Sei eine Menge $\mathcal{A}$ von Aussagenvariablen $A, B, C, D \ldots$ gegeben. Die *aussagenlogischen Formeln* über $\mathcal{A}$ sind durch folgende BNF Grammatik definiert.

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

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

Syntaxbäume

Formal ist eine Formel ein Syntaxbaum:



Der Baum 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 $\phi \land \psi$ ist wahr, wenn ϕ und ψ beide wahr sind. Ist entweder ϕ oder ψ falsch, so ist $\phi \land \psi$ falsch.
- Eine Formel der Form $\phi \lor \psi$ ist wahr, wenn ϕ wahr ist oder wenn ψ wahr ist. Nur wenn ϕ und ψ beide falsch sind, ist die Formel $\phi \lor \psi$ 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

Eine *Belegung* ist eine Funktion, die aussagenlogischen Variablen einen Wahrheitswert zuweist.
Einer Formel ϕ wird für jede Belegung η ein Wahrheitswert $[\![\phi]\!]\eta$ zugeordnet.

$$
\begin{aligned}
[\![A]\!]\eta &= \eta(A) & [\![\neg\phi]\!]\eta &= ![\![\phi]\!]\eta \\
[\![\top]\!]\eta &= \mathit{true} & [\![\bot]\!]\eta &= \mathit{false} \\
[\![\phi \wedge \psi]\!]\eta &= [\![\phi]\!]\eta \,\&\, [\![\psi]\!]\eta & [\![\phi \vee \psi]\!]\eta &= [\![\phi]\!]\eta \,||\, [\![\psi]\!]\eta \\
[\![\phi \Rightarrow \psi]\!]\eta &= ![\![\phi]\!]\eta \,||\, [\![\psi]\!]\eta & [\![\phi \Leftrightarrow \psi]\!]\eta &= ([\![\phi]\!]\eta = [\![\psi]\!]\eta)
\end{aligned}
$$

wobei !, $\&$ und $||$ durch folgende Wahrheitstafeln gegeben sind.

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

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

!	*false*	*true*
	true	*false*

Allgemeingültigkeit

Definition

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

Beispiele für allgemeingültige Formeln

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

Erfüllbare Formeln

Erfüllbarkeit

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

Unerfüllbarkeit

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

Beweis der ersten Aussage (Rest Übung):
ϕ ist allgemeingültig
gdw. $[\![\phi]\!]\eta = \mathit{true}$ für jede Belegung η
gdw. $[\![\neg\phi]\!]\eta = \mathit{false}$ für jede Belegung η
gdw. Es gibt keine Belegung η mit $[\![\neg\phi]\!]\eta = \mathit{true}$
gdw. $\neg\phi$ ist unerfüllbar

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 alle $i, j \in \{0, \ldots, 8\}$ 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)
- $\bigwedge_{i \neq i'} \bigwedge_j \bigwedge_k \neg(x_{ijk} \wedge x_{i'jk})$ (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 $u \neq u' \mid\mid v \neq v'$ *then* $\neg(x_{3z+u,3w+v,k} \wedge x_{3z+u',3w+v',k})$ *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: $\bigwedge_{(i,j,k)\in I} x_{ijk}$ wobei

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

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 groß sein, im schlimmsten Fall exponentiell.
Besteht man nur auf Erfüllbarkeitsäquivalenz, so geht es effizienter:

Satz

Zu jeder Formel ϕ kann in polynomieller Zeit eine KNF Γ linearer Größe (bezogen auf die Größe von ϕ) berechnet werden, für die gilt:

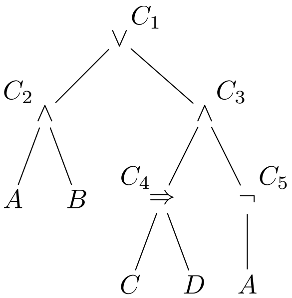
- Γ und ϕ sind erfüllbarkeitsäquivalent.
- Jede Belegung, die Γ erfüllt, erfüllt auch ϕ.

Man definiert rekursiv eine Funktion *KNF*, die zu jeder Formel ϕ eine Variable A und eine KNF Γ linearer Größe liefert, so dass jede erfüllende Belegung von Γ auch $A \Leftrightarrow \phi$ erfüllt.
Die gesuchte KNF ist dann $A \wedge \Gamma$.

Beweisidee

Idee: Führe für jeden inneren Knoten des Syntaxbaums eine neue Variable ein. Erzwinge die richtigen Werte durch eine Konjunktion von “kleinen” Formeln (mit je höchstens 3 Variablen).

Beispiel: $(A \wedge B) \vee ((C \Rightarrow D) \wedge (\neg A))$



Erfüllbarkeitsäquivalente Formel:

$$C_1 \wedge (C_1 \iff C_2 \vee C_3) \wedge (C_2 \iff A \wedge B) \wedge (C_3 \iff C_4 \wedge C_5)$$
$$\wedge (C_4 \iff C \Rightarrow D) \wedge (C_5 \iff \neg A)$$

Beweisidee

Bringe die Teilformeln $(C_1 \iff C_2 \vee C_3), \ldots$ in KNF.

Da jede Teilformel höchstens drei Variablen hat, erhält man je höchstens acht Klauseln. Die Größe ist insgesamt linear in der Größe des Syntaxbaums.

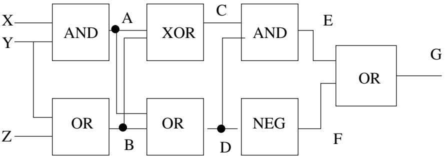
$$\begin{aligned} C_1 \; &\wedge (\neg C_1 \vee C_2 \vee C_3) \wedge (C_1 \vee \neg C_2) \wedge (C_1 \vee \neg C_3) \\ &\wedge (\neg C_2 \vee A) \wedge (\neg C_2 \vee B) \wedge (\neg C_2 \vee A \vee B) \\ &\wedge (\neg C_3 \vee C_4) \wedge (\neg C_3 \vee C_5) \wedge (\neg C_3 \vee C_5 \vee C_5) \\ &\wedge (\neg C_4 \vee \neg C \vee D) \wedge (C_4 \vee C) \wedge (C_4 \vee \neg D) \\ &\wedge (C_5 \vee A) \wedge (\neg C_5 \vee \neg A) \end{aligned}$$

Definition der Funktion *KNF* (Tseitin Übersetzung)

- *KNF*$(A) = (A, \top)$ (leere KNF)
- *KNF*$(\neg\phi) = (C, \Gamma \wedge (C \vee B) \wedge (\neg C \vee \neg B))$ falls *KNF*$(\phi) = (B, \Gamma)$, wobei C eine frische Variable ist.
- *KNF*$(\phi_1 \vee \phi_2) =$
  $(C, \Gamma_1 \wedge \Gamma_2 \wedge (\neg C \vee B_1 \vee B_2) \wedge (\neg B_1 \vee C) \wedge (\neg B_2 \vee C))$,
  falls *KNF*$(\phi_1) = (B_1, \Gamma_1)$ und *KNF*$(\phi_2) = (B_2, \Gamma_2)$, wobei C eine frische Variable ist.
- Rest als Übung.

