



# Symbolisches Model Checking und BDD- Semantik

H. Peter Gumm

Philipps-Universität Marburg

Sommersemester 2007



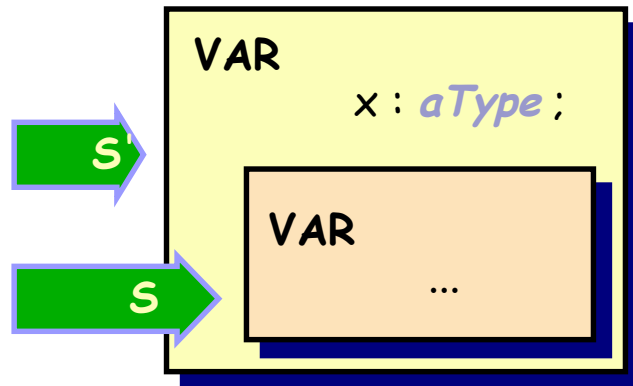
# Explizites Model Checking

- Repräsentation
  - proportional zur Größe des Modells  $O(|S|)$
  - zusätzlich für CTL:
    - proportional zur Größe der Formel  $O(|\varphi|)$
  - zusätzlich für LTL:
    - exponentiell in Größe der Formel  $O(2^{|\varphi|})$
- Modell wächst exponentiell mit #Variablen
  - Reale Modelle benötigen viele Variablen
    - 20, 50, 100, ...
  - $2^{100} = 10^{30}$  ...
    - Die Sonne wiegt ca.  $10^{34}$  g !



# Zustandsexplosion

- Hinzunahme einer Variablen  $x$   
`VAR  $x$  :  $aType$ ;`



$$S' = S \times aType,$$

also

$$|S'| = |S| * |aType|$$

- Symbolisches Model Checking:
  - repräsentiere  $S$  symbolisch - als Formel



# Symbolisches Model Checking

- Repräsentiere  $S$  symbolisch
  - als Formel
  - als Boolescher Term
  - als Entscheidungsgraph
- Vorteil
  - Vermeidung der Zustandsexplosion
- erfordert
  - symbolische Repräsentation von
    - Zustandsraum
    - Transitionsrelation
    - Spezifikation
- Algorithmen müssen auf symbolischer Repräsentation arbeiten
  - entsprechende Datenstrukturen erforderlich
  - BDD-Paket
    - gibt es schon einige unter GPL





# Symbolische Repräsentation

Teilmengen P einer festen Menge

$S = \{0,1\}^k$ ,  $x_1, \dots, x_k$  Variablen. P Teilmenge von S.  
Repräsentiere P durch *charakteristische Funktion* :

$$\chi_P = \lambda(x_1, \dots, x_k). \text{ if } (x_1, \dots, x_k) \in P \text{ then } 1 \text{ else } 0$$

Dafür gibt es einen Booleschen Term  $p(x_1, \dots, x_k)$  mit

$$\chi_P = \lambda(x_1, \dots, x_k). p(x_1, \dots, x_k)$$

Boolesche Operatoren.

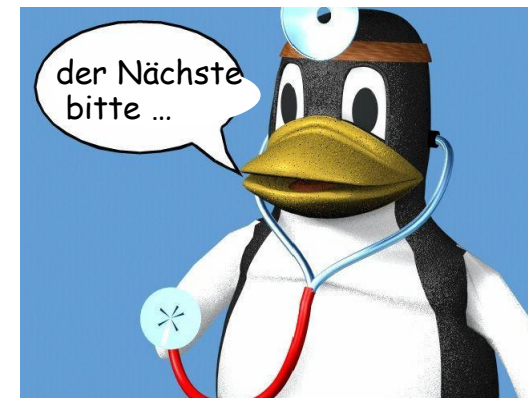
Sind  $P, Q \subseteq S$  repräsentiert durch  $\lambda(x_1, \dots, x_k). p$   
und  $\lambda(x_1, \dots, x_k). q$ , so entsprechen sich:

S	und	$\lambda(x_1, \dots, x_k). \text{tt}$
$\emptyset$	und	$\lambda(x_1, \dots, x_k). \text{ff}$
S-P	und	$\lambda(x_1, \dots, x_k). \neg p$
$P \cap Q$	und	$\lambda(x_1, \dots, x_k). p \wedge q$
$P \cup Q$	und	$\lambda(x_1, \dots, x_k). p \vee q$



# next-Variable – $x'$

- Für die Darstellung der Transitionsrelation benötigt:
  - Paare von Variablen  $(x, x')$
  - Relation drückt Zusammenhang zwischen  $x$  und  $x'$  aus
  - Zu jeder Variablen  $x_1, \dots, x_n$  führe entsprechende next-Variable  $x'_1, \dots, x'_n$  ein.
- next-Eigenschaft
  - $p = p(x_1, \dots, x_n)$  sei Eigenschaft
  - $p$  gilt im nächsten Zustand, falls  $p(x'_1, \dots, x'_n)$
  - $p'$  entstehe aus  $p$  durch Ersetzen jeder Variablen  $x_i$  durch  $x'_i$ .
- Formale Definition
  - $(x_i)' := x'_i$
  - $tt' := tt, ff' := ff$
  - $(\neg p)' := \neg(p')$
  - $(p \wedge q)' = p' \wedge q'$
  - $(p \vee q)' = p' \vee q'$
- Next-Tupel
  - $x = (x_1, \dots, x_n)$ , dann  $x' := (x'_1, \dots, x'_n)$

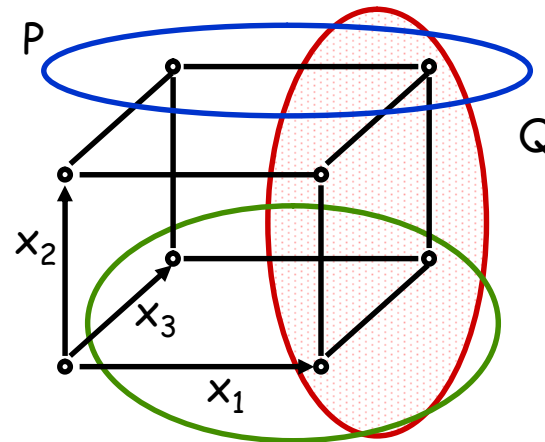




# Repräsentation von Relationen

- Zweistellige Relation  $R$ :
  - $R \subseteq \{0,1\}^k \times \{0,1\}^k = \{0,1\}^{2k}$
- es gibt einen Booleschen Term  $r(x_1, \dots, x_k, x'_1, \dots, x'_k)$  mit
  - $R = \lambda((x_1, \dots, x_k), (x'_1, \dots, x'_k)). r(x_1, \dots, x_k, x'_1, \dots, x'_k)$ .

Beispiel : Sei  $S = 2^3$ .



$$\begin{aligned} p(x_1, x_2, x_3) &:= x_2 \wedge x_3 \\ q(x_1, x_2, x_3) &:= x_1 \\ r(x_1, x_2, x_3) &:= \neg x_2 \wedge (x_1 \vee x_3) \end{aligned}$$

Die Relation "**diametral**" wird repräsentiert durch

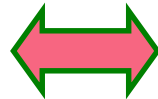
$$r(x_1, x_2, x_3, x'_1, x'_2, x'_3) := (x_1 \Leftrightarrow \neg x'_1) \wedge (x_2 \Leftrightarrow \neg x'_2) \wedge (x_3 \Leftrightarrow \neg x'_3),$$

wobei wir  $u \Leftrightarrow v$  als Abkürzung für  $(u \wedge v) \vee (\neg u \wedge \neg v)$  verwenden.



# Programme

```
VAR
  x, y: {0,1}=1;
BEGIN
  WHILE true DO
    x := x+y
  END
```



$$S = \{0,1\} \times \{0,1\}.$$

$$S_0 = \{ (1,1) \}$$

$$R = \{ ((0,0),(0,0)), ((0,1),(1,1)), ((1,0),(1,0)), ((1,1),(0,1)) \}$$

Formal:

$$S = \{ (x,y) \mid x \in \{0,1\} \wedge y \in \{0,1\} \}.$$

$$S_0 = \{ (x,y) \mid x=1 \wedge y=1 \}$$

$$R = \{ ((x,y),(x',y')) \mid x'=x+y \wedge y'=y \}$$

Charakteristische Funktionen  
als Boolesche Terme:

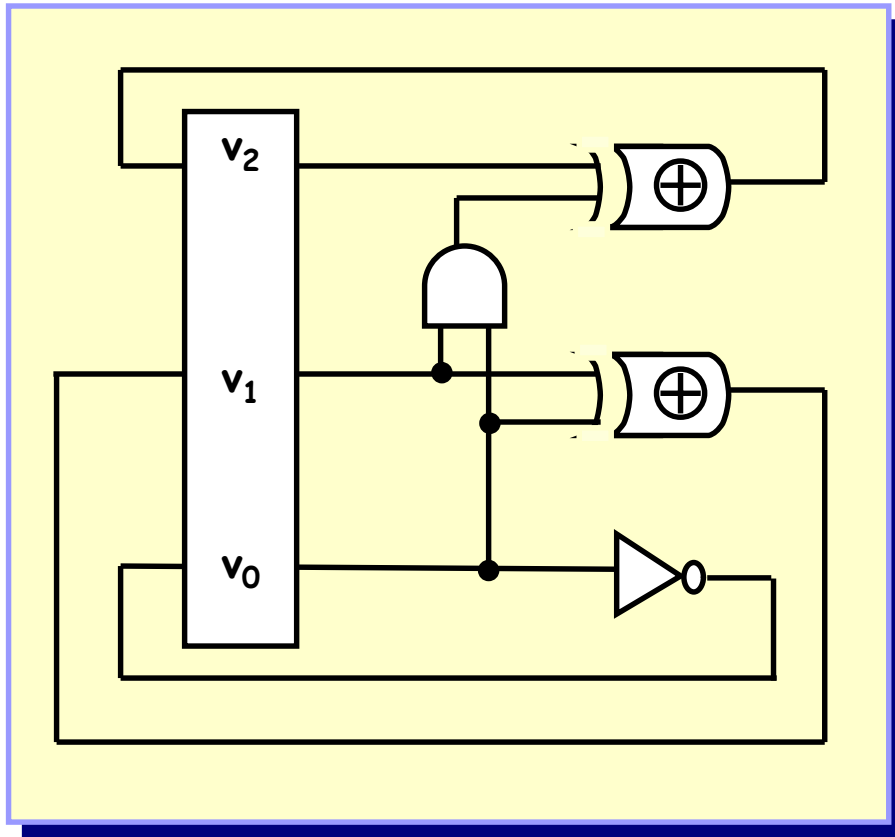
$$P_S(x,y) := \text{true}$$

$$P_I(x,y) := x \wedge y$$

$$P_R(x,y,x',y') := (x' \Leftrightarrow x+y \wedge y' \Leftrightarrow y)$$



# Eine Synchrone Schaltung



$$p_S(v_0, v_1, v_2) := \text{tt}$$

Die einzelnen Transitionen sind

$$v_0' = \neg v_0$$

$$v_1' = v_0 \oplus v_1$$

$$v_2' = (v_0 \wedge v_1) \oplus v_2$$

Also

$$p_{R0} = v_0' \Leftrightarrow \neg v_0$$

$$p_{R1} = v_1' \Leftrightarrow v_0 \oplus v_1$$

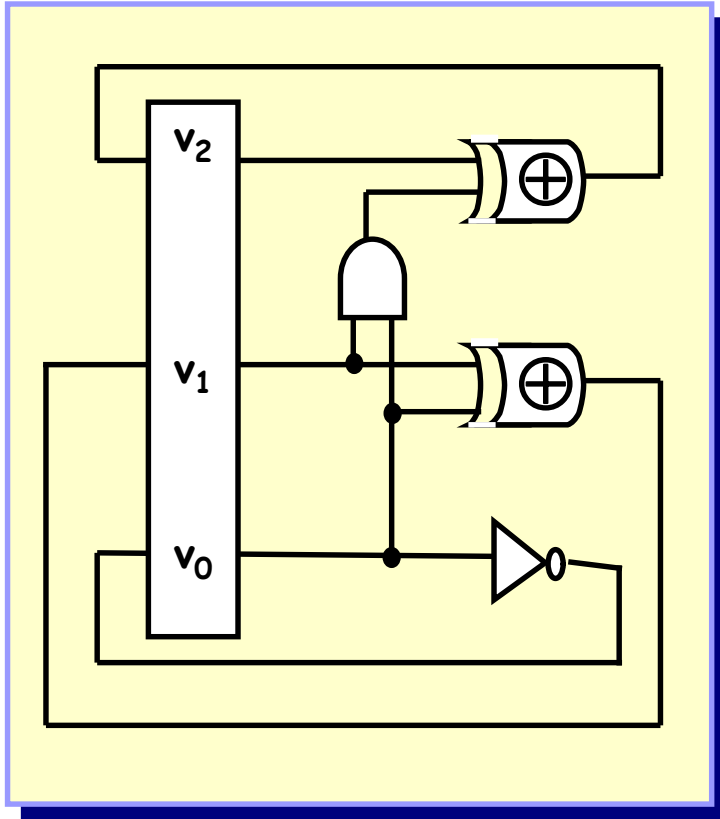
$$p_{R2} = v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2$$

Falls alle Register gleichzeitig (synchron) ihren Wert erneuern, dann gilt:

$$p_R = p_{R0} \wedge p_{R1} \wedge p_{R2} := (v_0' \Leftrightarrow \neg v_0) \wedge (v_1' \Leftrightarrow v_0 \oplus v_1) \wedge (v_2' \Leftrightarrow ((v_0 \wedge v_1) \oplus v_2))$$



# Asynchrone Schaltung



Wird die gleiche Schaltung asynchron betrieben, dann kann jedes Register unabhängig seinen Wert neu berechnen, während alle anderen fest bleiben:

$$P_{R0} = (v_0' \Leftrightarrow \neg v_0) \wedge (v_1' \Leftrightarrow v_1) \wedge (v_2' \Leftrightarrow v_2)$$

$$P_{R1} = (v_1' \Leftrightarrow v_0 \oplus v_1) \wedge (v_0' \Leftrightarrow v_0) \wedge (v_2' \Leftrightarrow v_2)$$

$$P_{R2} = (v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2) \wedge (v_0' \Leftrightarrow v_0) \wedge (v_2' \Leftrightarrow v_2)$$

Falls die Register **asynchron** ihren Wert erneuern:

$$P_R = P_{R0} \vee P_{R1} \vee P_{R2} =$$

$$(v_0' \Leftrightarrow \neg v_0) \wedge (v_1' \Leftrightarrow v_1) \wedge (v_2' \Leftrightarrow v_2)$$

$$\vee (v_1' \Leftrightarrow v_0 \oplus v_1) \wedge (v_0' \Leftrightarrow v_0) \wedge (v_2' \Leftrightarrow v_2)$$

$$\vee (v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2) \wedge (v_0' \Leftrightarrow v_0) \wedge (v_2' \Leftrightarrow v_2)$$



# Allgemeine Schaltungen

Eine asynchrone Schaltung bestehe aus den **Schaltgliedern**  $f_1, \dots, f_n$  die mit den **Drähten**  $d_1, \dots, d_m$  verbunden sind.

Für jeden Draht  $d_k$  führe zwei **Variablen**,  $v_k$  und  $v'_k$  ein. Setze  $V = \{v_1, \dots, v_m\}$ ,  $V' = \{v'_1, \dots, v'_m\}$ .

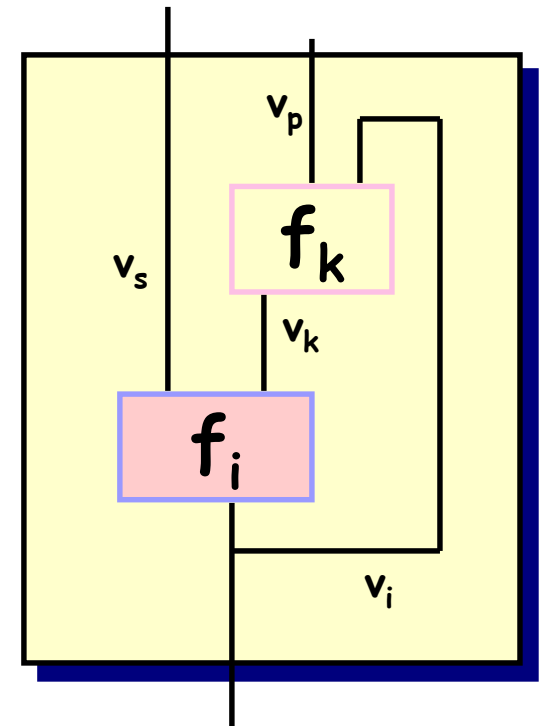
Jedes Schaltglied beschreibt eine Funktion  $v'_k = f_k(V)$ .

Dies ergibt mögliche **Transitionsrelationen**:

$$R_k(V, V') \equiv (v'_k \Leftrightarrow f_k(V)) \wedge \bigwedge_{i \neq k} (v'_i \Leftrightarrow v_i).$$

**Beschreibung der asynchronen Schaltung:**

$$p_R(V, V') = p_{R1}(V, V') \vee \dots \vee p_{Rn}(V, V')$$



Annahme: Draht  $v_k$  ist am Ausgang höchstens eines Schaltglieds  $f_k$ .



# Boolesche Quantifikation

Um die CTL-Operatoren bequem charakterisieren zu können führen wir noch die sogenannte Boolesche Quantifikation von Booleschen Termen ein :

$$\exists x. p = p[x = \text{tt}] \vee p[x = \text{ff}]$$

Boolesche Quantifikation einer Variablen

Beispiel:  $\exists x_1. (x_1 \vee x_3 \Rightarrow \neg x_2) = (\text{tt} \vee x_3 \Rightarrow \neg x_2) \vee (\text{ff} \vee x_3 \Rightarrow \neg x_2)$   
 $= \neg x_2 \vee (x_3 \Rightarrow \neg x_2)$   
 $= (x_3 \Rightarrow \neg x_2)$

Boolesche Quantifikation über eine Folge von Variablen kann direkt oder auch induktiv definiert werden:

$$\begin{aligned} \exists (x_1, \dots, x_n). p &= \bigvee_{(x_1, \dots, x_n) \in \{\text{tt}, \text{ff}\}^n} p \\ &= \exists x_1. \exists (x_2, \dots, x_n). p \\ &= \exists x_1. \exists x_2. \exists (x_3, \dots, x_n). p \\ &= \dots \\ &= \exists x_1. \exists x_2 \dots \exists x_n. p \end{aligned}$$



# Darstellung von CTL-Operatoren

Sei eine Relation  $R$  durch den Booleschen Term  $r(x_1, \dots, x_k, x_1', \dots, x_k')$  gegeben, und eine Eigenschaft  $P$  durch  $p(x_1, \dots, x_k)$  also

so gilt

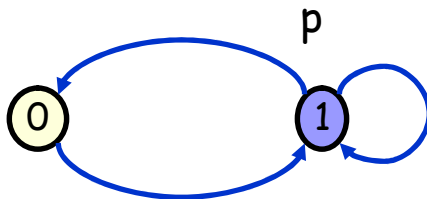
$$(\text{EX } P)(x_1, \dots, x_n) = \exists (x_1', \dots, x_n'). r(x_1, \dots, x_k, x_1', \dots, x_k') \wedge p(x_1', \dots, x_k')$$

also

$$\text{EX } p = \exists \underline{x}'. (r \wedge p')$$

Beispiel:

$S = 2$ . Die Relation  $R$  sei durch die Pfeile dargestellt.



