
# Kripke Strukturen und SMV

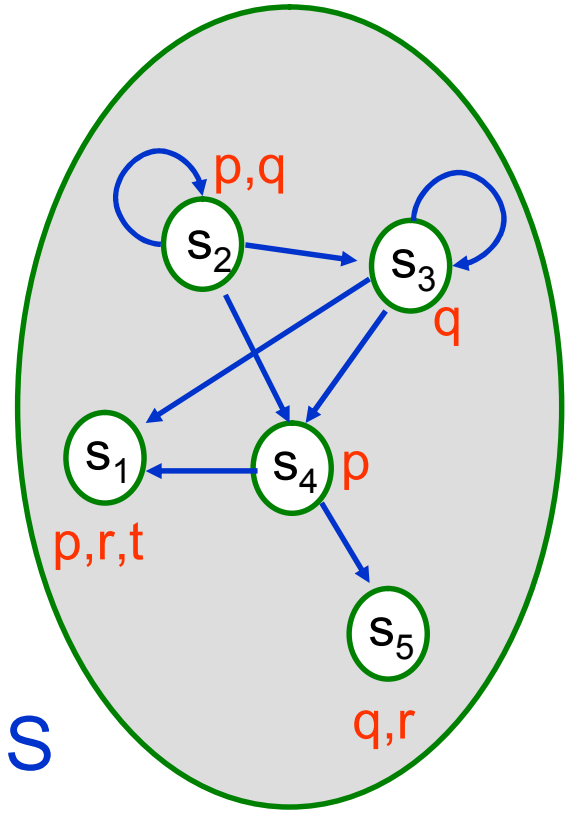
H. Peter Gumm

Philipps-Universität Marburg

Sommersemester 2007

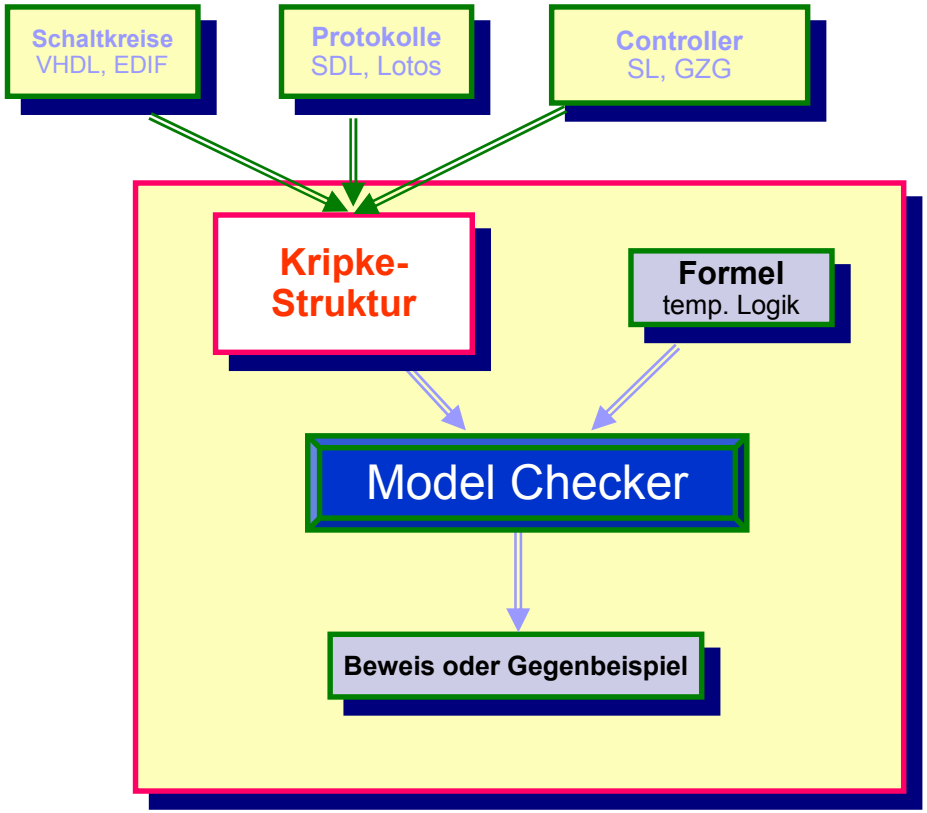
# Kripke Strukturen

- Einfache mathematische Modelle
- Modellieren fast alle Arten von Systemen
  - Automaten
  - Schaltungen
  - Programme
  - Protokolle
- Universelle Zwischenstruktur für Model Checking
- Klare Semantik für temporale Logik
  - Saul Kripke hat sie für diesen Zweck eingesetzt
- Andere Bezeichnung
  - Zustandsübergangsgraph



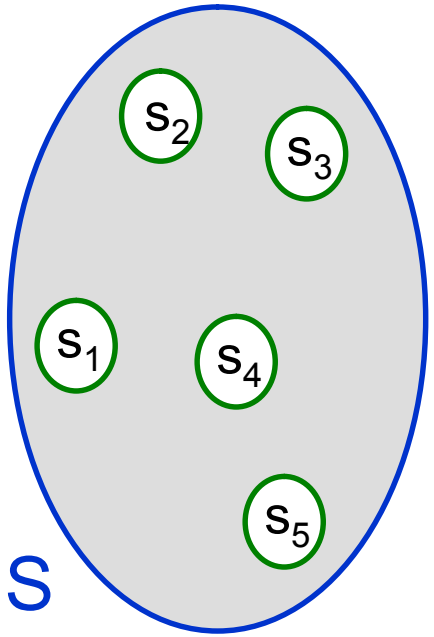
# Kripke Strukturen als universelle Modelle

- Basisstruktur für Model Checker
- Automatische Übersetzungen von anderen Beschreibungssprachen in Kripke Strukturen sind vorhanden
- Wir werden einige Übersetzungen studieren
- Kripke Strukturen können automatisch in SMV ( und in anderen Model Checkern) modelliert werden.



# Zustände

Das Verhalten technischer Systeme ist von internen Zuständen (engl.: *state* oder *mode*) abhängig.




- **Digitaluhr**:
  (Stoppuhr-Modus, Zeitanzeige-Modus, Einstell-Modus, Batterieladezustand, Schwingungszähler des Quarzes, ...)
- **Fernsehen**:
  Videotext oder Bild, Kanäle, etc.
- **Prozesse**:
  idle, running, waiting, blocked, terminated, ...
- **Programme**:
  Programmzähler, Variableninhalt, Filepointer, etc. ...

Zustände bilden endliche Menge **S**. Typische Elemente von S: $s, s', s_0, s_1, \ldots$
Jede endliche Menge kann man als Menge von Zuständen deuten.

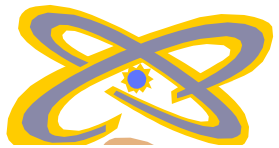
In der Praxis: Zustände durch Inhalt von Variablen bestimmt.
Einige dieser Variablen sind *beobachtbar*, einige versteckt ( *hidden* ).

# Atomare und Boolesche Eigenschaften

- Eine Eigenschaft heißt atomar, wenn sie sich nicht in einfachere Eigenschaften logisch zerlegen lässt.

- Atomare Eigenschaften sind meist durch Vergleich zweier Terme gegeben

  - `seite = gruen`
  - `request > 0`
  - `(timer +1) mod 10 < 5`
  - $ack_1 + ack_2 + ack_3 <= 1$
  - `timeout = TRUE`

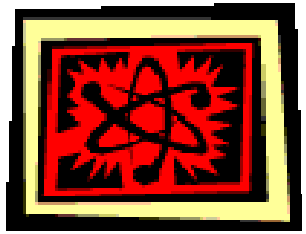


- Eigenschaften kann man durch die logischen Operatoren verknüpfen

  - $\wedge$ ( and, und, **&** )
  - $\vee$ ( or, oder, **|** )
  - $\Rightarrow$ ( implies, wenn..dann.., **->** )
  - $\Leftrightarrow$ ( iff, ==, gdw. , **<->**) )
  - $\neg$ ( not, nicht, **!**)
  - $\oplus$ ( xor, exclusives oder, **xor** )

# Atomare Aussagen - abstrakt

- Beobachtbare elementare Eigenschaften von Zuständen.
- Jeder Zustand bestimmt eine Menge von atomaren Aussagen, die wahr sind (gelten). Alle anderen atomaren Aussagen sind falsch.
- Abstrakt:
  Aussagen bilden eine Menge AP (atomic propositions).
  - typische Elemente von AP bezeichnen wir mit p, q, r, ..., $p_1$, $p_2$, ...
- Es gibt eine Funktion L: $S \to \mathbb{P}(AP)$.
  - Jedem Zustand $s \in S$ ordnen wir eine Menge $L(s) \subseteq AP$ zu.
  - L(s) ist die Menge aller der atomaren Aussagen, die in Zustand s gelten.



# Aussagen

- Für den Rest der Vorlesung betrachten wir eine feste Menge $AP$ von atomaren Eigenschaften .
  - $p, q, r, \ldots, p_1, p_2, \ldots \in P$
- Mit $P$ bezeichnen wir die Menge aller Aussagen (Propositions) d.h. aller Booleschen Kombinationen von Eigenschaften aus AP, also z.B.
  - $p \wedge q,\ \ p_1 \Rightarrow (p_2 \vee \neg p_3),\ tt \Rightarrow p \Rightarrow r$, etc.

  Typische Elemente aus $P$ bezeichnen wir ebenfalls mit
  - p,q,r, ...

