Symbolisches Model Checking und BDD-Semantik

H. Peter Gumm

Philipps-Universität Marburg

Sommersemester 2007

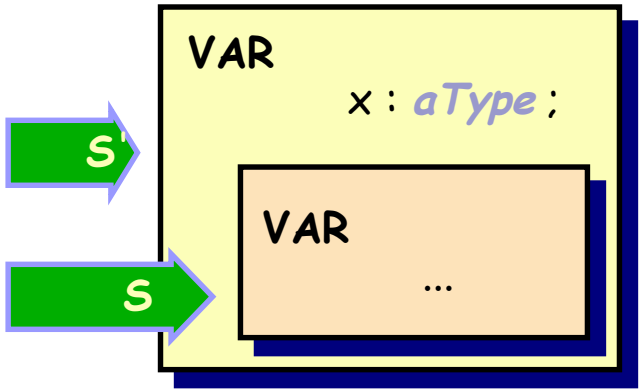
Explizites Model Checking

- 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

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

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

Dafür gibt es einen Booleschen Term $\mathbf{p}(x_1,..,x_k)$ mit

$$\chi_P = \lambda(x_1,..,x_k).\mathbf{p}(x_1,..,x_k)$$

Boolesche Operatoren.

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

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

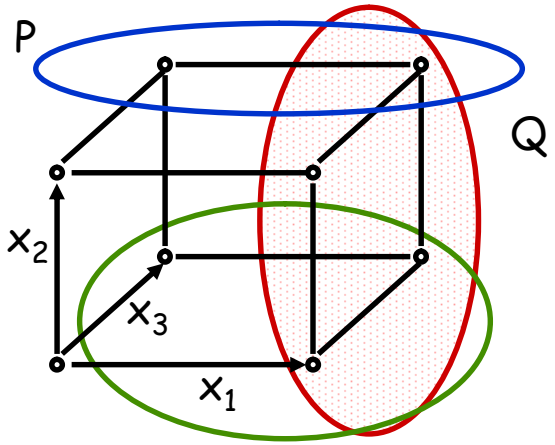
next-Variable – x‘

- Für die Darstellung der Transitionsrelation benötigt:
  - Paare von Variablen (x, x')
  - Relation drückt Zusammenhang zwischen x und x' aus
  - Zu jeder Variablen $x_1, \ldots, x_n$ führe entsprechende next-Variable $x'_1, \ldots, x'_n$ ein.
- next-Eigenschaft
  - $p=p(x_1,..,x_n)$ sei Eigenschaft
  - p gilt im nächsten Zustand, falls $p(x'_1,\ldots,x'_n)$
  - 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
  - $x = (x_1,\ldots,x_n)$, dann $x' := (x'_1,\ldots,x'_n)$

Repräsentation von Relationen

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

Beispiel : Sei $S = 2^3$.



$\mathbf{p}(x_1,x_2,x_3) := x_2 \wedge x_3$
$\mathbf{q}(x_1,x_2,x_3) := x_1$
$\mathbf{r}(x_1,x_2,x_3) := \neg x_2 \wedge (\, x_1 \vee x_3\,)$

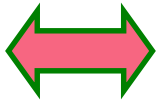
Die Relation "**diametral**" wird repräsentiert durch

$$\mathbf{r}\,(x_1,x_2,x_3,x_1',x_2',x_3') := (x_1 \Leftrightarrow \neg x_1') \wedge (x_2 \Leftrightarrow \neg x_2') \wedge (x_3 \Leftrightarrow \neg x_3'),$$

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:

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

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

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

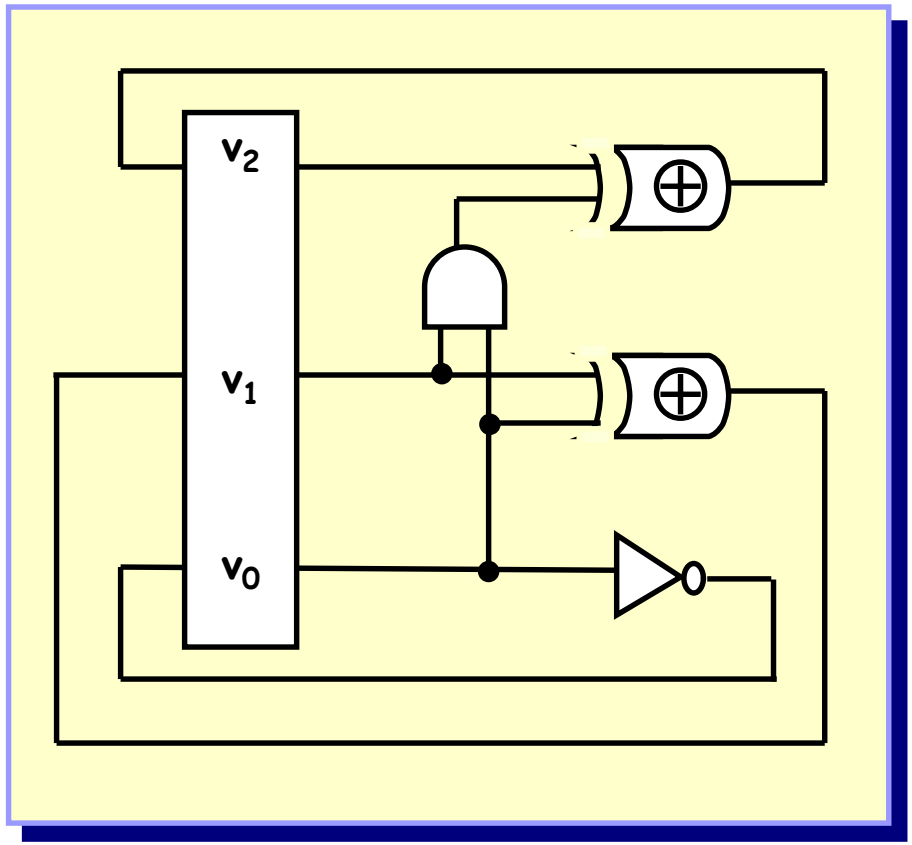
Charakteristische Funktionen als Boolesche Terme:

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

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

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

Eine Synchrone Schaltung



$p_S(v_0,v_1,v_2) := tt$

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

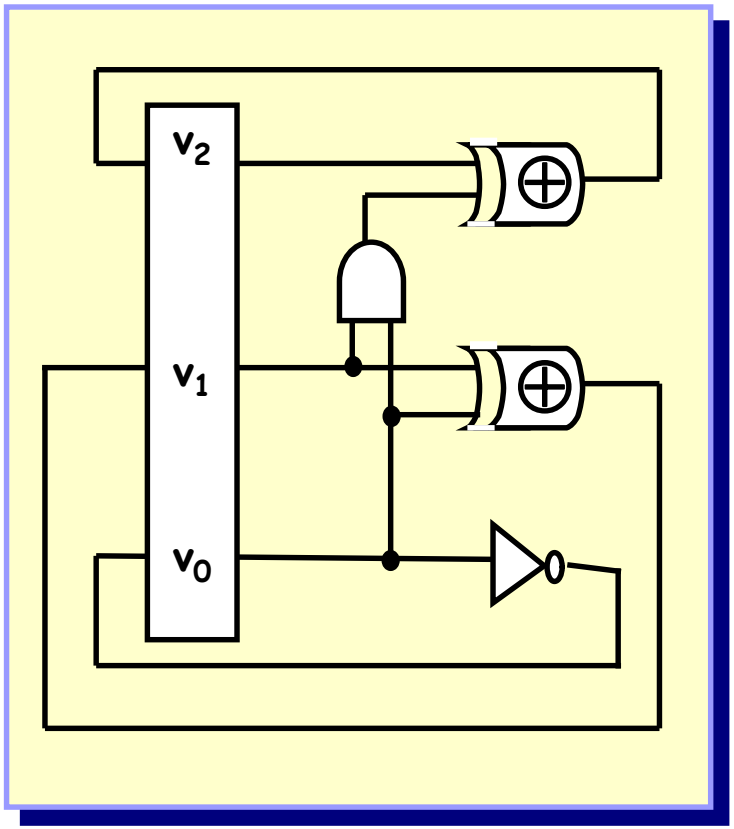
$p_{R0} = v'_0 \Leftrightarrow \neg v_0$

$p_{R1} = v'_1 \Leftrightarrow v_0 \oplus v_1$

$p_{R2} = v'_2 \Leftrightarrow (v_0 \wedge v_1) \oplus v_2$

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:

$$p_R = p_{R0} \vee p_{R1} \vee p_{R2} =$$

$$\begin{aligned} & (v_0' \Leftrightarrow \neg v_0) \wedge (v_1' \Leftrightarrow v_1) \wedge (v_2' \Leftrightarrow v_2) \\ \vee\ & (v_1' \Leftrightarrow v_0 \oplus v_1) \wedge (v_0' \Leftrightarrow v_0) \wedge (v_2' \Leftrightarrow v_2) \\ \vee\ & (v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2) \wedge (v_0' \Leftrightarrow v_0) \wedge (v_2' \Leftrightarrow v_2) \end{aligned}$$

Allgemeine Schaltungen

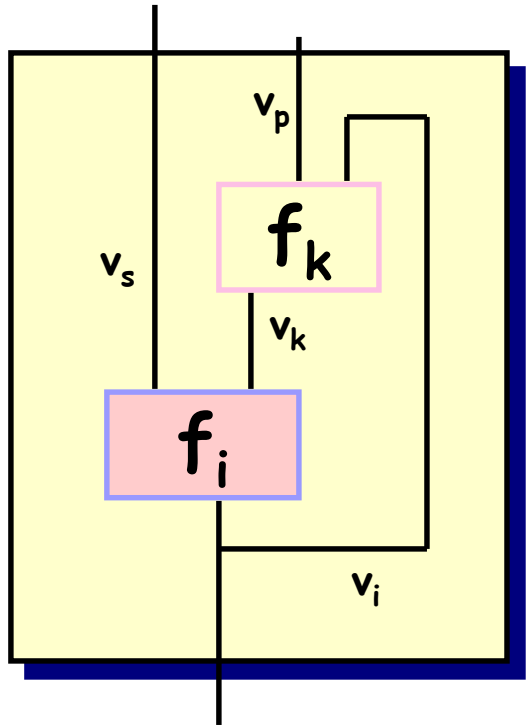
Eine asynchrone Schaltung bestehe aus den **Schaltgliedern** $f_1, \ldots, f_n$ die mit den **Drähten** $d_1, \ldots, d_m$ verbunden sind.

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

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

Dies ergibt mögliche **Transitionsrelationen**:

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



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

Beschreibung der asynchronen Schaltung:

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

Boolesche Quantifikation

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

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

Boolesche Quantifikation einer Variablen

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

$= \neg x_2\ \vee (\ x_3 \Rightarrow \neg x_2\ )$

$= (\ x_3 \Rightarrow \neg x_2\ )$

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

$$\exists\, (x_1,\ldots,x_n).\mathbf{p} = \bigvee (x_1,\ldots,x_n) \in \{tt,ff\}^n.\ \mathbf{p}$$

$= \exists\, x_1 \exists\, (x_2,\ldots,x_n).\mathbf{p}$

$= \exists\, x_1 \exists\, x_2 \exists\, (x_3,\ldots,x_n).\mathbf{p}$

$= \ .\ .\ .$

$= \exists\, x_1 \exists\, x_2 \ldots \exists\, x_n\ .\ \mathbf{p}$

Darstellung von CTL-Operatoren

Sei eine Relation **R** durch den Booleschen Term $\mathbf{r}(x_1,..,x_k, x_1',..,x_k')$ gegeben, und eine Eigenschaft **P** durch $\mathbf{p}(x_1,..,x_k)$ also

so gilt

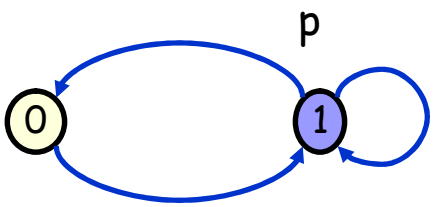
$$(\mathbf{EX}\ \mathbf{P})(x_1,...,x_n) \;=\; \exists\,(x_1',...,x_n').\ \mathbf{r}\ (x_1,..,x_k, x_1',..,x_k') \wedge\ \mathbf{p}\ (x_1',..,x_k')\ ,$$

also

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

Beispiel:

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

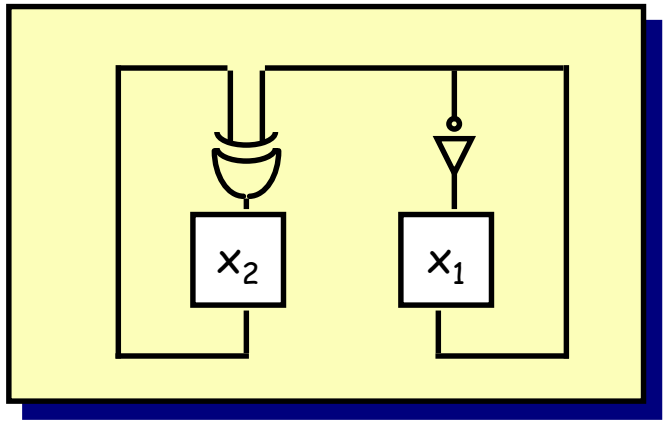


$\mathbf{r} = (x \vee x')$

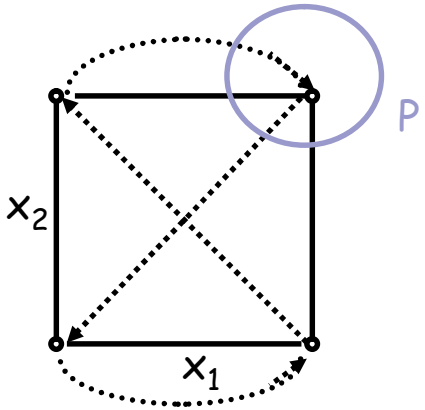
$\mathbf{p} = x$

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

Beispiel: Ein 2-Bit Zähler



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



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

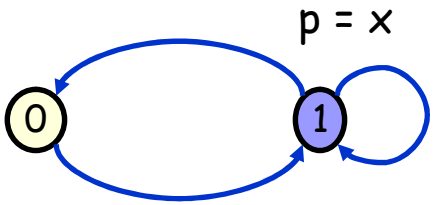
$$
\begin{aligned}
\mathbf{EX\ p} &= \exists \underline{\mathbf{x}}'.(\mathbf{r} \wedge \mathbf{p}') \\
&= \exists (x_1', x_2'). ((x_1' \Leftrightarrow \neg x_1) \wedge (x_2' \Leftrightarrow x_1 \oplus x_2) \wedge (x_1 \wedge x_2)' \\
&= \exists (x_1', x_2'). (x_1' \Leftrightarrow \neg x_1) \wedge (x_2'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

Berechnung von CTL-Formeln

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

$$\mathbf{EF}\ \mathbf{p} = \bigvee n.\ \tau^n(\mathbf{ff})$$

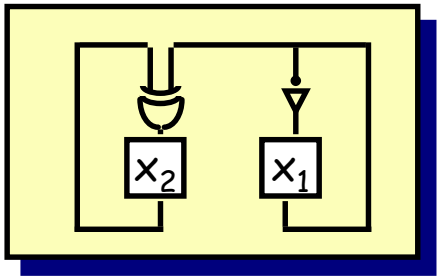
Beispiel 1:



Berechne **EF** x :

$\tau^1(\mathbf{ff})$ = x $\vee$ **EX**(**ff**)
= x.
$\tau^2(\mathbf{ff})$ = x $\vee$ **EX** x
= x $\vee \exists$ (x'). (x $\vee$ x') $\wedge$ x'
= **tt**.
$\tau^3(\mathbf{ff})$ = x $\vee \exists$ (x'). (x $\vee$ x') $\wedge$ **tt**'
= **tt**.
Also gilt im Beispiel :
EF x = **tt**.

