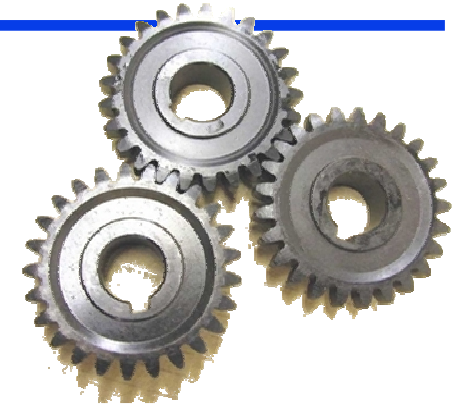
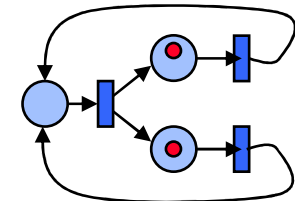
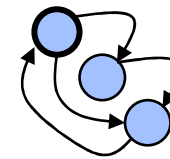

Modellierung und Analyse eingebetteter und verteilter Systeme



Thread „Funktionalität“ *Teil 2*

- ◆ Einleitung
- ◆ Zustandstransitionssysteme
- ◆ Petrinetz und Partialordnungsmodelle
- ◆ Prozessalgebra: CCS
- ◆ Temporale Logik: LTL, CTL, CTL*
- ◆ Erreichbarkeitsanalyse und Model Checking
- ◆ Eigenschaftsbeweise im STS
 - Safety: Zustandsinvarianten und Induktionsbeweis
 - Liveness: Leads-to-Ketten, Fairness- und Lattice-Regeln



$$X = (a.b.c.Y + D) \mid X$$

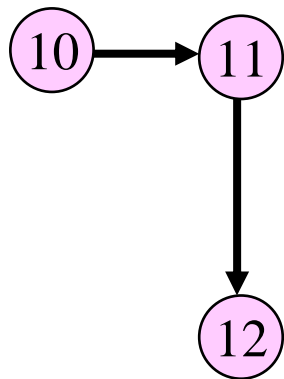
$$E((EX.P)U(AG.Q))$$

F: Funktionaler Thread – Inhalte

1. Einleitung
2. Erweiterter Mealy-Automat
3. Petri Netz
4. Gefärbtes Transitionssystem (LTS)
5. Calculus of Communicating Systems (CCS)
- 6. Einfaches Zustandstransitionssystem (STS)**
- 7. Safety und Liveness im STS**
- 8. Erreichbarkeitsanalyse**
- 9. Logiken (LTL, CTL, CTL*)**
- 10. Model Checking**
- 11. Safety- und Livenessbeweise**

F6: Einfaches Zustandstransitionssystem (STS)

- ◆ Definition
- ◆ Erreichbarer Zustand
- ◆ Definition über Variablen und Aktionen
- ◆ Hilfsvariablen
- ◆ Abläufe als Zustandsfolgen
- ◆ Unendliche Zustandsfolgen und Stotterschritte