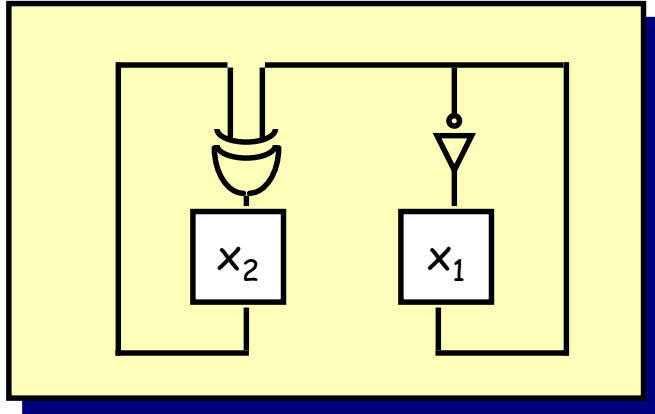
$$r = (x \vee x')$$

$$p = x$$

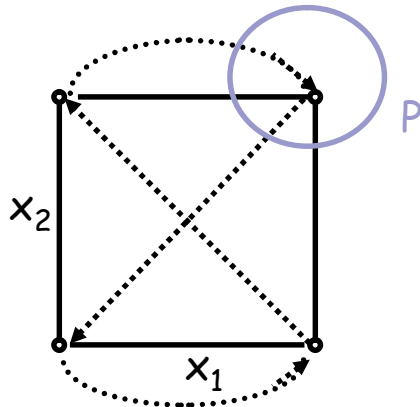
$$\begin{aligned} \text{EX } \neg p &= \exists \underline{x}'. (r \wedge (\neg p)') \\ &= \exists (x'). (x \vee x') \wedge \neg x' \\ &= \exists (x'). x \wedge \neg x' \\ &= (x \wedge \neg \text{tt}) \vee (x \wedge \neg \text{ff}) \\ &= x. \end{aligned}$$



# Beispiel: Ein 2-Bit Zähler



$S = 2^2$ . Die Relation  $R$  ist durch die gestrichelten Pfeile dargestellt.



$$r = (x_1' \Leftrightarrow \neg x_1) \wedge (x_2' \Leftrightarrow x_1 \oplus x_2),$$
$$p = x_1 \wedge x_2$$

$$\begin{aligned} \text{EX } p &= \exists \underline{x}' . (r \wedge p') \\ &= \exists (x_1', x_2') . ((x_1' \Leftrightarrow \neg x_1) \wedge (x_2' \Leftrightarrow x_1 \oplus x_2) \wedge (x_1 \wedge x_2)') \\ &= \exists (x_1', x_2') . (x_1' \Leftrightarrow \neg x_1) \wedge (x_2' \Leftrightarrow x_1 \oplus x_2) \wedge x_1' \wedge x_2' \\ &= \exists (x_1', x_2') . \neg x_1 \wedge (x_1 \oplus x_2) \wedge x_1' \wedge x_2' \\ &= \exists (x_1') . (\neg x_1 \wedge (x_1 \oplus x_2) \wedge x_1' \wedge \mathbf{tt}) \vee \\ &\quad (\neg x_1 \wedge (x_1 \oplus x_2) \wedge x_1' \wedge \mathbf{ff}) \\ &= \exists (x_1') . \neg x_1 \wedge (x_1 \oplus x_2) \wedge x_1' \\ &= (\neg x_1 \wedge (x_1 \oplus x_2) \wedge \mathbf{tt}) \vee (\neg x_1 \wedge (x_1 \oplus x_2) \wedge \mathbf{ff}) \\ &= \neg x_1 \wedge x_2 \end{aligned}$$

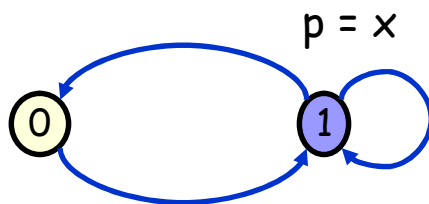


# Berechnung von CTL-Formeln

Beliebige CTL-Operatoren lassen sich wie bisher als Fixpunkte berechnen.  
So gilt z.B. wegen  $EF p = \mu y. p \vee EX y$ , dass

$$EF p = \bigvee_n. \tau^n(ff)$$

Beispiel 1:

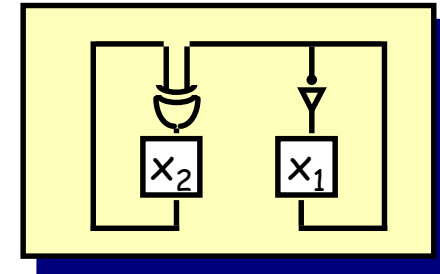


Berechne  $EF x$ :

$$\begin{aligned} \tau^1(ff) &= x \vee EX(ff) \\ &= x. \\ \tau^2(ff) &= x \vee EX x \\ &= x \vee \exists (x'). (x \vee x') \wedge x' \\ &= \mathbf{tt}. \\ \tau^3(ff) &= x \vee \exists (x'). (x \vee x') \wedge \mathbf{tt}' \\ &= \mathbf{tt}. \end{aligned}$$

Also gilt im Beispiel :  
 $EF x = \mathbf{tt}$ .

Beispiel 2:



Berechne  $EF x_1 \wedge x_2$ :

$$\begin{aligned} \tau^1(ff) &= (x_1 \wedge x_2) \vee EX(ff) \\ &= x_1 \wedge x_2. \\ \tau^2(ff) &= (x_1 \wedge x_2) \vee EX (x_1 \wedge x_2) \\ &= x_2. \\ \tau^3(ff) &= (x_1 \wedge x_2) \vee EX x_2 \\ &= x_1 \vee x_2 \\ \tau^4(ff) &= (x_1 \wedge x_2) \vee EX x_1 \vee x_2 \\ &= \mathbf{tt} \end{aligned}$$

Also gilt im Beispiel :  
 $EF (x_1 \wedge x_2) = \mathbf{tt}$ .



# Algorithmus

Sei  $p$  nun ein beliebiger CTL-Ausdruck. Zur Berechnung von  $p$  dient dann die folgende Funktion **eval** mit Hilfsfunktionen **evalEX**, **evalEG** und **evalEU** :

```
function eval(p)
case
  p ∈ AP      : p
  p = ¬q      : ¬eval(q)
  p = p ∨ q    : eval(p) ∨ eval(q)
  p = EXq     : evalEX(eval(q))
  p = E[p U q] : evalEU(eval(p), eval(q))
  p = EG p    : evalEG(eval(p))
end_case
```

```
function evalEX(p) = ∃  $\underline{x}'$ . (r ∧ p')
```

```
function evalEG(y)
  y' := y ∧ evalEX(y)
  if (y'=y) return(y)
  else return evalEG(y')
```

```
function evalEU(p,y)
  y' := y ∨ (p ∧ evalEX(y))
  if (y'=y) return(y)
  else return evalEU(p,q,y')
```

Beachte: Die Boolesche Repräsentation  $r$  der Transitionsrelation  $R$  ist (via **evalEX**) implizit in allen obigen Funktionen, genaugenommen müsste sie als weiteres Argument übergeben werden.



# Datenstrukturen und Operationen

Wie immer wir Boolesche Terme symbolisch repräsentieren, wir müssen folgende Operationen möglichst effizient implementieren:

```
function evalEX(p) =  $\exists \underline{x}' . (r \wedge p')$ 
```

```
function evalEG(y)
```

```
   $y' := y \wedge \text{evalEX}(y)$ 
```

```
  if  $(y' = y)$  return(y)
```

```
  else return evalEG(  $y'$  )
```

$( p(x), r(x, x') ) \mapsto \exists x' . (r \wedge p')$

$(p, q) \mapsto p \wedge q$

$(p, q) \mapsto p = q$



# Repräsentation von Daten

- Angenommen, wir arbeiten mit Daten im Bereich 0 .. 15 :



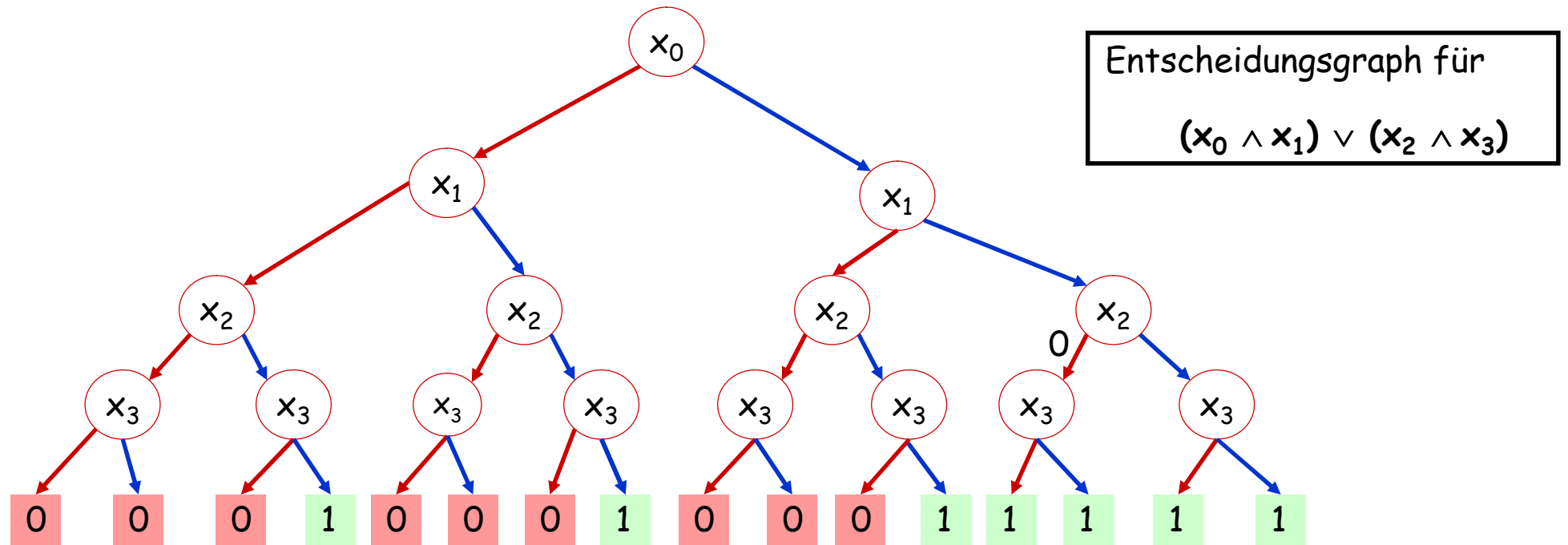
- Die gezeigte Menge kann man repräsentieren durch

- Aufzählung
  - { 0, 2, 4, 6, 8, 10, 11, 12, 14, 15 }
- logische Formel
  - { n | n mod 2 = 0  $\vee$  n = 11  $\vee$  n = 15 }
- Bit-Array
  - [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1 ]
- explizite Angabe der Bitfolgen
  - { 0000, 0010, 0100, 0110, 1000, 1010, 1011, 1100, 1110, 1111 }
- die charakteristische Funktion
  - $\chi_M(n) = \text{if } n \bmod 2 = 0 \vee n = 11 \vee n = 15 \text{ then } 1 \text{ else } 0$
- die charakteristische Funktion der Bitpositionen
  - $$\chi(x_3, x_2, x_1, x_0) = \neg x_3 \neg x_2 \neg x_1 \neg x_0 + \neg x_3 \neg x_2 x_1 \neg x_0 + \neg x_3 x_2 \neg x_1 \neg x_0 + \neg x_3 x_2 x_1 \neg x_0 + x_3 \neg x_2 \neg x_1 \neg x_0$$
$$+ x_3 \neg x_2 x_1 \neg x_0 + x_3 \neg x_2 x_1 x_0 + x_3 x_2 \neg x_1 \neg x_0 + x_3 x_2 x_1 \neg x_0 + x_3 x_2 x_1 x_0$$



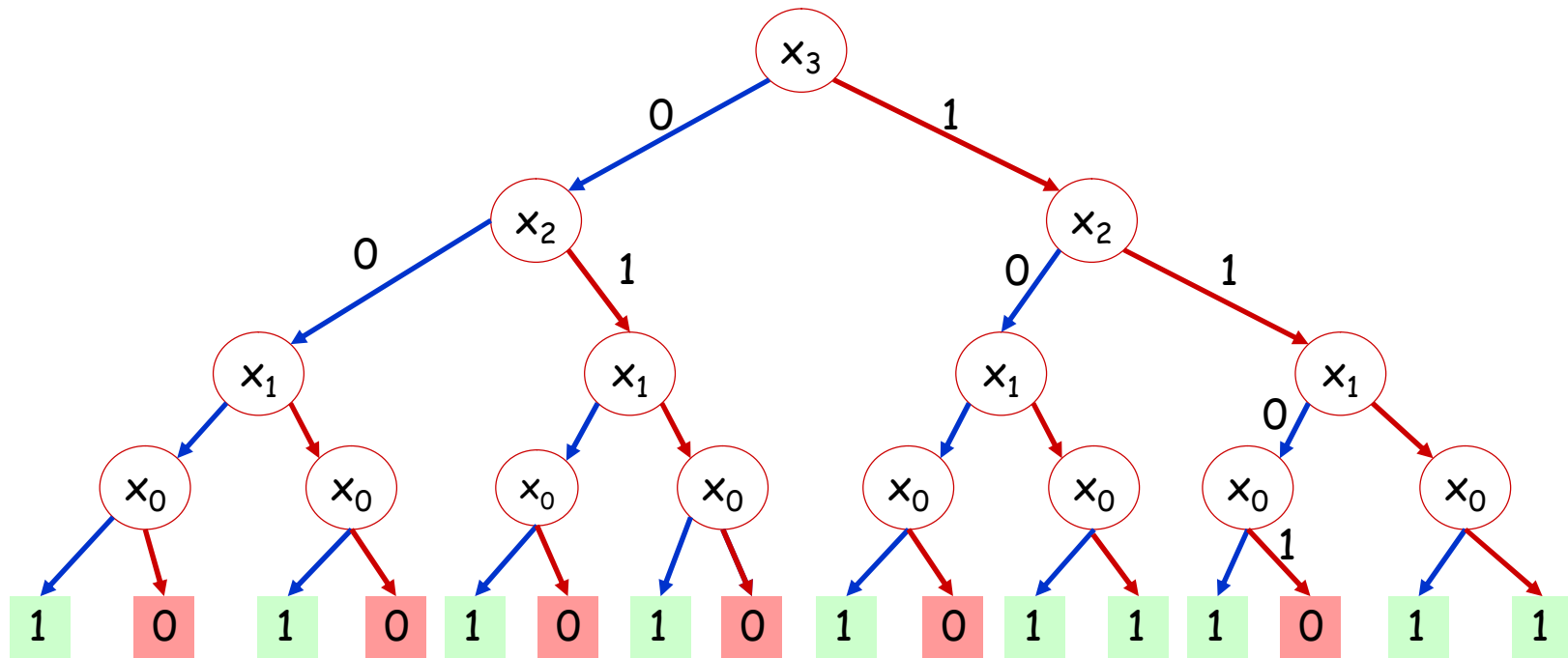
# Darstellung boolescher Terme

- ... durch einen Entscheidungsbaum





# Jeder Entscheidungsbaum ...



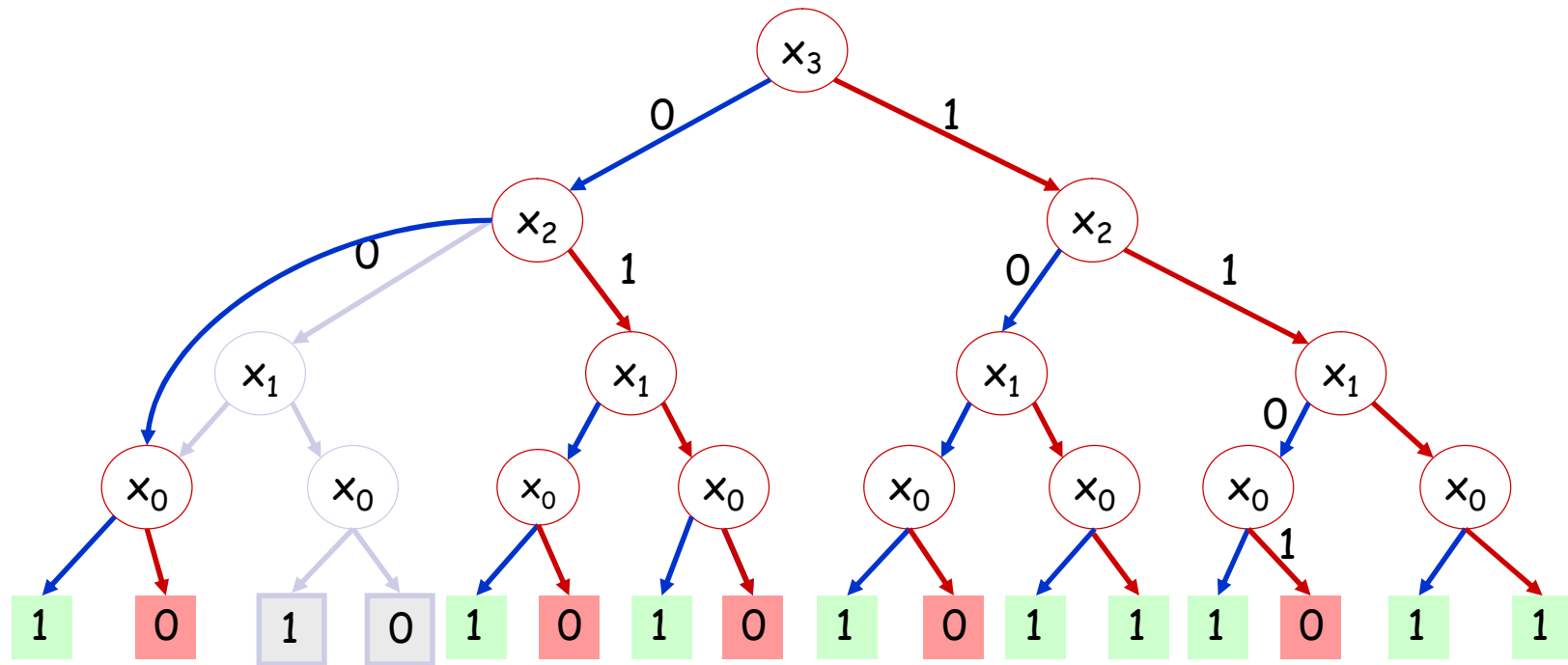
... repräsentiert Boolesche Funktion  $f : 2^4 \rightarrow 2$ :