# Kontext

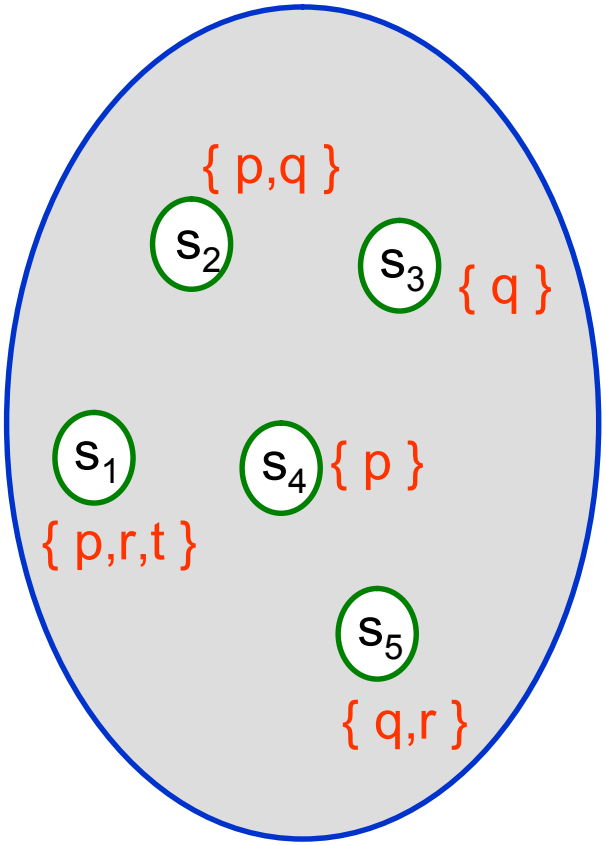
Ein Kontext (oder ein aussagenlogisches Modell über AP ist ein Paar $C = (S,AP,L)$ bestehend aus

- $S$ : einer Menge (von Zuständen),
- $AP$ : einer Menge (von atomaren Aussagen)
- $L$ : $S \to \mathbb{P}(AP)$ einer Abbildung.

<u>Beispiel</u>: $S = \{ s_1, s_2, s_3, s_4, s_5 \}$

$L(s_1) = \{ p,r,t\}$
$L(s_2) = \{p,q\}$
$L(s_3) = \{q\}$
$L(s_4) = \{p\}$
$L(s_5) = \{q,r\}$

S

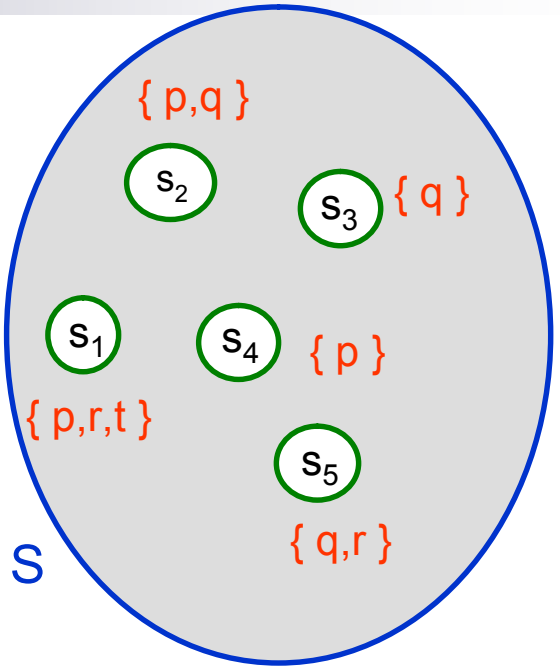



S steht für state
AP für atomic propositions
L für labeling

Die Mengenklammern lassen wir aus Bequemlichkeit oft weg.

# Kontext als Tabelle

- Kontext C=(S,AP,L) kann man als Tabelle darstellen, z.B.:
  - eine Zeile pro Zustand
  - eine Spalte pro atomarer Eigenschaft
  - 1 in Zeile s und Spalte p,
    - falls $p \in L(s)$
    - d.h., falls $C,s \models p$
  - 0 sonst.

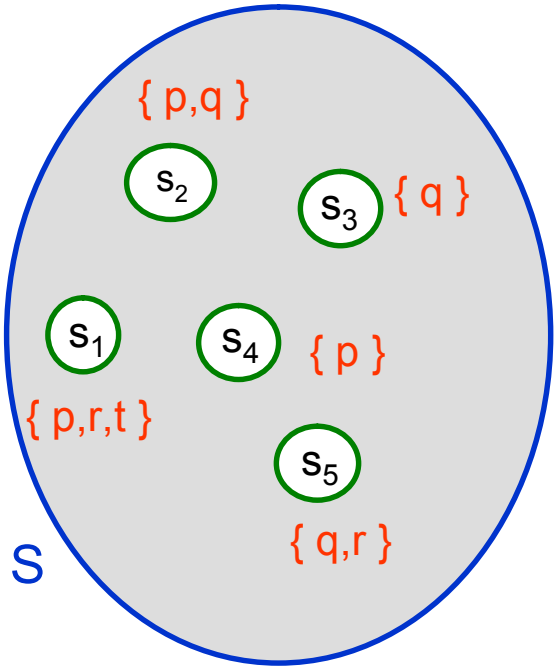



| $\models$ | p | q | r | t |
|---|---|---|---|---|
| $s_1$ | 1 | 0 | 1 | 1 |
| $s_2$ | 1 | 1 | 0 | 0 |
| $s_3$ | 0 | 1 | 0 | 0 |
| $s_4$ | 1 | 0 | 0 | 0 |
| $s_5$ | 0 | 1 | 1 | 0 |

# Gültigkeit

- Definition: In Zustand *s* von Kontext *C*=(*S*,*AP*,*L*) *gilt* die atomare Aussage p:
  - $C,s \models p \quad :\Leftrightarrow \quad p \in L(s)$
- Erweiterungen auf p aus P:
  - $C,s \models p \wedge q \quad :\Leftrightarrow \quad C,s \models p$ und $C,s \models q$
  - $C,s \models \neg p \quad :\Leftrightarrow \quad C,s \not\models p$
- Die anderen Junktoren sind Abkürzungen
  - $C,s \models p_1 \vee p_2 \quad :\Leftrightarrow \quad C,s \models \neg (\neg p \wedge \neg q)$
  - $C,s \models p_1 \Rightarrow p_2 \quad :\Leftrightarrow \quad C,s \models \neg p \vee q$
  - $C,s \models p_1 \Leftrightarrow p_2 \quad :\Leftrightarrow \quad C,s \models (p \Rightarrow q) \wedge (q \Rightarrow p)$
  - $C,s \models p_1 \oplus p_2 \quad :\Leftrightarrow \quad C,s \models (p \wedge \neg q) \vee (q \wedge \neg p)$

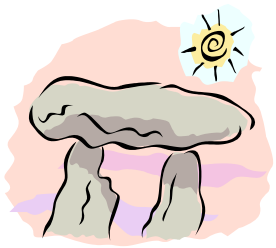
$C,s_1 \models p,$
$C,s_1 \models p \wedge r,$
$C,s_1 \models \neg q,$




$C,s_2 \models r \Leftrightarrow t,$
$C,s_2 \models p \Leftrightarrow \neg t,$

$C,s_4 \models q \Rightarrow p,$
$C,s_4 \not\models p \Rightarrow q,$

# Modelle und Äquivalenz



Sei $p \in P$ eine Aussage, $C = (S, AP, L)$ ein **Kontext** und $s \in S$.

$C,s$ heißt ein ***Modell von p***, falls :

$$C,s \models \mathbf{p}.$$

p heißt

- ***erfüllbar***, falls es ein Modell von **p** gibt.
- ***widersprüchlich***, falls **p** nicht erfüllbar ist
- ***allgemeingültig***, falls jeder Zustand jedes Kontextes ein Modell von p ist.

Zwei Aussagen $p, q \in P$ heißen ***semantisch äquivalent***, wenn für jeden Kontext $C = (S,AP,L)$ und jedes $s \in \mathbf{S}$ gilt :

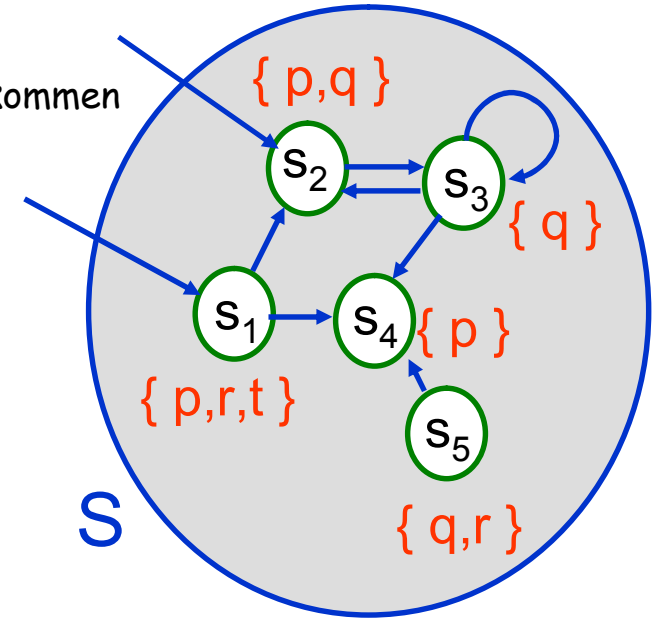
$$C,s \models p \quad \text{gdw.} \quad C,s \models q.$$

Beispiele : Mit rot, gelb,grün $\in$ **AP** gilt :

| | |
|---|---|
| $\neg$ rot $\Rightarrow$ (grün $\wedge$ gelb) | ist erfüllbar, aber nicht allgemeingültig |
| rot $\wedge$ (rot $\Rightarrow \neg$( grün $\vee$ rot )) | ist widersprüchlich |
| rot $\Rightarrow$ (grün $\Rightarrow$ rot) | ist allgemeingültig |

# Kripke Strukturen

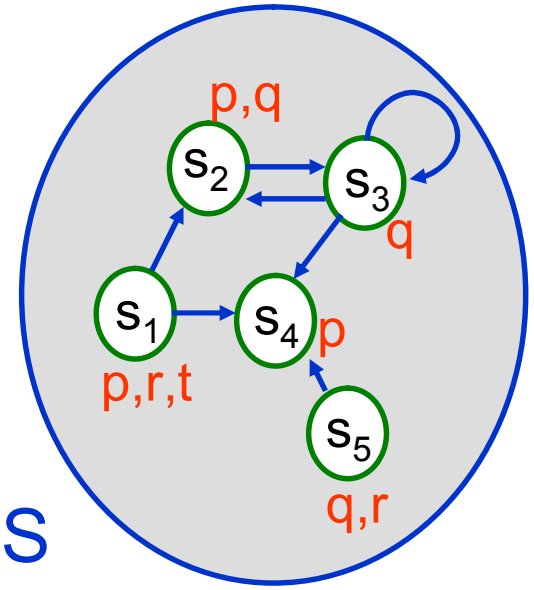
- Eine Kripke-Struktur ist ein Tupel K = (S, I, R, AP,L), wobei
  - (S,AP, L) ein Kontext ist und
  - I $\subseteq$ S ist eine Menge von *Anfangszuständen* *)
  - R $\subseteq$ S $\times$ S eine beliebige *Relation*.
- Schreibweisen:
  - s R s‘ statt (s , s‘) $\in$ R
  - $\rightarrow$ oder $\rightarrow_R$ statt R
  - Anfangszustände markiert man mit Pfeilen, die von außen kommen
- s $\rightarrow_R$ s‘ heißt also (s,s‘) $\in$ R
  - System kann von Zustand s in Zustand s‘ übergehen
- Kripke Strukturen sind Graphen
  - Graph G=(S,R) mit Knotenbeschriftung L:S$\rightarrow\mathbb{P}$(AP).
  - In der Sprache der Graphen heißen Zustände „Knoten“ und Paare (s , s‘) $\in$ R Kanten.