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 Schaltkreise



$$
\begin{aligned}
&(A \vee \neg X \vee \neg Y) \wedge (\neg A \vee X) \wedge (\neg A \vee Y) \wedge \\
&(\neg B \vee Y \vee Z) \wedge (B \vee \neg Y) \wedge (B \vee \neg Z) \wedge \\
&(\neg C \vee A \vee B) \wedge (\neg C \vee \neg A \vee \neg B) \wedge (C \vee \neg A \vee B) \wedge (C \vee A \vee \neg B) \wedge \\
&(\neg D \vee A \vee B) \wedge (D \vee \neg A) \wedge (D \vee \neg B) \wedge \\
&(E \vee \neg C \vee \neg D) \wedge (\neg E \vee C) \wedge (\neg E \vee D) \wedge \\
&(\neg F \vee \neg D) \wedge (F \vee D) \wedge \\
&(\neg G \vee E \vee F) \wedge (G \vee \neg E) \wedge (G \vee \neg F) \wedge \\
&G
\end{aligned}
$$

Das SAT-Problem

Das *SAT-Problem* besteht darin, für eine gegebene KNF zu entscheiden, ob sie erfüllbar 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, Picosat, $\ldots$.

SAT-Solver geben nicht nur “erfüllbar” order “unerfüllbar” als Antwort. Bei erfüllbaren Formeln wird auch eine erfüllende Belegung als Zeuge ausgegeben.

SAT-Solver

SAT-Solver erwarten eine KNF als Eingabe.

- Mit der Tseitin-Übersetzung können wir für jede Formel ϕ eine nicht zu große KNF ausrechnen.
- Gibt der SAT-Solver eine erfüllende Belegung der KNF zurück, so macht diese auch ϕ wahr.

Das Äquivalenzproblem lässt sich ebenfalls auf SAT reduzieren: Zwei Formeln ϕ und ψ sind äquivalent, genau dann wenn $\neg(\phi \Leftrightarrow \psi)$ unerfüllbar ist.

DIMACS-Format

SAT-Solver akzeptieren eine KNF in einem einfachen 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 5 0
```

Die Zuordnung von Variablen zu Zahlen muss man sich merken.
Hier $A = 1, B = 2, C = 3, D = 4, E = 5$.

DPLL-Algorithmus

Aktuelle 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*(η, Γ) 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

DPLL$(\eta, \Gamma) =$

1. Vereinfache Γ und η mit *Unit Propagation* zu Γ' und η'.
2. Wenn η' alle Variablen in Γ' mit Werten belegt, dann gib η' als Ergebnis zurück.
3. Wähle andernfalls eine beliebige von η' noch nicht belegte Variable X aus und berechne rekursiv $DPLL(\eta'[X \mapsto \mathit{false}], \Gamma')$. Ist das Ergebnis eine Belegung, so wird sie zurückgegeben.
   Ist das Ergebnis UNSAT, dann probiere noch $DPLL(\eta'[X \mapsto \mathit{true}], \Gamma')$.
   Waren die rekursiven Aufrufe beide nicht erfolgreich, so wird UNSAT zurückgegeben.

N.B. $\eta[X \mapsto \mathit{true}]$ bezeichnet die Belegung, die X auf *true* setzt und alle anderen Variablen wie η belegt.

Unit Propagation

Funktionsweise der Unit Propagation:

1a. Vereinfache alle Klauseln in Γ gemäß η:
   - Entferne aus den Klauseln alle Literale, die durch η falsch gemacht werden.
   - Entferne alle Klauseln, die ein wahres Literal enthalten.

1b. Falls Γ die leere Klausel enthält, dann gib UNSAT zurück.

1c. Falls Γ die Einerklausel A enthält, so setze $\eta(A) =$ *true* und beginne wieder bei Schritt 1a.

1d. Falls Γ die Einerklausel $\neg A$ enthält, so setze $\eta(A) =$ *false* und beginne wieder bei Schritt 1a.

Beispiele für Unit Propagation

- $\Gamma = (A) \wedge (B \vee \neg C)$ und $\eta(A) =$ *false*
  In $1a$ wird A aus der ersten Klausel entfernt. Es entsteht die leere Klausel, also wird in $1b$. UNSAT zurückgegeben.
- $\Gamma = (\neg A) \wedge (B \vee \neg C)$ und $\eta(A) =$ *false*
  In $1a$ wird die ganze erste Klausel entfernt. Das Ergebnis der Unit Propagation ist $\Gamma' = (B \vee \neg C)$, $\eta' = \eta$.
- $\Gamma = (A \vee B) \wedge \ldots$ und $\eta(A) =$ *false*.
  In Schritt $1a$ wird $(A \vee B)$ zu (B).
  In Schritt $1c$ wirt $\eta(B) =$ *true* gesetzt.
  Die Unit Propagation wird dann neu begonnen.

Beispiel für DPLL

Ausführung von DPLL mit $\eta = [X \mapsto \textit{false}, A \mapsto \textit{false}]$ und

$$\begin{aligned}\Gamma = \quad & (A \vee B \vee C) \wedge (A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C \vee X) \wedge \\ & (A \vee \neg B \vee \neg C \vee \neg X) \wedge (\neg A \vee \neg B \vee C \vee X)\end{aligned}$$

- Unit Propagation liefert $\eta' = \eta$ und

$$\Gamma' = \quad (B \vee C) \wedge (B \vee \neg C) \wedge (\neg B \vee C) \ .$$

- Der Aufruf $\mathit{DPLL}(\eta[B \mapsto \textit{false}], \Gamma')$ gibt UNSAT zurück.
- Der Aufruf $\mathit{DPLL}(\eta[B \mapsto \textit{true}], \Gamma')$ liefert eine Belegung η'' mit $\eta''(X) = \textit{false}$ und $\eta''(A) = \textit{false}$ und $\eta''(B) = \textit{true}$ und $\eta''(C) = \textit{true}$, die als erfüllende Belegung zurückgegeben wird.

DPLL Algorithmus: Anmerkungen

- Vorsicht: Bei der Implementierung kann man Γ nicht als globale Variable speichern, da sonst der Aufruf *DPLL*$(\Gamma, \eta[X \mapsto$ *false*$])$ das Γ modifiziert 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 Stack abgelegt. In der Praxis implementiert man diesen Stack explizit und speichert auf diesem nur jeweils die Änderungen ab.

Optimierungen

- Anstatt explizit eine neue KNF Γ' zu erzeugen, merkt man sich nur für jede Klausel in Γ, welche Literale bereits gesetzt sind. Um Einerklauseln für neue Setzungen schnell zu finden, "beobachtet" jede Klausel die Variablen zweier ihrer unbesetzten Literale (*watched literals*). Wird eines 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 entsprechenden Grund mit ab.

Beispiel

$$\begin{aligned}\Gamma = \quad & (A \vee B \vee C) \wedge (A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C \vee X) \wedge \\ & (A \vee \neg B \vee \neg C \vee X) \wedge (\neg A \vee \neg B \vee C \vee \neg X)\end{aligned}$$

Im DPLL-Algorithmus wird erst $X = $ *false* und dann $A = $ *false* und dann $B = $ *false* gesetzt. Durch Unit Propagation erhalten wir aus den ersten beiden Klauseln einen Konflikt. Beteiligt waren nur die Setzungen $A = B = $ *false*. Wir können daher die Klausel $A \vee B$ lernen. Sie hilft uns später bei $X = $ *true*.

Anwendungen von SAT-Solvern

Auf der Vorlesungshomepage finden Sie Haskell- und Java-Programme, welche die Tseitin-Transformation, die Übersetzung ins DIMACS-Format und den Aufruf eines SAT-Solvers implementieren.
Das Haskell-Programm hat folgenden Datentyp für Formeln.