0000, 0010, 0100, 0110, 1000, 1010, 1011, 1100, 1110, 1111  $\mapsto 1$

0001, 0011, 0101, 0111, 1001, 1101  $\mapsto 0$



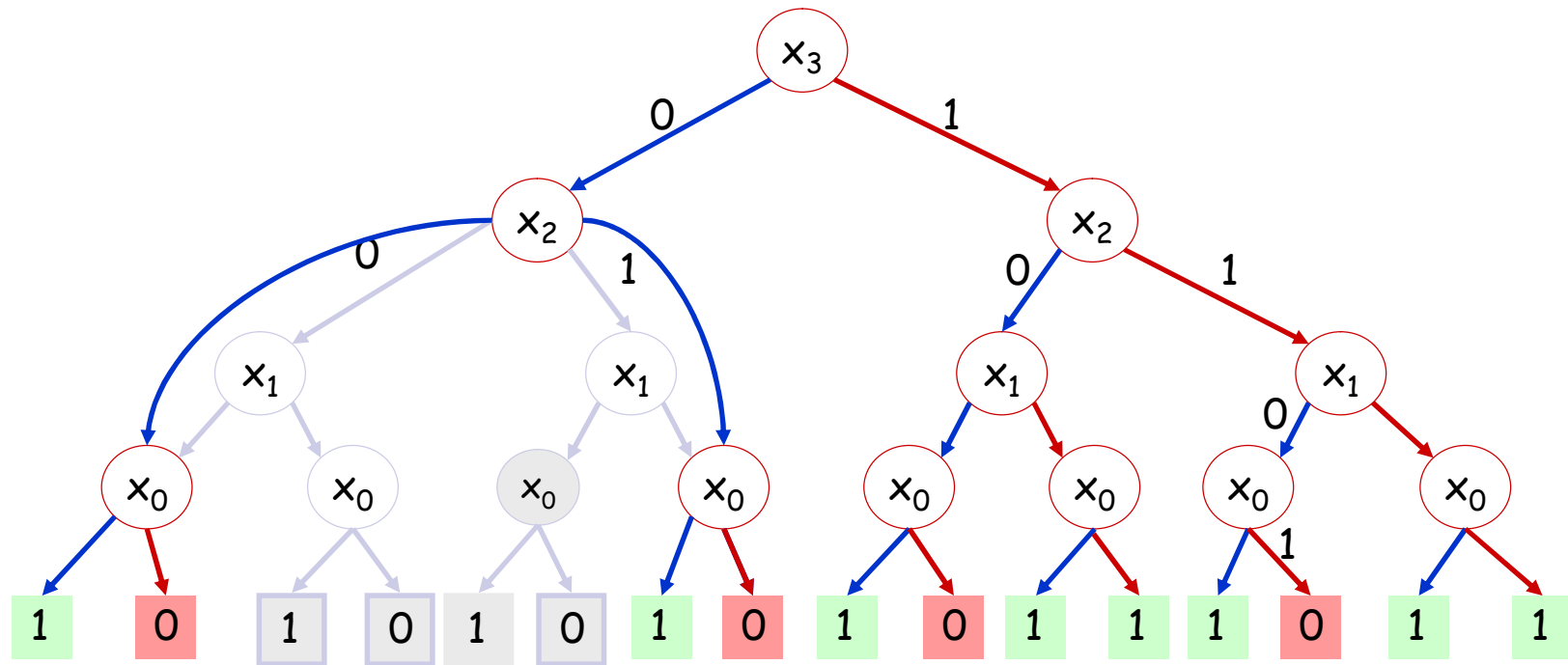
# Entscheidungsbaum



0000, 0010, 0100, 0110, 1000, 1010, 1011, 1100, 1110, 1111



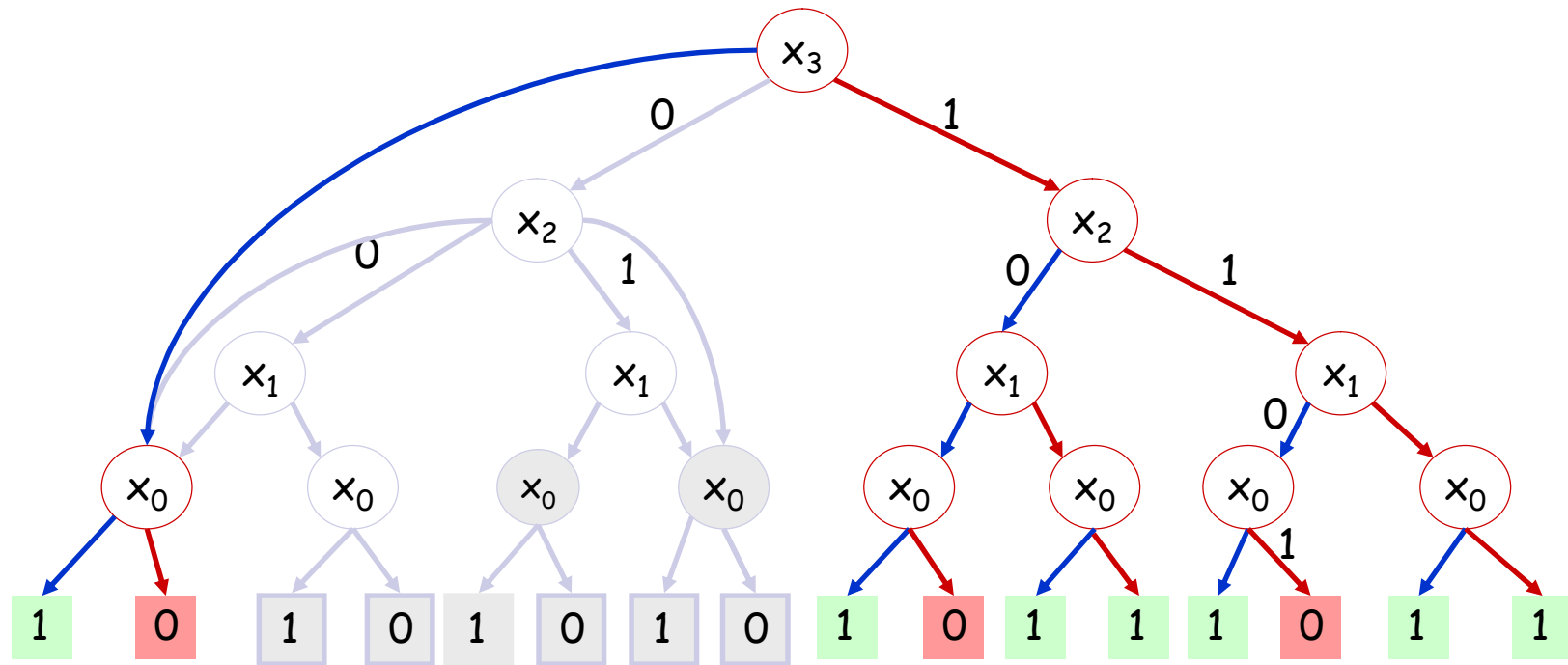
# Entscheidungsbaum



0000, 0010, 0100, 0110, 1000, 1010, 1011, 1100, 1110, 1111



# Entscheidungsbaum

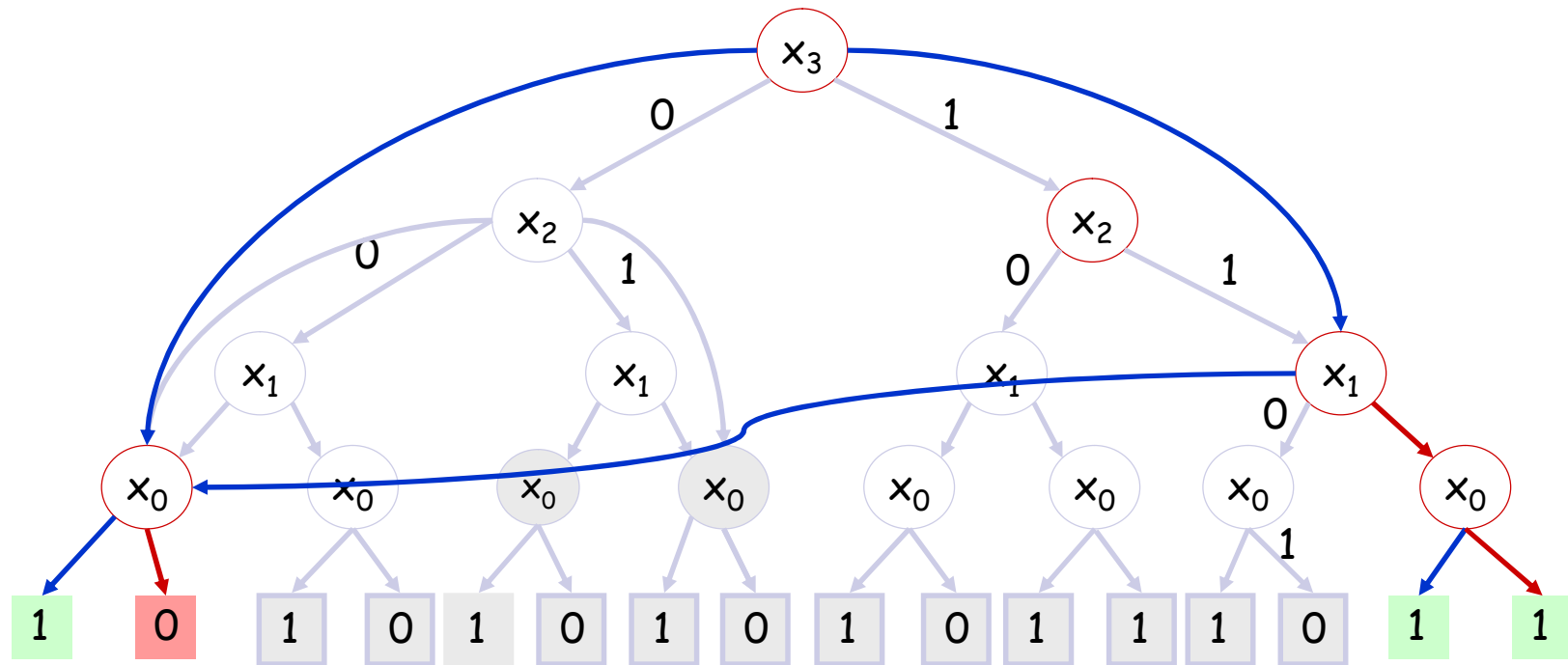


0000, 0010, 0100, 0110, 1000, 1010, 1011, 1100, 1110, 1111





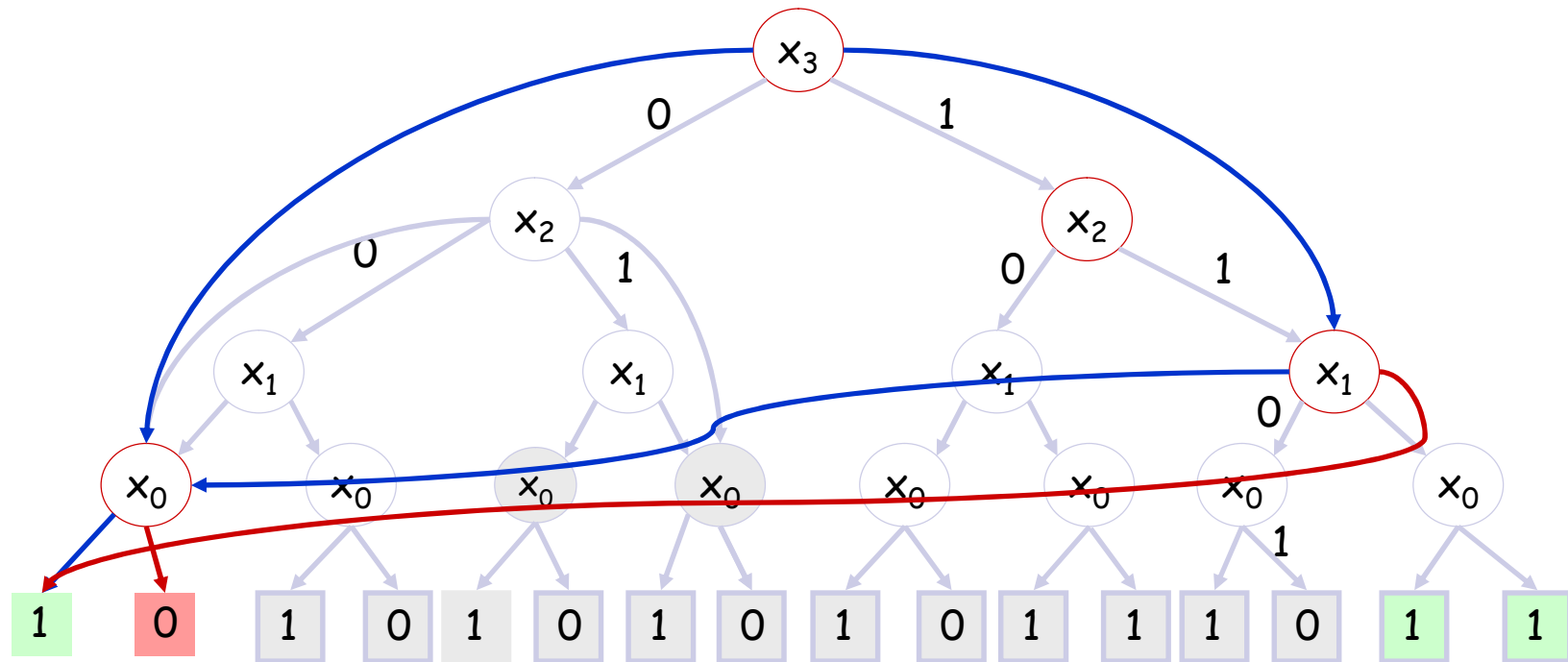
# Entscheidungsbaum



0000, 0010, 0100, 0110, 1000, 1010, 1011, 1100, 1110, 1111



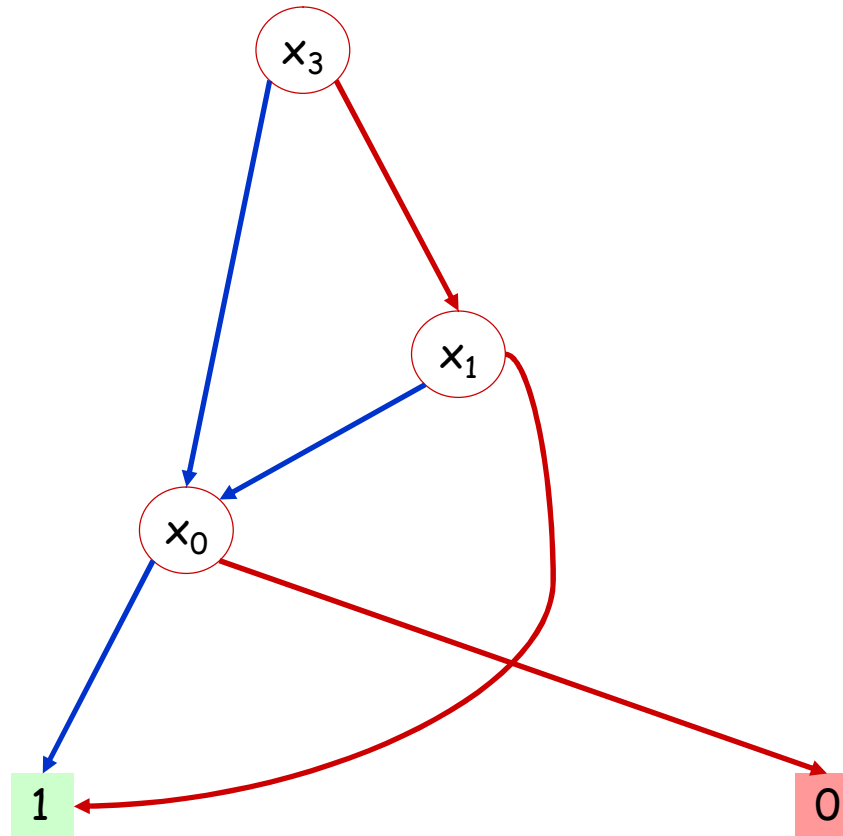
# Entscheidungsbaum



0000, 0010, 0100, 0110, 1000, 1010, 1011, 1100, 1110, 1111



# Entscheidungsgraph

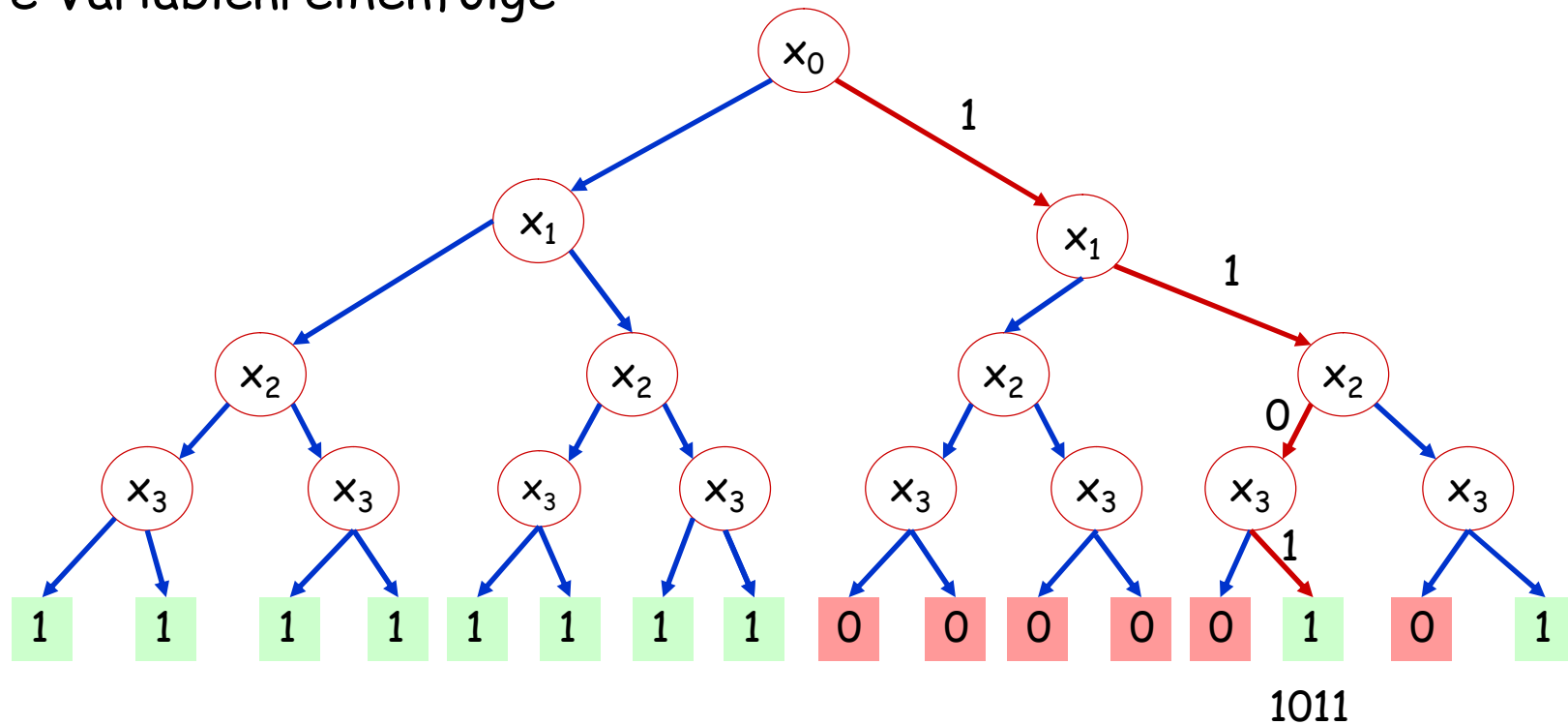


0000, 0010, 0100, 0110, 1000, 1010, 1011, 1100, 1110, 1111



# Alternativer Entscheidungsbaum

andere Variablenreihenfolge



0000, 0010, 0100, 0110, 1000, 1010, 1011, 1100, 1110, 1111



# Entscheidungsgraph

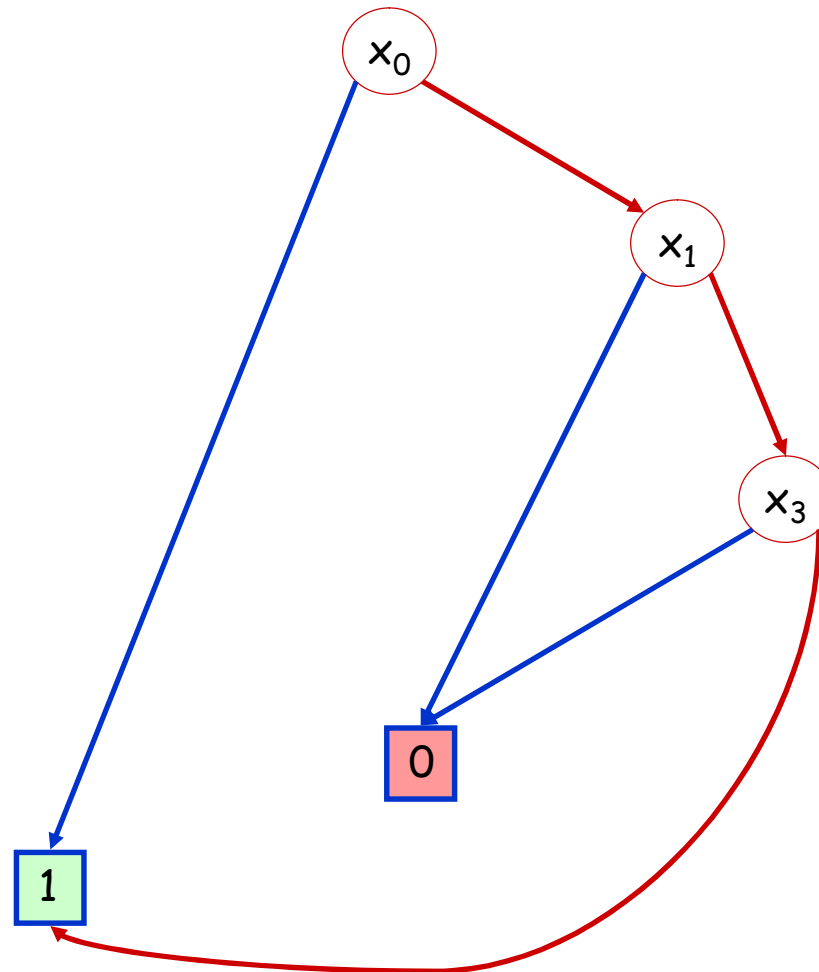
Logische Darstellung:

$$(x_0 ? (x_1 ? x_3 : 0), 1)$$

wobei

$$(x ? y : z) := (\neg x \vee y) \wedge (x \vee z)$$

= if  $x$  then  $y$  else  $z$

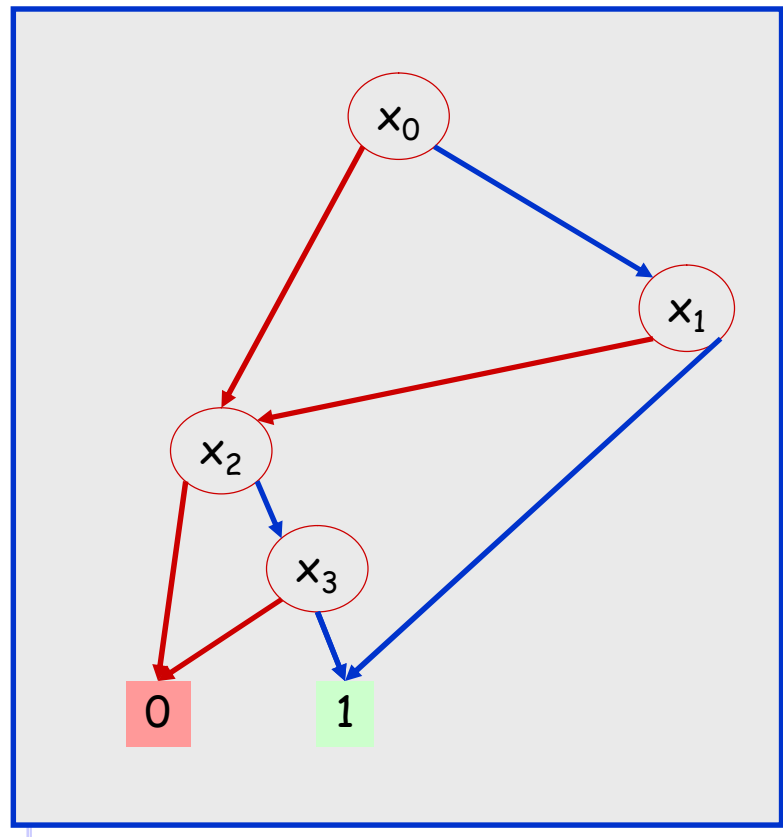


0000, 0010, 0100, 0110, 1000, 1010, 1011, 1100, 1110, 1111



# Darstellung boolescher Terme

- ... durch ein Entscheidungsdiagramm
- ... binary decision diagram (BDD)
- immer ein azyklischer Graph
  - DAG (directed acyclic graph)



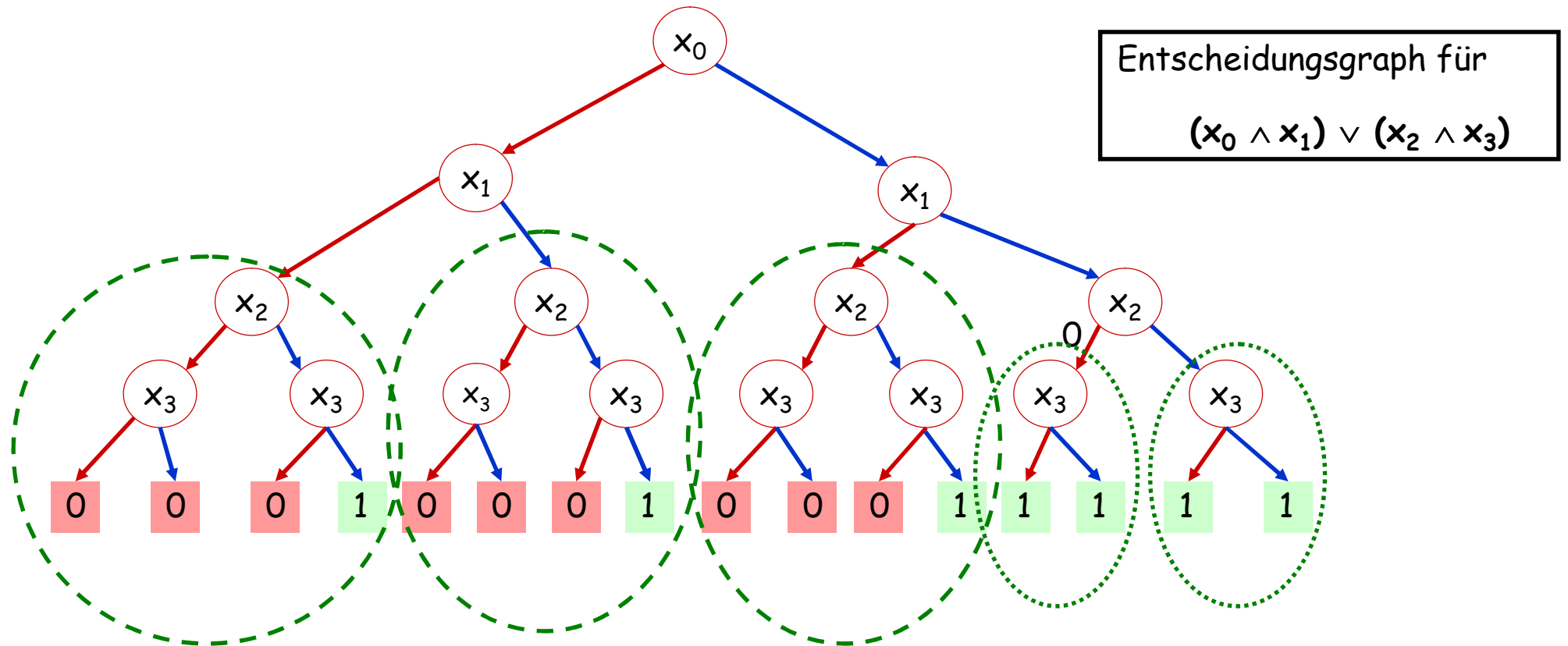
Entscheidungsdiagramm für

$$(x_0 \wedge x_1) \vee (x_2 \wedge x_3)$$



# Entscheidungsbaum $\rightarrow$ BDD

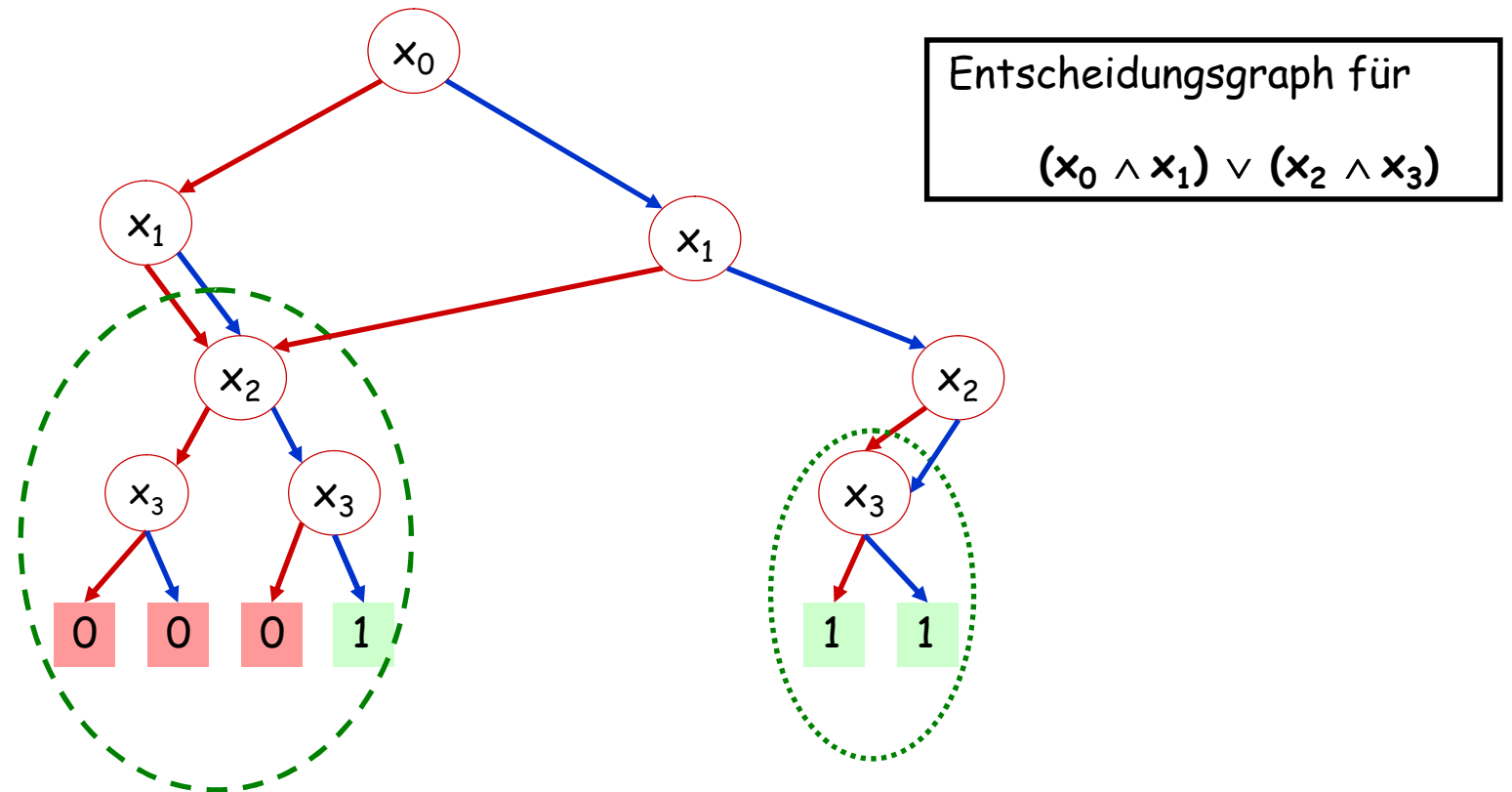
- ... Verschmelzen gleicher Teilbäume





# Entscheidungsbaum $\rightarrow$ BDD

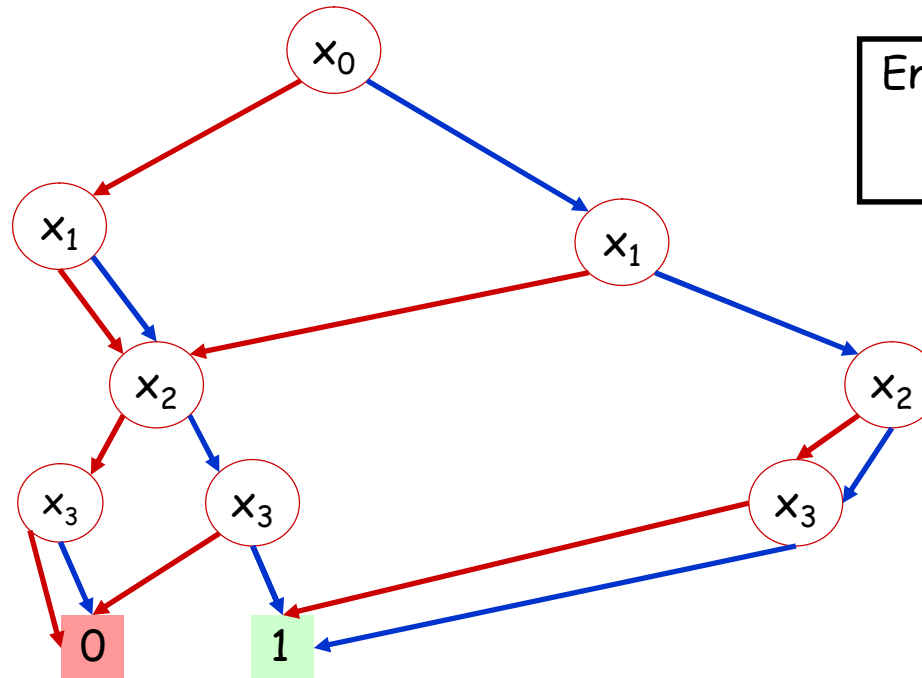
- ... Verschmelzen gleicher Teilbäume





# Entscheidungsbaum $\rightarrow$ BDD

- ... Verschmelzen gleicher Teilbäume



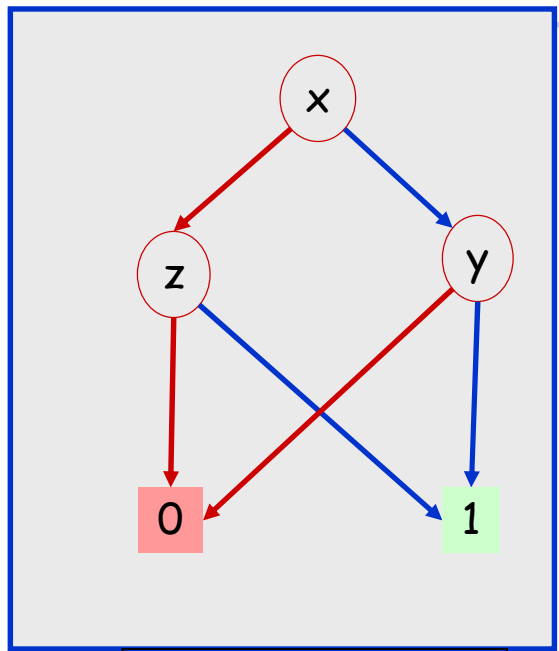
Entscheidungsgraph für  
 $(x_0 \wedge x_1) \vee (x_2 \wedge x_3)$



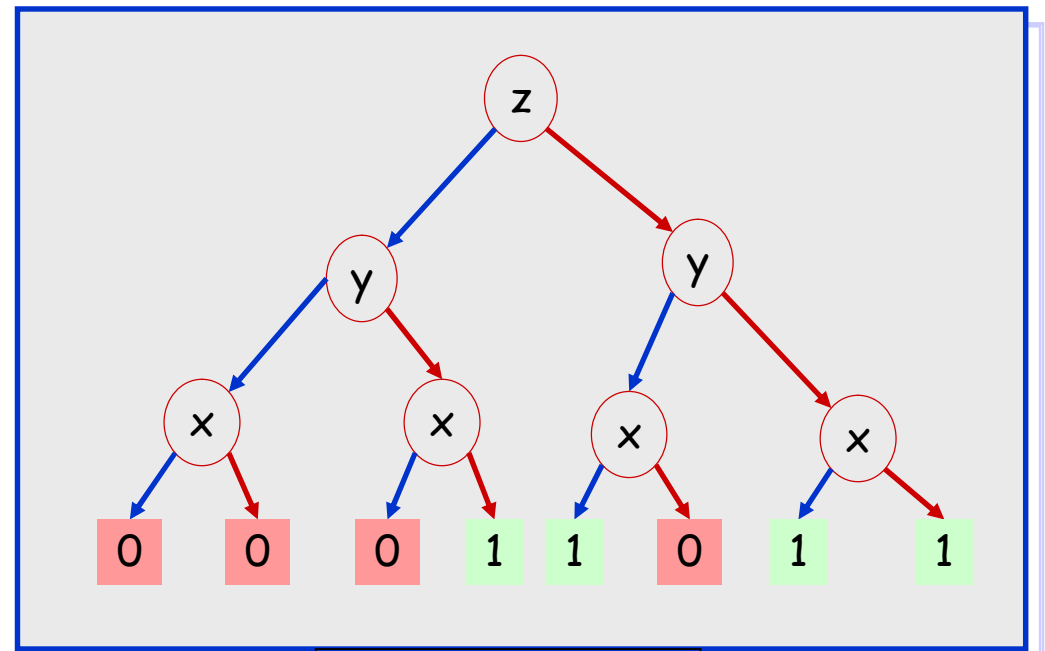
# Variablenordnung

Sei  $p : [2^k \rightarrow 2]$  eine  $k$ -stellige Boolesche Funktion,  $p = \lambda x_1, \dots, x_k. p$ . Die Größe der BDD-Darstellung von  $p$  kann sehr stark von der Variablenordnung abhängen.