)* Wenn I fehlt, setzt man I=S

# Bezeichnungen

- Eine Relation R heißt
  - total, falls zu jedem Zustand ein Nachfolgezustand existiert, d.h.
    $\forall s \in S. \exists s' \in S. (s,s') \in R$.
  - deterministisch, falls der Nachfolgezustand eindeutig ist, d.h.
    $\forall s, s_1, s_2 \in S. (s, s_1) \in R \wedge (s, s_2) \in R \Rightarrow s_1 = s_2$.
- Statt $(s,s') \in R$, sagt man auch
  - s' ist Nachfolger von s
  - s ist Vorgänger von s'
- Eine zweistellige Relation $R \subseteq S \times S$ heißt
  - *reflexiv*, falls $\forall s \in S. (s,s) \in R$.
  - *symmetrisch*, falls $\forall s, s' \in S. (s,s') \in R \Rightarrow (s',s) \in R$.
  - *transitiv*, falls $\forall s,s',s'' \in S. (s,s') \in R \wedge (s',s'') \in R \Rightarrow (s,s'') \in R$.




Die hier gezeigte Kripke-Struktur ist weder total, noch deterministisch

# Relationenalgebra

- Bezeichnungen
  - $id_S := \{ (s,s) \mid s \in S \}$ heißt Identität oder Diagonale von S
  - $R^{-1} := \{ (s',s) \mid (s,s') \in R \}$ heißt konverse Relation zu R
- Sind R und T zweistellige Relationen auf S, so heißt

  $$R \circ T := \{ (s,t) \mid \exists u \in S.\ (s,u) \in R,\ (u,t) \in T \}$$

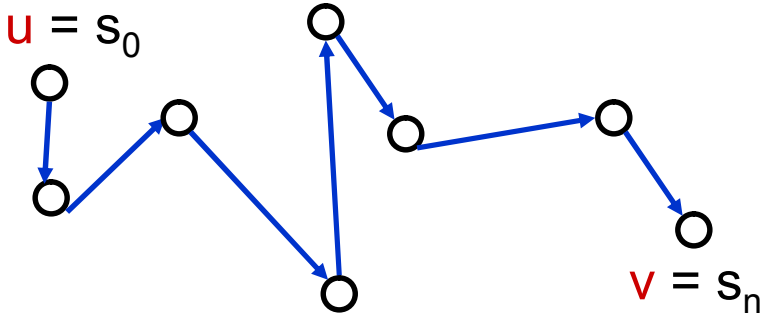
  das Relationenprodukt von R und T.

Vorsicht: Im Allgemeinen ist $R \circ R^{-1} \neq id$

- Eine Relation R ist
  - reflexiv gdw. $id_S \subseteq R$
  - symmetrisch gdw. $R^{-1} \subseteq R$
  - transitiv gdw. $R \circ R \subseteq R$

# Transitive Hülle

- Setze $R^0 := id_S$ , $R^1 = R$ und $R^{n+1} := R^n \circ R$
  - $R^* := \bigcup\{ R^n \mid n \geq 0 \}$ und
  - $R^+ := \bigcup\{ R^n \mid n > 0 \}$.




- $R^+$ ist die transitive Hülle von R
  - dh. kleinste transitive Relation, die R umfasst
- $R^*$ ist die reflexiv transitive Hülle von R, d.h.
  - kleinste reflexive und transitive Relation, die R enthält.
  - Alternative Beschreibung
    - $(u,v) \in R^+$ gdw. $\exists\, n \in \mathbb{N}.\ \exists s_0,\ldots,s_n \in S.\ u=s_0,\ v=s_n\ ,\ \forall\, i < n: (s_i,s_{i+1}) \in R$
    - $R^* = R^+ \cup id_S$

# Relationen und Abbildungen

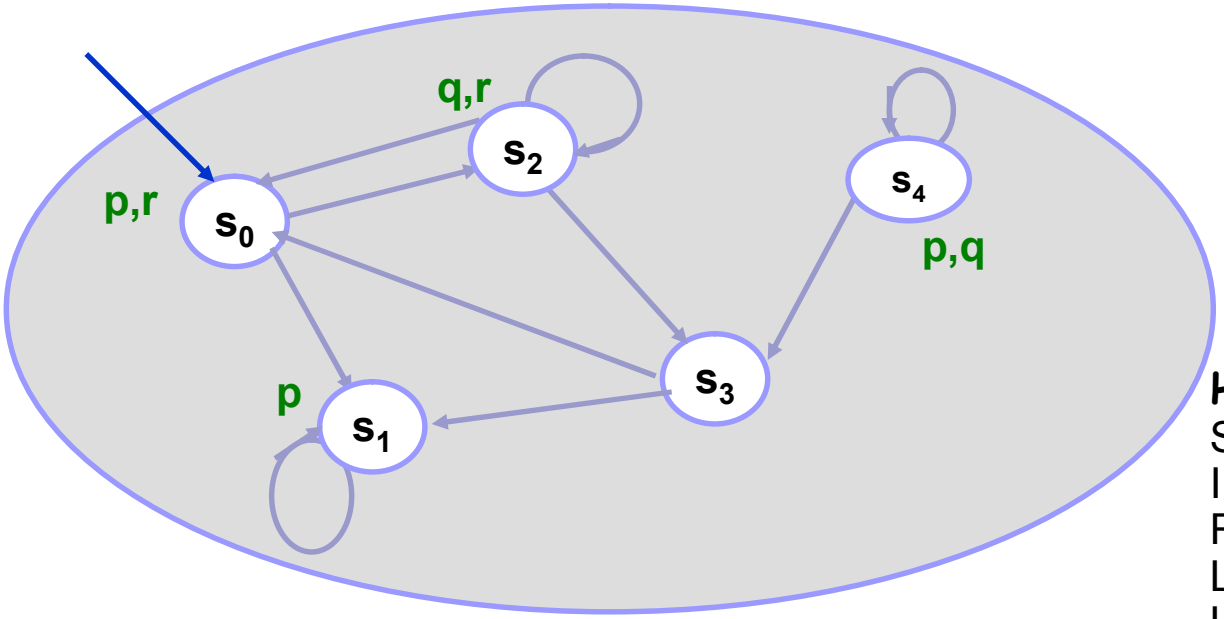
Jede Relation $R \subseteq S \times T$ induziert zwei Abbildungen

- $R[-] : S \to \mathbb{P}(T)$ ( Nachfolgerabbildung )
  - $R[s] := \{ t \in T \mid s\ R\ t \}$
- $R^{-1}[-] : T \to \mathbb{P}(S)$ ( Vorgängerabbildung )
  - $R^{-1}[t] := \{ s \in S \mid s\ R\ t \}$
- umgekehrt lässt sich R aus einer Abbildung $N : S \to \mathbb{P}(T)$ zurückgewinnen
  - $s\ R\ t :\Leftrightarrow t \in N(s)$
  - Es gilt dann $N(s) = R[s]$
- ebenso aus einer Abbildung $V: T \to \mathbb{P}(S)$ als
  - $s\ R\ t :\Leftrightarrow s \in V(t)$
  - Es gilt dann $V(t) = R^{-1}[t]$

# Kripke Struktur

Der Einfachheit halber betrachten wir (wie auch NuSMV) in der Zukunft nur Kripke-Strukturen mit totaler Transitionsrelation. Also: jeder Zustand hat mindestens einen Nachfolger, notfalls sich selber

| | |
|---|---|
| $S$ | Menge von Zuständen (States) |
| $R \subseteq S \times S$ | totale Transitionsrelation. |
| $L : S \to \mathbb{P}(AP)$ | Abbildung (labeling) |
| $I \subseteq S$ | Anfangszustände |




**Hier:**

$S = \{ s_0, s_1, s_2, s_3, s_4 \}$

$I = \{ s_0 \}$

$R = \{ (s_0, s_1), (s_0, s_2), (s_1, s_1), \ldots \}$

$L(s_0) = \{ p,r \}$, $L(s_1) = \{ p \}$,

$L(s_3) = \{ \}$, $L(s_4) = \{ p, q \}$,

# Andere Sichtweise von Kripke Strukturen

$R \subseteq S \times S$ totale Transitionsrelation
$I \subseteq S$
$L : S \to \mathbb{P}(AP)$ Abbildung (labeling)

R kann man auch auffassen …

… als charakteristische Abbildung $\chi_R : S \times S \to 2$

… oder als Nachfolgerabbildung $\nu_R : S \to \mathbb{P}(S)$

L kann man auch modellieren …

… charakteristische Abbildung $\chi_L : S \times AP \to 2$

… als Gültigkeitsrelation $\models \subseteq S \times AP$

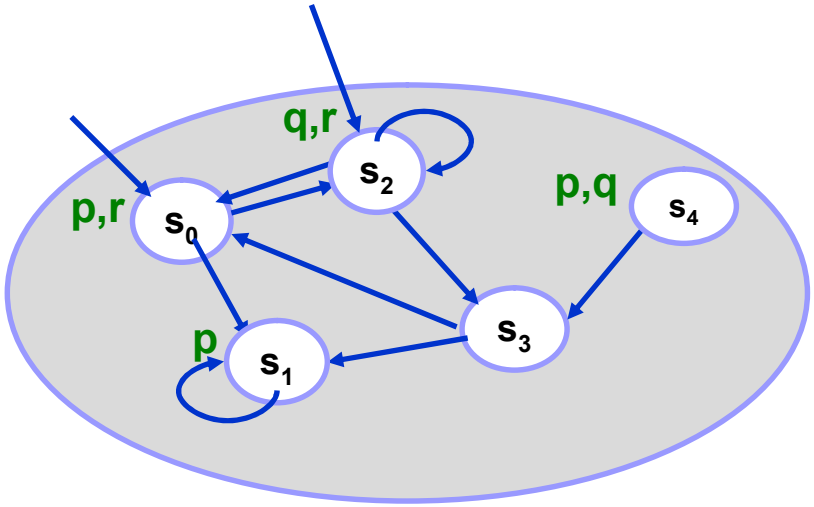
… oder als Abbildung $\mu_L : AP \to \mathbb{P}(S)$