```
data Formula a = Var a
                | TT
                | FF
                | Neg (Formula a)
                | Or (Formula a) (Formula a)
                | And (Formula a) (Formula a)
                | Xor (Formula a) (Formula a)
                | Imp (Formula a) (Formula a)
                | Iff (Formula a) (Formula a)
                | BigOr [Formula a]
                | BigAnd [Formula a]
```

Tseitin-Übersetzung und Sat-Solver in Haskell

Das Modul `Tseitin` implementiert die Tseitin-Übersetzung und kann auch gleich einen Sat-Solver aufrufen.

```
data Solution a
  = Unsatisfiable
  | Satisfiable (Map a Bool)

satisfiable :: Ord a => Formula a -> IO (Solution a)
```

Beim Ausführung von `satisfiable phi` wird phi mit der Tseitin-Übersetzung in KNF umgewandelt und es wird Picosat aufgerufen. Bei Unerfüllbarkeit wird `Unsatisfiable` zurückgegeben, sonst `Satisfiable(eta)`, wobei `eta` eine erfüllende Belegung ist.

Beispiel

```
module Main where
import Formula
import Tseitin

phi :: Formula String
phi = And (Var "x") (Var "y")

main = do
  answer <- satisfiable phi
  case answer of
    Unsatisfiable ->
      putStr "phi ist nicht erfüllbar."
    Satisfiable eta ->
      putStr ("phi ist erfüllbar und zwar mit Belegung "
              ++ show eta)
```

Compilierung und Ausführung

Kompilieren und Ausführen:

```
$ ghc Main.hs
$ ./Main
```

Interaktiv:

```
$ ghci Tseitin.hs

*Tseitin> satisfiable (And (Var "x") (Var "y"))
Satisfiable (fromList [("x",True),("y",True)])

*Tseitin> satisfiable (And (Var "x") (Neg (Var "x")))
Unsatisfiable
```

Sudoku

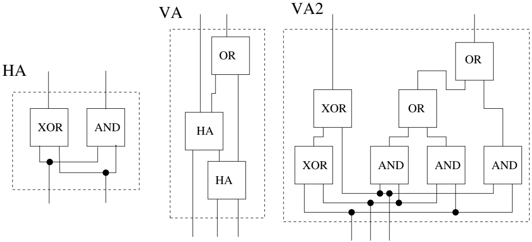
Die Sudoku-Formel kann in einem Programm erzeugt werden.

```
sudoku :: Formula (Int, Int, Int)
sudoku = BigAnd [ at_least, one_number_each_cell
                , row_cond, col_cond,
                , no_number_twice_in_square, givens ]
  where at_least =
          BigAnd [ BigOr [ Var (row, col, i) | i <- [1..9] ]
                 | row <- [0..8], col <- [0..8] ]
        one_number_each_cell =
          BigAnd [ (Neg (Var (row, col, i)))
                   `Or` (Neg (Var (row, col, j)))
                 | row <- [0..8], col <- [0..8],
                   i <- [1..9], j <- [1..9], i < j ]
        ...
```

Der Aufruf `satisfiable sudoku` liefert eine Lösung des Sudokus oder sagt, dass das Problem unlösbar ist.

Anwendung: Schaltkreisverifikation

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

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

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:
$X = Y = \mathit{true}, C = \mathit{false}$

Gleichheit von Addierern

Halbaddierer:

```
ha x y = (x `Xor` y, x `And` y)
```

Volladdierer:

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

Volladdierer (zweite Variante):

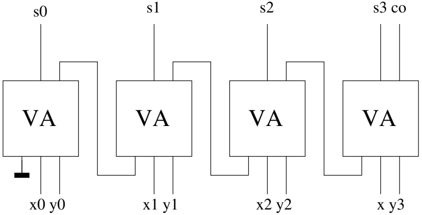
```
va2 x y c =
  let s = (x `Xor` y) `Xor` c
      cr = ((c `And` y) `Or` (y `And` x)) `Or` (c `And` x)
  in (s, cr)
```

Gleichheit von Addierern

```
equiv =
  let (s1, c1) = va (Var "x") (Var "y") (Var "c")
      (s2, c2) = va2 (Var "x") (Var "y") (Var "c")
  in (s1 `Iff` s2) `And` (c1 `Iff` c2)

main = do
  answer <- satisfiable (Neg equiv)
  case answer of
    Unsatisfiable ->
      putStr "equiv ist allgemeingültig."
    Satisfiable eta ->
      putStr ("equiv wird falsch durch die Belegung "
              ++ show eta)
```

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: Der 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 Eingaben direkt zu berechnen.

carry, gen und prop

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

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

Idee: *gen*$_{ij}$ — Am Ende des Blocks $i \dots j$ entsteht ein Übertrag.
prop$_{ij}$ — Der Block $i \dots j$ propagiert einen einlaufenden Übertrag.
Da sich das Intervall jeweils halbiert, ist die Rekursionstiefe und damit die Tiefe der entstehenden Schaltkreise $O(\log(n))$.

Implementierung in Haskell (1)

Den Ripple-Addierer implementiert man so:

```
rippleadd 0 = ([], FF)
rippleadd i =
  let (sumbits, c) = rippleadd (i - 1)
      (sum, c') = va (x (i - 1)) (y (i - 1)) c
  in (sum:sumbits, c')
```

Es gilt insbesondere:

`rippleadd size`$= ([s_{size-1}, \ldots, s_0], co)$.

Implementierung in Haskell (2)

Die gen- und prop-Schaltkreise als Formeln:

```
carry 0 = false
carry i = gen 0 (i - 1)

gen i j =
  if i == j then
    (x i) `And` (y i)
  else
    (gen k j) `Or` ((gen i (k-1)) `And` (prop k j))
    where k = (j + 1 - i) `div` 2 + i

prop i j =
  if i == j then
    (x i) `Or` (y i)
  else
    (prop k j) `And` (prop i (k-1))
    where k = (j + 1 - i) `div` 2 + i
```

Implementierung in Haskell (3)

Lookahead-Addierer:

```
lookaheadadd 0 = ([], FF)
lookaheadadd i =
    let (ss, _) = lookaheadadd (i - 1)
        (s, d) = va2 (x (i - 1)) (y (i - 1)) (carry (i - 1))
    in (s:ss, d)
```

Implementierung in Haskell (4)

Äquivalenz der Addierer:

```
equals size =
  let (sumbits1, carryout1) = rippleadd size
      (sumbits2, carryout2) = lookaheadadd size
  in (carryout1 `Iff` carryout2) `And`
    (BigAnd [s1 `Iff` s2 | (s1, s2) <- zip sumbits1 sumbits2])
```

Der Aufruf

```
satisfiable (Neg (equals 64))
```

prüft nun, ob die beiden 64-Bit-Addierwerke für *alle* Eingaben die gleichen Ergebnisse liefern: Die Formel `(Neg (equals 64))` ist unerfüllbar gdw. `(equals 64)` allgemeingültig ist.

Verifikation nebenläufiger Systeme

Wir wollen Eigenschaften von nebenläufigen Systemen überprüfen.

Beispiel:

Wechselseitiger Ausschluss (Mutual Exclusion)

- P Prozesse arbeiten nebenläufig.
- Die Prozesse teilen sich eine Ressource.
- Es dürfen nicht zwei Prozesse gleichzeitig auf die Ressource zugreifen.

Verschieden Lösungen:

- Semaphoren
- Synchronisierung durch gemeinsame Variablen (z.B. Peterson Algorithmus)

Beispiel: Mutual Exclusion mit Semaphor

Es gibt k Prozesse mit Pseudocode

```
sleep: goto wait;
wait: if (sem == free) { sem = occ; goto work } else { goto wait }
work: /* kritischer Bereich abstrahiert */ sem = free; goto sleep
```

sowie einen Semaphor mit zwei möglichen Zuständen *free* und *occ*.

Wir entfernen aus gegebenen Programmen alle uninteressanten Details. Hier betrachten wir nur die Prozesssynchronisierung.

Pseudocode:

- Jede Zeile hat einen Label.
- In einem Schritt wird eine ganze Zeile auf einmal ausgeführt.

Modellierung nebenläufiger Systeme

Modellierung als Zustandsübergangssystem:

- Eine Menge Z, die Menge aller möglichen Systemzustände.
- Eine Menge $I \subseteq Z$ von möglichen Anfangszuständen.
- Eine Relation $\mapsto \subseteq Z \times Z$. Die Aussage $z \mapsto z'$ drückt aus, dass $z' \in Z$ ein möglicher Nachfolgezustand von $z \in Z$ ist.
- Eine Menge $V \subseteq Z$ von verbotenen bzw. unerwünschten Zuständen.

Frage: Kann das System einen verbotenen Zustand erreichen?

$$I \ni z_0 \mapsto z_1 \mapsto \cdots \mapsto z_n \in V$$

Beispiel: Mutual Exclusion mit Semaphor

Es gibt k Prozesse mit Pseudocode

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

sowie einen Semaphor mit zwei möglichen Zuständen *free* und *occ*.

Modellierung:

$$Z = \{(p_1, \ldots, p_k, s) \mid p_i \in \{\mathit{sleep}, \mathit{wait}, \mathit{work}\}, s \in \{\mathit{free}, \mathit{occ}\}\}$$

$$I = \{(\mathit{sleep}, \ldots, \mathit{sleep}, \mathit{free})\}$$

$$V = \{(p_1, \ldots, p_k, s) \in Z \mid p_i = \mathit{work}, p_j = \mathit{work}, i \neq j\}$$

Zu jedem Zeitpunkt macht *genau ein* Prozess und möglicherweise der Semaphor eine Aktion. Möglich sind:

- $(\ldots, \mathit{sleep}, \ldots, s) \mapsto (\ldots, \mathit{wait}, \ldots, s)$
- $(\ldots, \mathit{wait}, \ldots, \mathit{free}) \mapsto (\ldots, \mathit{work}, \ldots, \mathit{occ})$
- $(\ldots, \mathit{work}, \ldots, s) \mapsto (\ldots, \mathit{sleep}, \ldots, \mathit{free})$

Kodierung in Aussagenlogik

Man kann SAT-Solver dazu verwenden, um zu zeigen, dass ein System nach n Schritten keinen verbotenen Zustand erreicht:

Definiere dazu eine Formel, die genau dann erfüllbar ist, wenn es einen Pfad der Form $I \ni z_0 \mapsto z_1 \mapsto \cdots \mapsto z_t \in V$ für $t \leqslant n$ gibt.

Wähle eine frische Variable q_{zt} für alle $z \in Z$ und $0 \leqslant t \leqslant n$. (Intention: q_{zt} sagt, dass z der t-te Zustand in einem schlechten Pfad ist.)

Klauseln:

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

Kodierung in Aussagenlogik

Ist die Zustandsmenge durch Tupel gegeben, d.h. $z = (z_1, \ldots, z_\ell)$, so kann man auch Variablen $q_{1z_1t}, \ldots, q_{\ell z_\ell t}$ statt q_{zt} einführen.

Bei vier Komponenten à 10 Möglichkeiten benötigt man dann $10 + 10 + 10 + 10$ Variablen statt $10 \cdot 10 \cdot 10 \cdot 10$.

Die Intention von q_{iz_it} ist: Der t-te Zustand in einem schlechten Pfad hat z_i als i-te Komponente.

Die Aussage q_{zt} kann man durch $q_{1z_1t} \wedge \cdots \wedge q_{\ell z_\ell t}$ ausdrücken.

Semaphor-Beispiel mit Aussagenlogik

Variablen:

Für $0 < p \leqslant k$ und $z \in \{\textit{sleep}, \textit{wait}, \textit{work}\}$ und $t \leqslant n$ je eine Variable q_{pzt}. Außerdem für $w \in \{\textit{free}, \textit{occ}\}$ und $t \leqslant n$ je eine Variable s_{wt}.

Klauseln:

- Für $t < n, 0 < p \leqslant k$, genau ein q_{pzt} und genau ein s_{wt} gesetzt.
- $\bigwedge_p q_{p\ \textit{sleep}\ 0} \wedge s_{\textit{free}\ 0}$
- $\bigvee_t \bigvee_{p_1 \neq p_2} q_{p_1 \textit{work}\ t} \wedge q_{p_2 \textit{work}\ t}$
- $\bigwedge_{t<n} \bigvee_p \left(\bigwedge\limits_{p_1 \neq p} \bigwedge\limits_{z} q_{p_1 z t} \Leftrightarrow q_{p_1 z (t+1)} \right) \wedge$
  $$\bigvee_{(z,w,z',w') \in R} q_{pzt} \wedge q_{pz'(t+1)} \wedge s_{wt} \wedge s_{w'(t+1)}$$

Hier ist $R = \{(\textit{sleep}, w, \textit{wait}, w) \mid w \in \{\textit{free}, \textit{occ}\}\} \cup$
$\{(\textit{wait}, \textit{free}, \textit{work}, \textit{occ})\} \cup \{(\textit{work}, w, \textit{sleep}, \textit{free}) \mid w \in \{\textit{free}, \textit{occ}\}\}$.

Peterson Algorithmus

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

Zwei Prozesse 0,1. Jeder hat ein Flag $\mathit{flag}[i]$ und es gibt eine Variable $\mathit{turn} : \{0, 1\}$, 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 /* kritischer Bereich hier */; flag[p] := 0; goto 0
```