Beispiel :  $p(x,y,z) = \text{if } x \text{ then } y \text{ else } z$



Variablenordnung  
 $x > y > z$



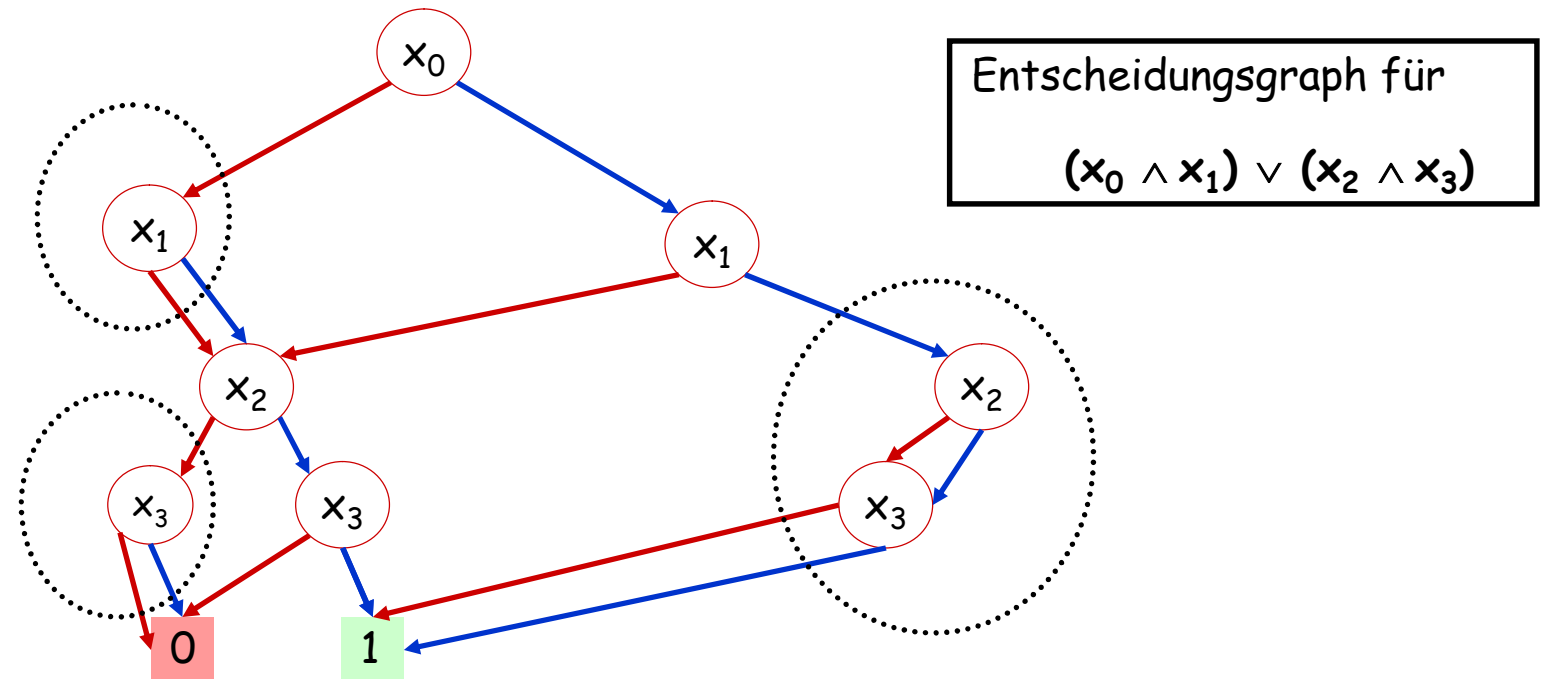
Variablenordnung  
 $z > y > x$

Legt man für alle BDDs eine gemeinsame Variablen-Ordnung fest, so spricht man von **OBDDs**



# Entscheidungsbaum $\rightarrow$ OBDD

- Ist die Reihenfolge der Knoten von der Wurzel zu den Blättern festgelegt, so können indifferente Knoten entfernt werden.
  - Hier: indifferent = linker und rechter Teilbaum identisch

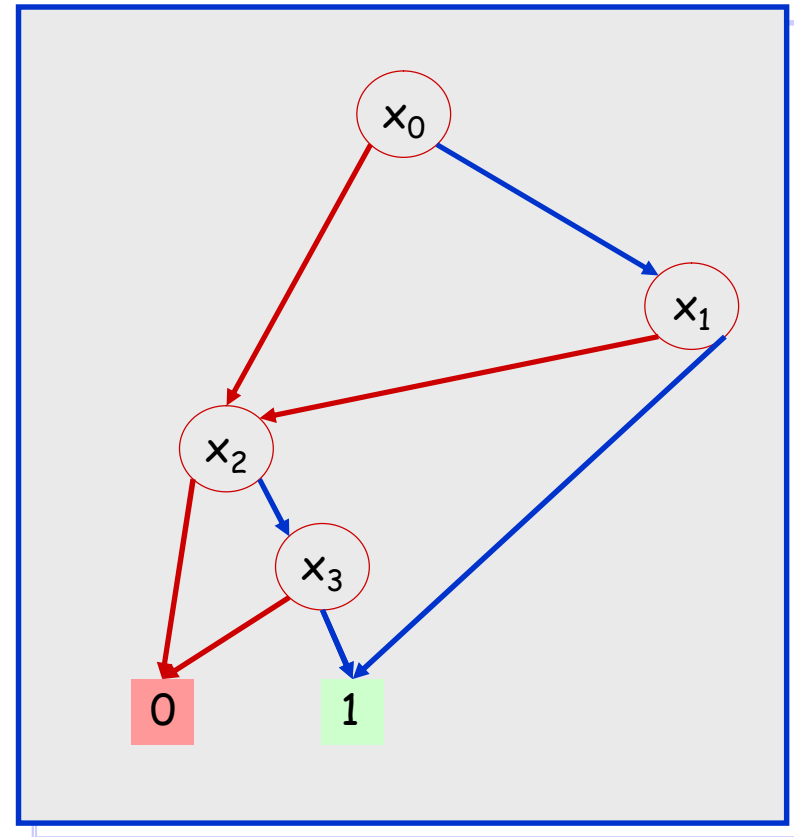




# Entscheidungsbaum $\rightarrow$ OBDD

- Entfernen indifferenter Knoten
  - Knoten mit linkem Teilbaum = rechtem Teilbaum
- Resultat: Geordnetes BDD :

OBDD

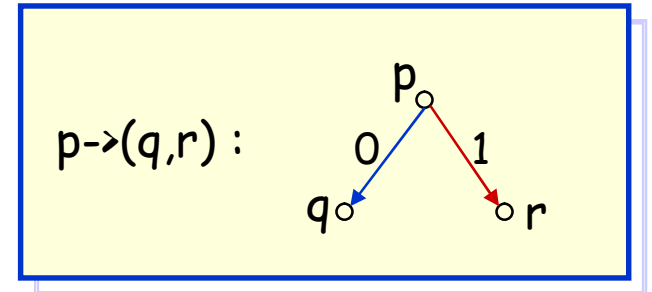


Entscheidungsgraph für

$$(x_0 \wedge x_1) \vee (x_2 \wedge x_3)$$



# OBDDs als Funktionen



Wir benutzen im Folgenden die Notation " $p \rightarrow (q, r)$ " für den Booleschen Term  $(p \wedge r) \vee (\neg p \wedge q)$ .

In Programmiersprachennotation entspricht dies gerade " $p ? r : q$ "

Für Bitvektoren  $\underline{v} = (v_1, v_2, \dots, v_k) \in 2^k$  sei  $\underline{v}[i]$  die  $i$ -te Komponente, also  $\underline{v}[i] = v_i$ .

Jedes OBDD  $n$  repräsentiert eine Funktion  $f_n : 2^k \rightarrow 2$ . Wenn  $\underline{v} = (v_1, v_2, \dots, v_k)$  ein Bitvektor ist, dann sei

$$\begin{aligned} f_0 &= \lambda \underline{v}. \mathbf{ff}, \\ f_1 &= \lambda \underline{v}. \mathbf{tt} \\ f_n &= \lambda \underline{v}. v_i \rightarrow (f_l \underline{v}, f_r \underline{v}), \text{ falls } n = \text{node}(x_i, l, r). \end{aligned}$$

In einer nichtfunktionalen Sprache, (z.B. Prolog) würden wir die Applikation eines OBDDs so programmieren. Wir schreiben dafür kurz :

$$f_{\text{node}(x_i, l, r)} = x_i \rightarrow (f_l, f_r).$$



# OBDD ist Normalform

Satz (Bryant) : Seien  $m, n$  OBDDs. Wenn  $f_m = f_n$ , dann folgt  $m=n$ .

Zu jeder Booleschen Funktion gibt es also einen eindeutigen OBDD. Ob zwei Boolesche Funktionen gleich sind, kann man testen, indem man die entsprechenden OBDDs konstruiert und überprüft, ob diese gleich sind.

Lemma: Sei die Variablenordnung  $x_1 > x_2 > \dots > x_k$  gegeben und  $n = \text{node}(x_i, l, r)$  ein OBDD. Dann hängt  $f_n$  nicht von den Variablen vor  $x_i$ , also von  $x_1, \dots, x_{i-1}$  ab.

Beweis des Satzes : Wir nehmen an, daß der Satz richtig ist für alle  $m', n'$  mit  $m' < m$  und  $n' < n$ . Wir zeigen, daß unter dieser Voraussetzung der Satz auch für  $m, n$  richtig ist .

1. Fall :  $m, n \in \{0, 1\}$ . Dieser Fall ist trivial, da aus  $f_m = f_n$  sofort folgt  $m=n$ .

2. Fall :  $m \in \{0, 1\}, n = \text{node}(x_i, l_i, r_i) \notin \{0, 1\}$ .  $f_n$  hängt nicht von  $x_i$  ab, also  $f_{l_i} = f_{r_i}$ , also  $l_i = r_i$  nach Ind.vor., also  $n$  kein OBDD.

3. Fall :  $m, n \notin \{0, 1\}$ . Dann gilt  $m = \text{node}(x_{i_1}, l_1, r_1)$  und  $n = \text{node}(x_{i_2}, l_2, r_2)$ .

Wenn  $x_{i_1} = x_{i_2}$ , dann folgt  $f_{l_1} = f_{l_2}$  und  $f_{r_1} = f_{r_2}$ . Nach Ind. Voraussetzung folgt  $l_1 = l_2$  und  $r_1 = r_2$ , also  $m=n$ .

Wenn  $x_{i_1} \neq x_{i_2}$ , dann gilt o.B.d.A.  $x_{i_1} > x_{i_2}$ . Nach dem Lemma hängt dann aber  $f_n$  nicht von  $x_{i_1}$  ab, folglich kann auch  $f_m$  nicht von  $x_{i_1}$  abhängen. Es folgt also  $f_{l_1} = f_{r_1}$  und, wieder nach Ind.Voraussetzung  $l_1 = r_1$ . Dann war aber  $m$  kein OBDD !



# Operationen auf OBDDs

Wenn wir Boolesche Terme  $p$  und  $q$  durch OBDDs  $\text{obdd}(p)$  und  $\text{obdd}(q)$  repräsentiert haben, so stellt sich die Frage, wie wir aus  $\text{obdd}(p)$ ,  $\text{obdd}(q)$  die Booleschen Kombinationen  $\text{obdd}(p \vee q)$ ,  $\text{obdd}(p \wedge q)$  und  $\text{obdd}(\neg p)$  berechnen.

Allgemeiner zeigen wir, wie eine beliebige Operation  $g : [2^k \rightarrow 2]$  auf OBDDs übertragen werden kann. Wir definieren eine Funktion **apply**, die dies leistet.

$$\text{obdd}(g(p,q)) = \text{apply}(g, \text{obdd}(p), \text{obdd}(q))$$

Wir benutzen dazu die folgende Beobachtung, die für beliebige einstellige Funktionen  $f : [2 \rightarrow 2]$ , zweistellige  $g : [2^2 \rightarrow 2]$ , (analog auch für beliebige  $k$ -stellige Operationen) gilt :

$$f(x \rightarrow (p,q)) = x \rightarrow (f(p), f(q))$$

$$g(x \rightarrow (p_1, q_1), x \rightarrow (p_2, q_2)) = x \rightarrow (g(p_1, q_2), g(q_1, q_2))$$

Wenn  $g$  nicht von  $x$  abhängt, gilt :

$$g(x \rightarrow (p, q)) = g(p) = g(q)$$



# Apply

Seien  $p_1, p_2$  durch OBDDs  $p_1$  bzw.  $p_2$  gegeben und sei  $g : [2^2 \rightarrow 2]$ . Wir liften  $g$  nun zu einer Operation auf OBDDs :

```
apply(g,p1,p2) = if p1, p2 ∈ { 0,1 } then g(p1,p2)
                 else if p1 > p2 then
                   let node(v,l1,r1) = p1,
                   in mk_obdd (v,apply(g,l1,p2),apply(g,r1,p2))

                 else if p1 < p2 then
                   ... symmetrisch ...

                 else                                     % gleiche Variable v !
                   let p1=node(v,l1,r1)
                   p2 = node(v,l2,r2)
                   in mk_obdd(v,apply(g,l1,l2),apply(g,r1,r2))
```

Hier gilt:  $p_1 < p_2 \Leftrightarrow p_1 = \text{node}(v_1, l_1, r_1) \wedge p_2 = \text{node}(v_2, l_2, r_2) \wedge v_1 < v_2$

mk\_obdd ist eine Funktion, die aus einer Variablen und zwei OBDDs ein neues OBDD konstruiert :

```
mk_obdd(v,l,r) = if l = r then l
                 else node(v,l,r)
```



# AndExists

$$EX p = \exists \underline{x}'.(r \wedge p')$$

$p_1, p_2$  seien OBDDs. Kann man Ausdrücke der Form  $\exists \underline{v}.(p_1 \wedge p_2)$  effizienter berechnen ?

Ja: Ziehe Quantifikation nach innen: "early quantification".

Für eine einzelne Variable  $x$  gilt nämlich :

$$\exists x. \text{node}(y, l, r) = \text{if } x = y \text{ then } l \vee r \\ \text{else } \text{mk\_obdd}(\text{node}(y, \exists x.l, \exists x.r))$$

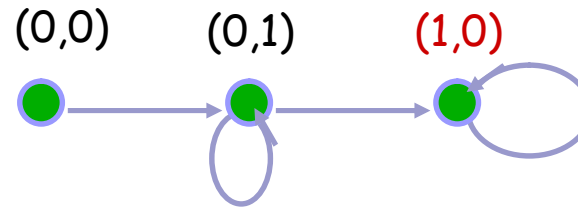
Für einen Vektor  $\underline{x}$  von Variablen soll  $\text{andEx}(\underline{x}, p_1, p_2) = \text{obdd}(\exists \underline{x}.(p_1 \wedge p_2))$  gelten:

```
andEx( $\underline{x}$ ,  $p_1$ ,  $p_2$ ) = if  $p_1, p_2 \in \{0,1\}$  then  $p_1 \wedge p_2$ 
                        else if  $p_1 > p_2$  then
                            let  $\text{node}(y, l, r) = p_1$ 
                                in if  $y \in \underline{x}$  then andEx( $\underline{x}, l, p_2$ )  $\vee$  andEx( $\underline{x}, r, p_2$ )
                                    else  $\text{mk\_obdd}(y, \text{andEx}(\underline{x}, l, p_2), \text{andEx}(\underline{x}, r, p_2))$ 
                        else if  $p_1 < p_2$  then
                            ... symmetrisch ...
                        else
                            % gleiche Variable  $x$  !
                            let  $\text{node}(y, l_1, r_1) = p_1$ 
                                 $\text{node}(y, l_2, r_2) = p_2$ 
                                in if  $y \in \underline{x}$  then andEx( $\underline{x}, l_1, l_2$ )  $\vee$  andEx( $\underline{x}, r_1, r_2$ )
                                    else  $\text{mk\_obdd}(\text{node}(y, \text{andEx}(\underline{x}, l_1, l_2), \text{andEx}(\underline{x}, r_1, r_2)))$ .
```



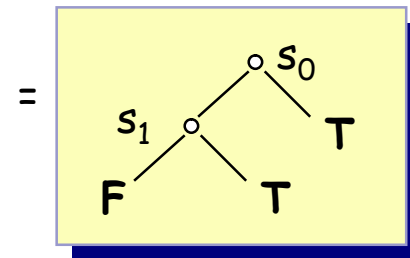
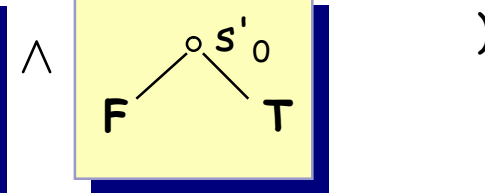
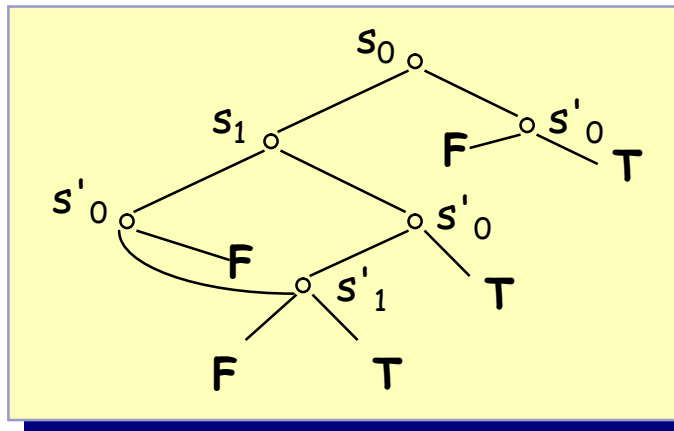
# Beispiel

- $S = \{0,1,2\} = \{(0,0), (0,1), (1,0)\}$



$R^{-1}(s=(1,0)) = \exists s' . R(s,s') \wedge s'=(1,0)$  führt zu den OBDDs (siehe späteres Kapitel)

$= \exists (s'_0, s'_1) . ($



$= (s = (0,1) \vee s = (1,0) \vee s = (1,1) )$



# Anwendung von AndExists

Der AndExists-Algorithmus ist nicht nur bei der Berechnung von EXp nützlich.