I kann man auch modellieren

…. durch ein *init* $\in$ AP $I = \{ s \in S \mid s \models init \}$

Saul Kripke

# Repräsentation von Kripke-Strukturen



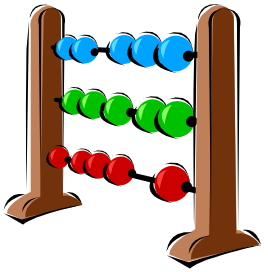

- Eine Kripke Struktur lässt sich durch Tabellen repräsentieren:
  - Eine Tabelle für die Bewertung:
    $\models : S \times AP \to 2$
  - Eine Tabelle für die Übergangsrelation:
    $\chi_R : S \times S \to 2$
  - Eine Liste von Anfangszutänden

| $\models$ | p | q | r |
|---|---|---|---|
| $s_0$ | 1 | 0 | 1 |
| $s_1$ | 1 | 0 | 0 |
| $s_2$ | 0 | 1 | 1 |
| $s_3$ | 0 | 0 | 0 |
| $s_4$ | 1 | 1 | 0 |

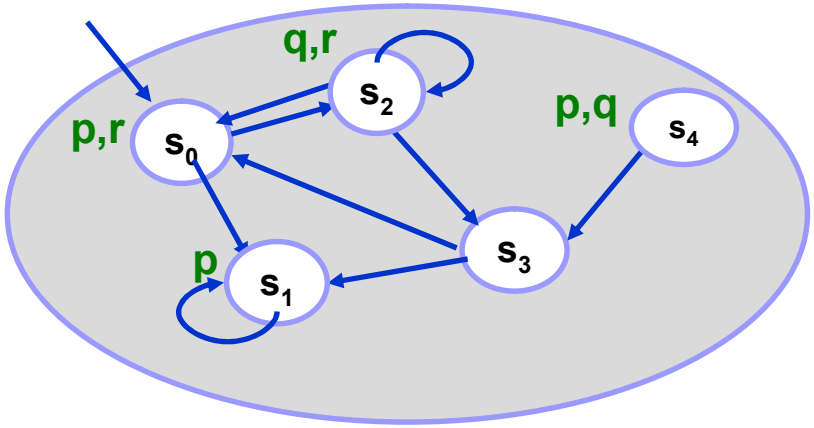
| R | $s_0$ | $s_1$ | $s_2$ | $s_3$ | $s_4$ |
|---|---|---|---|---|---|
| $s_0$ | 0 | 1 | 1 | 0 | 0 |
| $s_1$ | 0 | 1 | 0 | 0 | 0 |
| $s_2$ | 1 | 0 | 1 | 1 | 0 |
| $s_3$ | 1 | 0 | 0 | 0 | 0 |
| $s_4$ | 0 | 0 | 0 | 1 | 0 |

| I | |
|---|---|
| $s_0$ | 1 |
| $s_1$ | 0 |
| $s_2$ | 1 |
| $s_3$ | 0 |
| $s_4$ | 0 |

# Pfade und Berechnungen



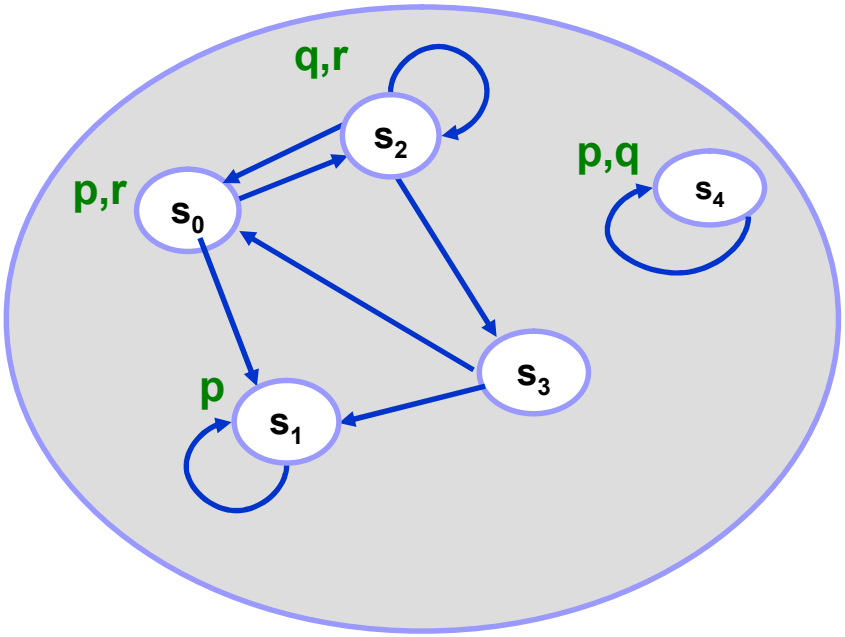
- Sei K=(S,I, R,L,AP) eine Kripke Struktur über P.
- Ein *Pfad* ist eine unendliche Folge von Zuständen
  $\sigma = (s_i)_{i\in\mathbb{N}} = (s_0,s_1, \dots)$
  mit
  - $\forall\, i\in\mathbb{N}.\ (s_i,s_{i+1}) \in R$ ( Folgezustände sind in der Transitionsrelation )
- Eine *Berechnung* ist ein Pfad mit $s_0 \in I$




Ein Pfad ist z.B.
$\sigma = (s_3,s_0,s_2,s_0,s_2,s_2,s_0,\dots)$

eine Berechnung ist
$\pi = (s_0,s_2,s_3,s_0,s_1,s_1,s_1,\dots)$

# Modellierung in SMV




S

R

UltraEdit-32 - [C:\Documents and Setting...

File Edit Search Project View Format Colum

kripke.smv

```
-- kripke.smv

MODULE main
VAR
  state : {s0, s1, s2, s3, s4 };

ASSIGN
  next(state) := case
    state = s0 : {s1, s2};
    state = s2 : {s0, s2, s3};
    state = s3 : {s0, s1};
    1          : state ;  -- default
  esac;
```

For Help, press F1 Ln 18, Col. 26, C0 DOS M

Hier:

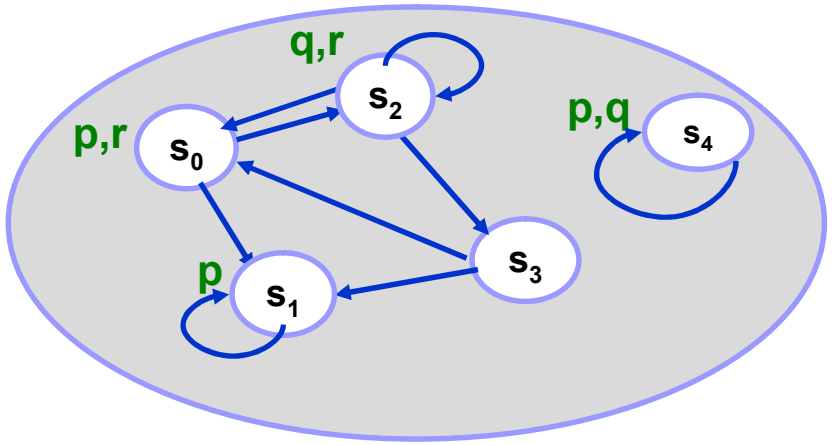
$S = \{s_0, s_1, s_2, s_3, s_4\}$

$R = \{ (s_0, s_1), (s_0, s_2), (s_1, s_1), \ldots \}$

$L(s_0) = \{ p, r \}$, $L(s_1) = \{ p \}$,

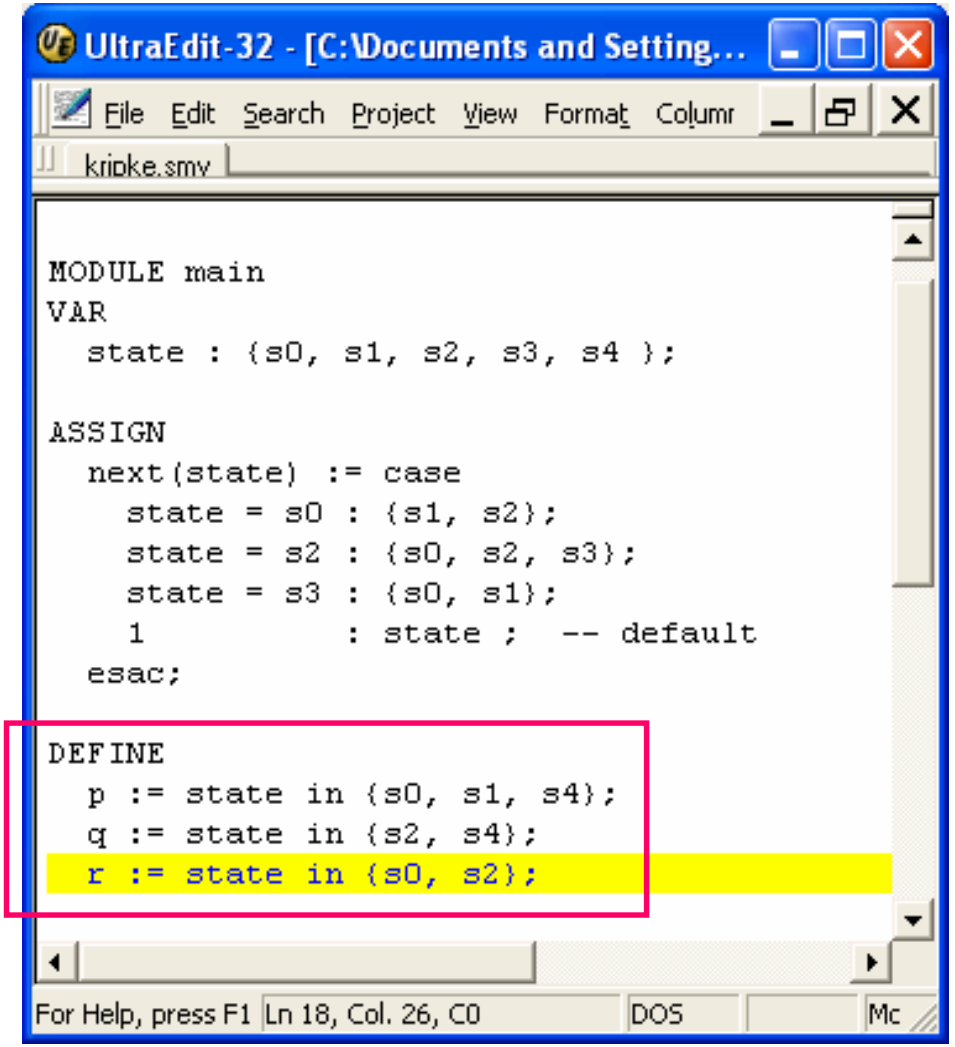
$L(s_3) = \{ \}$, $L(s_4) = \{ p, q \}$

# Modellierung der Eigenschaften

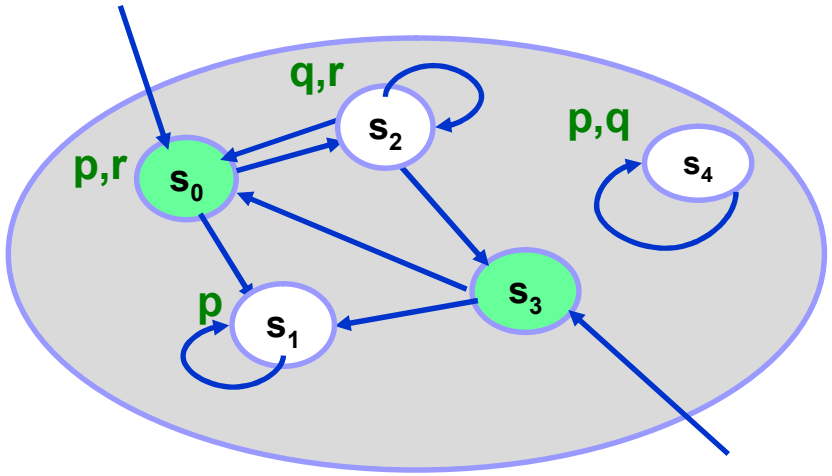



- Eigenschaften kann man gut mit der DEFINE-Deklaration modellieren
- DEFINE ist ein Macro
  - textuelle Ersetzung
  - der Zustandsraum wird nicht vergrößert

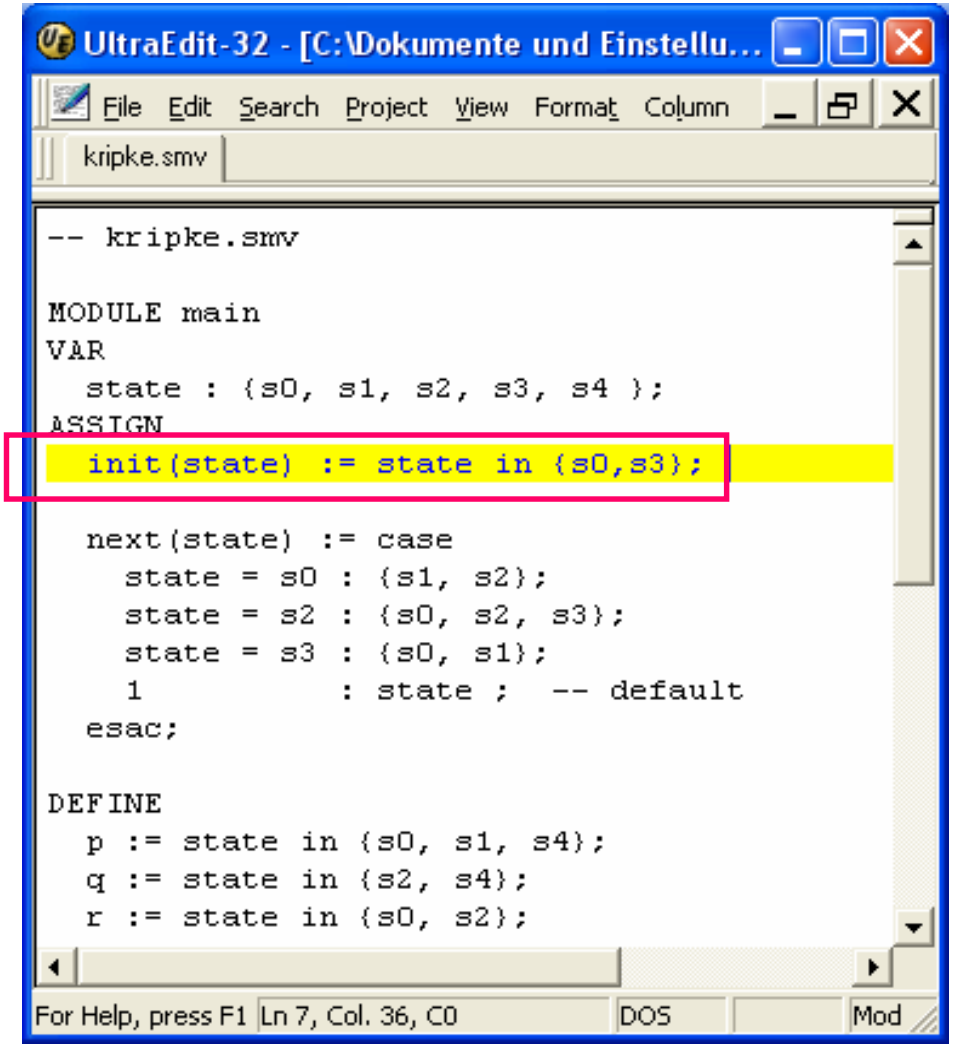
AP



# Anfangszustände - Erreichbarkeit




- Man interessiert sich meist nur für die von einem Anfangszustand aus erreichbaren Zustände.
- s' heißt erreichbar, falls $\exists\, s \in I.\ s\ R^*\ s'$.
- Im Beispiel sind $s_0$ und $s_3$ Anfangszustände. $s_0$, $s_1$, $s_2$ und $s_3$ erreichbar, $s_4$ nicht.
- In SMV kennzeichnet man Anfangszustände durch **init**.



# Trivial-Programm als Kripke Struktur

```
{
   boolean n,m=1;

   m = (m+n)%2;
}
```

Kann als Kripke Struktur z.B. über

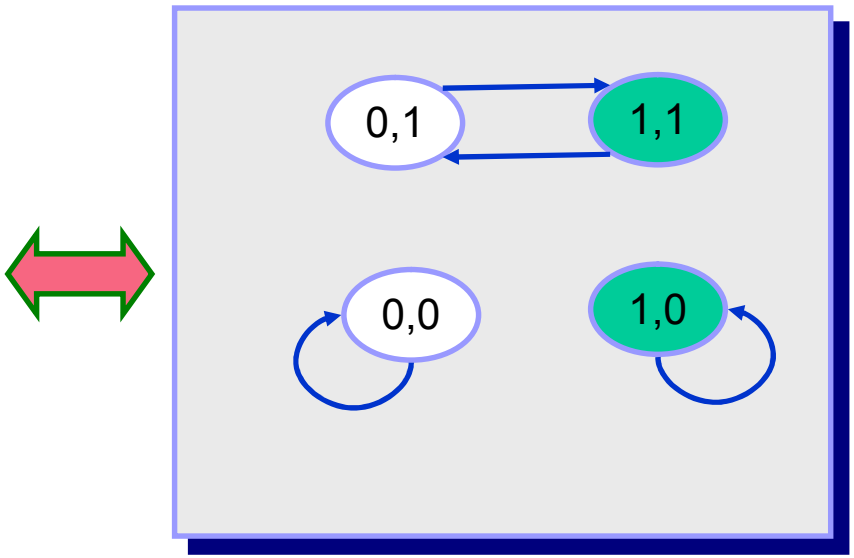
**AP** = { m = 0, m = 1, n = 0, n = 1 }

aufgefasst werden:

**S** = {0,1} × {0,1}.
**I** = { (1,0),(1,1) }

**R** = { ((0,0),(0,0)), ((0,1),(1,1)),
((1,0),(1,0)), ((1,1),(0,1)) }

**L**((0,0)) = { m=0, n=0 }, **L**((0,1)) = { m=0, n=1 },
**L**((1,0)) = { m=1, n=0 }, **L**((1,1)) = { m=1, n=1 }.




Diese Kripke Struktur ist **deterministisch**, d.h. **R** ist eine **Abbildung**.

# Das Trivial-Programm in SMV

```
{
  boolean n,m=1;

  m = (m+n)%2;
}
```

```
1
2  MODULE main
3  VAR
4    m : boolean;
5    n : boolean;
6
7  ASSIGN
8   init(m) := 1;
9
10  next(m) := (m + n) mod 2;
11
12  next(n) := n;  --nicht vergessen !!
13
```

- next(n) := n
  - notwendig, ansonsten ist n im Folgezustand unbestimmt !
- Vorsicht:
  - NuSMV hat viele Schlüsselwörter:

MODULE, DEFINE, CONSTANTS, VAR, IVAR, INIT, TRANS, INVAR, SPEC, CTLSPEC, LTLSPEC, PSLSPEC COMPUTE, INVARSPEC, FAIRNESS, JUSTICE, COMPASSION, ISA, ASSIGN, CONSTRAINT, SIMPWFF, CTLWFF, LTLWFF, PSLWFF, COMPWFF, IN, MIN, MAX, process, array, of, boolean, integer, real, word, word1, bool, EX, AX, EF, AF, EG, AG, E, F, O, G, H, X, Y, Z, A, U, S, V, T, BU, EBF, ABF, EBG, ABG, case, esac, mod, next, init, union, in, xor, xnor, self, TRUE, FALSE

# Beliebiges Programm → Kripke Struktur

```
 0:{
    boolean done;
    byte x, y;
 1: x = 0;
 2: done = false;
 3: while (!done) {
   4: done = (x*x == y);
   5: x = x+1;
 6:}
7: }
```

Zustand **S** ergibt sich aus

- dem aktuellen Wert der Variablen
- **und** dem aktuellen Wert des Programmzählers

Wir führen **label** (Marken) ein. Diese entsprechen den Punkten, wo der Scheduler das Programm unterbrechen darf. Zuweisungen nehmen wir als **atomar** an.