Beispiel 2:



Berechne **EF** $x_1 \wedge x_2$:

$\tau^1(\mathbf{ff})$ = $(x_1 \wedge x_2) \vee$ **EX**(**ff**)
= $x_1 \wedge x_2$.
$\tau^2(\mathbf{ff})$ = $(x_1 \wedge x_2) \vee$ **EX** $(x_1 \wedge x_2)$
= x_2.
$\tau^3(\mathbf{ff})$ = $(x_1 \wedge x_2) \vee$ **EX** x_2
= $x_1 \vee x_2$
$\tau^4(\mathbf{ff})$ = $(x_1 \wedge x_2) \vee$ **EX** $x_1 \vee x_2$
= **tt**
Also gilt im Beispiel :
EF $(x_1 \wedge x_2)$ = **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) = ∃ 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 $\wedge$ **evalEX**(y)

if (y'=y) return(y)

else return **evalEG**(y')

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

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

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

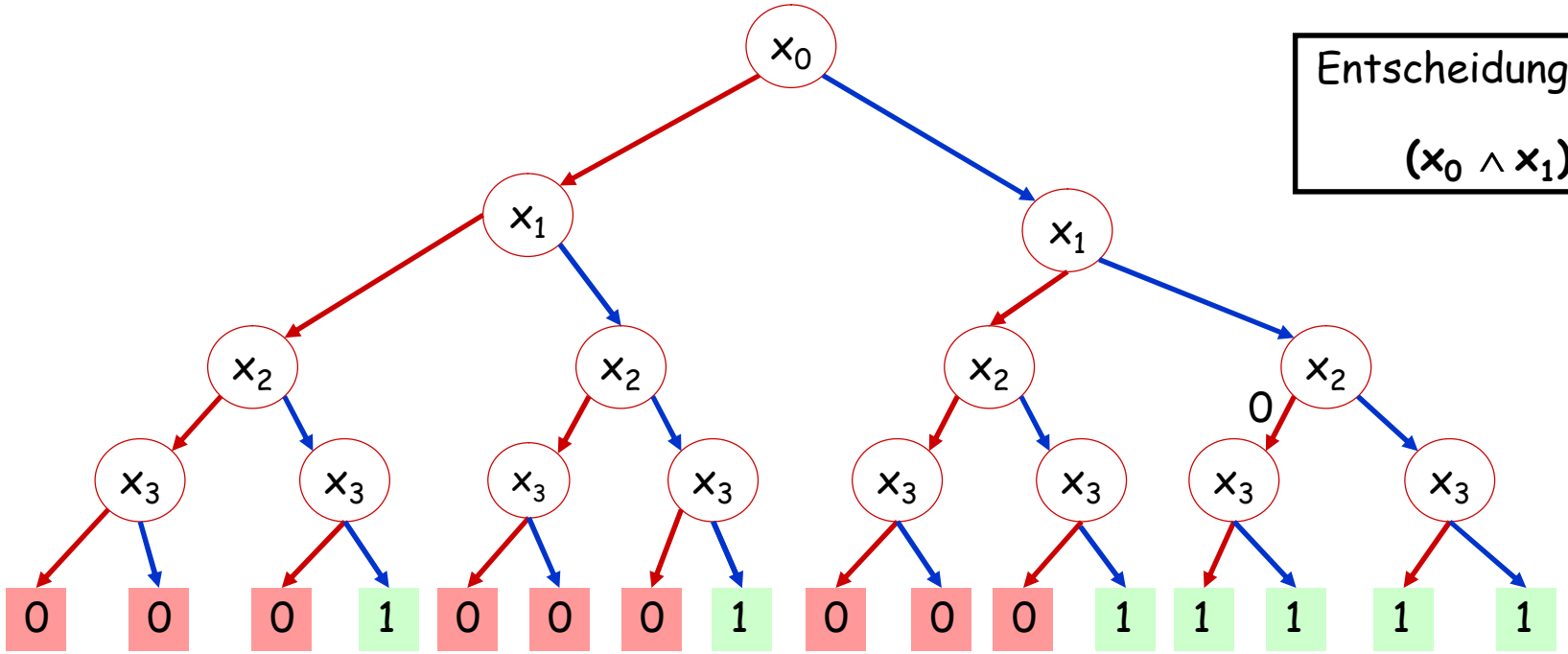
Repräsentation von Daten

- 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
    - $\{ n \mid n \bmod 2 = 0 \vee n = 11 \vee n = 15 \}$
  - Bit-Array
    - [1, 0,1,0,1,0,1,0,1,0,1,1,1,0,1,1]
  - explizite Angabe der Bitfolgen
    - { 0000, 0010, 0100, 0110, 1000, 1010, 1011, 1100, 1110, 1111 }
  - die charakteristische Funktion
    - $\chi_M (n) = \text{if } n \bmod 2 = 0 \vee n = 11 \vee n = 15 \text{ then 1 else 0}$
  - 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

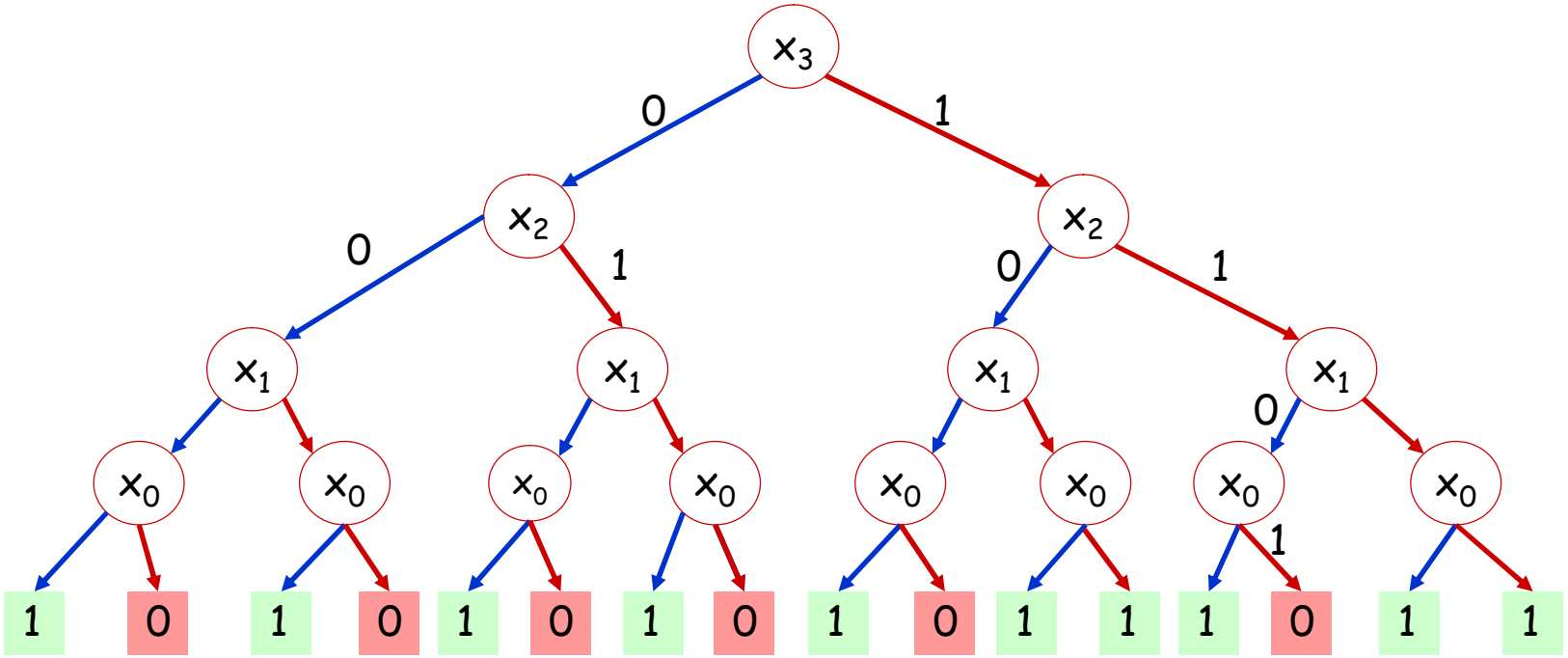
- ... durch einen Entscheidungsbaum



Entscheidungsgraph für

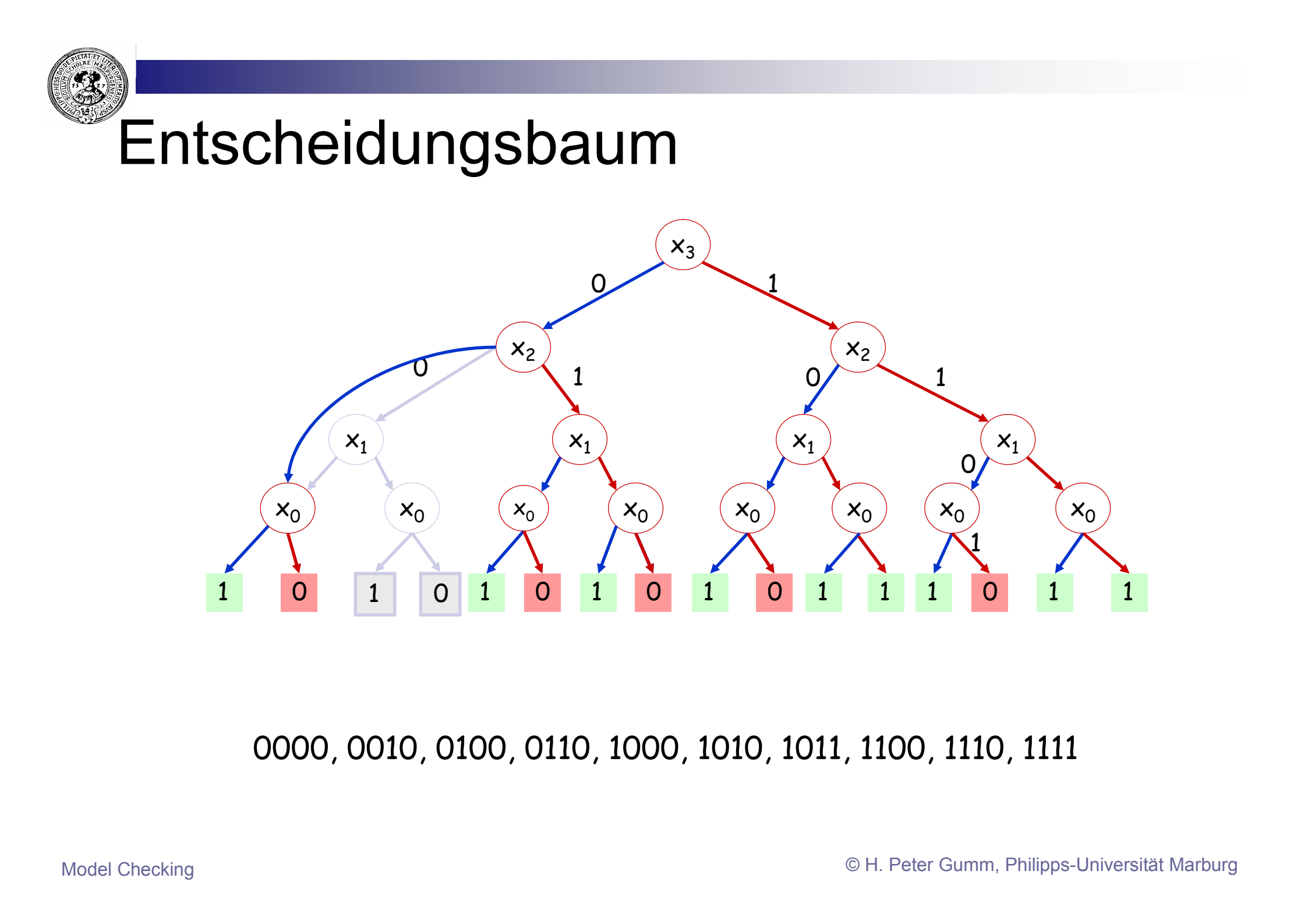
$(x_0 \wedge x_1) \vee (x_2 \wedge x_3)$

Jeder Entscheidungsbaum …



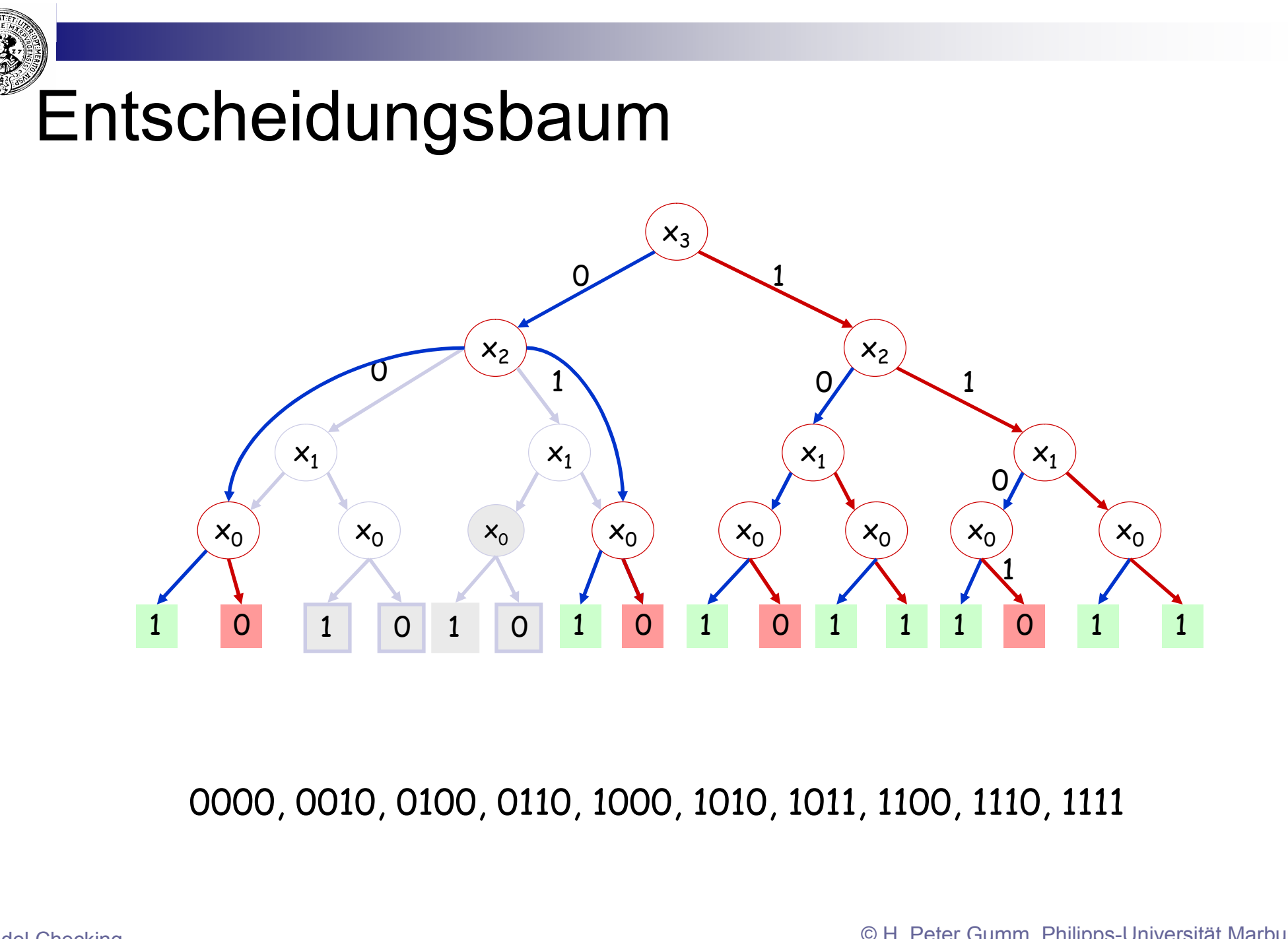
… repräsentiert Boolesche Funktion $f : 2^4 \to 2$:

$$0000, 0010, 0100, 0110, 1000, 1010, 1011, 1100, 1110, 1111 \mapsto 1$$
$$0001, 0011, 0101, 0111, 1001, 1101 \mapsto 0$$