$f[v \leftarrow w]$  sei der OBDD, der aus  $f$  entsteht, wenn man die Variable  $v$  durch  $w$  ersetzt.  
Es gilt :

$$f[v \leftarrow w] = \exists v. (v=w \wedge f)$$

Diese Beobachtung erleichtert die Berechnung der Anwendung einer Booleschen Funktion auf einen Wert  $w$  :

Es gilt nämlich :

$$\begin{aligned} (\lambda v. f) w &= f[v \leftarrow w] \\ &= \exists v. (v = w \wedge f) \end{aligned}$$

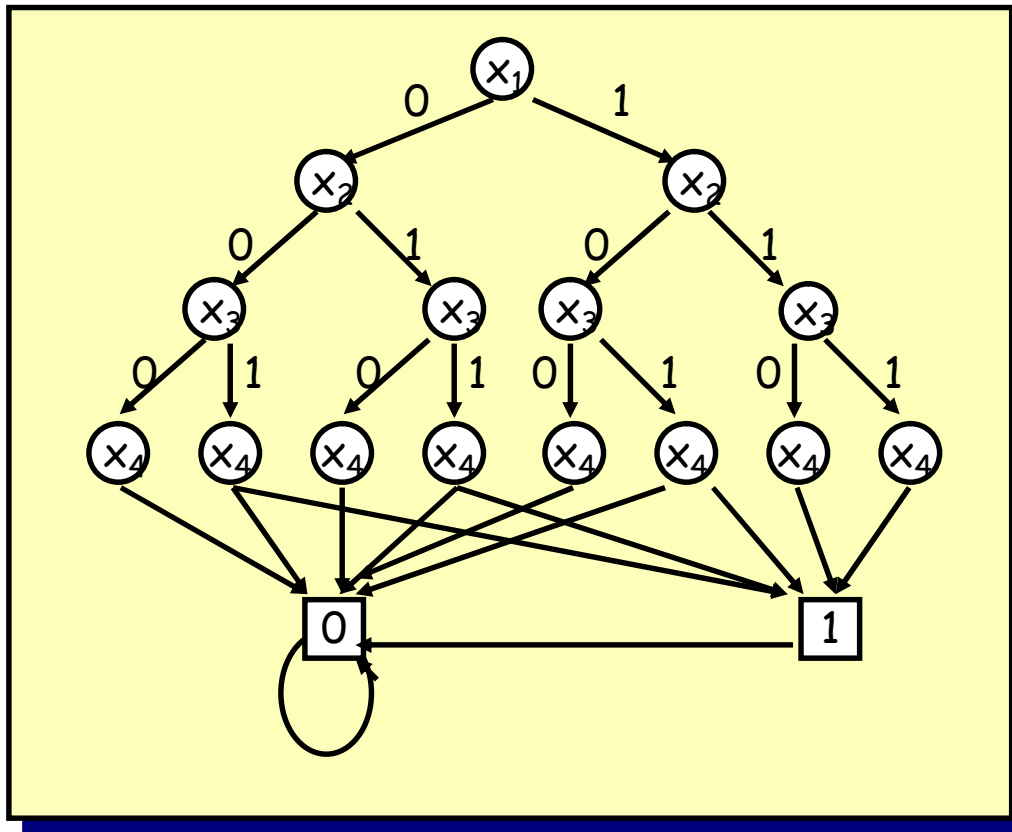
Damit kann man auch die Funktion **Compose** darstellen.  $\text{Compose}(\underline{v}, p, \underline{q})$  für einen OBDD  $p$  mit Variablen  $\underline{v} = (v_1, \dots, v_n)$  und einen Vektor von OBDDs  $\underline{q} = (q_1, \dots, q_n)$  berechnet  $p[v_1=q_1, \dots, v_n=q_n]$

$$\begin{aligned} \text{Compose}(\underline{v}, p, \underline{q}) &= p[\underline{v} \leftarrow \underline{q}] \\ &= \text{andEx}(\underline{v}, v_1=q_1 \wedge \dots \wedge v_n=q_n \wedge p). \end{aligned}$$



# BDDs und Automaten

Sei  $p(x_1, \dots, x_n)$  ein Boolescher Term, und  $L = \{ (b_1, \dots, b_n) \mid b_i \in \{0,1\}, p(b_1, \dots, b_n) = 1 \}$ , dann ist  $L$  eine endliche Menge von Worten über dem Alphabet  $\{0,1\}$ . Folglich gibt es einen Automaten, der gerade  $L$  erkennt. Den Entscheidungsbaum von  $p$  können wir als einen solchen Automaten auffassen. Dazu setzen wir  $\underline{1}$  als einzigen terminalen Knoten und setzen fehlende Pfeile auf  $\underline{0}$ .

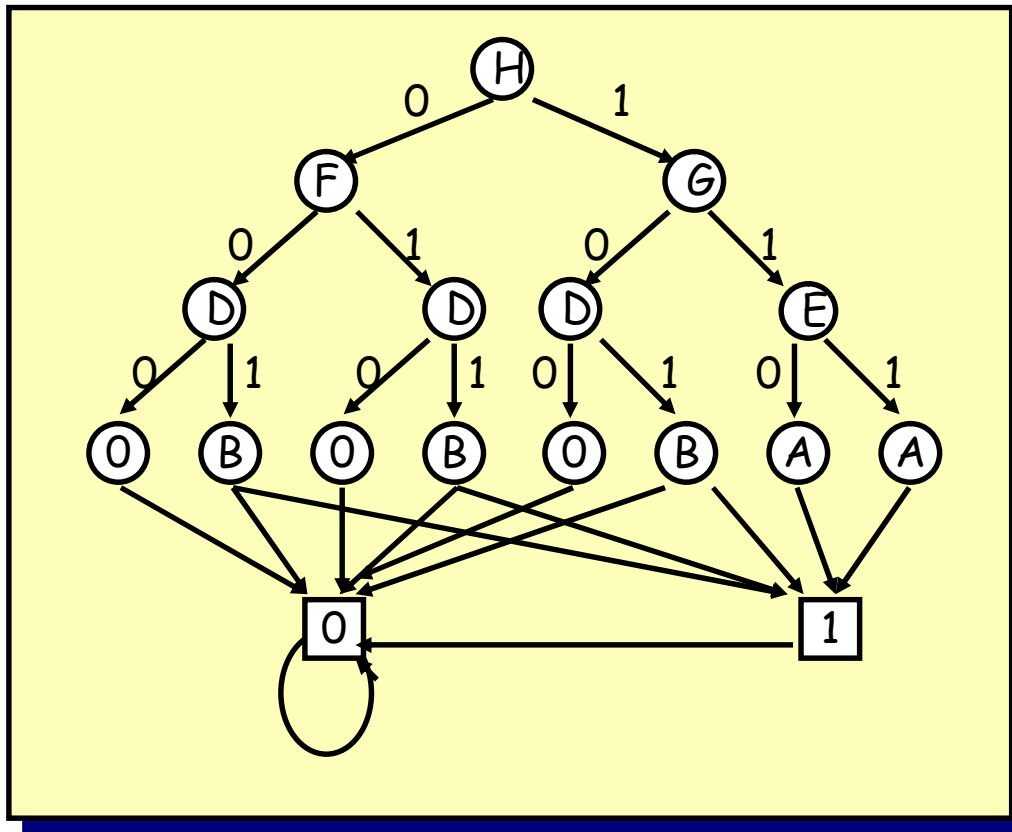


Automat für  
 $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$



# Minimierung

Das aus der Automatentheorie bekannte Minimierungsverfahren kann auch hier angewendet werden und führt zu dem minimalen Automaten, der gerade dem (minimalen) BDD entspricht :

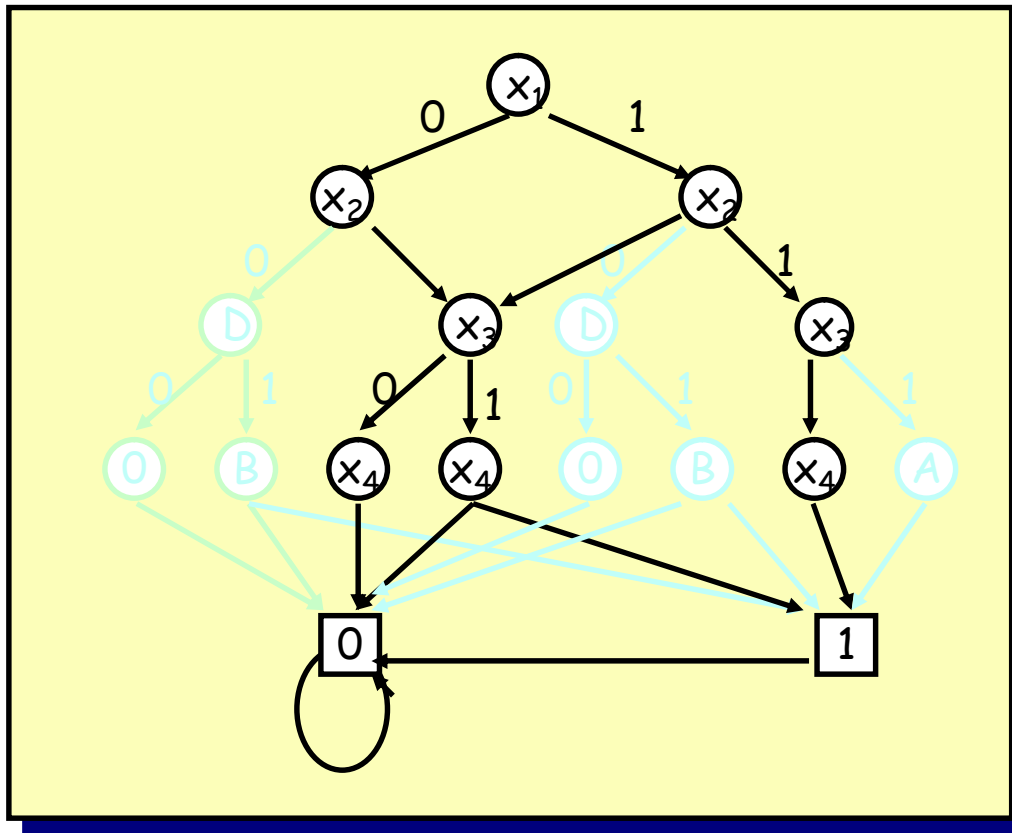


Minimaler Automat für  
 $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$

Klasseneinteilung.  
Knoten mit derselben Aufschrift  
gehören zur selben Klasse.



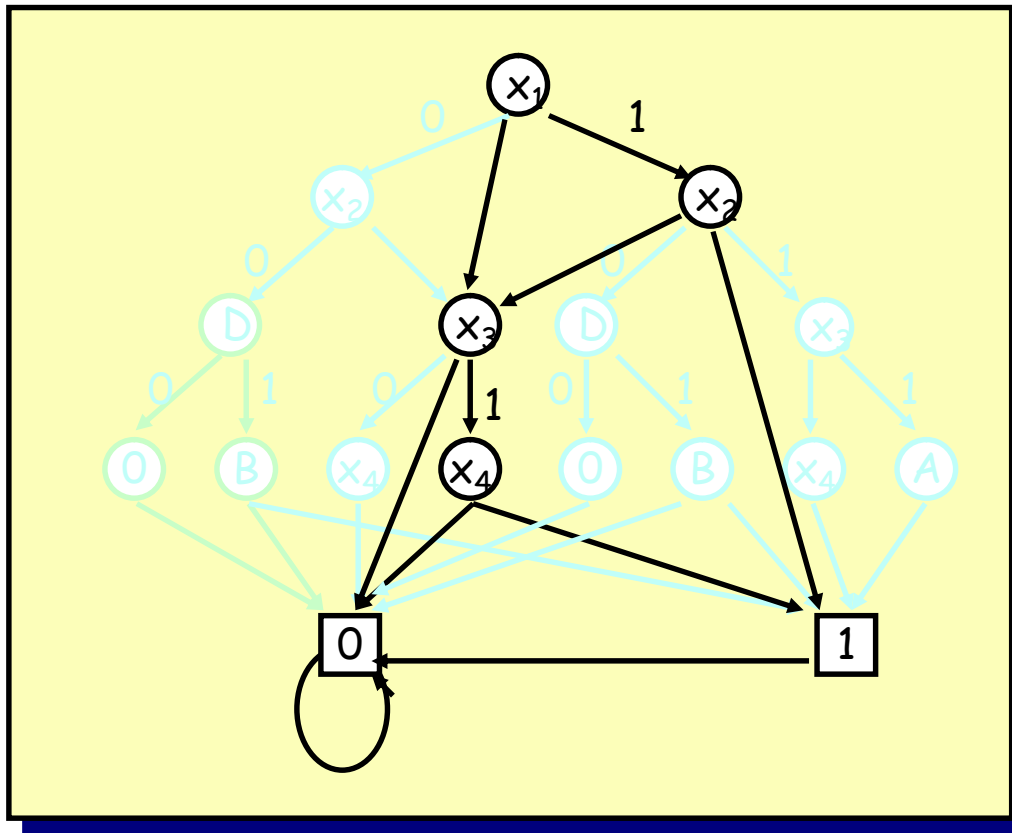
# Minimalautomat



Jeder Knoten wird mit  $x_i$  bezeichnet, wobei  $i$  die Ebene anzeigt :



# Ordnung der Zustände

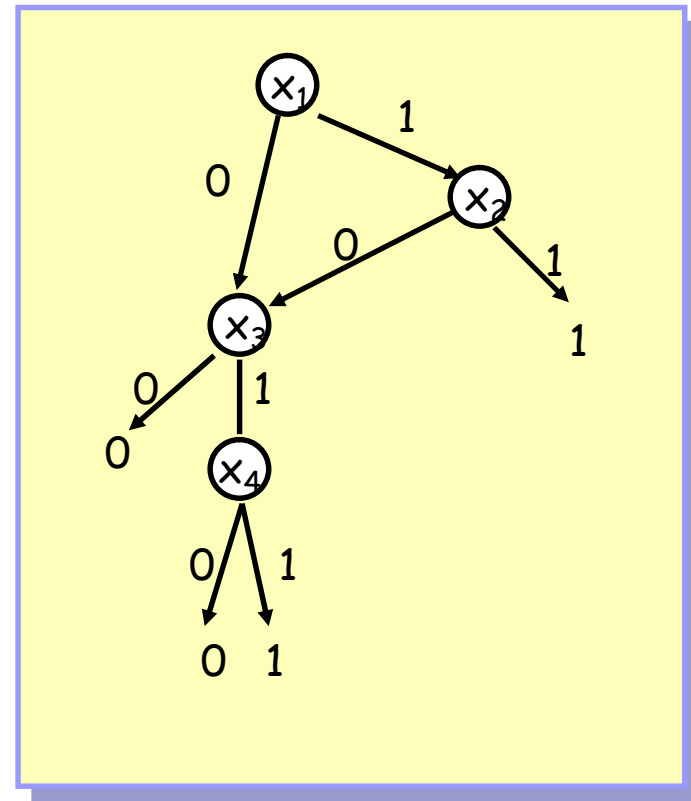
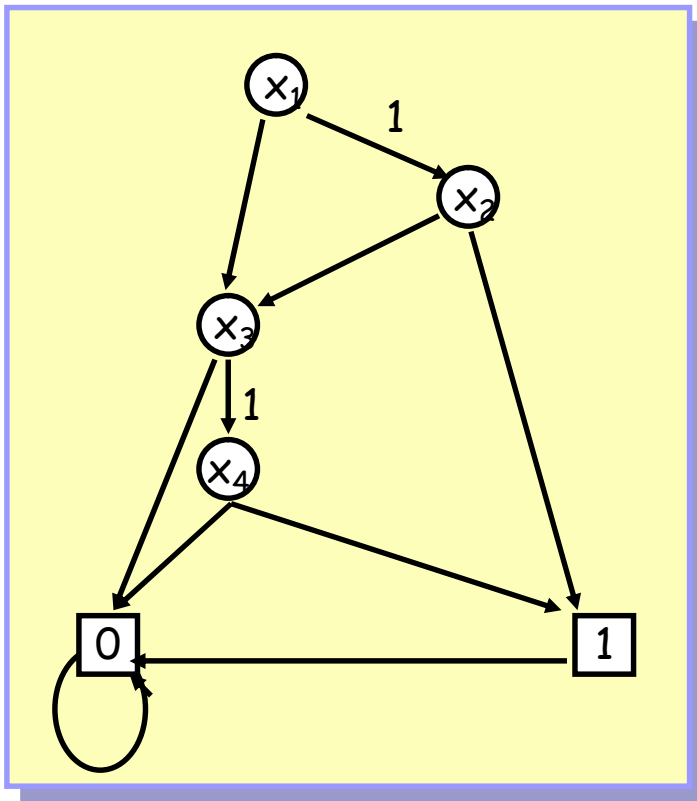


Jeder Zustand, bei dem alle ausgehenden Kanten das gleiche Ziel haben, wird entfernt.



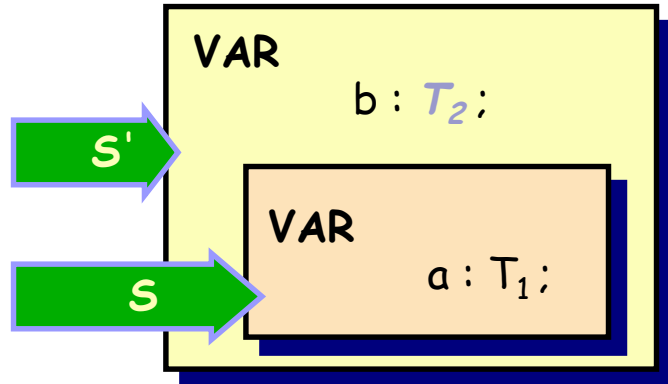
# Minimaler Automat und OBDD

Durch Gegenüberstellung erkennt man leicht, daß der so erhaltene minimale Automat und der OBDD übereinstimmen.





# Vermeiden der Zustandsexplosion



Ursprünglich Zustandsbereich  $S_1$   
Erweitert zu Zustandsbereich  $S = S_1 \times S_2$   
also  $S_1 = T_1$ ,  $S_2 = T_2$  und  $S = T_1 \times T_2$ .

$S_1$  repräsentiert durch Boolesche Variablen  $x_1, \dots, x_k$ .  
 $S_2$  repräsentiert durch Boolesche Variablen  $y_1, \dots, y_r$ .

Eine  $n$ -stellige Relation  $R$  auf  $S = S_1 \times S_2$  heißt **unabhängig von  $S_2$** , falls für beliebige  $a_1, \dots, a_n \in S_1$ ,  $b_1, \dots, b_n, c_1, \dots, c_n \in S_2$  gilt :

$$R((a_1, b_1), \dots, (a_n, b_n)) \Leftrightarrow R((a_1, c_1), \dots, (a_n, c_n))$$

Ist  $R$  unabhängig von  $S_2$ , so wird die BDD-Darstellung von  $R$  nicht von  $y_1, \dots, y_r$  abhängen:

Ist  $R$  unabhängig von  $S_2$ , dann kommen in der OBDD-Darstellung von  $R$  die Variablen  $y_1, \dots, y_r$  nicht vor.

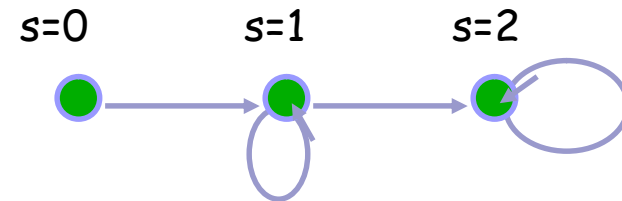
Obwohl sich beim Übergang von  $S_1$  nach  $S_2$  die Anzahl der Zustände vervielfacht hat, hat sich die BDD-Darstellung von  $R$  nicht verändert. Die Zustandsexplosion schlägt also nicht auf die Repräsentation der Teilmengen und Relationen durch. Dies ist der wahre Vorteil des symbolischen Model-Checking.



# SMV-Modell → BDD-Semantik

Das folgende einfache SMV-Programm dient im ersten Teil dieses Kapitels als *Generalbeispiel*.

```
MODULE main
VAR
  s: 0 .. 2;
ASSIGN
  init(s) := { 0, 1 };
  next(s) :=
    case
      s=0 : 1;
      s=1 : 1 union 2;
      s=2 : 2
    esac;
SPEC
  AF AG s=2 /* falsch */
SPEC
  EF AG s=2 /* richtig */
```





# Codierungen

Zunächst codieren wir nicht-Boolesche Variablen durch Boolesche Vektoren. Sei also

$\text{VAR } x : a .. b$

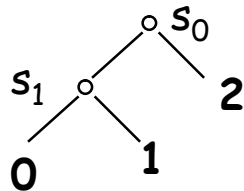
Dann codiere :

```
encode( x, i, a .. a) = a
encode( x, i, a .. b) =
  x.i → ( encode(x,i+1,a .. ⌊(a+b)/2 ⌋),
          encode(x,i+1,⌊(a+b)/2 ⌋+1 .. b) )
```

Die Codierung der Variablen  $x$  in unserem Beispiel veranschaulichen wir auf drei Weisen :

1.