$S = \{ (d,u,v,pc) : d \in \{0,1\}, u,v \in \{0,..,255\}, pc \in \{0,..,7\} \} = \{0,1\} \times \{0,..,255\} \times \{0,\ldots,255\} \times \{0,..,7\}$

$\Rightarrow$

$|S| = 2 \times 2^8 \times 2^8 \times 2^3 = 2^{20} =$ **1.048.576** Zustände, allerdings viele nicht erreichbar !

# Markierungsalgorithmus

Der Algorithmus **Lbl** soll ein beliebiges **while**-Programm **P** mit Marken versehen.

```
Lbl(P)        =        l: M(P) l':
```

Marken an Anfang und Ende

Hilfsfunktion M erzeugt innere Marken :

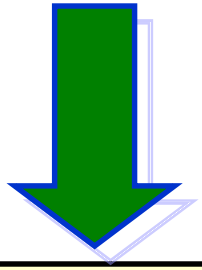
```
M(v=t;)                          = v=t;

M(P1 P2 )                        = M(P1) l: M(P2)

M(if b then P1 else P2 )= if b then l:M(P1) else l':M(P2)

M(while b do P)                  = while b do l: M(P)
```

`l, l'` jedesmal *neue* Marken. Dienen später als Werte des Programmzählers ***pc***.

# Die Dynamik

- Nachdem Marken eingefügt sind, verwandle Programm in ein *Goto-Programm*
- Ein *Goto-Programm* besteht nur aus
  - (bedingten) Zuweisungen
  - (bedingten) Sprüngen
- Jedes sequentielle Programm lässt sich in ein äquivalentes Goto-Programm übersetzen.
- Umwandlung kann automatisch erfolgen.

```
 0:{
    boolean done;
    byte x, y;
 1: x = 0;
 2: done = false;
 3: while (!done) {
 4:   done = (x*x == y);
 5:   x = x+1;
 6:}
7: }
```



```
0:
1: x = 0;
2: done = false;
3: if done goto 7;
4: done = (x*x == y);
5: x = x+1;
6: Goto 3;
7:
```

# Goto-Programm ⇒ Kripke-Struktur

- Sei $T_1\ x_1; T_2\ x_2; \ldots T_n\ x_n;$ die Variablendeklaration, und $M = \{0 .. m-1\}$ die Marken (Zeilenr.) des Goto-Programms P.
- Ein Zustand ist ein (n+1)-Tupel $(v_1,..,v_n, pc) \in T_1 \times \ldots \times T_n \times M$.
- Es gilt $( (v_1,..,v_n,pc) , (w_1,..,w_n,pc') ) \in R$ , falls

`m: if b(x1,..,xn) then goto k` $\in P$ und

$pc = m,$
$b(v_1,..,v_n) = true,$
$w_k = v_k$ für alle k,
$pc' = k.$

`m: if b(x1,..,xn) then xi=t(x1,…,xn);` $\in P$ und

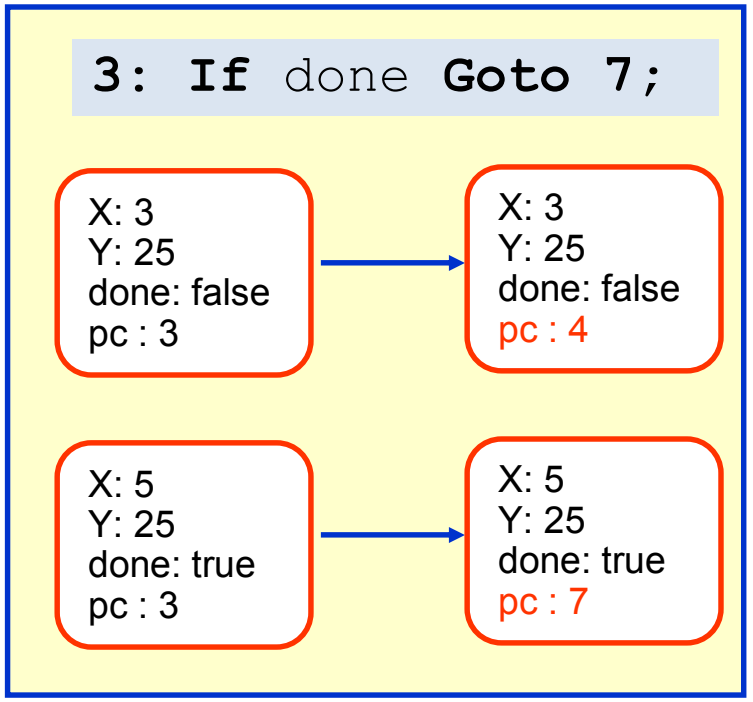
$pc = m,$

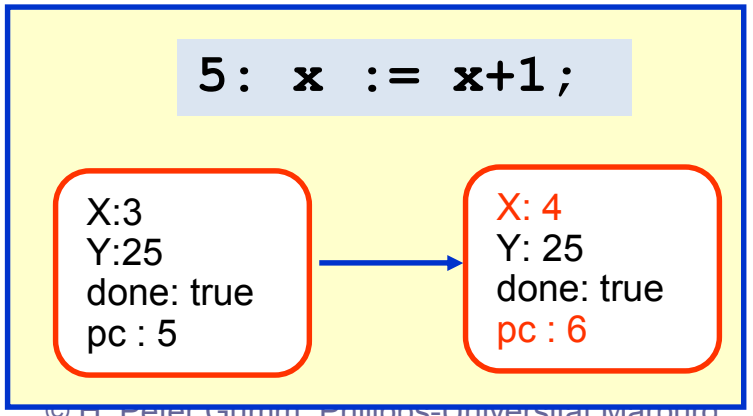
$b(v_1,..,v_n) = true,$

$w_i = t(v_1,\ldots,v_n),$

$w_k = v_k$ für $k \neq i$ ,

$pc' = m+1.$



# Goto ⇒ SMV

```
VAR
     done : boolean;
     x, y :{0 .. 255};
0:
1: x = 0;
2: done = false;
3: If done Goto 7;
4: done = (x*x == y);
5: x = x+1
6: Goto 3;
7:
```

Für jede Variable (Programmvariable und pc) spezifiziere

-Anfangszustand

- Wert zum jeweils nächsten Zeitpunkt

- auch wenn dieser sich nicht ändert !!

UltraEdit-32 - [C:\Dokumente und Einstellu...

File Edit Search Project View Format Column

wurzel.smv*

```
VAR
  done : boolean;
  x : 0 .. 255;
  y : 0 .. 255;
  pc : 0 .. 7;     -- Programmzähler

ASSIGN
 init(pc) := 0;

 next(pc) := case
   pc = 3 & done : 7;
   pc = 6        : 3;
   pc = 7        : 7;
   TRUE          : (pc+1) mod 8;
 esac;

 next(done) := case
    pc = 2 : FALSE;
    pc = 4 : x*x=y;
    TRUE   : done;
 esac;

 next(x) := case
   pc = 1 : 0;
   pc = 5 : (x+1) mod 256;
   TRUE   : x;
 esac;

 next(y) := y;    -- nicht vergessen
```

For Help, press F1 | Ln 6, Col. 1, C0 | DOS | Mod

# Verteilte Programme

```
Pg ::= cobegin P1 || ... || Pn coend

Jedes Pi ist ein Prozess, d.h. ein While-
Programm mit syntaktischen Erweiterungen:

P     ::= skip | wait(b) | lock(v) | unlock(v)
        | v = t;
        | P1 P2
        | if( b ) P1 else P2
        | while(b) P
```

Intendierte Semantik:

- Die $P_i$ werden verteilt (asynchron) ausgeführt.
- **atomar** (d.h. vom scheduler nicht unterbrechbar) sind :

  `v:= t, skip, wait(b), lock(v), unlock(v).`
- Nach jeder atomaren Aktion kann der laufende Prozess unterbrochen werden.

# Semantik verteilter Programme

```
cobegin P1 || P2 || ... || Pn coend
```

Zunächst werden alle Prozesse dem Scheduler übergeben,

- dann wird jeweils ein Schritt in irgendeinem Prozess ausgeführt,
- wenn alle Prozesse fertig sind, stoppt das Programm.

```
wait(b)
```

Prozess wartet auf das Eintreten der Bedingung **b**. (busy waiting)

```
lock(v)
```

```
unlock(v)
```

Setzen und Rücksetzen eines binären Semaphors.