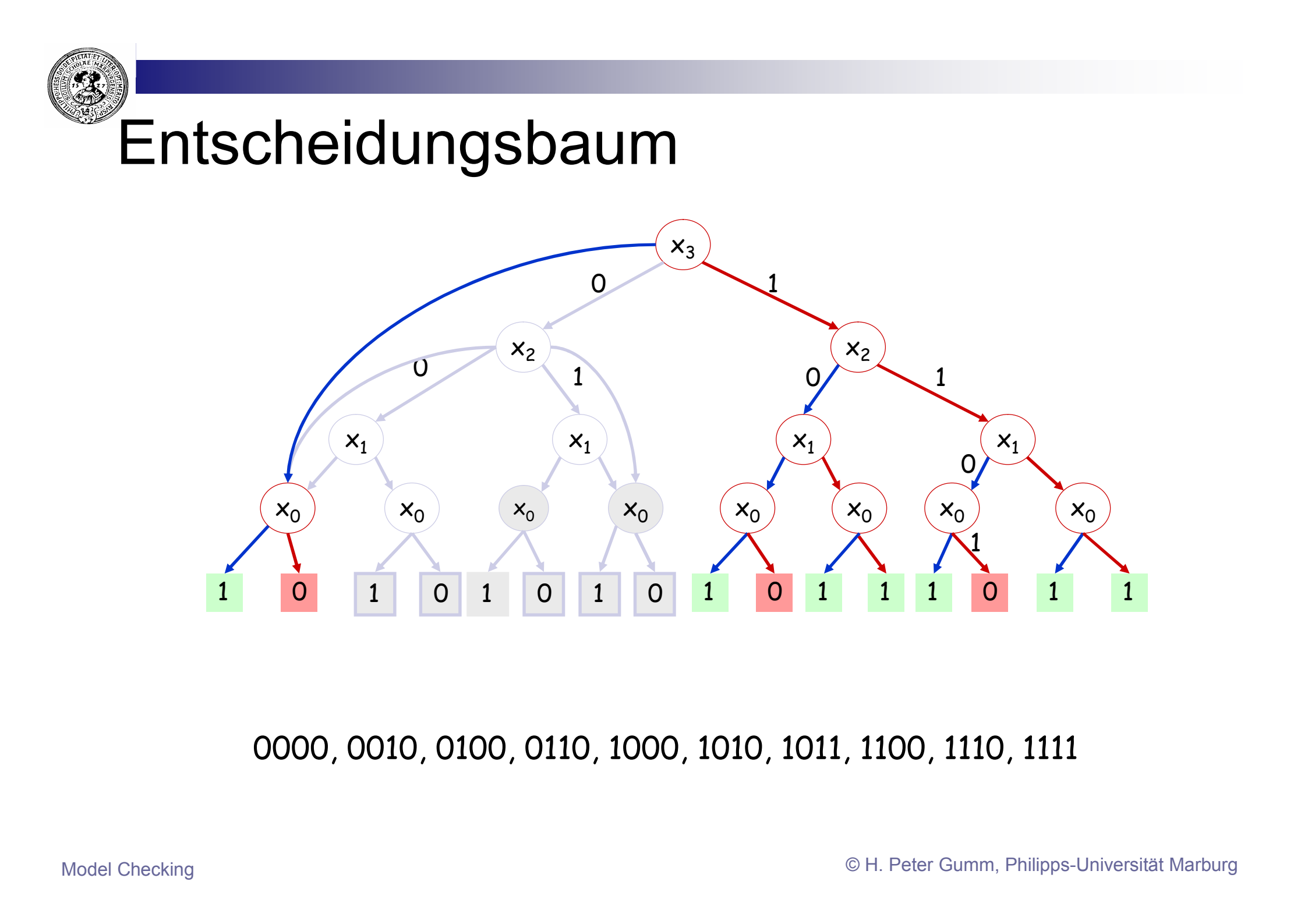
Entscheidungsbaum

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



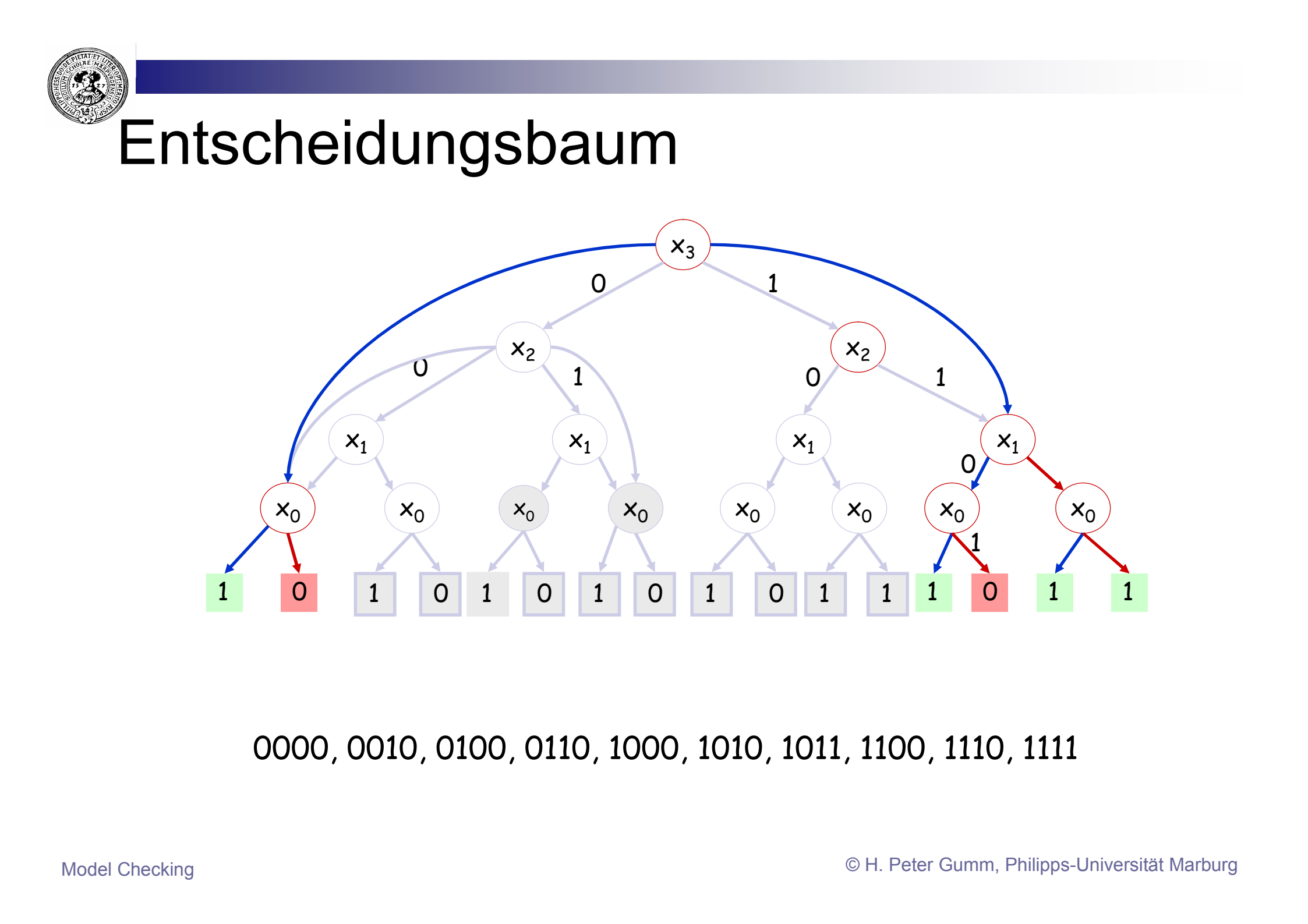
Entscheidungsbaum

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



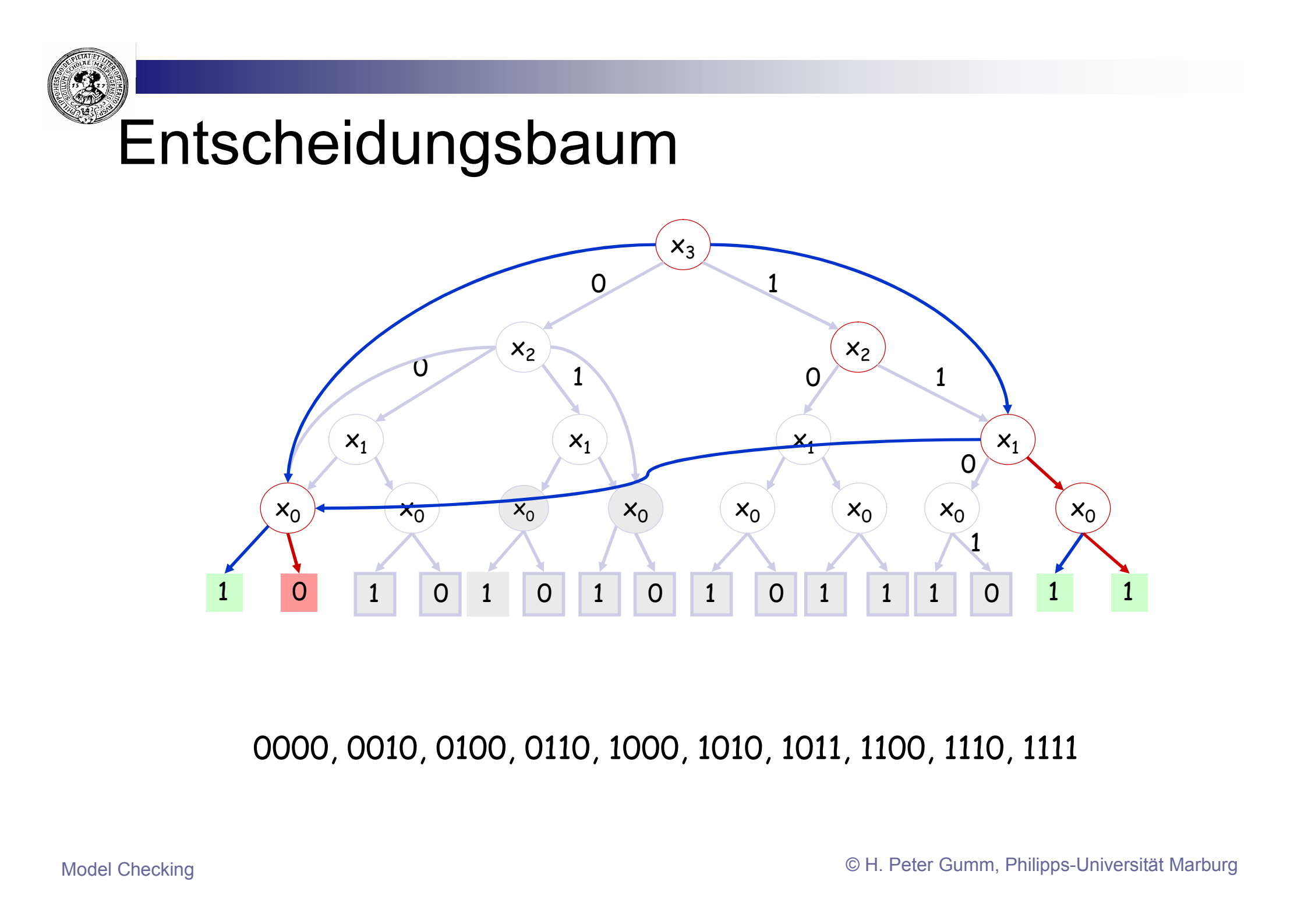
Entscheidungsbaum

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



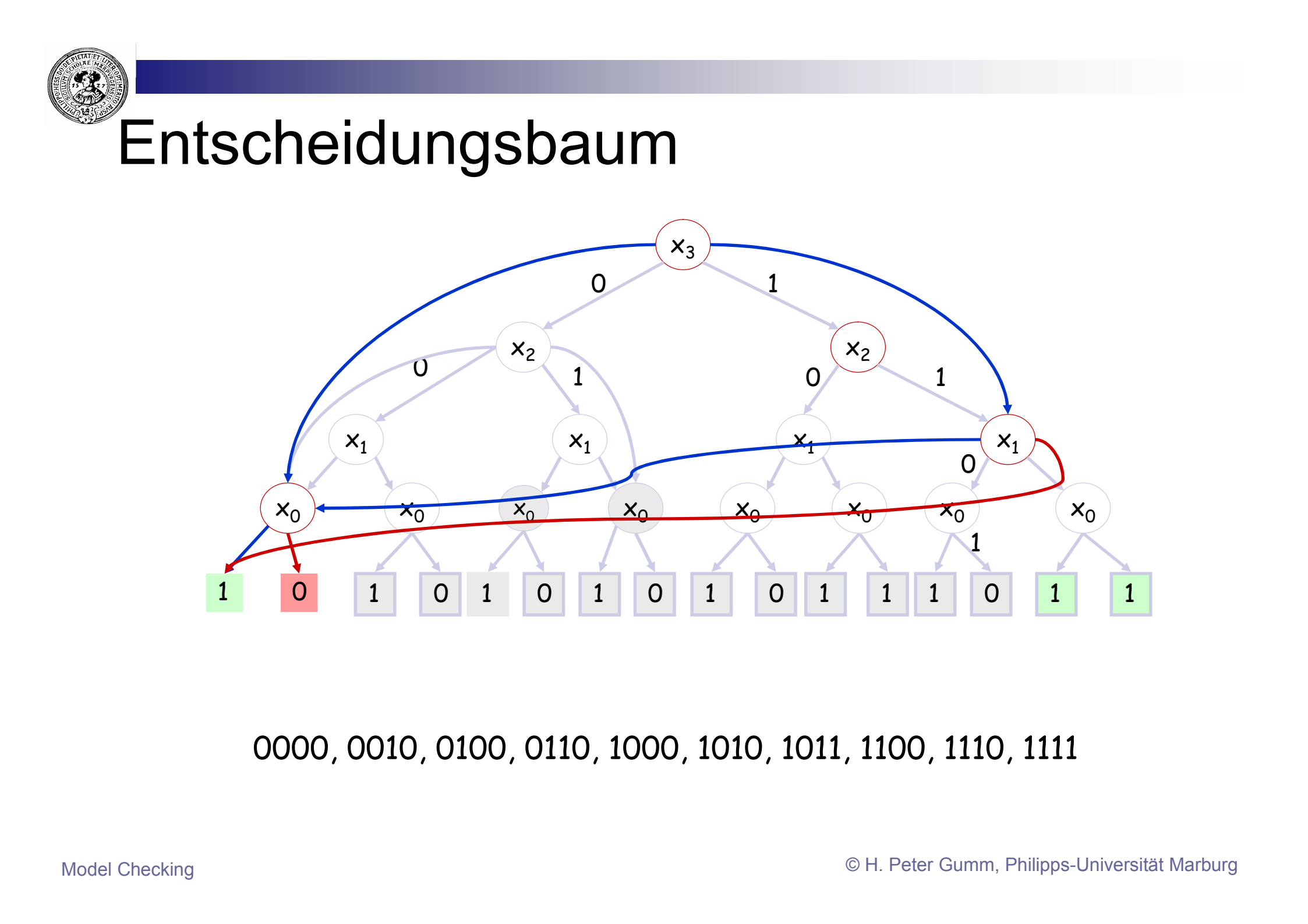
Entscheidungsbaum