Codierung der Variablen  
als Binärbaum:



2.

Textuelle Repräsentation  
des Binärbaumes :

$s_0 \rightarrow (s_1 \rightarrow (0, 1), 2)$

3.

Als Mengen Boolescher Vektoren gleicher Länge  
 $0 = \{ (0,0) \}$   $1 = \{ (0,1) \}$  ,  $2 = \{ (1,0), (1,1) \}$

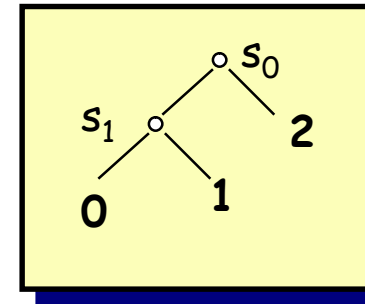


# Codierung von *Init* als OBDD

Programmtext :

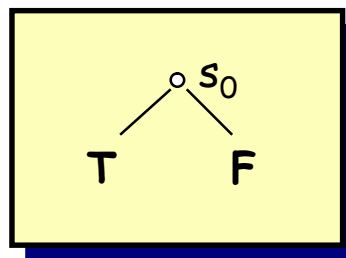
```
init(s) := { 0, 1 }
```

Codierung der Variablen s :



Ersetze  
0 und 1 durch **T**  
2 durch **F**  
und bilde den OBDD

BDD für "s in { 0 , 1 }" :



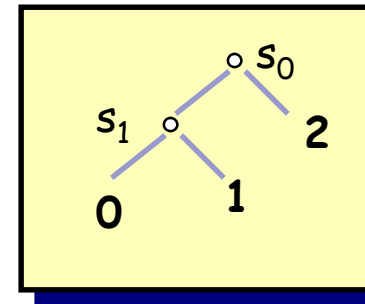


# Codierung von R als OBDD

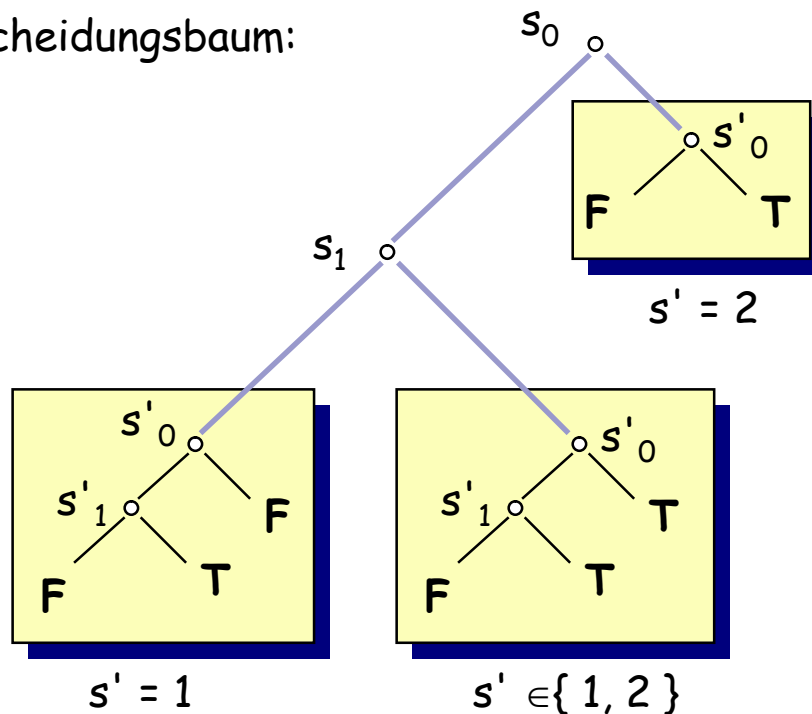
## 1. Programmtext :

```
next(s) :=  
  case  
    s=0 : 1;  
    s=1 : 1 union 2;  
    s=2 : 2  
  esac;
```

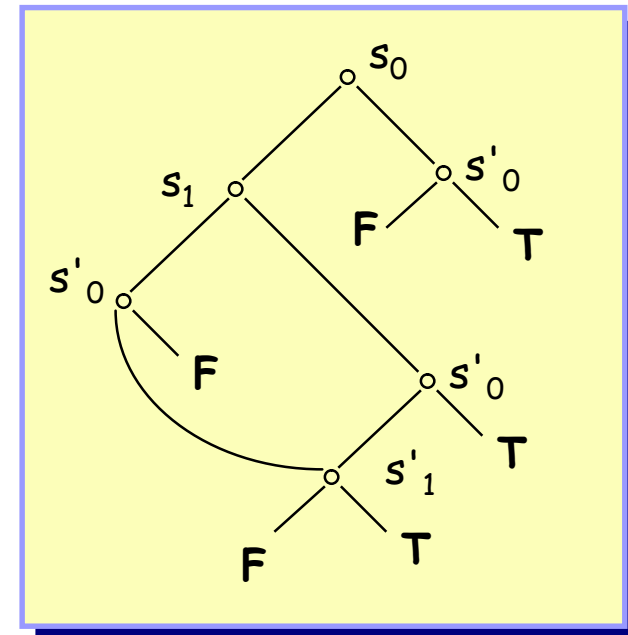
## 2. Codierung der Variablen s :



## 3. Entscheidungsbaum:



## 4. Der fertige BDD :





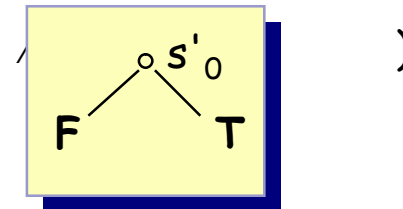
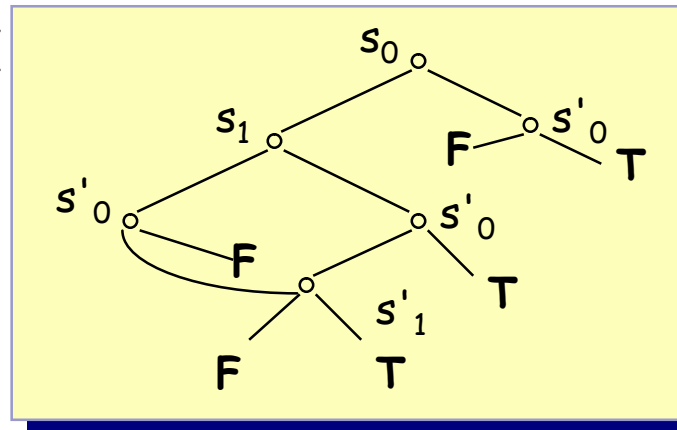
# R<sup>-1</sup> als Funktion

$$\begin{aligned}
 R^{-1}(p) &= \text{EX } p \\
 &= \lambda(s_0, s_1). \exists(s'_0, s'_1). R \wedge p'
 \end{aligned}$$

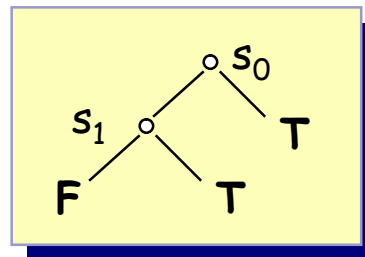
Insbesondere erhalten wir in unserem Beispiel :

$$R^{-1}(s=2) = \exists s' . R(s, s') \wedge s'=2$$

$$= \exists(s'_0, s'_1). ($$



=



$$= (s=1 \vee s=2)$$



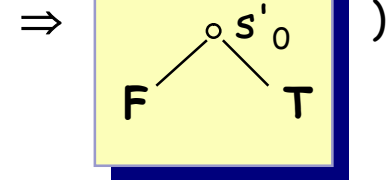
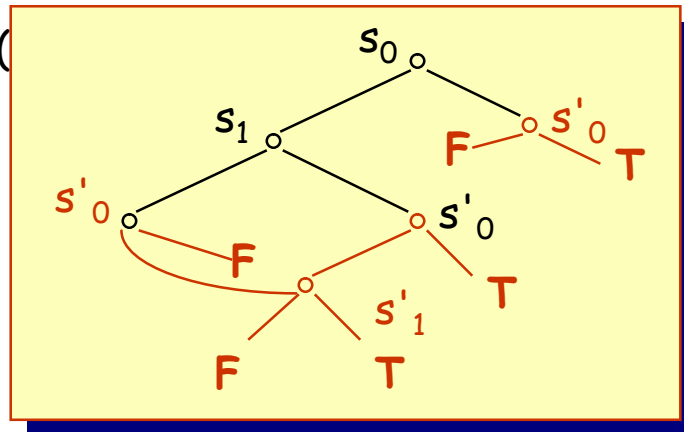
# wp(R,-) als Funktion

$$\begin{aligned} \text{wp}(R, -) &= \text{AX} \\ &= \lambda p. \lambda(s_0, s_1). \forall (s'_0, s'_1). R \Rightarrow p' \end{aligned}$$

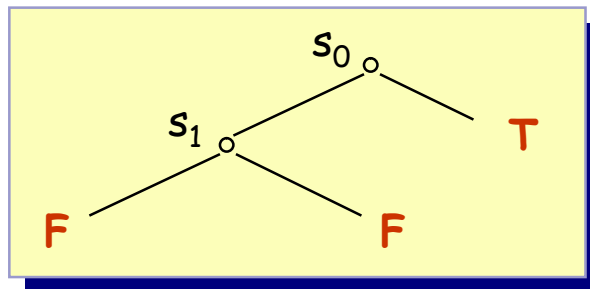
Insbesondere erhalten wir in unserem Beispiel :

$$\text{wp}(R, s=2) = \forall s'. R(s, s') \Rightarrow s'=2$$

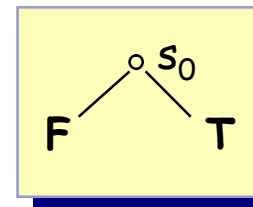
$$= \forall (s'_0, s'_1). ($$



=



=



= (s = 2)

Im nächsten Schritt erhält man dann :  $R^{-1}(s=1 \vee s=2) = T$ .



# Berechnung der ersten SPEC

**AF AG s=2**

1. Berechne

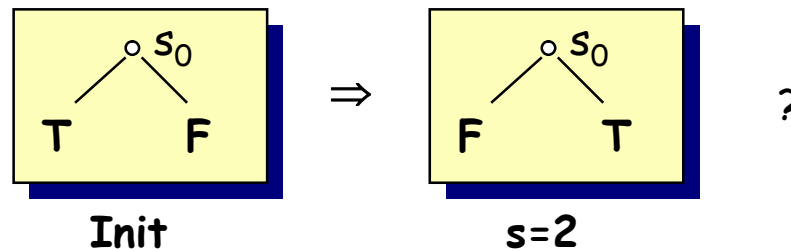
$$\begin{aligned} \mathbf{AG} (s=2) &= \forall Y. s=2 \wedge \mathbf{AX} Y \\ &= s=2 \wedge \mathbf{wp}(R, s=2) \wedge \dots \\ &= s=2 \wedge s=2 \wedge \dots \\ &= s=2. \end{aligned}$$

2. Berechne

$$\begin{aligned} \mathbf{AF} \mathbf{AG} (s=2) &= \mathbf{AF} s=2 \\ &= \mu Y. s=2 \vee \mathbf{AX} Y \\ &= s=2 \vee \mathbf{wp}(R, s=2) \vee \dots \\ &= s=2. \end{aligned}$$

3. Prüfe

$$R \models \text{Init} \Rightarrow \mathbf{AF} \mathbf{AG} s=2 ?$$



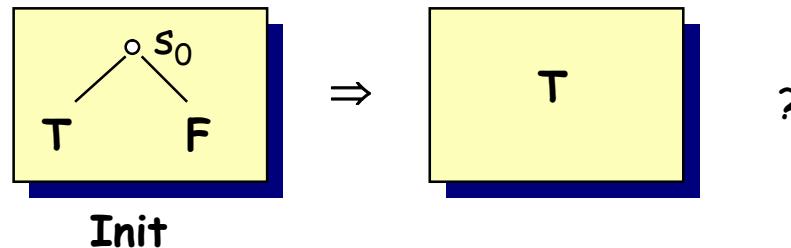
**Nein !**



# Berechnung der zweiten SPEC

EF AG s=2

1. Berechne  
 $AG (s=2)$  = ... wie vorher ..  
= s=2.
2. Berechne  
 $EF AG (s=2)$  = EF s=2  
=  $\mu Y. s=2 \vee EX Y$   
=  $s=2 \vee R^{-1}(s=2) \vee R^{-1}( \dots ) \dots$   
=  $s=2 \vee (s=1 \vee s=2) \vee R^{-1}(s=1 \vee s=2) \vee \dots$   
= T
3. Prüfe  
 $R \models Init \Rightarrow EF AG s=2 ?$



Ja !



# Formale Semantik

Wir betrachten zunächst nur die Semantik eines SMV-Programmes, das aus einem einzigen (parameterlosen) Modul "*main*" besteht.

Semantische Funktionen :

$[  \dots  ]_I$	:	Initialbedingung
$[  \dots  ]_T$	:	Transition <b>R.</b>
$[  \dots  ]_S$	:	Spezifikation
$[  \dots  ]_E$	:	Ausdrucksemantik

Wenn der Index fehlt, so gilt die angegebene semantische Klausel für alle semantischen Funktionen. Alle semantischen Funktionen, bis auf  $[| \dots |]_E$  liefern als Ergebnis einen OBDD.  $[| \dots |]_E$  hingegen liefert einen OBDD, an dessen Blättern sich Konstanten des Datenbereiches befinden.

Ein SMV-Programm ist eine durch ";" getrennte **Liste** von Fragmenten. Die Semantik einer solchen Liste ist die Konjunktion der Semantiken der Fragmente, d.h. wir erhalten als erste semantische Klausel :

$$(1) \quad [| \text{head} ; \text{rest} |] = [| \text{head} |] \wedge [| \text{rest} |].$$



# Programmkopf

In unserem einfachen Falle ist der Programmkopf immer

```
MODULE main.
```

Da dies auch der einzige Modul ist, gilt automatisch immer *running=tt*.  
Folglich ist auch diese Variable überflüssig. Daher ist die Kopfzeile  
semantisch neutral, also

```
(2) [| MODULE main |] = tt.
```



# Variablen

Variablendeklarationen stellen einen Kontext her, in dem der Rest des Programmes ausgerechnet wird.

Variablen sind von Hause aus Boolesch :

(3)  $[| \text{VAR } x : \text{boolean } |] = \text{tt.}$

Nicht-Boolesche Variablen werden durch einen Entscheidungsbaum dargestellt. Das gilt für Bereichstypen, wie auch für Aufzählungstypen.

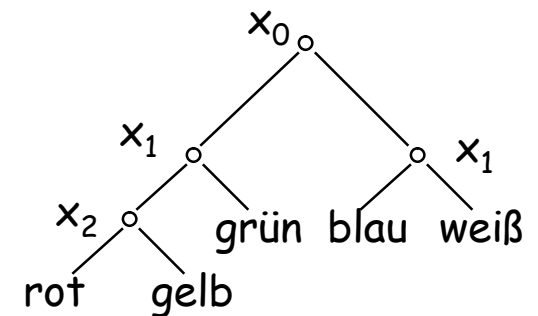
(4)  $[| \text{VAR } x : a .. b |]_E = ( x = \text{encode}(x, 0, a .. b) )$

Beispiel :

$[| \text{VAR } x : (\text{rot}, \text{gelb}, \text{grün}, \text{blau}, \text{weiß}) |]$

$= \text{encode}(x, 0, (\text{rot}, \text{gelb}, \text{grün}, \text{blau}, \text{weiß})) =$

$= x.0 \rightarrow (x.1 \rightarrow (x.2 \rightarrow (\text{rot}, \text{gelb}), \text{grün}), x.1 \rightarrow (\text{blau}, \text{weiß})).$





# ASSIGN

Die Initialbedingung ist nur für  $[| \dots |]_I$  interessant, alle anderen semantischen Funktionen liefern  $\top$  :

$$(5) \quad [| \text{ASSIGN init}(x) := e |]_I := x \in [| e |]_E.$$

Alle anderen ASSIGN-statements sind nur für  $[| \dots |]_T$  relevant. Die übrigen semantischen Funktionen liefern  $\top$  :

$$(6) \quad [| \text{ASSIGN } x := e |]_T := x \in [| e |]_E.$$

$$(7) \quad [| \text{ASSIGN next}(x) := e |]_T := x' \in [| e |]_E.$$



# Ausdruckssemantik

Jedem Ausdruck wird ein OBDD zugeordnet, an dessen Blättern sich Konstanten des zum Ausdruck gehörenden Datentyps befinden.

$$(8) \quad [| c |]_E = \{ c \}, \text{ falls } c \text{ eine Konstante ist.}$$

$$(9) \quad [| x |]_E = \text{encode}(x, 0, \text{Typ}), \\ \text{falls } x \text{ erklärt ist als "VAR } x : \text{Typ".}$$

Falls • irgendeine Operation des Datentyps ist, also

•  $\in \{ +, -, *, /, \text{mod}, =, <, <=, >, >=, |, \&, \rightarrow, \leftrightarrow \}$

dann wird • zu den Blättern propagiert und dort ausgeführt :

$$(10) \quad [| e_1 \bullet e_2 |]_E = [| e_1 |]_E \bullet [| e_2 |]_E,$$

wobei die Propagation nach folgenden Regeln stattfindet :

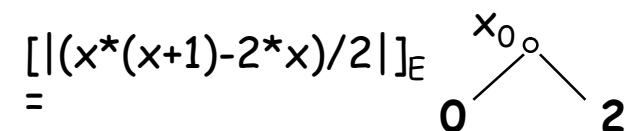
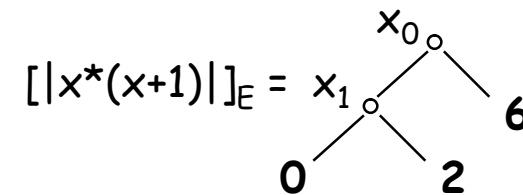
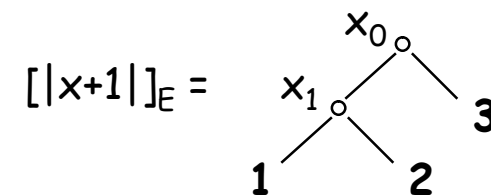
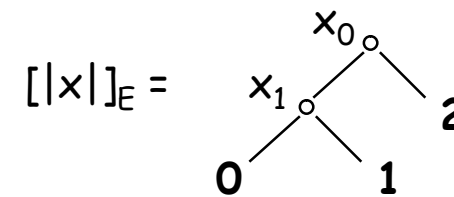
$$(x \rightarrow (y, z)) \bullet a = x \rightarrow ((y \bullet a)[x/0], (z \bullet a)[x/1])$$

$$a \bullet (x \rightarrow (y, z)) = x \rightarrow ((a \bullet y)[x/0], (a \bullet z)[x/1])$$

und folgende Vereinfachungen Anwendung finden :

$$x \rightarrow (y, y) = y$$

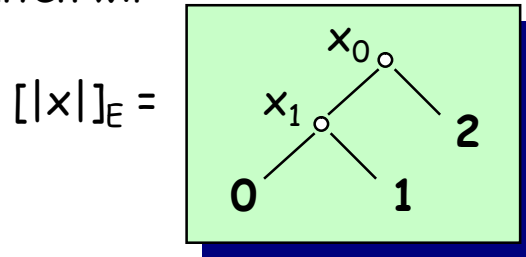
$$(x \rightarrow (y, z)) \bullet (x \rightarrow (u, v)) = (x \rightarrow (y \bullet u, z \bullet v)).$$