Peterson Algorithmus

- Variablen $line_{p,i,t}$ für alle $p \in \{0,1\}, i \in \{0,1,2,3,4\}, t < n$.
  Zum Zeitpunkt t ist Prozess Nummer p in Zeile i.
- Variablen $flag_{f,p,t}$ für alle $p, f \in \{0,1\}, t < n$.
  Zum Zeitpunkt t hat flag$[p]$ den Wert f.
- Variablen $turn_{p,t}$ für alle $p \in \{0,1\}, t < n$.
  Zum Zeitpunkt t hat turn den Wert p.

Die Klauseln erzwingen, dass die Variablen die gewünschte Bedeutung haben, sowie dass zwei Prozesse den kritischen Bereich irgendwann gleichzeitig betreten (erfüllbar gdw. System inkorrekt).

- `sanity`: Zu jeder Zeit ist jeder Prozess in genau einer Zeile, und die Variablen *flag* und *turn* haben genau einen Wert.
- `initial`: Anfangszustand des Systems
- `transitions`: Aufzählung aller möglichen Zustandsübergänge von Zeit t zu $t+1$, für alle $t < n$.
- `undesired`: Ungewünschter Systemzustand

Zusammenfassung: SAT-Solver

- Kodierung von Problemen als Erfüllbarkeitsprobleme in Aussagenlogik
  - Sudoku
  - Schaltkreisverifikation
  - Nebenläufige Systeme
- Tseitin-Übersetzung
- DPLL-Verfahren

Binäre Entscheidungsdiagramme

Boole'sche Funktionen sind Funktionen, die Wahrheitswerte auf Wahrheitswerte abbilden.

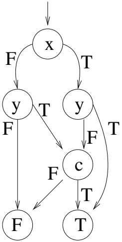
Aussagenlogische Formeln repräsentieren Boole'sche Funktionen.
Beispiel: $(x \vee \neg y) \wedge (\neg x \vee y) \wedge \neg(x \wedge y)$ repräsentiert $f(x, y)$ mit:

x	y	$f(x, y)$
false	*false*	*true*
false	*true*	*false*
true	*false*	*false*
true	*true*	*false*

Binäre Entscheidungsdiagramme (binary decision diagrams, BDDs) sind eine effiziente Repräsentierung Boole'scher Funktionen.

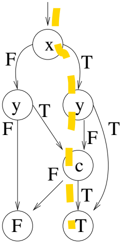
BDDs Grundidee

Man kann Boole'sche Funktionen durch Entscheidungsdiagramme 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.: $x = c =$ *true* (im Diagramm "T") und $y =$ *false* (im Diagramm "F"):



Definition BDD

Es wird eine totale Ordnung auf den Variablen fixiert.

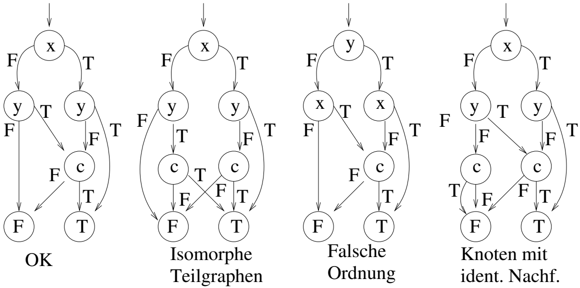
Definition (BDD)

Ein BDD ist ein gerichteter azyklischer Graph, so dass:

- Ein Knoten ist als Wurzelknoten ausgezeichnet.
- Jeder Knoten ist entweder mit T, F oder einer Variablen beschriftet; jede Kante ist entweder mit T oder F beschriftet.
- Knoten, die mit T oder F beschriftet sind, haben keine ausgehenden Kanten.
- Knoten, die mit einer Variablen x beschriftet sind, haben genau zwei ausgehende Kanten. Die beiden Nachfolgerknoten dürfen jeweils nur mit T, F oder einer Variablen y beschriftet sein, die in der Variablenordnung nach x kommt.
- Es gibt keine zwei verschiedene isomorphe Teil-BDDs.

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 zweiten Beispiel sind die Teil-BDDs mit Wurzel c isomorph.

Erzeugung von BDDs

Ein *nichtreduziertes BDD* ist ein BDD, welches isomorphe verschiedene Teil-BDDs und identische Nachfolger enthalten kann. Etwa das zweite 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 identischen Nachfolgern (T und F Kanten) verschmelzen.
- Alle T-Blätter verschmelzen; alle F-Blätter verschmelzen.

Satz: Kann keine dieser Reduktionsregeln 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 die Äquivalenz von 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 $f[x{:=}\textit{true}]$ und $f[x{:=}\textit{false}]$ bezeichnet man die Funktionen, die man erhält, wenn x entsprechend vorbesetzt wird.