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



Entscheidungsbaum

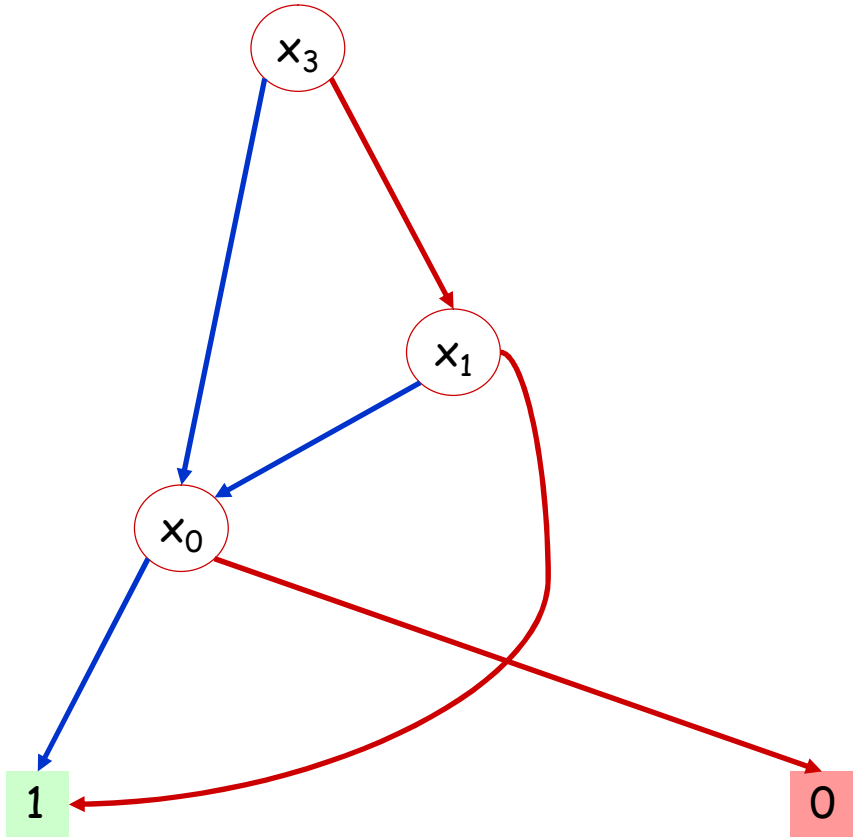
0000, 0010, 0100, 0110, 1000, 1010, 1011, 1100, 1110, 1111



Entscheidungsbaum

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

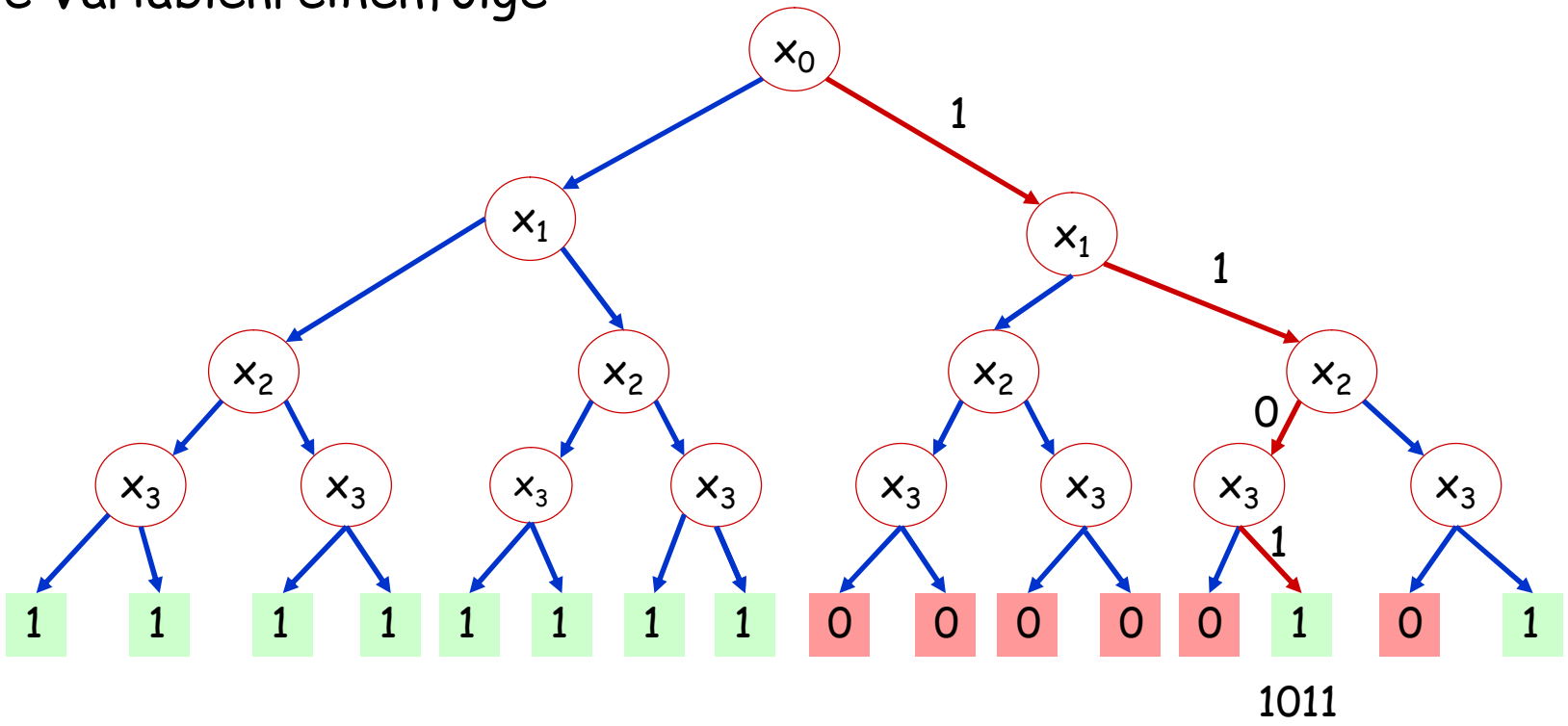
Entscheidungsgraph



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

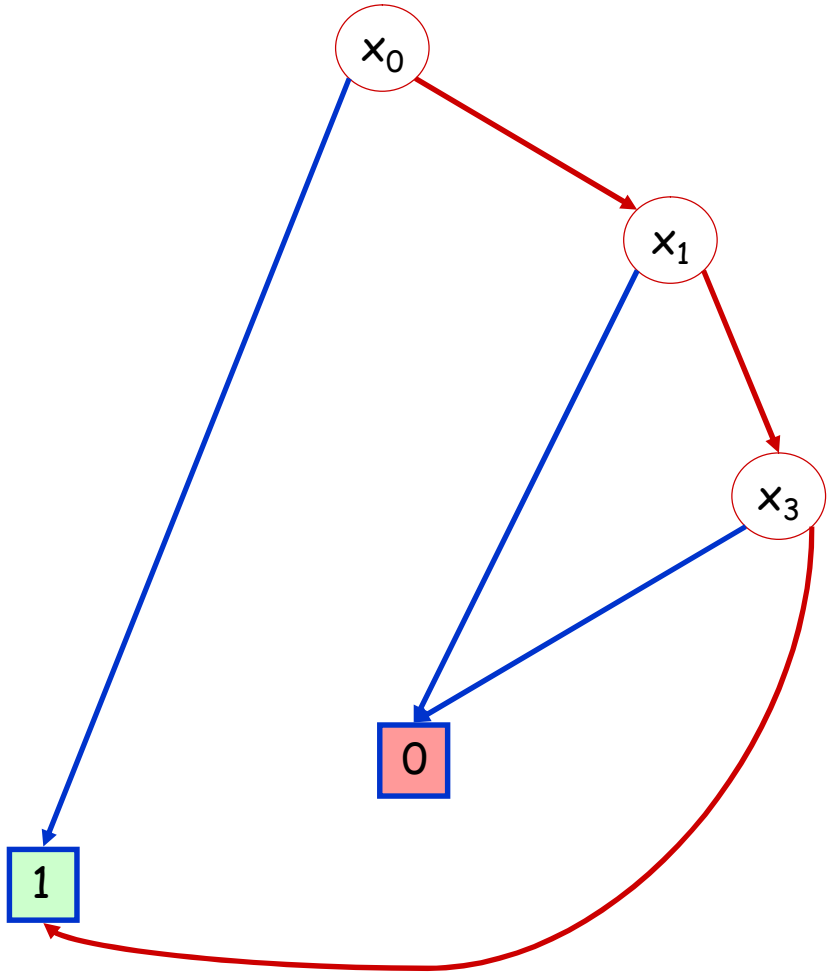
Logische Darstellung:

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

wobei

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

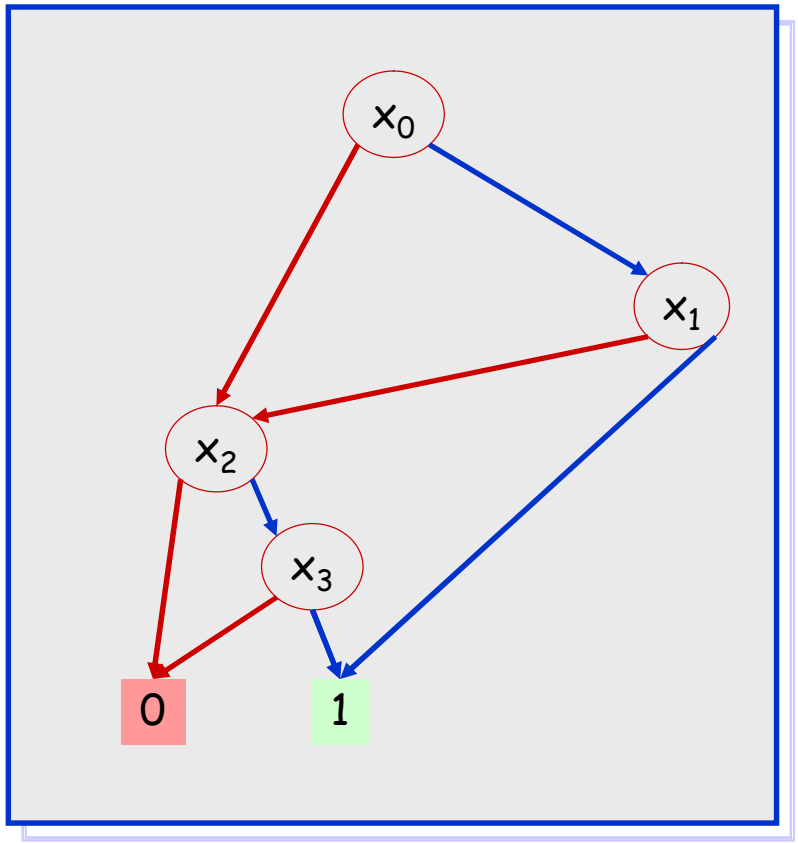
= **if** x **then** y **else** z



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

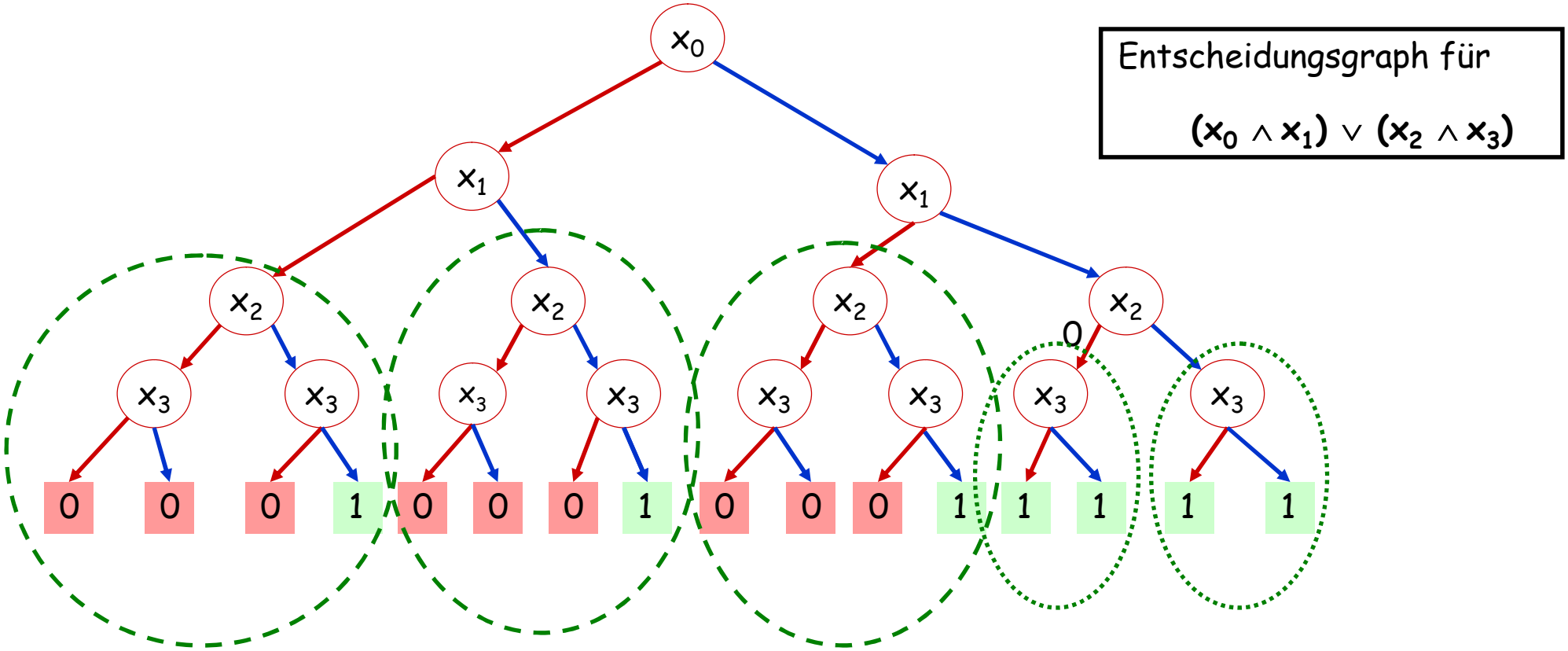


Entscheidungsdiagramm für

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

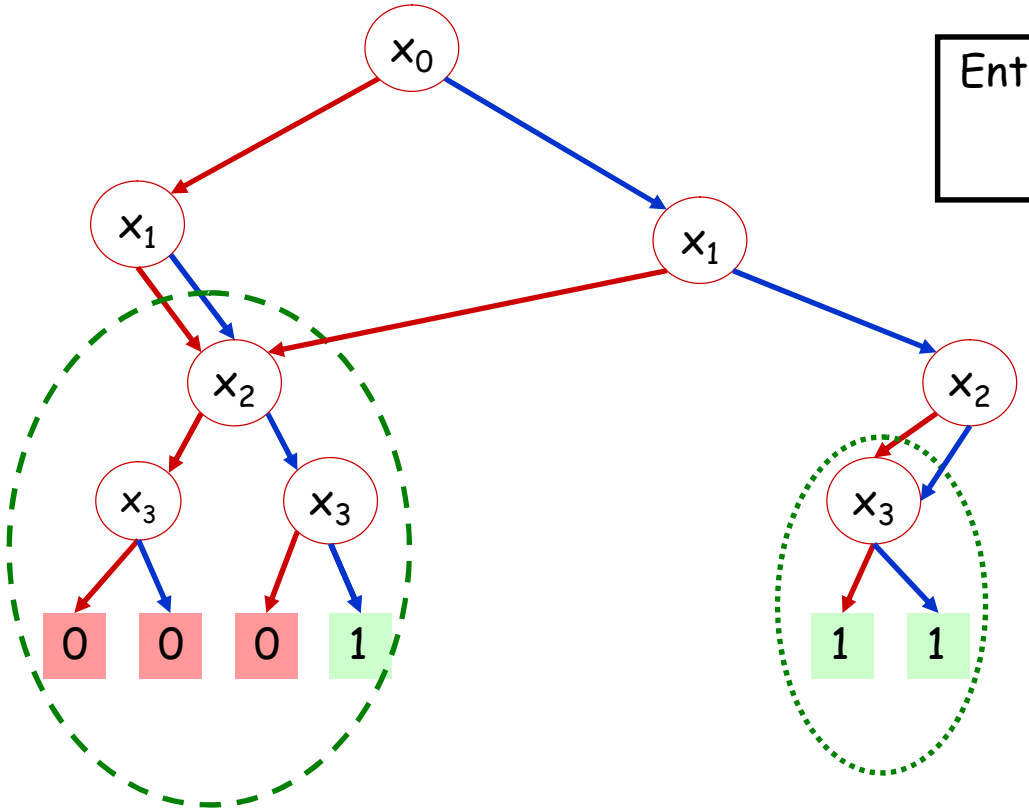
Entscheidungsbaum → BDD

- ... Verschmelzen gleicher Teilbäume



Entscheidungsbaum → BDD

- ... Verschmelzen gleicher Teilbäume

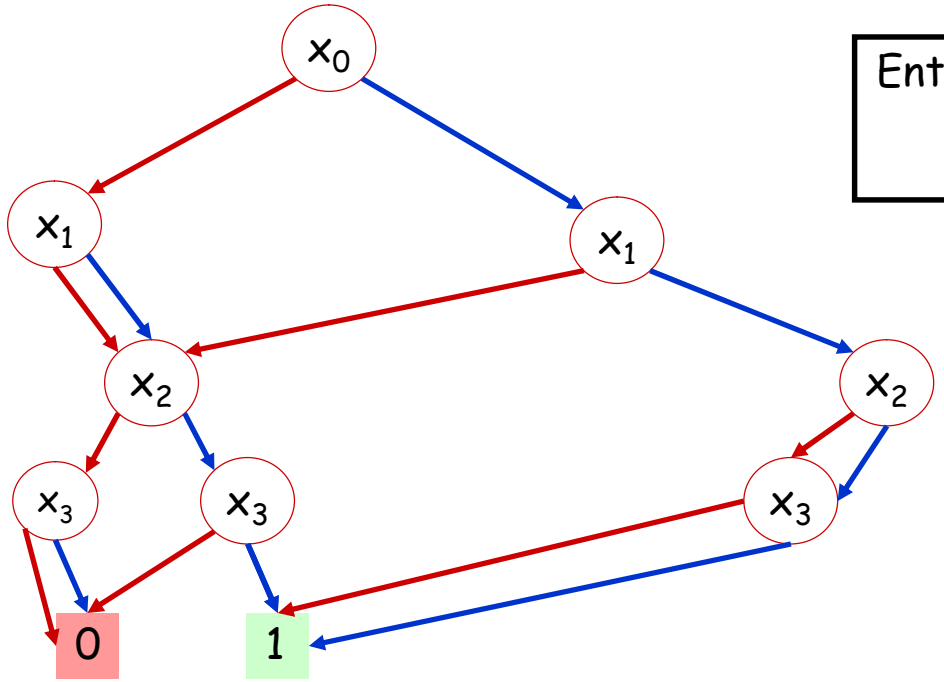


Entscheidungsgraph für

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

Entscheidungsbaum → BDD

- ... Verschmelzen gleicher Teilbäume



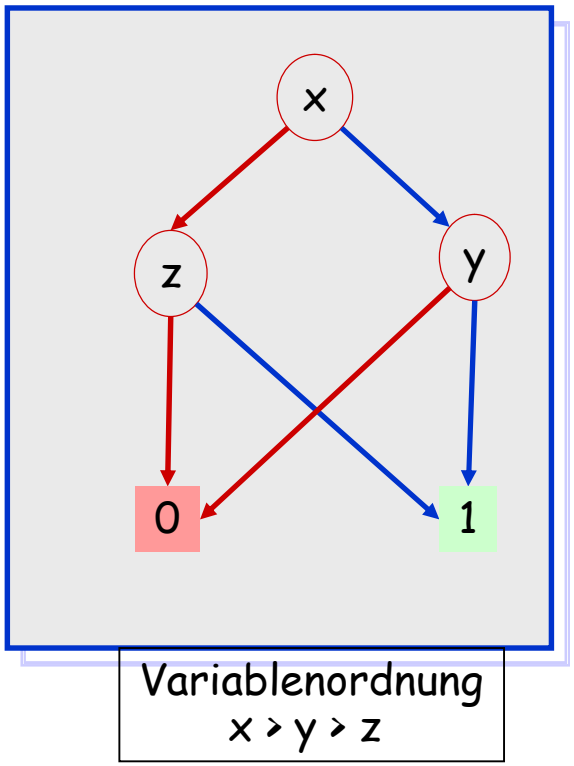
Entscheidungsgraph für

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

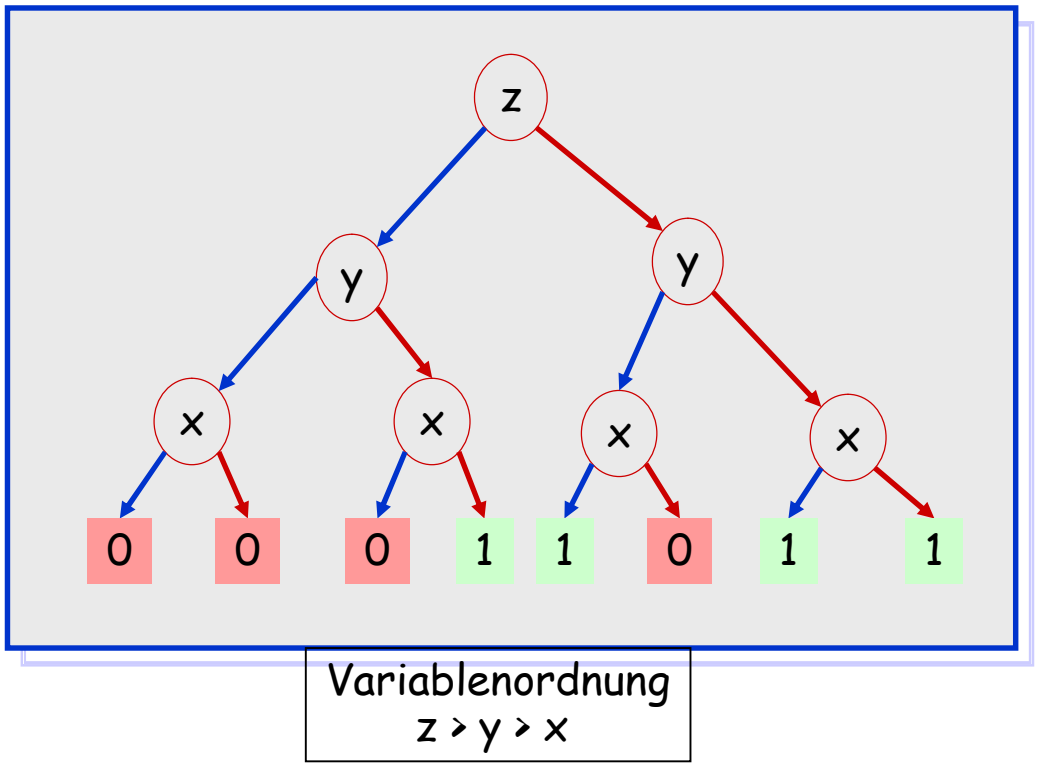
Variablenordnung

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

Beispiel : p(x,y,z) = if x then y else z



Variablenordnung
x > y > z

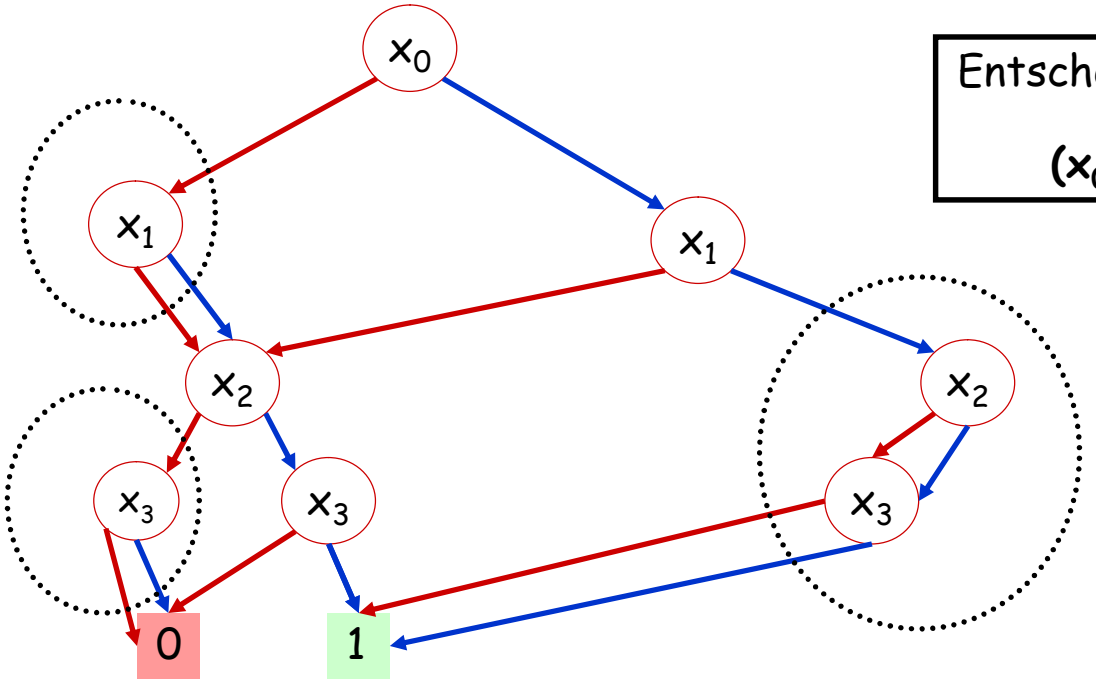


Variablenordnung
z > y > x

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

Entscheidungsbaum → 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



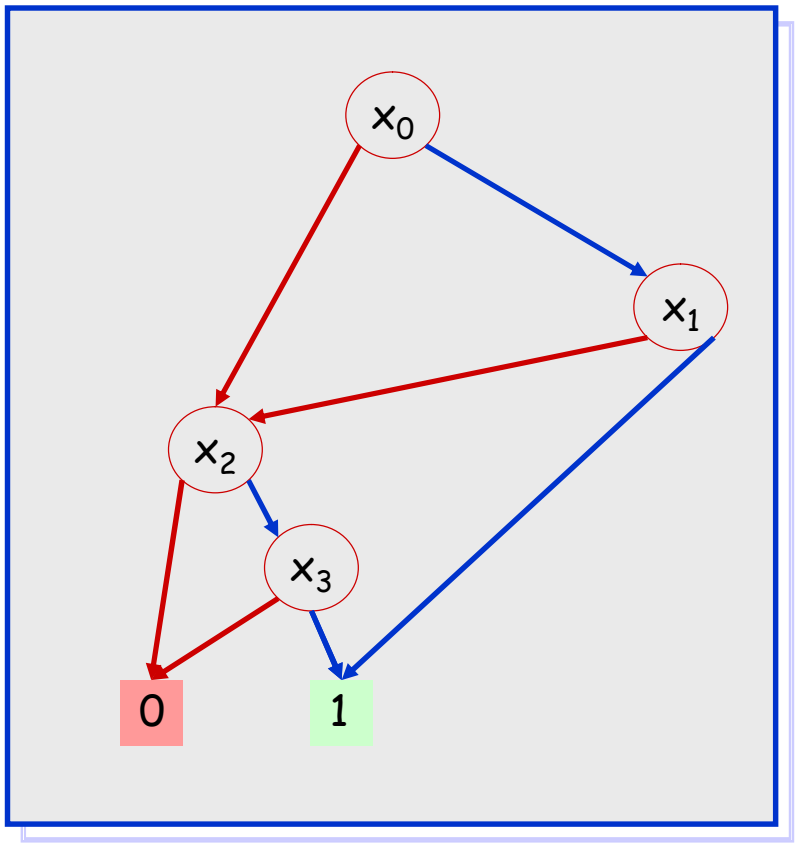
Entscheidungsgraph für

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

Entscheidungsbaum → OBDD

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

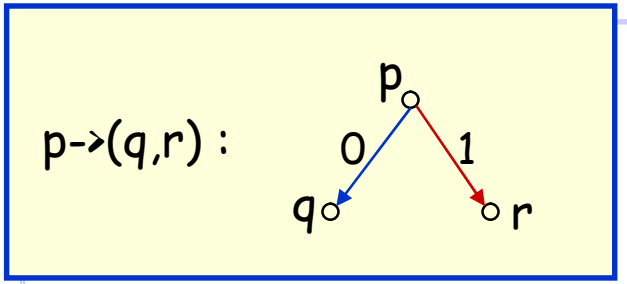
  OBDD



Entscheidungsgraph für

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

OBDDs als Funktionen



Wir benutzen im Folgenden die Notation "p -> (q,r) "
für den Booleschen Term $(p \wedge r) \vee (\neg p \wedge q)$.
In Programmiersprachennotation entspricht dies gerade "p ? r : q"

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

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

$f_0 = \lambda \underline{\mathbf{v}}.\ \mathbf{ff}$,
$f_1 = \lambda \underline{\mathbf{v}}.\ \mathbf{tt}$
$f_n = \lambda \underline{\mathbf{v}}.\ v_i \to (f_l\ \underline{\mathbf{v}}\ ,\ f_r\ \underline{\mathbf{v}})$, falls n =node(x_i,l,r).