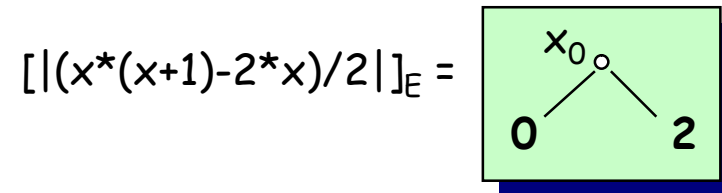


# Berechnung der Transitionsemantik

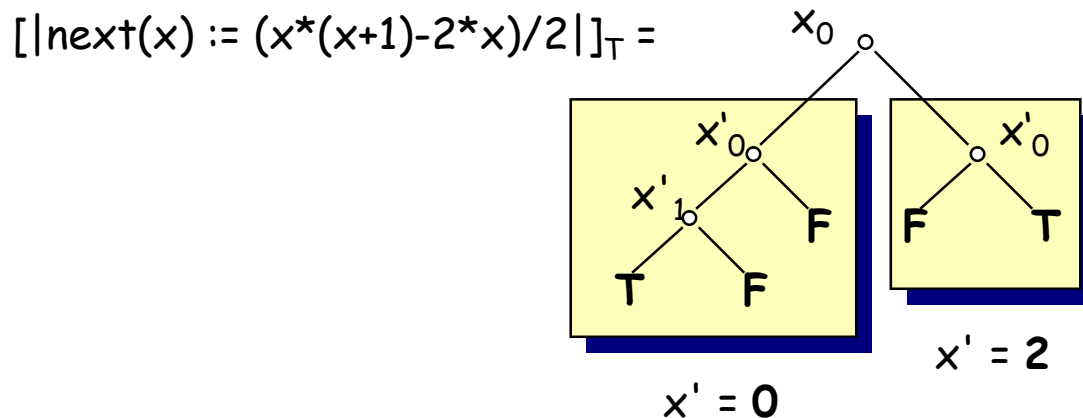
1. Aus der Deklaration "VAR x : 0 .. 2 ;" erhalten wir :



2. Für den Ausdruck " $(x*(x+1)-2*x)/2$ " ergibt sich :



3. Die Semantik der Transition " $next(x) := (x*(x+1)-2*x)/2$ " liefert schließlich den gesuchten OBDD :





# Nichtdeterminismus

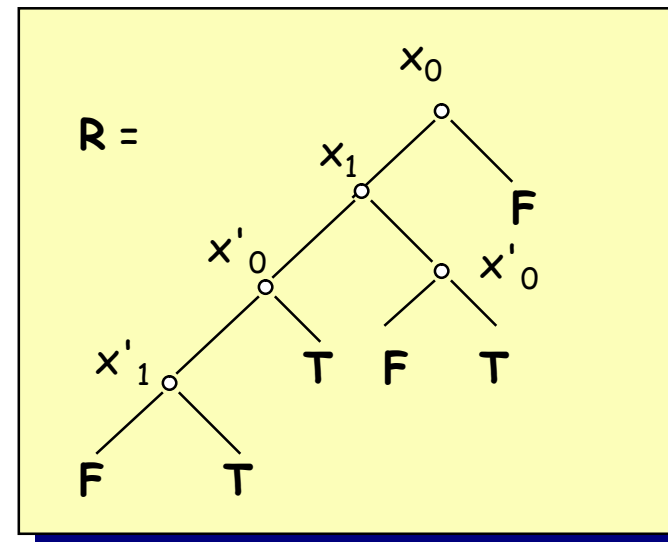
SMV gestattet nichtdeterministische Transitions-Relationen, weil der Wert eines Ausdruckes eine *Menge* von Datenobjekten sein kann. Allerdings muß die linke Seite einer Zuweisung immer entweder  $x$  oder  $next(x)$  für eine Variable  $x$  sein.

Eine nichtdeterministische Spezifikation durch Angabe eines beliebigen Boolesch-wertigen Ausdruckes wäre aber leicht zu realisieren :

Beispiel :

```
VAR
  x : 0 .. 2;
ASSIGN
  2*next(x) > x + next(x).
```

Als Transitionsrelation erhalten wir den OBDD :





# Fairness

Die formale Semantik der Fairness-Bedingungen ist einfach :

Dem angegebenen CTL-Ausdruck wird ein implizites **GF** vorangestellt :

$$(11) \quad [ [ \text{FAIRNESS } e ] ]_F = \text{GF} [ [ e ] ]_E )$$

Allerdings verläuft die Auswertung in SMV auf eine andere Weise, indem nämlich, wie in Kapitel 6 erläutert, die Pfad-Quantoren **A** und **E** durch die mittels der Fairness-Bedingungen **C** modifizierten Pfad-Quantoren **A<sub>C</sub>** und **E<sub>C</sub>** ersetzt werden.



# Spezifikationssemantik und Korrektheitsbedingung

Die Semantik der Spezifikation liefert eine Funktion, die den Fixpunkt berechnet welcher zu dem CTL-Ausdruck gehört.

$$(12) \quad [ \text{SPEC } f ]_S := \lambda r. \text{eval}(f, r).$$

Wir erinnern uns :

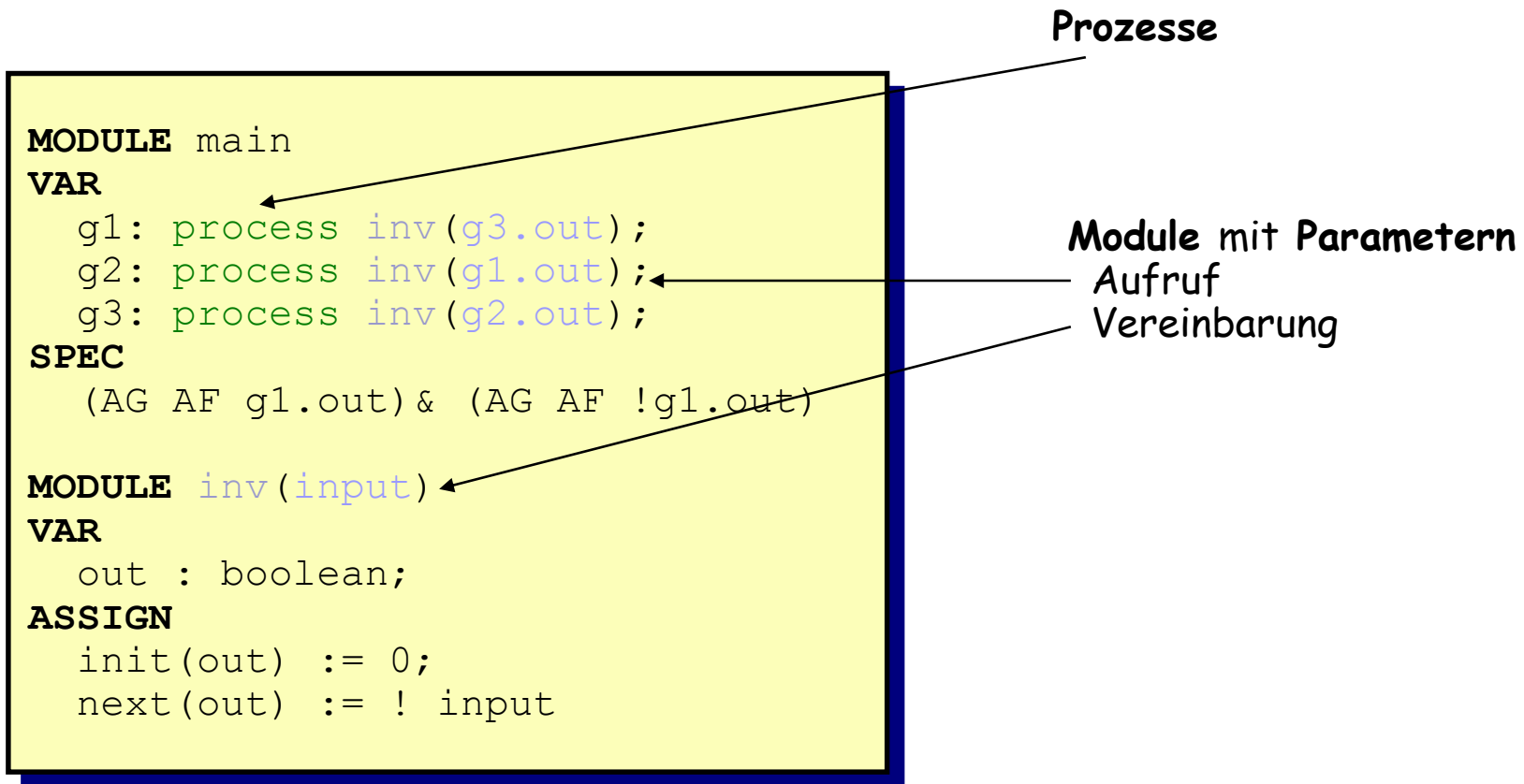
$$\begin{aligned} \text{eval}(\text{EF } g, r) &= \text{evalEF}(\text{eval}(g, r), \text{tt}, r) \\ \text{eval}(\text{E}[f \text{ U } g], r) &= \text{evalEU}(\text{eval}(g, r), \text{eval}(f, r), \text{ff}), \\ &\dots \text{ etc. } \dots \end{aligned}$$

Die Korrektheitsbedingung eines SMV-Programmes **prog** lautet schließlich :

$$(13) \quad [ \text{prog} ]_I \wedge [ \text{prog} ]_F \Rightarrow [ \text{prog} ]_S ( [ \text{prog} ]_R )$$



# Module, Parameter, Prozesse





# Semantik von Moduldeklaration

Ein Modul mit  $k$  Parametern wird zu einer  $k$ -stelligen Funktion :

$$(14) \quad [ [ \text{MODULE } m(p_1, p_2, \dots, p_k) \text{ body } ] ] \\ = ( m = \lambda p_1. p_2. \dots p_k. [ [ \text{body } ] ] ).$$

Da eine Instanz eines Moduls nur dann aktiv ist, wenn seine Variable "running" wahr ist, müssen wir zunächst eine leichte Modifikation an der Assignmentsemantik vornehmen :

$$(7') \quad [ [ \text{ASSIGN } \text{next}(x) := e ] ] := \text{running} \rightarrow (x' = x, x' \in [ [ e ] ]_E)$$



# Semantik der Modulinstanziierung

Eine lokale Variable  $x$  in einer Instanz  $a$  eines Moduls  $m$  wird durch die Punktnotation  $a.x$  referenziert. Für das folgende erweist es sich als bequem, diese Punktnotation durch folgende Regeln auf beliebige Ausdrücke zu erweitern :

$$\begin{aligned} a . ( b ) &= a.b \\ a.(bc) &= a.b a.c \\ a.c &= c , \text{ falls } c \text{ eine Konstante ist.} \end{aligned}$$

Mit den spitzen Klammern " $\langle$ " und " $\rangle$ " *schützen* wir ganze Ausdrücke vor der Punktnotation :

$$a . \langle e \rangle = e.$$

Die Instanziierung eines Moduls  $m$  mit  $k$  Parametern wird dann beschrieben durch :

$$(15) \quad [ | \text{VAR } x : m(e_1, e_2, \dots, e_k) | ] \\ = ( x . m \langle e_1 \rangle \langle e_2 \rangle \dots \langle e_k \rangle ) [ \text{running} / \langle \text{running} \rangle ] .$$

$m$  wird also auf  $e_1, \dots, e_k$  angewendet, im Ergebnis wird jeder Variablen außer "running" ein "x." vorangestellt. Dadurch wird erreicht, daß "running" vom Vatermodul geerbt wird.



# Beispiel der Modulinstanziierung

Für den Modul "inv" ergibt sich :

```
[| MODULE inv(input) ... |] = ( inv = λ input. [| next(out) := ! input |] )
```

Für die Instanziierung berechnet man :

```
[| VAR x : inv(g3.out) |] = ( x. inv(g3.out)[running/<running>]  
= ( x.λ input. [| next(out) := ! input |][running/<running>]) g3.out  
= x. [| next(out) := ! g3.out |] [running/<running>]  
= x. ( running →( out' = ! g3.out , out' = out) [running/<running>])  
= x. ( <running> →( out' = ! g3.out , out' = out) )  
= running →( x.out' = ! g3.out , x.out' = out)
```



# Prozesse

Ein Prozess darf nur aktiviert werden, falls

- der Vaterprozess running ist und
- kein Bruderprozess läuft.

Um letzteres zu garantieren, führen wir eine Variable "blocked" ein, die jeden Bruderprozess blockieren kann. In der folgenden Klausel seien in "rest" die Deklarationen aller Bruderprozesse zusammengefasst :

Die Instanziierung eines Moduls  $m$  mit  $k$  Parametern wird dann beschrieben durch :

$$(16) \quad [ | \text{VAR process } x : m(e_1, e_2, \dots, e_k) ; \text{rest} \ | ] \\ = ( x.m \langle e_1 \rangle \langle e_2 \rangle \dots \langle e_k \rangle ) \\ \wedge ( x.\text{running} \Rightarrow \neg \text{blocked} \wedge \text{running} ) \\ \wedge [ | \text{rest} \ | ] [ \text{blocked} / \text{blocked} \vee x.\text{running} ]$$

In der ersten Klausel braucht jetzt  $\langle \text{running} \rangle$  nicht mehr geschützt werden, denn in der zweiten Klausel folgt die Bedingung " $x.\text{running} \Rightarrow \text{running}$ " und zusätzlich: " $x.\text{running} \Rightarrow \neg \text{blocked}$ " und in jedem anderen Bruderprozess wird "blocked" ersetzt durch " $\text{blocked} \vee x.\text{running}$ ".

Die Klausel ist nicht mehr "denotational", weil auch die Semantik der "Umgebung", genauer der Bruderprozesse (rest) verändert wird.



# Beispiel

$$\begin{aligned} (16) \quad & [ | \text{VAR process } x : m(e_1, e_2, \dots, e_k) ; \text{rest} \quad | ] \\ & = ( x.m \langle e_1 \rangle \langle e_2 \rangle \dots \langle e_k \rangle ) \\ & \wedge ( x.\text{running} \Rightarrow \neg \text{blocked} \wedge \text{running} ) \\ & \wedge [ | \text{rest} \quad | ] [ \text{blocked} / (\text{blocked} \vee x.\text{running}) ] \end{aligned}$$

Die obige Regel ergibt in unserem Beispiel :

$$\begin{aligned} & [ | \text{VAR} \\ & \quad \text{process } g_1 : \text{inv}(x_1) ; \\ & \quad \text{process } g_2 : \text{inv}(x_2) ; \\ & \quad \text{process } g_3 : \text{inv}(x_3) ; \dots \quad | ] \\ & = \quad g_1.\text{inv} \langle x_1 \rangle \wedge g_2.\text{inv} \langle x_2 \rangle \wedge g_3.\text{inv} \langle x_3 \rangle \\ & \quad \wedge g_1.\text{running} \Rightarrow \neg (\text{blocked} \vee g_2.\text{running} \vee g_3.\text{running}) \wedge \text{running} \\ & \quad \wedge g_2.\text{running} \Rightarrow \neg (\text{blocked} \vee g_1.\text{running} \vee g_3.\text{running}) \wedge \text{running} \\ & \quad \wedge g_3.\text{running} \Rightarrow \neg (\text{blocked} \vee g_1.\text{running} \vee g_2.\text{running}) \wedge \text{running} \\ & \quad \wedge \dots \end{aligned}$$

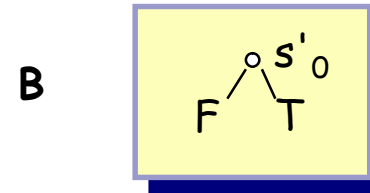
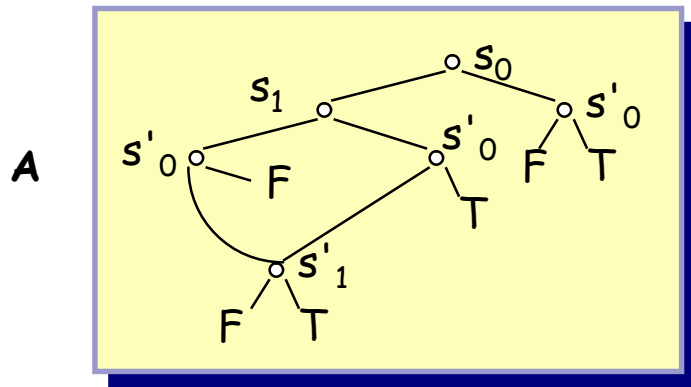


# Aufgaben

1. Mit den Variablenordnungen  $x_1 \succ x_2 \succ x_3 \succ x_4$  bzw.  $x_4 \succ x_3 \succ x_2 \succ x_1$  bestimmen Sie die OBDDs zu

$$\begin{aligned} &x_0 \vee x_1 \Rightarrow x_2 \wedge \neg x_3 \\ &(x_0 \wedge x_3) \vee x_2 \\ &\neg x_0 \Rightarrow (x_3 \vee x_1) \wedge x_2 \end{aligned}$$

2. Mit der Variablenordnung  $s_0 \succ s_1 \succ s'_0 \succ s'_1$  seien die OBDDs



gegeben. Bestimmen Sie die OBDDs zu

- a)  $A \wedge B, A \vee B$  und  $A \Rightarrow B$
- b)  $\exists (s'_0, s'_1). (A \wedge B)$
- c)  $\forall (s'_0, s'_1). (A \Rightarrow B)$



# Aufgaben (b)

3. Geben Sie eine einfache Methode an, um Ausdrücke der Art 2b) und 2c) zu berechnen. Beschreiben Sie also einen einfachen Algorithmus, der für die Variablenordnung  $x_1 \succ \dots \succ x_n \succ x'_1 \succ \dots \succ x'_n$  Ausdrücke der Art

$$\lambda x_1 \dots x_n . \exists (x'_1, \dots, x'_n) . A \wedge p'$$

bzw.

$$\lambda x_1 \dots x_n . \exists (x'_1, \dots, x'_n) . A \Rightarrow p'$$

berechnet. Dabei sei  $p'$  ein OBDD, der nur von  $(x'_1, \dots, x'_n)$  abhängt.

4. Der Datentyp  $\text{Farbe} = \{\text{rot}, \text{gelb}, \text{grün}\}$  sei folgendermaßen durch Boolesche Vektoren codiert :  
 $\text{rot} = (0,0)$ ,  $\text{gelb} = (0,1)$ ,  $\text{grün} = (1,0)$

a) Finden Sie den OBDD, der die Teilmenge  $\{\text{rot}, \text{gelb}\}$  repräsentiert.

b) Finden Sie den OBDD, der die Relation

$$R = \{ (s, s') \mid \begin{array}{l} \text{if } s = \text{rot} \text{ then } s' = \text{gelb} \\ \text{else if } s = \text{gelb} \text{ then } s' \in \{ \text{gelb}, \text{grün} \} \\ \text{else } s' = \text{grün} \end{array} \}$$

repräsentiert. Wählen Sie die Variablenordnung

$$s_0 \succ s_1 \succ s'_0 \succ s'_1.$$