Es gilt (Shannon-Zerlegung):

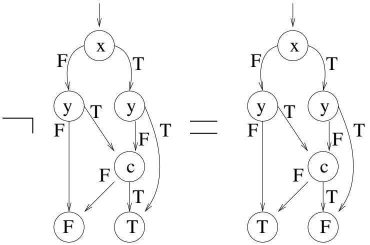
$$\begin{aligned} f &= x \wedge f[x{:=}\textit{true}] \vee \neg x \wedge f[x{:=}\textit{false}] \\ &= (\neg x \vee f[x{:=}\textit{true}]) \wedge (x \vee f[x{:=}\textit{false}]) \end{aligned}$$

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 $f_p[x{:=}\textit{false}]$ und $f_p[x{:=}\textit{true}]$ (wobei f_p die durch p definierte Boole'sche Funktion ist).

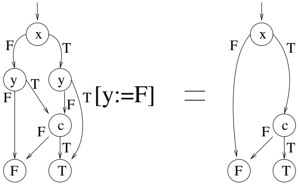
Negation und einstellige Operationen

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

Die Berechnung von zweistelligen Booleschen Operationen auf BDDs ($\vee$, $\wedge$, $\Rightarrow$, $\ldots$) kann durch einen einzigen Algorithmus erklärt werden.

Sei $\oplus$ eine zweistellige Boolesche Funktion.
(Infix-Schreibweise, d.h. $x \oplus y$ statt $\oplus(x, y)$)

Gegeben:
BDDs p_1 und p_2 bezüglich der gleichen Variablenordnung.

Aufgabe:
Berechnung eines BDDs $p_1 \oplus p_2$ mit Boolescher Funktion $f_{p_1 \oplus p_2}(\vec{x}) = f_{p_1}(\vec{x}) \oplus f_{p_2}(\vec{x})$.

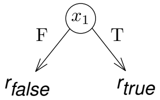
Zweistellige Boole'sche Operationen (2)

Rekursiver Algorithmus zur Berechnung von $p_1 \oplus p_2$.

- Fall: p_1 ist ein Blatt mit Beschriftung b_1.
  Man erhält $p_1 \oplus p_2$, indem man die einstellige Funktion $b_1 \oplus -$ auf die Blätter von p_2 anwendet und das Ergebnis reduziert.
- Fall: Die Wurzeln von p_1 sind p_2 ist innere Knoten mit Variablen x_1 und x_2.
  - Fall $x_1 \leqslant x_2$. Dann sind $p_1[x_1{:=}b]$ und $p_2[x_1{:=}b]$ Sub-BDDs von p_1 bzw. p_2. Berechne rekursiv:

    $$\begin{aligned} r_{\mathit{false}} &:= p_1[x_1{:=}\mathit{false}] \oplus p_2[x_1{:=}\mathit{false}] \\ r_{\mathit{true}} &:= p_1[x_1{:=}\mathit{true}] \oplus p_2[x_1{:=}\mathit{true}] \end{aligned}$$

    Das Ergebnis $p_1 \oplus p_2$ erhält man durch Reduktion von:

  - Fall $x_1 > x_2$. Analog mit x_2 statt x_1.

Zweistellige Boole'sche Operationen (3)

Implementierung mit dynamischer Programmierung

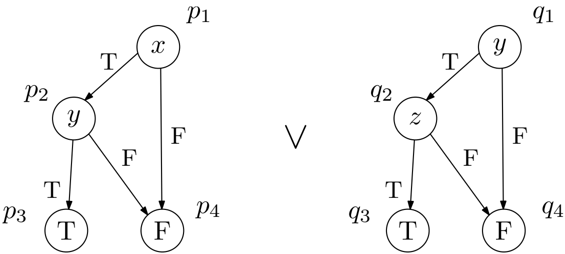
Bei der Berechnung von $p_1 \oplus p_2$ werden rekursiv BDDs $q_1 \oplus q_2$ berechnet, wobei q_1 und q_2 Sub-BDDs von p_1 bzw. p_2 sind.

Um die mehrfache Berechnung von $q_1 \oplus q_2$ zu verhindern, legt man eine Tabelle an, in der für jedes Paar (q_1, q_2) von Sub-BDDs von p_1 bzw. p_2 das Ergebnis $q_1 \oplus q_2$ gespeichert wird.

Die Tabelle wird dann sukzessive ausgefüllt.

Beispiel

Man berechne:



Die Teil-BDDs sind mit p_1, p_2, p_3, p_4 und q_1, q_2, q_3, q_4 bezeichnet.
Zu berechnen ist also $p_1 \vee q_1$.

Beispiel

Nebenrechnungen (1):

$$p_3 \lor q_2 = \begin{tikzcd} & z \arrow[dl, "T"'] \arrow[dr, "F"] & \\ T & & T \end{tikzcd} = T$$

$$p_4 \lor q_4 = F$$

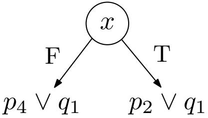
$$p_2 \lor q_1 = \begin{tikzcd} & y \arrow[dl, "F"'] \arrow[dr, "T"] & \\ p_4 \lor q_4 & & p_3 \lor q_2 \end{tikzcd} = \begin{tikzcd} & y \arrow[dl, "F"'] \arrow[dr, "T"] & \\ F & & T \end{tikzcd}$$

Beispiel

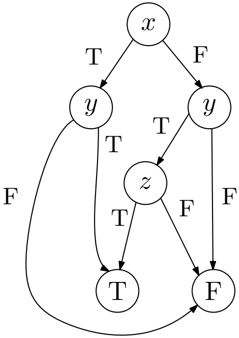
Nebenrechnungen (2):

$$p_4 \vee q_1 = q_1$$

$p_1 \vee q_1 =$



$=$



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 …

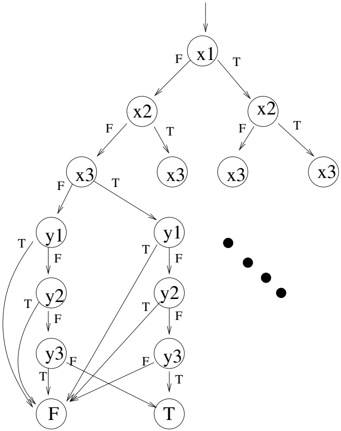
Beispiel einer Familie von BDDs exponentieller Größe

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.