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

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

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 $x_1 > x_2 > \ldots > x_k$ gegeben und $n = node(x_i,l,r)$ ein OBDD.
Dann hängt f_n **nicht** von den Variablen vor x_i, also von $x_1,\ldots,x_{i-1}$ 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 Vorraussetzung 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 : $m \in \{0,1\}$, $n=node(x_i,l_i,r_i) \notin \{0,1\}$. f_n hängt nicht von x_i ab, also $f_{li}=f_{ri}$, also $l_i=r_i$ nach Ind.vor., also n kein OBDD.

3.Fall : $m,n \notin \{0,1\}$. Dann gilt $m=node(x_{i1},l_1,r_1)$ und $n=node(x_{i2},l_2,r_2)$.
Wenn $x_{i1} = x_{i2}$, dann folgt $f_{l1}=f_{l2}$ und $f_{r1}=f_{r2}$. Nach Ind. Vorraussetzung folgt $l_1=l_2$ und $r_1=r_2$, also m=n.
Wenn $x_{i1} \neq x_{i2}$, dann gilt o.B.d.A. $x_{i1} > x_{i2}$. Nach dem Lemma hängt dann aber f_n nicht von x_{i1} ab, folglich kann auch f_m nicht von x_{i1} abhängen. Es folgt also $f_{l1} = f_{r1}$ und, wieder nach Ind.Vorraussetzung $l_1=r_1$. Dann war aber m kein OBDD !

Operationen auf OBDDs

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

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

obdd(**g**(**p**,**q**)) = **apply**(**g**,obdd(**p**),obdd(**q**))

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

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

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

Wenn g nicht von x abhängt, gilt :

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

Apply

Seien p_1,p_2 durch OBDDs $\underline{p_1}$ bzw. $\underline{p_2}$ 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: $p_1 < p_2 \;:\Leftrightarrow\; p_1=\text{node}(v_1,l_1,r_1) \wedge p_1=\text{node}(v_2,l_2,r_2) \wedge v_1 < v_2$

mk_obdd ist eine Funktion, die aus einer Variablen und zwei OBBDS 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 :

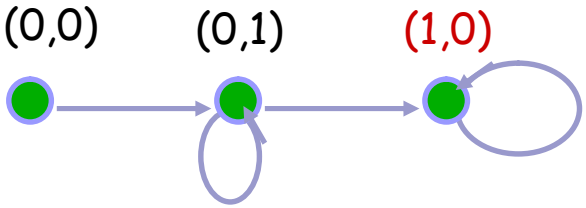
$\exists$ x. node(y,l,r) = **if** x = y **then** l $\vee$ r
else mk_obdd(node(y, $\exists$ x.l , $\exists$ x.r)

Für einen Vektor $\underline{x}$ von Variablen soll andEx($\underline{x}$,p1,p2) = obdd($\exists \underline{x}.(p_1 \wedge p_2)$ gelten:

```
andEx(x, p1, p2) = if p1, p2 ∈ { 0,1 } then p1 ∧ p2
                   else if p1 > p2 then
                          let node(y, l, r)= p1
                          in if y ∈ x then andEx(x,l, p2 ) ∨ andEx(x, r, p2 )
                             else mk_obdd(y,andEx(x,l,p2),andEx(x,r,p2))
                   else if p1 < p2 then
                          ... symmetrisch ...
                   else                                  % gleiche Variable x !
                          let node(y,l1,r1) = p1
                              node(y,l2,r2) = p2
                          in if y ∈ x then andEx(x,l1, l2 ) ∨ andEx(x, r1, r2 )
                             else mk_obdd(node(y,andEx(x,l1,l2),andEx(x,r1,r2))).
```

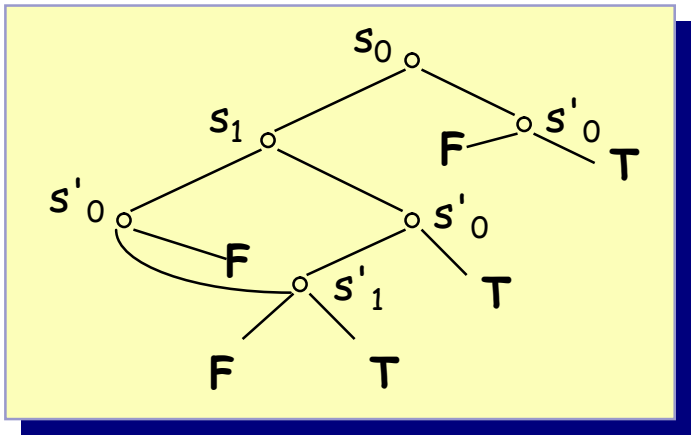
Beispiel

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

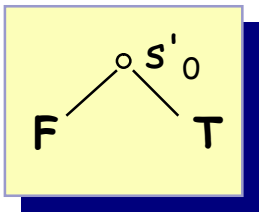


R^{-1} (**s=(1,0)**) = $\exists s'$. $R(s,s') \wedge s'=(1,0)$ führt zu den OBDDs *(siehe späteres Kapitel)*

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

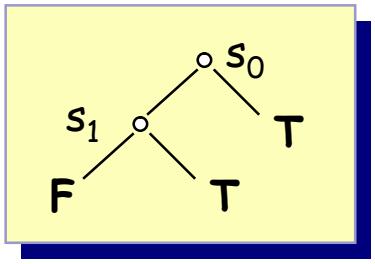


$\wedge$



)

$=$



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

Anwendung von AndExists

Der AndExists-Algorithmus ist nicht nur bei der Berechnung von **EX**p nützlich.

f [v ← 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 :

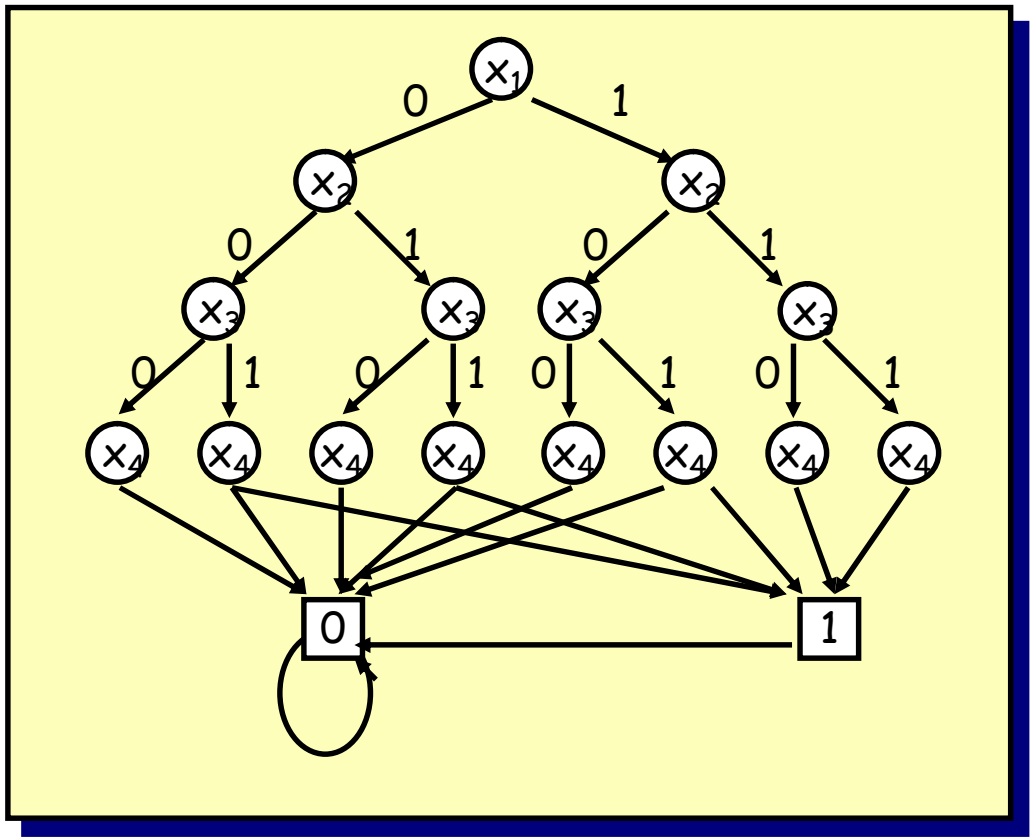
$$(\lambda v.\ f)\ w = f[v \leftarrow w] = \exists v.(v = w \wedge f)$$

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

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

BDDs und Automaten

Sei $p(x_1,\ldots,x_n)$ ein Boolescher Term, und $L = \{ (b_1,\ldots,b_n) \mid b_i \in \{0,1\}, p(b_1,\ldots,b_n)=1 \}$, 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 1 als einzigen terminalen Knoten und setzen fehlende Pfeile auf 0.

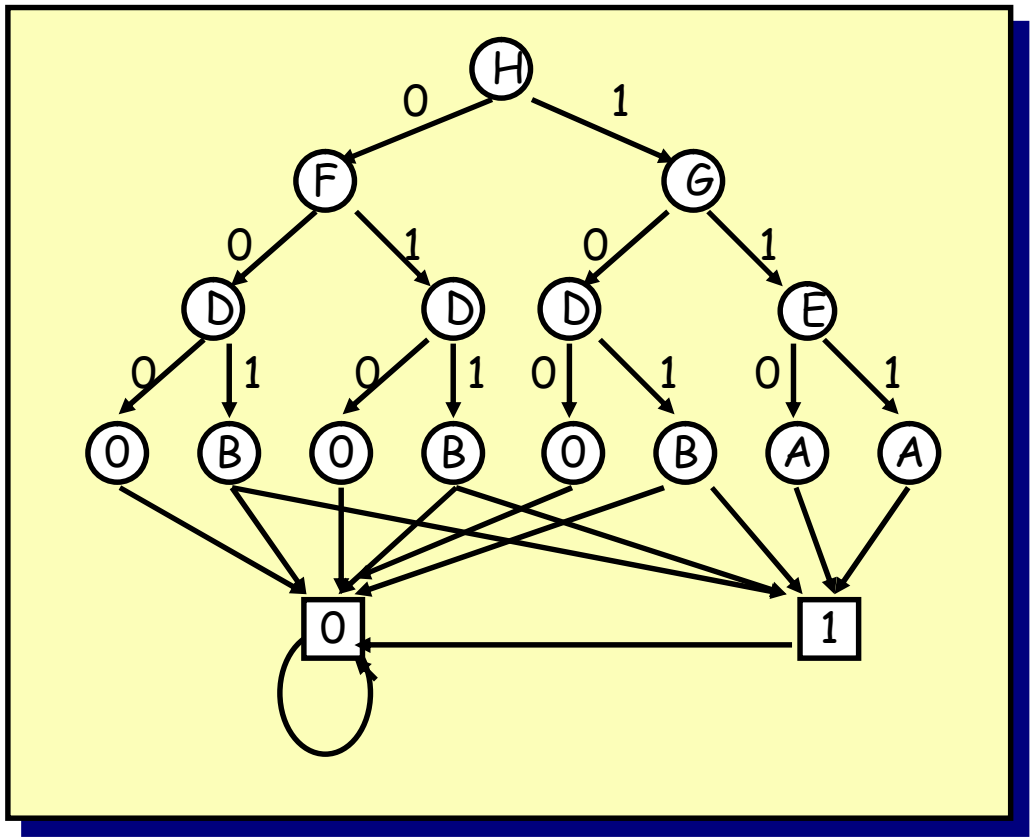


Automat für

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

Minimierung

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

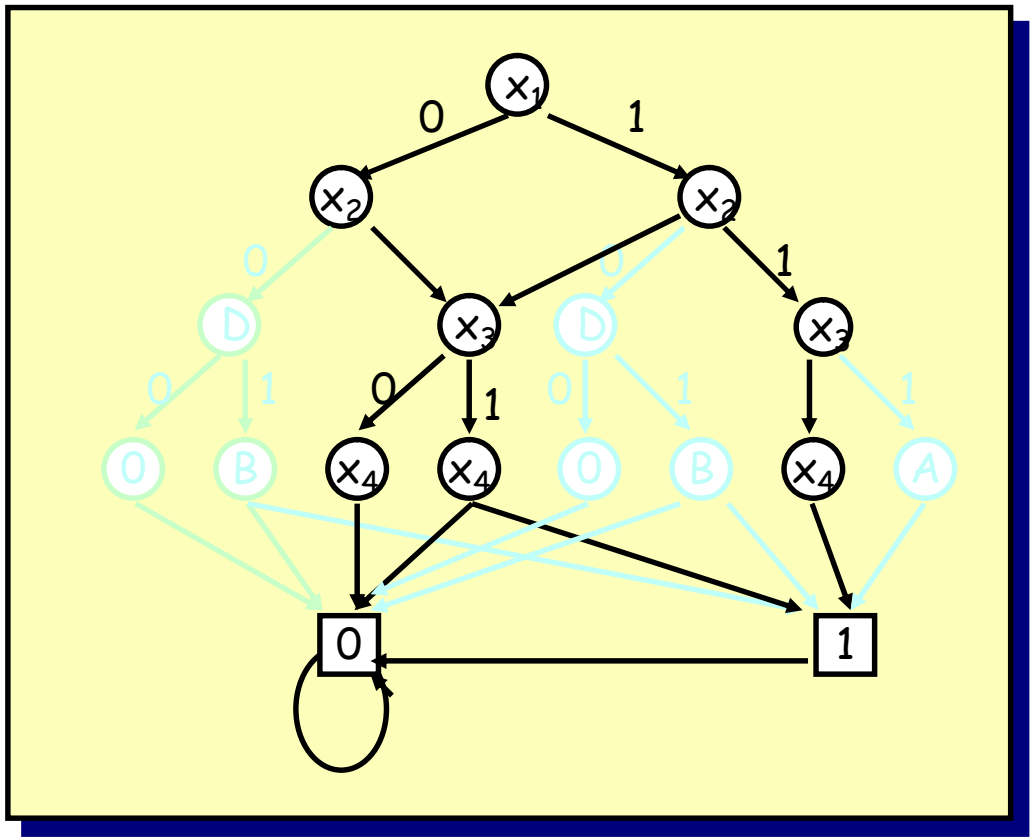


Minimaler Automat für

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

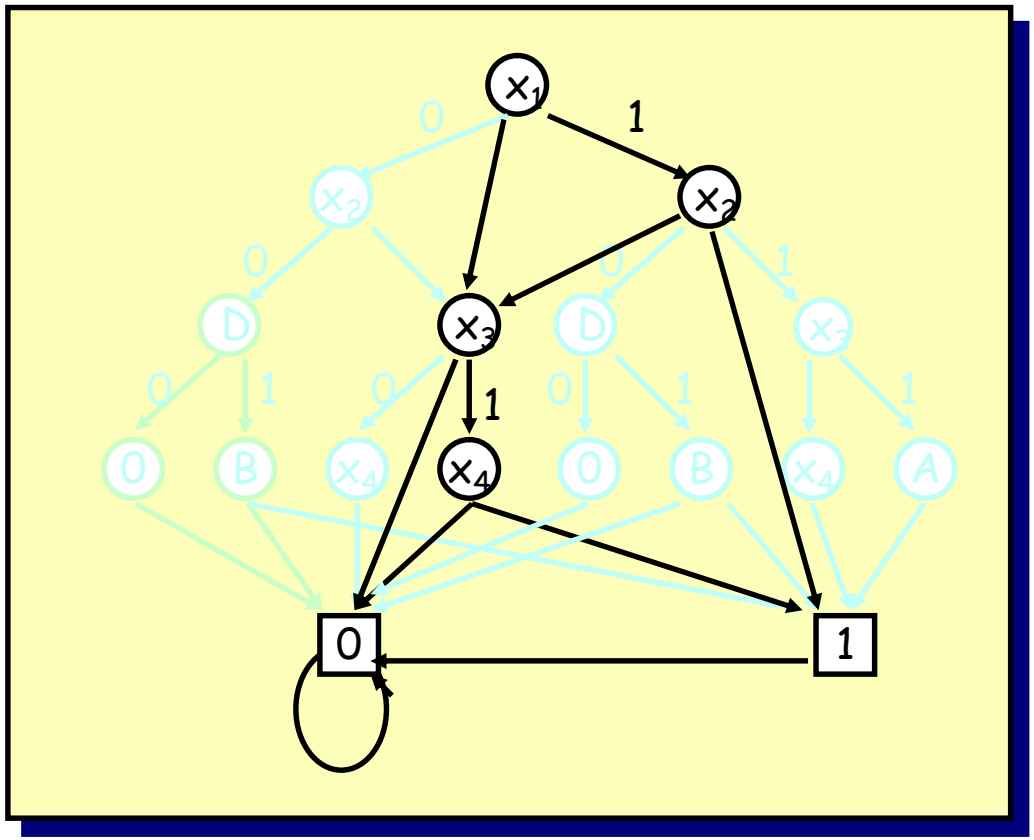
Klasseneinteilung.
Knoten mit derselben Aufschrift gehören zur selben Klasse.