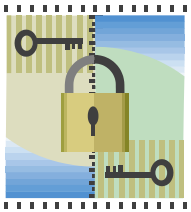
# Mutual exclusion

```
boolean turn = 0;

cobegin P0 || P1 coend
```

wobei

$P_i$ =

```
l: while(true)
     {
      nc: wait(turn == i);
      cs: turn = (1-i);
     }
```



# Verteilte Programme → Kripke Strukturen

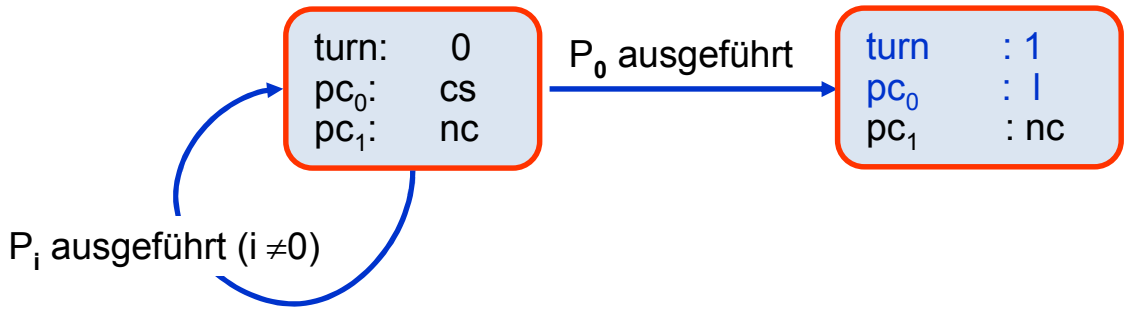
- Ein Zustand besteht jetzt aus
  - allen Variableninhalten
  - allen Programmzählern

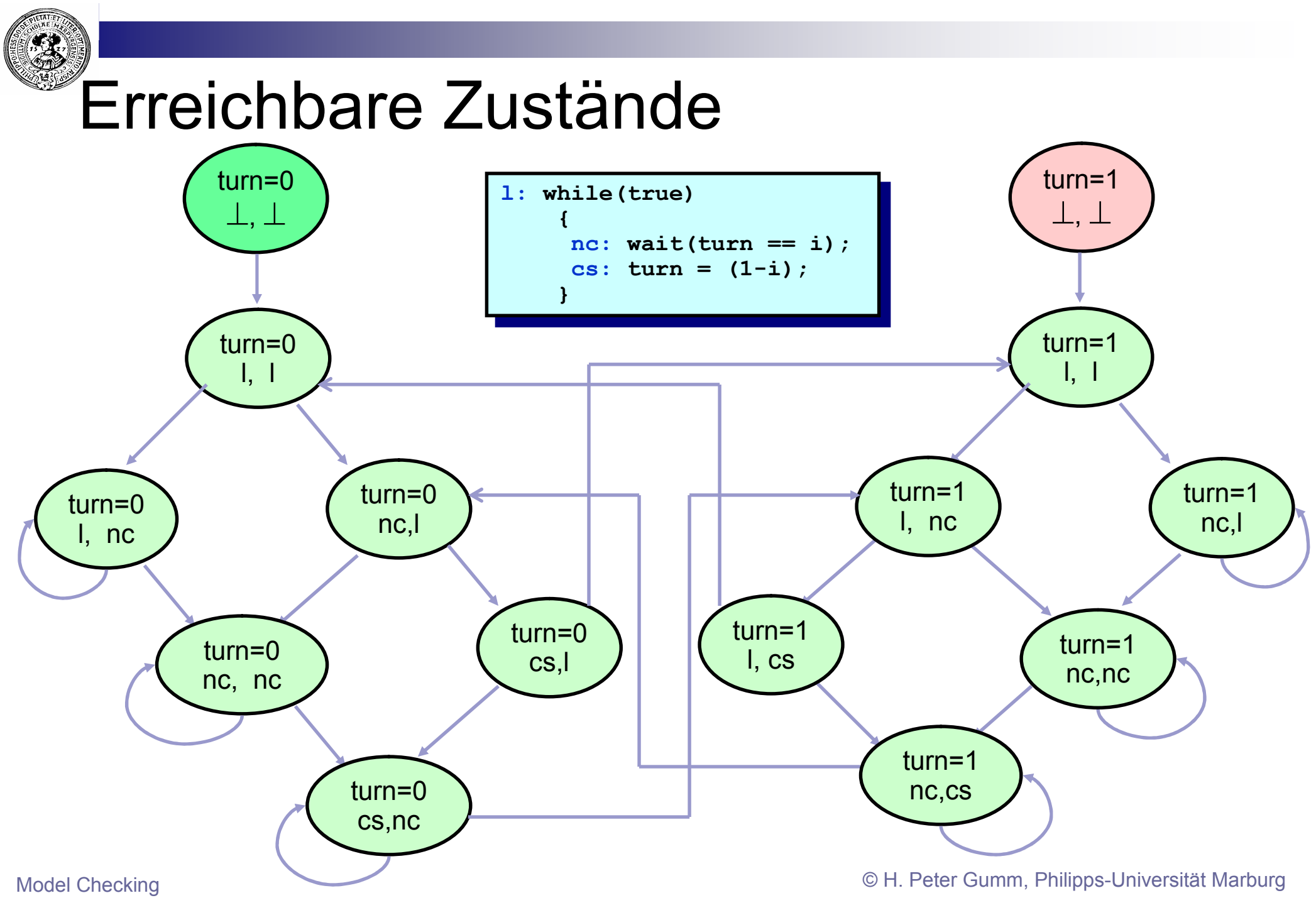
Jeder Zustand hat (höchstens) so viele Nachfolger wie es laufende Prozesse gibt.

```
boolean turn = 0;

cobegin P0 || P1 coend
```

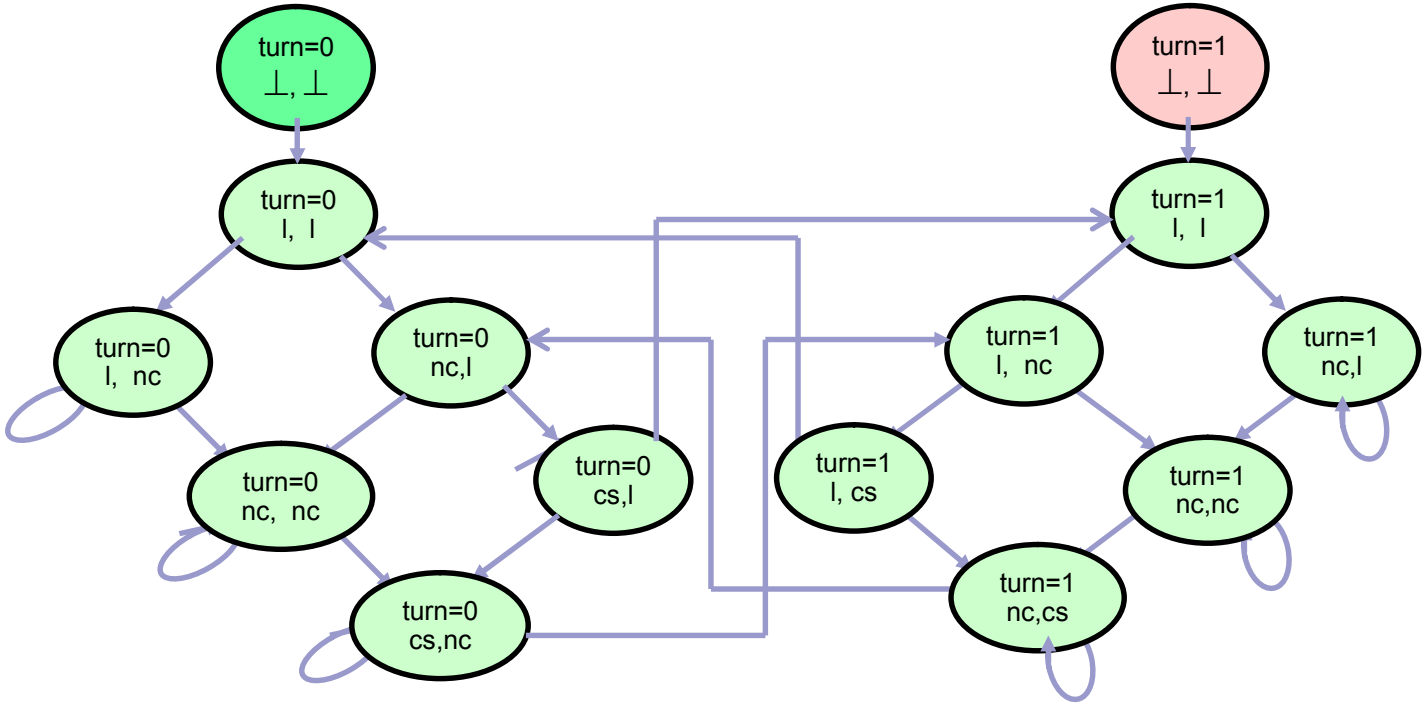
$P_i$ =

```
l: while(true)
    {
     nc: wait(turn == i);
     cs: turn = (1-i);
    }
```





# Eigenschaften der Kripke Struktur




- An der Kripke Struktur sind u.a. folgende Eigenschaften ablesbar oder widerlegbar:
  - In keinem erreichbaren Zustand gilt $pc_0 = pc_1 = cs$
  - Von nc kommt jeder Prozess unweigerlich irgendwann zu cs
  - Von nc kann ein Prozess immer zu nc gelangen
  - Kein Prozess kann den andern verhungern lassen
  - Es kommt nie zu einer Verklemmung
  - Nachteil: Programme kommen nur abwechselnd in ihren kritischen Abschnitt

# Modellierung in SMV

-- Petersens Lösung für Mutex

```
boolean   turn = 0;
boolean[] flag = {0,0};

cobegin P0 || P1 coend
```

```
process Pi:
   while(true){
      flag[i] = true;
      turn    = 1-i;
      wait (!flag[1-i]|(turn==i));
      Skip;                  -- Kritisch
      flag[i] = false;
   }
```

**Sicherheit**:
**Nie beide Prozesse im kritischen Bereich**

**Lebendigkeit**:
**Fairness des Prozess-Schedulers vorausgesetzt, wird jeder Prozess unendlich oft kritisch.**

**No deadlock**:
**Die Prozesse bleiben nicht beide dauernd im Wartezustand**

UltraEdit-32 - [C:\Documents and Settings\g...

File Edit Search Project View Format Column !

Bakery.smv SimplePetereson.smv*

```
MODULE main
VAR
  turn : 0 .. 1;
  flag : array 0 .. 1 of boolean;
  P0 : process user(0,turn,flag);
  P1 : process user(1,turn,flag);

ASSIGN
  init(turn)    := 0;
  init(flag[0]) := FALSE;
  init(flag[1]) := FALSE;
```

```
LTLSPEC  -- Sicherheit: Mutex
  G !(P0.critical & P1.critical)

LTLSPEC  -- Lebendigkeit
  G F P0.critical & G F P1.critical;

LTLSPEC  -- No deadlock
  G (P0.waiting & P1.waiting
      -> F (!P0.waiting | !P1.waiting));
```

For Help, press F1 | Ln 55, Col. 2, C0 | DOS | Mod: :

# Der user-Prozess

```
Process Pi:
  while(true){
    flag[i] = true;
    turn    = 1-i;
    wait (!flag[1-i]|(turn==i));
    skip;              -- Kritisch
    flag[i] = false;
  }
```

Umwandlung in Goto-Programm

```
Process Pi:
  0 : flag[i] := true;
  1 : turn    := 1-i;
  2 : if (flag[1-i] & !(turn==i)
      goto 2;
  3 : skip;     -- Kritisch
  4 : flag[i] := false;
  5 : goto 0
```

UltraEdit-32 - [C:\Documents and Settings\g...