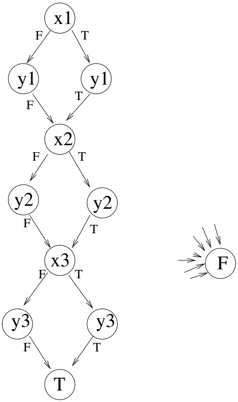


Selbe Funktion, andere Ordnung

Mit der Variablenordnung

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

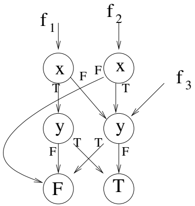
haben die BDDs lineare Größe.
Alle fehlenden Kanten führen in das *false*-Blatt.



Implementierung I

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

Treten zum Beispiel die Funktionen $f_1 = x \Leftrightarrow y$, $f_2 = x \wedge \neg y$, $f_3 = \neg y$ 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 eine 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 viele verschiedene Programmiersprachen. Wir verwenden die C-Bibliothek BuDDy mit Bindings für Haskell und Java.

Die Bibliothek BuDDy

Bibliothek für BDDs in C/C++

- Datentyp für BDDs
- Konstruktion von BDDs: $\neg, \vee, \wedge, \exists, \forall, \ldots$
- Variablenumbenennung, Variablenumordnung, ...
- Siehe Dokumentation für weitere Funktionen.

Haskell Binding:

- haskell-buddy:
  `https://github.com/jwaldmann/haskell-buddy`

Java-Interface für BDDs auf der Vorlesungshomepage:

- Enthält Java-Version von BuDDy und benötigt keine Installation von Bibliotheken
- Implementiert die eben vorgestellten Algorithmen
- Beispielprogramme

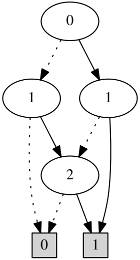
Beispiel in Java

```
// Erzeuge Variablen in der Reihenfolge ihrer Ordnung
BDDVar x = freshBDDVar(); // Variable mit Nummer 0
BDDVar y = freshBDDVar(); // Variable mit Nummer 1
BDDVar c = freshBDDVar(); // Variable mit Nummer 2

// Verknuepfung von BDDs
BDD b = or(and(var(x), var(y)), and(var(x), var(c)),
           and(var(y), var(c)));

// Beispiel fuer Gleichheitstest
boolean unsat = equal(b, constantFalse());
if (unsat) { System.out.println("b ist unerfuellbar"); }

// Ausgabe als Graph
printGraph("b.dot", b);
```

(siehe Vorlesungshomepage für vollständiges Programm)

Beispiel in Java

Das Programm berechnet das BDD `b` und schreibt es im Graphviz-Format in die Datei `b.dot`.



Umwandlung ins PDF-Format (z.B. im CIP-Pool):

```
dot -Tpdf b.dot > b.pdf
```

Alternativ kann man Web-Tools zur Anzeige verwenden, z.B.:
`https://mdaines.github.io/viz.js/`

Beispiel: Lookahead Addierer mit BDDs

Die Korrektheit des Lookahead-Addierers kann auch mit BDDs überprüft werden.

Für den Ripple-Carry-Addierer und Carry-Lookahead-Addierer berechenen wir für jedes Summenbit und das Übertragsbit ein BDD.

- Wir berechnen das BDD genau wie die aussagenlogische Formel. Die Verknüpfungen $\wedge$, $\vee$ werden jetzt als binäre Operationen auf BDDs interpretiert.
- Anstelle von `cnf.or`, `cnf.and`, ... zur Konstruktion von Formeln rufen wir einfach durch `bdd.or`, `bdd.and`, ... zur Berechnung von BDDs auf.

Wir prüfen Gleichheit der beiden Addierer, indem wir die BDDs für die einzelnen Bits auf Gleichheit testen:
Der Aufruf `equal(b1, b2)` vergleicht BDDs `b1` und `b2`.

Nebenläufige Systeme mit BDDs

BDDs erlauben eine Modellierung nebenläufiger Systeme ohne Zeitschranke (“tmax”).

Idee:

- Gegeben ein endliches Zustandsübergangssystem, berechne die Menge aller Zustände, die von einem Anfangszustand erreichbar sind.
- Prüfe, ob diese Menge einen verbotenen Zustand enthält.

Problem:

Nebenläufige Systeme haben meist sehr viele Zustände.

- `HashSet` o.ä. ist als Datenstruktur nicht geeignet.
  Im Peterson-Beispiel mit drei Prozessen sind über 10^{16} Zustände erreichbar.
- Mit BDDs können große Mengen in vielen Fällen effizient repräsentiert werden.

BDDs zur Repräsentierung von Mengen

BDDs können als Datenstruktur zur Repräsentierung großer endlicher Mengen verwendet werden.

Sei Z eine endliche Menge. Angenommen für jedes Element von Z wurde eine eindeutige Kodierung mit n Bits festgelegt.

Dann kann jede Teilmenge von $X \subseteq Z$ als Boole'sche Funktion über den Variablen $x_0, x_1, \dots, x_{n-1}$ verstanden werden:

- Bedeutung der Variablen: x_i ist i-tes Bit der Binärkodierung.
- Die Funktion gibt genau dann *true* zurück, wenn die Variablen $x_0, x_1, \dots, x_{n-1}$ ein Element von X kodieren. (*charakteristische Funktion*)

BDDs zur Repräsentierung von Mengen

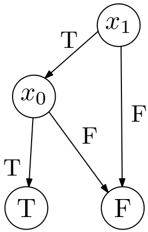
Beispiel

$$Z = \{1, 2, \ldots, 15\}$$
$$X = \{3, 7, 11, 15\}$$

Binärkodierung:

x_3	x_2	x_1	x_0	Element
false	*false*	*false*	*true*	1
false	*false*	*true*	*false*	2
false	*false*	*true*	*true*	3
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
true	*true*	*true*	*true*	15

BDD für X:



Operationen auf Mengen

Angenommen

- das BDD b1 repräsentiert $X_1 \subseteq Z$,
- das BDD b2 repräsentiert $X_2 \subseteq Z$,
- das BDD `sanity` repräsentiert $Z \subseteq Z$.

Dann gilt:

- b1 $\wedge$ b2 repräsentiert $X_1 \cap X_2$.
- b1 $\vee$ b2 repräsentiert $X_1 \cup X_2$.
- `sanity` $\wedge\ \neg$b1 repräsentiert $Z \backslash X_1$.

Eigenschaften der Mengen können durch Gleichheitstests berechnet werden.

- X_1 ist leer gdw. b1 $= \perp$.
- $X_1 \subseteq X_2$ gdw. b1 $\wedge$ b2 $=$ b1.

Erreichbarkeit mit BDDs

Ziel: Berechnung der Menge aller errechbaren Zustänge in einem Zustandsübergangssystem als BDD und entscheide, ob sie einen verbotenen Zustand enthält.