Minimalautomat



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

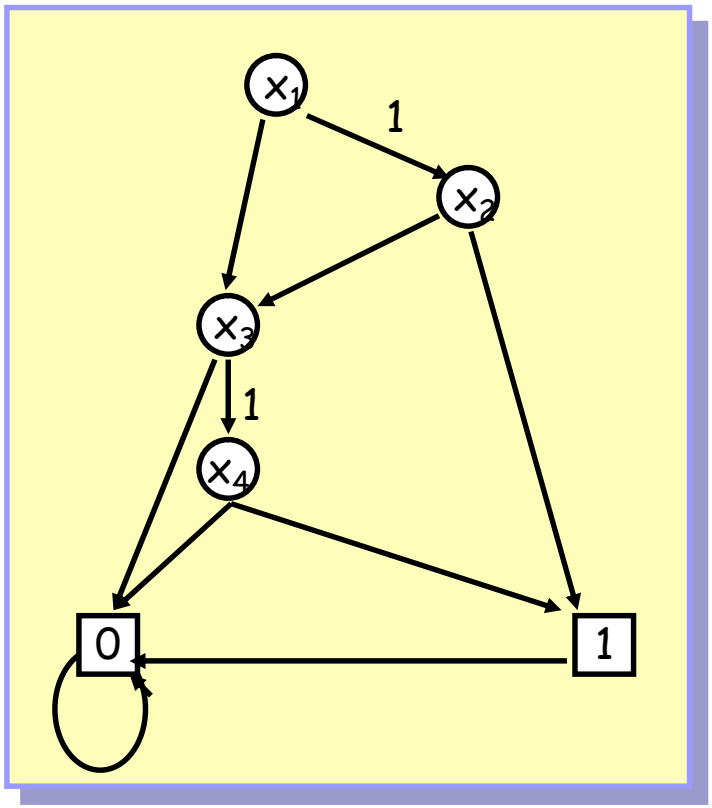
Ordnung der Zustände

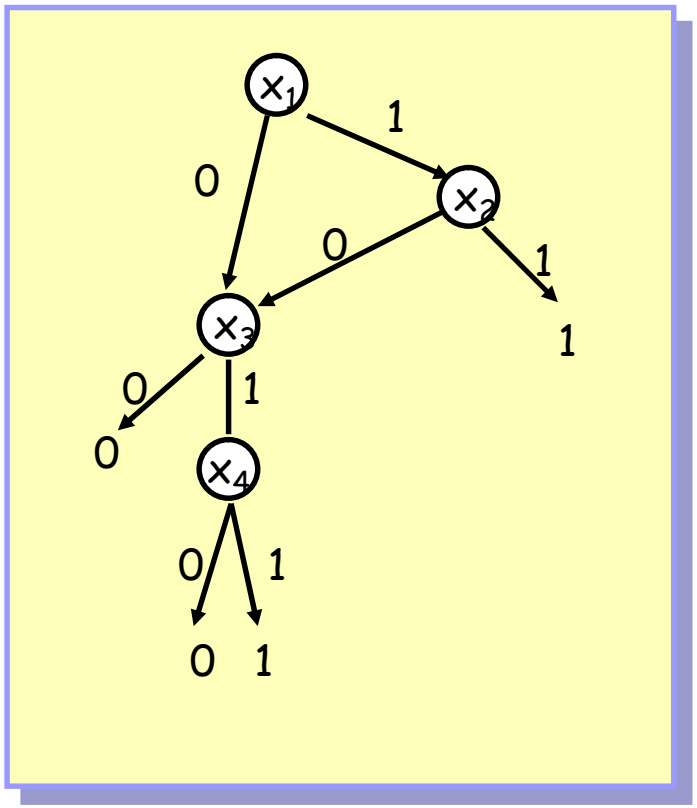


Jeder Zustand, bei dem alle ausgehenden Kanten das gleiche Ziel haben, wird entfernt.

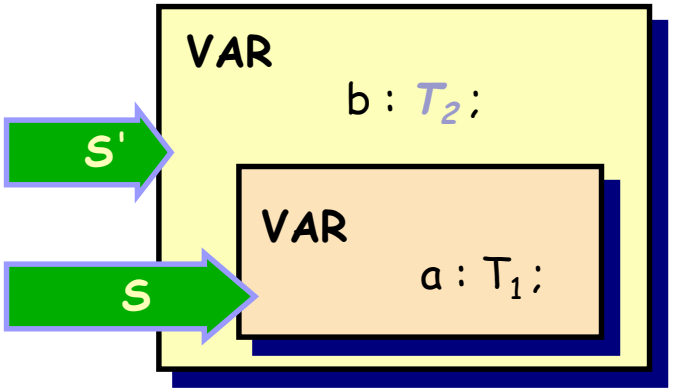
Minimaler Automat und OBDD

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

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

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

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

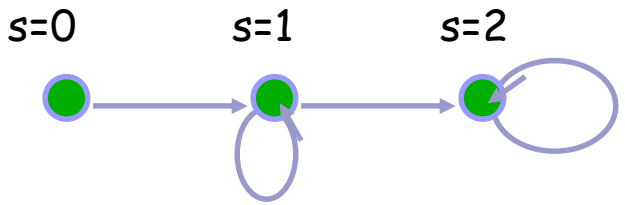
Ist R unabhängig von S_2, dann kommen in der OBDD-Darstellung von **R** die Variablen $y_1,\ldots,y_r$ nicht vor.

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

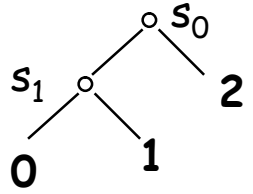
VAR x : a .. b

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 :

$s_0 \rightarrow (s_1 \rightarrow (\ \mathbf{0}, \mathbf{1}\ ) , \mathbf{2}\ )$

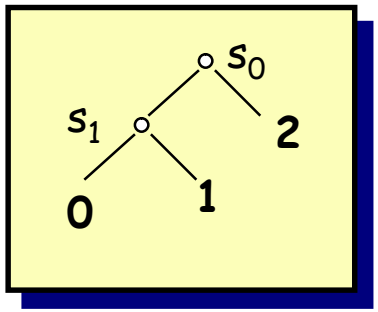
3. Als Mengen Boolescher Vektoren gleicher Länge
0 = { (0,0) } **1** = { (0,1) } , **2** = { (1,0) , (1,1) }

Codierung von *Init* als OBDD

Programmtext :

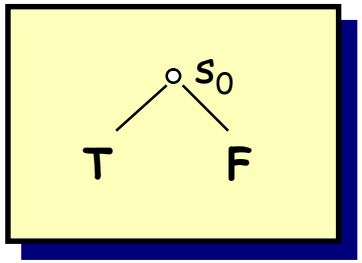
```
init(s) := { 0, 1 }
```

Codierung der Variablen **s** :



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

BDD für "**s in { 0 , 1 }**" :

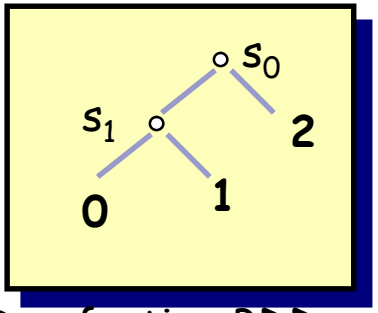


Codierung von R als OBDD

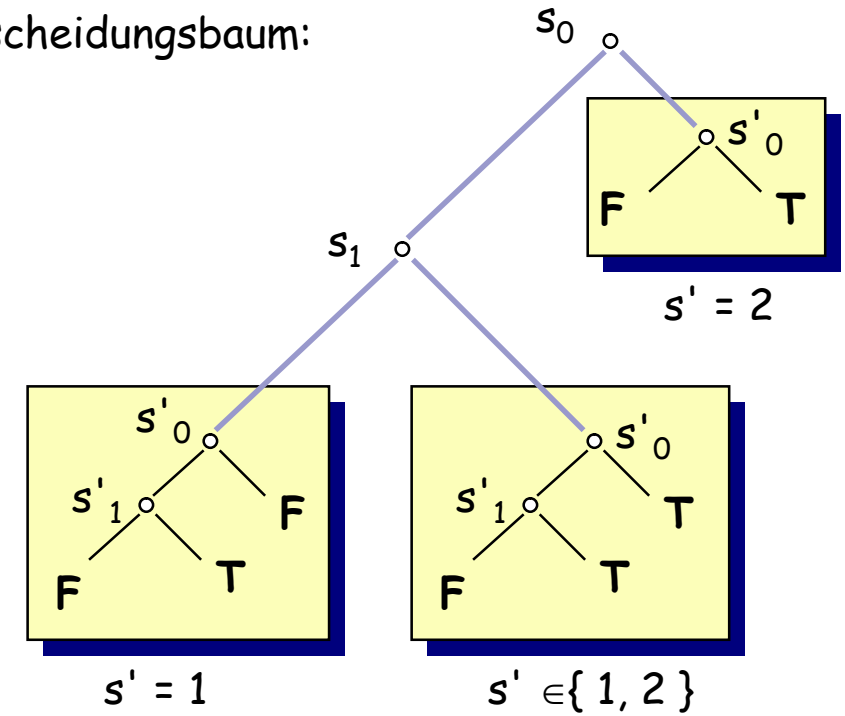
1. Programmtext :

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

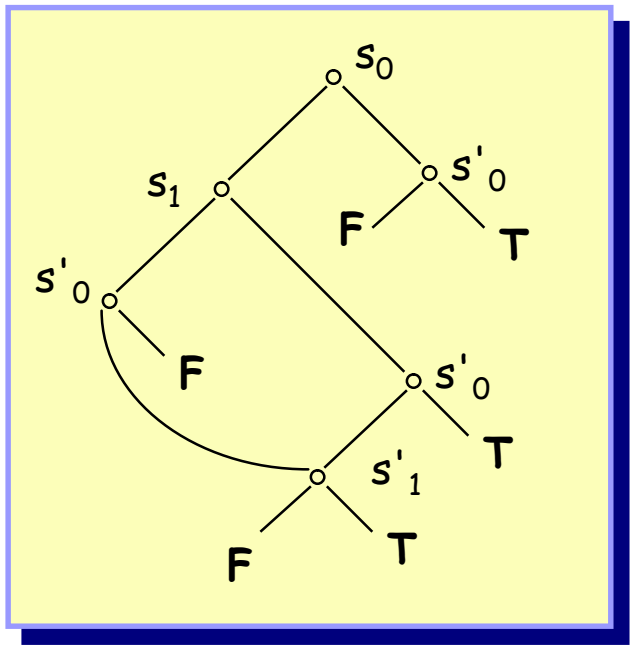
2. Codierung der Variablen **s** :



3. Entscheidungsbaum:



4. Der fertige BDD :



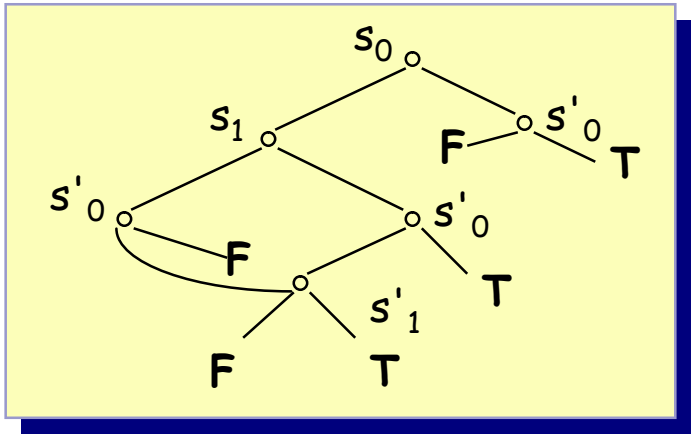
R^{-1} als Funktion

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

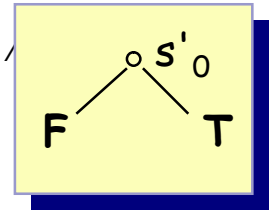
Insbesondere erhalten wir in unserem Beispiel :

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

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

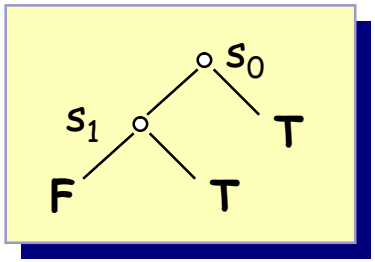


$\wedge$



)

$=$



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

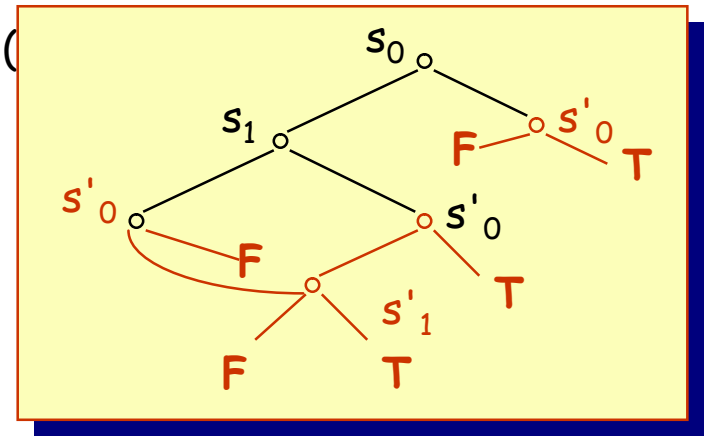
wp(R,-) als Funktion

$$wp(R,-) = AX = \lambda p.\ \lambda(s_0,s_1).\forall (s'_0, s'_1).\ R \Rightarrow p'$$