File Edit Search Project View Format Column !

Bakery.smv SimplePetereson.smv*

```
MODULE user(i,turn,flag)
VAR
  pc: 0 .. 5;
ASSIGN
-- Program counter : pc
  init(pc) := 0;
  next(pc) := case
     pc = 2 & flag[1-i] & turn=(1-i) : 2;
     1    : (pc + 1) mod 6;
  esac;

  next(turn) := case
        pc = 1 : 1-i;
        1      : turn;
  esac;

  next(flag[i]) := case
        pc = 0   : TRUE;
        pc = 4   : FALSE;
        1        : flag[i];
  esac;

DEFINE
    critical := pc = 3;
    waiting  := pc = 2;
FAIRNESS
  running
```

For Help, press F1 Ln 48, Col. 12, C0

Prozess $P_i$ muss seine pid i „kennen“

in der neuesten Version von NuSMV sind nur Konstanten in Arrayindizes erlaubt

# Granularität der Modellierung

Ein einfaches verteiltes Programm

```
byte x = 1;
byte y = 2;

cobegin
      x = x+y;  ||  y = y+x;
coend
```

Mögliche Endzustände :

$S_0$ : **x = 3, y = 5**
$S_1$ : **x = 4, y = 3**

Das gleiche ( ? ) in Assembler

```
byte x = 1;
byte y = 2;
cobegin
      load  R1,x
      add   R1,y
      store x,R1
 ||
      load  R2,y
      add   R2,x
      store y,R2
coend
```

Mögliche Endzustände

$S_0$ : **x = 3, y = 5**
$S_1$ : **x = 4, y = 3**
$S_2$ : **x = 3, y = 3**

**Zusätzlicher Endzustand :**

# Prädikatenlogik für Kripke-Strukturen

- Eine Kripke-Struktur K=(S,I, R,L) führt zu einer neuen Art von atomaren Propositionen
  - R(x,y) wobei x und y Elementvariablen sind
  - p(x) für jedes $p \in AP \cup \{init\}$
  - x = y
- Die prädikatenlogische Sprache über K hat Quantoren $\forall, \exists$
  - Quantifikation über Elemente von S
    - $\forall x.\ \exists y.\ R(x,y)$,
    - $\forall x.p(x) \Rightarrow \exists y.\ R(x,y) \wedge q(y)$

# Syntax der PL

- Variablen
  - $x,y,z,\ldots \in V$
- Atomare Ausdrücke
  - $p(x)$, falls $p \in AP$, $x=y$, $R(x,y)$, $p(x)$, $init(x)$ für alle Variablen $x,y,.. \in V$
- Ausdrücke

  p ::= a, falls a ein atomarer Ausdruck

  $| \; p_1 \vee p_2 \; | \; p_1 \wedge p_2 \; | \; p_1 \Rightarrow p_2 \; | \; \neg p$

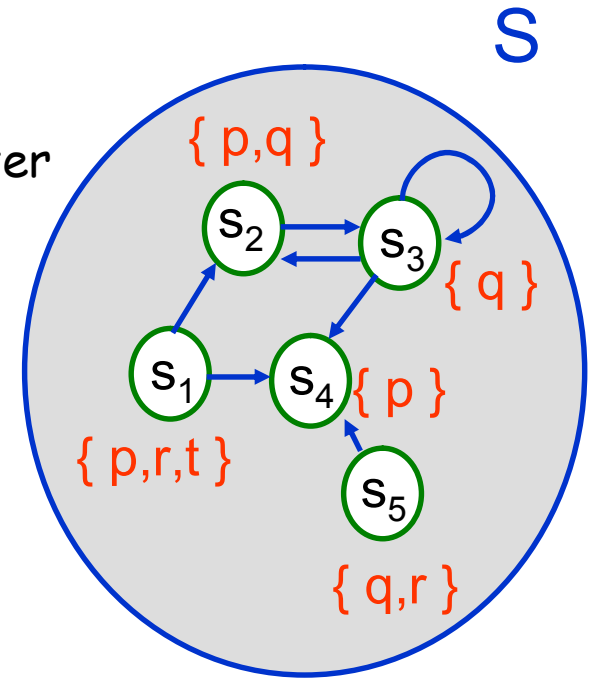
  $| \; \exists x.p \; | \; \forall x.p$
- Aussagen
  - Ausdrücke ohne freie Variablen
- Beispiele
  - Ausdruck A(y) mit einer freien Variablen:

    $\exists x. \neg x=y \vee R(x,y)$
  - Aussage (alle Variablen gebunden) :

    $\forall x. p(x) \Rightarrow \exists y. q(y) \wedge R(x,y)$

# Semantik der PL

- $K \models \varphi$
  - in K gilt Aussage $\varphi$
- Rekursiv über den Aufbau der Formel $\varphi$
  - Schwierigkeit:
    - Teilformeln von $\varphi$ sind Ausdrücke, nicht Aussagen
  - Lösung: *Belegung* $h:V \to S$ als zusätzlicher Parameter
- $K \models_h \varphi$
  - bei Belegung h gilt Formel $\varphi$ in K
- Beispiel:
  - In nebenstehender Kripke Struktur haben wir
  - $K \models_h p(x) \wedge R(x,y) \Rightarrow q(y)$ für $h : x \mapsto s_1 , y \mapsto s_2$
  - $K \not\models_h p(x) \wedge R(x,y) \Rightarrow q(y)$ für $h : x \mapsto s_1 , y \mapsto s_4$
  - $K \models_h \forall y.p(x) \wedge R(x,y) \Rightarrow q(y)$ für $h : x \mapsto s_2 , \ldots$



# $K \models_h \varphi$ - rekursive Definition

- $\varphi$ atomar
  - $K \models_h x=y \quad :\Leftrightarrow \quad h(x) = h(y)$
  - $K \models_h p(x) \quad :\Leftrightarrow \quad p \in L(x)$
  - $K \models_h R(x,y) \quad :\Leftrightarrow \quad (h(x),h(y)) \in R$
- $\varphi$ boolesch
  - $K \models_h \varphi_1 \wedge \varphi_2 \quad :\Leftrightarrow \quad K \models_h \varphi_1$ und $K \models_h \varphi_2$
  - …
- $\varphi$ quantifiziert
  - $K \models_h \exists x.\varphi \quad :\Leftrightarrow \quad$ Für mindestens ein $s \in S$ gilt: $K \models_{h[x:=s]} \varphi$
  - $K \models_h \forall x.\varphi \quad :\Leftrightarrow \quad$ Für jedes $s \in S$ gilt: $K \models_{h[x:=s]} \varphi$

  dabei ist die Belegung $h[x:=s]$ definiert durch $h[x:=s](v) := \text{if}(v=x) \text{ then } s \text{ else } h(v)$
- Falls $\varphi$ eine *Aussage* ist, ist h irrelevant
  - $K \models \varphi \quad :\Leftrightarrow \quad K \models_h \varphi$ für irgendein/jedes h

# Transitionssysteme

- T=(S,I, Act,$\rightarrow$,AP,L)
- (S,AP,L) Kontext und I $\subseteq$ S
  - Act Menge von Aktionsnamen
  - $\rightarrow \subseteq S \times Act \times S$ Transitionsrelation
  - statt $(s,\alpha,s')$ schreiben wir $s \xrightarrow{\alpha} s'$
- Andere Sichtweise:
  - Für jedes $\alpha \in$ Act gibt es eine Relation $\xrightarrow{\alpha}$
  - Kripkestruktur mit vielen Relationen
- Lauf ist jetzt Folge
  - $s_0, \alpha_0, s_1, \alpha_1, \ldots, \alpha_{n-1}\, s_n$ mit $s_0 \in I$ und $\forall\, 0 \leq i \leq n.\ (s_i,\alpha_i,s_{i+1}) \in \rightarrow$ .
- In SMV werden die $\alpha \in$ Act als **IVAR** gekennzeichnet

# Grenzen der Prädikatenlogik

- Einige relevanten Eigenschaften lassen sich in der PL nicht ausdrücken
  - Beispiel: Erreichbarkeit -- von jedem p-Zustand ist ein q-Zustand erreichbar:

    $\forall x.\ p(x) \Rightarrow \exists n \in N.\ \exists s_1.\ \exists s_2, \ldots, \exists s_n.\ x=s_1 \wedge R(s_1,s_2) \wedge \ldots \wedge R(s_{n-1},s_n) \wedge s_n = y \wedge q(y)$

    - Quantifikation nur über Elemente aus S erlaubt
    - „…" kein syntaktisch erlaubter Ausdruck
  - Formulierung in Prädikatenlogik würde unendliche Disjunktionen/Konjunktionen benötigen:
  - $\forall x.\ p(x) \Rightarrow \exists s_1.\ \exists s_2,.\ x=s_1 \wedge R(s_1,s_2) \wedge s_2 = y \wedge q(y)$

    $\vee\ \exists s_1.\ \exists s_2,\ \exists s_3.\ x=s_1 \wedge R(s_1,s_2) \wedge R(s_2,s_3) \wedge s_3 = y \wedge q(y)$

    $\vee\ \exists s_1.\ \exists s_2,\ \exists s_3.\ \exists s_4.\ x=s_1 \wedge R(s_1,s_2) \wedge R(s_2,s_3) \wedge R(s_3,s_4) \wedge s_4 = y \wedge q(y)$

    $\vee$ …. etc. …
- Überprüfung beliebiger prädikatenlogischer Eigenschaften aufwendig
  - exponentiell in der „Größe" der Formel

# Anforderung an Sprachen für verteilte Systeme

- Erreichbarkeit und zeitliche Verläufe einfach zu formulieren
  - irgendwann gilt p danach gilt immer q
  - es ist unmöglich in einen Zustand mit q zu gelangen
  - p gilt bis q eintritt
  - p gilt unendlich oft
- Die Sprache muss
  - einfach sein, aber
  - ausdrucksstark
- Eigenschaften müssen
  - automatisch nachprüfbar oder
  - widerlegbar sein.
- In den nächsten Kapiteln stellen wir drei solche Sprachen vor:
  - LTL - lineare temporale Logik
  - CTL - computation tree logic
  - CTL*