Repräsentiere die Mengen Z, I und V durch BDDs `sanity`, `initial` und `undesired` über den Variablen $x_0, \ldots, x_{n-1}$.

Repräsentiere die Relation $\mapsto$ durch ein BDD `next` mit der doppelten Anzahl von Variablen $x_0, x'_0, \ldots, x_{n-1}, x'_{n-1}$ wie folgt:

Angenommen

- die Werte von $x_0, \ldots, x_{n-1}$ kodieren ein Element $z \in Z$, und
- die Werte von $x'_0, \ldots, x'_{n-1}$ kodieren ein Element $z' \in Z$,

dann gibt der Wert von `next` an, ob $z \mapsto z'$ gilt.

Erreichbarkeit in i Schritten

Schreibe $R_i \subseteq Z$ für die Menge der Zustände, die man von einem Startzustand in höchstens i Schritten erreichen kann.

Rekursive Definition:

$$
\begin{aligned}
R_0 &= I \\
R_{i+1} &= R_i \cup \{z' \in Z \mid \exists z.\, z \in R_i \wedge z \mapsto z'\}
\end{aligned}
$$

Es gilt $R_0 \subseteq R_1 \subseteq R_2 \subseteq \dots$.

Implementierung mit BDDs

Berechne ein BDD $\texttt{reach}_i$ für R_i durch:

- $\texttt{reach}_0 := \texttt{initial}$
- Falls $\texttt{reach}_i$ bereits berechnet wurde, so berechne

  $$\texttt{reach}_{i+1} := \texttt{reach}_i \vee (\texttt{sanity} \wedge \texttt{rename}(\texttt{exists}(\texttt{reach}_i \wedge \texttt{next})))$$

Hierbei bezeichnet

- $\texttt{exists}$ die existentielle Quantifizierung der ungestrichenen Variablen, d.h. $\texttt{exists}(b) = \exists x_0. \exists x_1. \dots \exists x_{n-1}(b)$.
- $\texttt{rename}$ die Umbenennung der gestrichenen Variablen in die ungestrichenen, d.h. x_i' wird in x_i umbenannt, für alle $i = 0, \dots, n-1$.

Die Umbenennung ist nötig, da $\texttt{exists}(\texttt{reach}_i \wedge \texttt{next})$ zwar ein BDD für die Menge $\{z' \mid \exists z \in R_i.\, z \mapsto z'\}$ ist, allerdings in den Variablen $x_0', \dots, x_{n-1}'$, statt wie gewünscht mit $x_0, \dots, x_{n-1}$.

Erreichbarkeit

Nach Konstruktion ist R_i die Menge der Zustände, die man in höchstens i Schritten von einem Startzustand aus erreichen kann.

$$R_0 \subseteq R_1 \subseteq R_2 \subseteq \dots$$

Die Menge Z der Zustände ist endlich. Es muss daher ein k geben mit $R_k = R_{k+1}$. Die Menge R_k enthält dann *alle* erreichbaren Zustände.

Fixpunktiteration: Zur Berechnung aller erreichbaren Zustände genügt es also nacheinander $R_0, R_1, \dots$ zu berechnen, bis irgendwann $R_k = R_{k+1}$ gilt, d.h. bis ein Fixpunkt gefunden ist.

Erreichbarkeit

Der folgende Algorithmus berechnet also die erreichbaren Zustände als BDD.

$$
\begin{array}{l}
\mathtt{reach} := \mathtt{initial} \\
\text{repeat} \\
\quad \mathtt{old} := \mathtt{reach} \\
\quad \mathtt{reach} := \mathtt{old} \lor (\mathtt{sanity} \land \mathtt{rename}(\mathtt{exists}(\mathtt{old} \land \mathtt{next}))) \\
\text{until } \mathtt{reach} = \mathtt{old}
\end{array}
$$

Die Abbruchbedingung in der repeat-until Schleife kann mit BDDs in konstanter Zeit geprüft werden.

Nebenläufige Systeme mit BDDs

Beispiel: Wechselseitiger Ausschluss mit Semaphore

Repräsentation des Zustandsübergangssystems durch BDDs:

- Wähle folgende Variablen: Für $p < P$ (Anzahl der Prozesse) und $z \in \{\textit{sleep}, \textit{wait}, \textit{work}\}$ je eine Variable q_{pz}. Außerdem für $w \in \{\textit{free}, \textit{occ}\}$ je eine Variable s_w.
- Jeder Zustand im Semaphoren-Beispiel entspricht einer Belegung dieser Variablen.
- Boolesche Funktionen (und damit BDDs) über diesen Variablen repräsentieren Zustandsmengen.

Semaphor mit BDDs

Alle Zustände Z:

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

(Wir brauchen ein BDD für Z, da nicht alle Binärkodierungen ein Element von Z kodieren.)

Anfangszustände I:

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

Verbotene Zustände V:

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

Zustandsübergangsrelation

- Führe einen zweiten Satz Variablen für Folgezustand ein: q'_{pz} und s'_w für alle p, z und 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".

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

wobei $R = \{(\mathit{sleep}, w, \mathit{wait}, w) \mid w \in \{\mathit{free}, \mathit{occ}\}\} \cup \{(\mathit{wait}, \mathit{free}, \mathit{work}, \mathit{occ})\} \cup \{(\mathit{work}, w, \mathit{sleep}, \mathit{free}) \mid w \in \{\mathit{free}, \mathit{occ}\}\}$.

Iteration

Berechnung der erreichbaren Zustände.

$$\begin{array}{l} \texttt{reach} := \texttt{initial} \\ \text{repeat} \\ \quad \texttt{old} := \texttt{reach} \\ \quad \texttt{reach} := \texttt{old} \vee (\texttt{sanity} \wedge \texttt{rename}(\texttt{exists}(\texttt{old} \wedge \texttt{next}))) \\ \text{until } \texttt{reach} = \texttt{old} \end{array}$$

Zur Erinnerung:

- `exists` die existentielle Quantifizierung der ungestrichenen Variablen.
- `rename` die Ersetzung der gestrichenen Variablen durch die ungestrichenen.

Korrektheit

Wechselseitiger Ausschluss ist gegeben, wenn kein Zustand in V erreichbar ist.

Das ist genau dann der Fall wenn

$$\texttt{reach} \wedge \texttt{undesirable} = \bot$$

gilt.

Es genügt also, die beiden BDDs mit $\wedge$ zu verknüpfen und zu überprüfen, dass das Ergebnis das BDD F ist.

Zusammenfassung Kapitel 1

- Wiederholung Aussagenlogik
- Erfüllbarkeitsäquivalente 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.