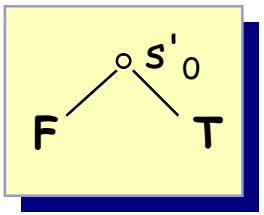
Insbesondere erhalten wir in unserem Beispiel :

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

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

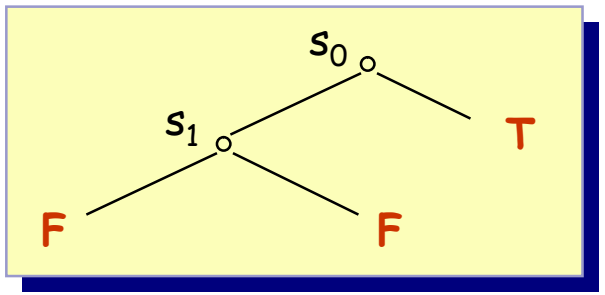


$\Rightarrow$

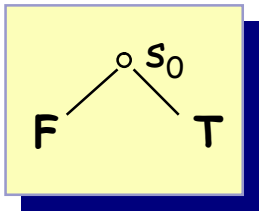


)

$=$



$=$



$= (s = 2)$

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

Berechnung der ersten SPEC

AF AG s=2

1. Berechne

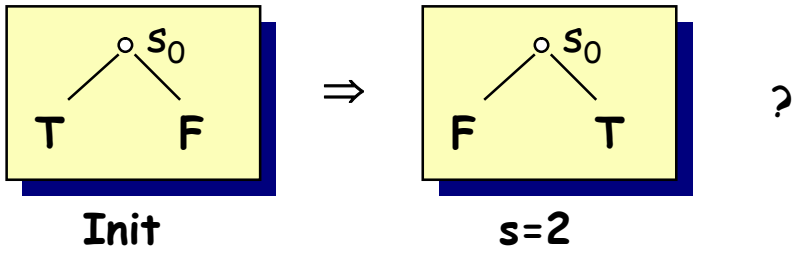
AG (s=2) $= \nu Y.\ s=2 \wedge \mathbf{AX}\ Y$
$= s=2 \wedge \mathbf{wp}(R,s=2) \wedge \ldots$
$= s=2 \wedge s=2 \wedge \ldots$
$= s=2.$

2. Berechne

AF AG (s=2) $= \mathbf{AF}\ s=2$
$= \mu Y.\ s=2 \vee \mathbf{AX}\ Y$
$= s=2 \vee \mathbf{wp}(R,s=2) \vee \ldots$
$= s=2.$

3. Prüfe

R |= **Init** $\Rightarrow$ **AF AG** s=2 ?



Nein !

Berechnung der zweiten SPEC

EF AG s=2

1. Berechne

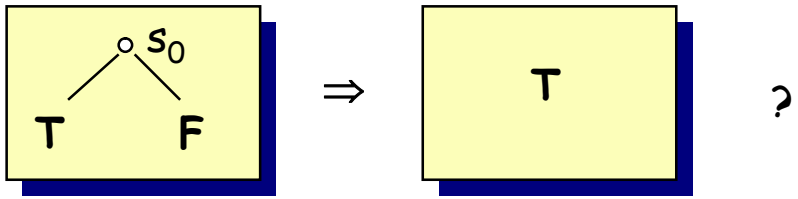
   AG (s=2) = ... wie vorher ..
   = s=2.

2. Berechne

   EF AG (s=2) = **EF** s=2
   = μY. s=2 $\vee$ **EX** Y
   = s=2 $\vee$ $\mathbf{R^{-1}}$(s=2) $\vee$ $\mathbf{R^{-1}}$(...) ...
   = s=2 $\vee$ (s=1 $\vee$ s=2) $\vee$ $\mathbf{R^{-1}}$(s=1 $\vee$ s=2) $\vee$...
   = **T**

3. Prüfe

   R |= **Init** $\Rightarrow$ **EF AG** s=2 ?



Init

Ja !

Formale Semantik

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

Semantische Funktionen :

$[\| \ldots \|]_I$	:	Initialbedingung
$[\| \ldots \|]_T$	:	Transition **R**.
$[\| \ldots \|]_S$	:	Spezifikation
$[\| \ldots \|]_E$	:	Ausdrucksemantik

Wenn der Index fehlt, so gilt die angegebene semantische Klausel für alle semantischen Funktionen. Alle semantischen Funktionen, bis auf $[\| \ldots \|]_E$ liefern als Ergebnis einen OBDD. $[\| \ldots \|]_E$ 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) $[\| \text{head ; rest} \|] = [\| \text{head} \|] \wedge [\| \text{rest} \|]$.

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) $[\![\texttt{MODULE main}]\!] = tt.$

Variablen

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

Variablen sind von Hause aus Boolesch :

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

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

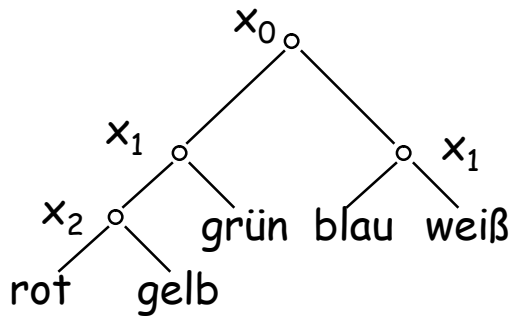
(4) $[\!|\ \text{VAR}\ x : a\ ..\ b\ |\!]_E = (\ x=\text{encode}(x,0,a\ ..\ b)\ )$

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 $[\!| \ldots |\!]_I$ interessant, alle anderen semantischen Funktionen liefern tt :

(5) $[\!|$ ASSIGN init(x) := e $|\!]_I := x \in [\!|$ e $|\!]_E$.

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

(6) $[\!|$ ASSIGN x := e $|\!]_T := x \in [\!|$ e $|\!]_E$.

(7) $[\!|$ 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) $[|\ c\ |]_E = \{\ c\ \}$, falls c eine Konstante ist.

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

Falls • irgendeine Operation des Datentyps ist, also
• $\in$ { +, -, *, /, mod, =, <, <=, >, >=, |, &, ->, <-> }
dann wird • zu den Blättern propagiert und dort ausgeführt :

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

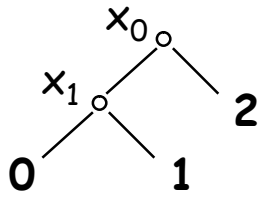
wobei die Propagation nach folgenden Regeln stattfindet :

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

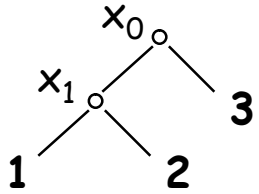
und folgende Vereinfachungen Anwendung finden :

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

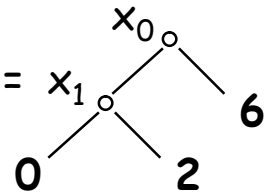
$[|x|]_E$ =



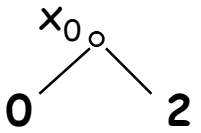
$[|x+1|]_E$ =



$[|x*(x+1)|]_E$ =



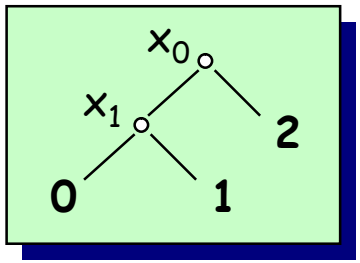
$[|(x*(x+1)-2*x)/2|]_E$
=



Berechnung der Transitionssemantik

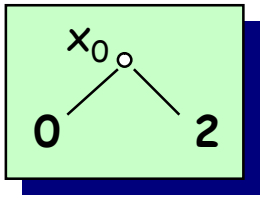
1. Aus der Deklaration "VAR x : 0 .. 2 ;" erhalten wir :

$[|x|]_E$ =



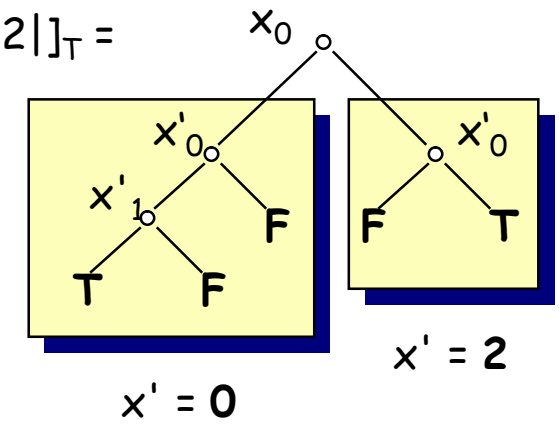
2. Für den Ausdruck " (x*(x+1)-2*x)/2" ergibt sich :

$[|(x*(x+1)-2*x)/2|]_E$ =



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

$[|next(x) := (x*(x+1)-2*x)/2|]_T$ =



Nichtdeterminismus

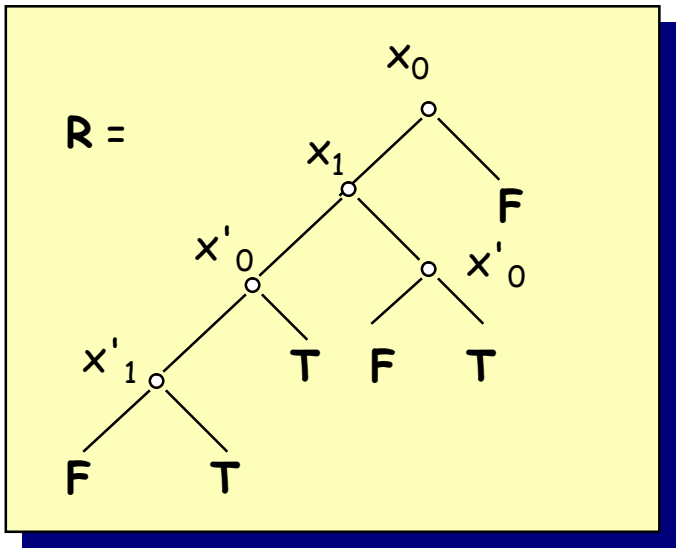
SMV gestattet nichtdeterministische Transitions-Relationen, weil der Wert eines Ausdruckes eine ***Menge*** von Datenobjekten sein kann. Allerdings muß die linke Seite einer Zuweisung immer entweder ***x*** oder ***next(x)*** für eine Variable x sein.

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) $[\![\textbf{FAIRNESS}\ e \,]\!]_F = \textbf{GF}\ [\![e \,]\!]_E$)

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 $\mathbf{A}_C$ und $\mathbf{E}_C$ ersetzt werden.

Spezifikationssemantik und Korrektheitsbedingung

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

(12) $[|\ \textbf{SPEC}\ f\ |]_S := \lambda\, r.\ \mathrm{eval}(f, r).$

Wir erinnern uns :

eval(**EF** g, r) = evalEF(eval(g,r),tt,r)
eval(**E**[f **U**g], r) = evalEU(eval(g,r),eval(f,r),ff),
.. etc. ..

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

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

Module, Parameter, Prozesse

```
MODULE main
VAR
  g1: process inv(g3.out);
  g2: process inv(g1.out);
  g3: process inv(g2.out);
SPEC
  (AG AF g1.out)& (AG AF !g1.out)

MODULE inv(input)
VAR
  out : boolean;
ASSIGN
  init(out) := 0;
  next(out) := ! input
```

Prozesse

Module mit **Parametern**
Aufruf
Vereinbarung

Semantik von Moduldeklaration

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

(14) $[\!|\ \textbf{MODULE}\ m(p_1,p_2,\ldots,p_k)\ \ body\ \ |\!] = (\ m = \lambda\ p_1.\ p_2.\ ..\ \ p_k.\ [\!|\ body\ |\!]\ )\ .$

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

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

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 :

a . (b) = a.b
a.(bc) = a.b a.c
a.c = c , falls c eine Konstante ist.

Mit den spitzen Klammern "<" und ">" *schützen* wir ganze Ausdrücke vor der Punktnotation :

a . < e > = e.

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

(15) [| **VAR** x : $m(e_1,e_2,\ldots,e_k)$ |]
= (x. m $<e_1>$ $<e_2>$ ... $<e_k>$) [running/<running>]) .

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

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

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

Beispiel

(16) [| VAR process x : $m(e_1,e_2,\ldots,e_k)$; rest |]

$= (x.\ m \langle e_1 \rangle \langle e_2 \rangle \ \ldots \langle e_k \rangle)$

$\wedge$ (x.running $\Rightarrow \neg$ blocked $\wedge$ running)

$\wedge$ [|rest |] [blocked / (blocked $\vee$ x.running)]

Die obige Regel ergibt in unserem Beispiel :

[| VAR

process g_1 : $inv(x_1)$;
process g_2 : $inv(x_2)$;
process g_3 : $inv(x_3)$; ... |]

$= \quad g_1.inv \langle x_1 \rangle \wedge g_2.inv\langle x_2 \rangle \wedge g_3.inv\langle x_3 \rangle$

$\wedge\ g_1.running \Rightarrow \neg$ (blocked $\vee\ g_2.running \vee g_3.running) \wedge$ running

$\wedge\ g_2.running \Rightarrow \neg$ (blocked $\vee\ g_1.running \vee g_3.running) \wedge$ running

$\wedge\ g_3.running \Rightarrow \neg$ (blocked $\vee\ g_1.running \vee g_2.running) \wedge$ running

$\wedge$ ...

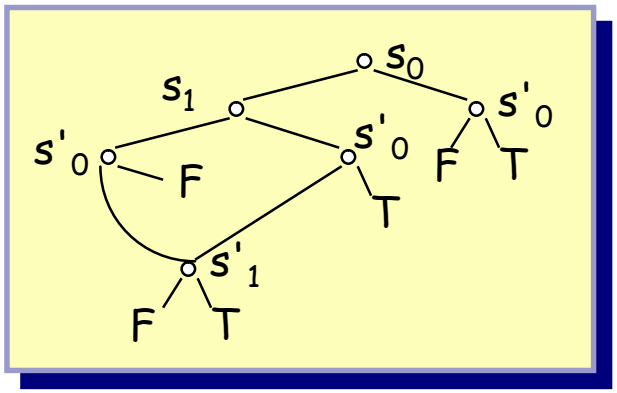
Aufgaben

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

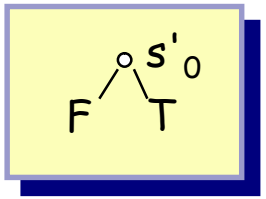
$x_0 \vee x_1 \Rightarrow x_2 \wedge \neg x_3$
$(x_0 \wedge x_3) \vee x_2$
$\neg x_0 \Rightarrow (x_3 \vee x_1) \wedge x_2$

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

A



B



gegeben. Bestimmen Sie die OBBDs 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 $x_1 > \ldots > x_n > x'_1 > \ldots > x'_n$ Ausdrücke der Art

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

bzw.

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

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

4. Der Datentyp Farbe = { rot, gelb, grün } sei folgendermaßen durch Boolesche Vektoren codiert :

rot = (0,0), gelb = (0,1), grün = (1,0)

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

b) Finden ´ Sie den OBDD, der die Relation

R = { (s,s') | if s= rot then s' = gelb
else if s=gelb then s' ∈{ gelb, grün }
else s' = grün }

repräsentiert. Wählen Sie die Variablenordnung

$s_0 > s_1 > s'_0 > s'_1$.