Literatur

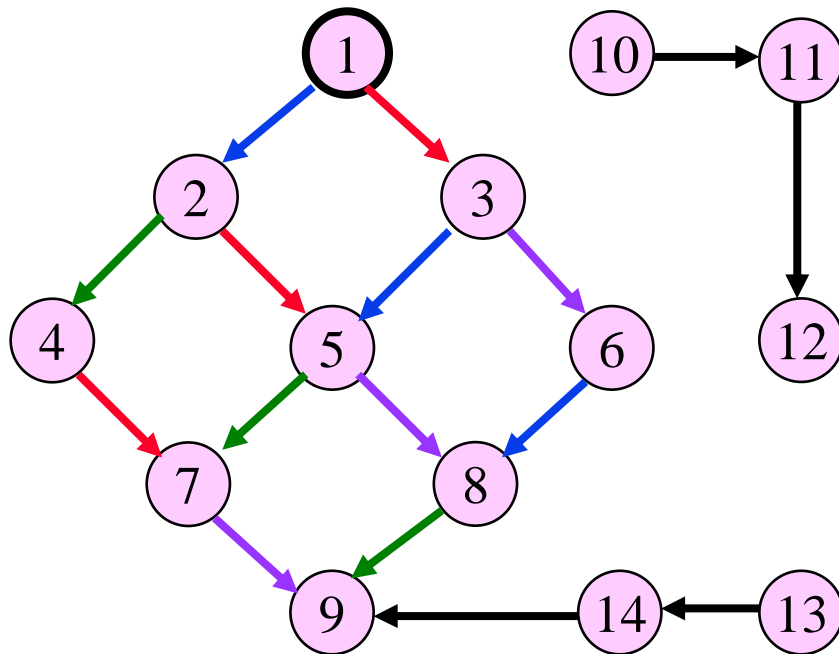
Leslie Lamport: The temporal logic of actions (TLA). ACM TOPLAS, 16(3) 872-923, May 1994.

F6: Definition STS

Zustandstransitionssystem (State Transition System STS)

STS = $\langle S, S_0, \text{Next} \rangle$

- **S** Menge von Zuständen (nicht unbedingt endlich)
- **S₀** Menge von Startzuständen, $S_0 \subset S$
- **Next** Folgezustandsrelation, $\text{Next} \subset S \times S$

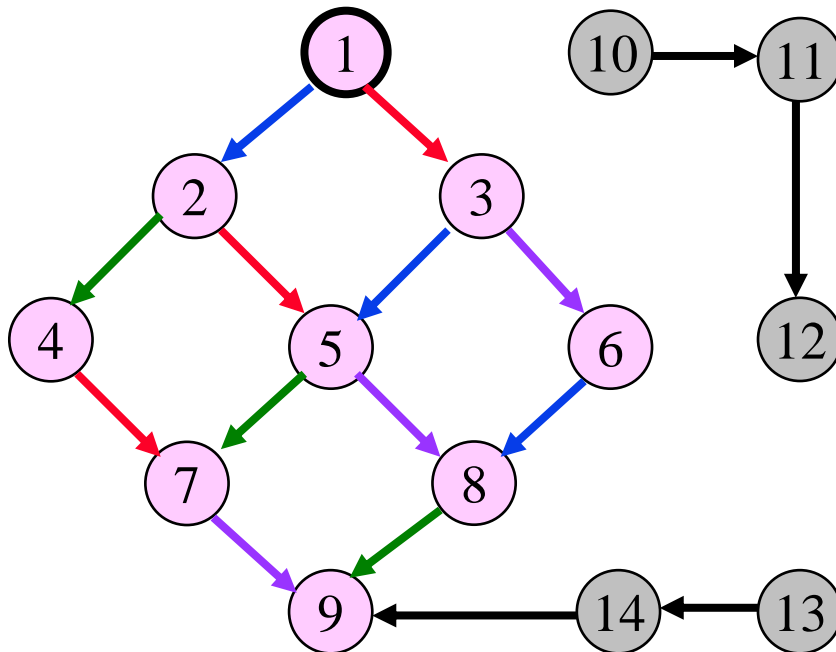


S = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
S₀ = {1}
Next = { $\langle 1, 2 \rangle$, $\langle 3, 5 \rangle$, $\langle 6, 8 \rangle$,
 $\langle 2, 4 \rangle$, $\langle 5, 7 \rangle$, $\langle 8, 9 \rangle$,
 $\langle 1, 3 \rangle$, $\langle 2, 5 \rangle$, $\langle 4, 7 \rangle$,
 $\langle 3, 6 \rangle$, $\langle 5, 8 \rangle$, $\langle 7, 9 \rangle$,
 $\langle 10, 11 \rangle$, $\langle 11, 12 \rangle$,
 $\langle 13, 14 \rangle$, $\langle 14, 9 \rangle$ }

F6: STS – Erreichbare Zustände

Ein Zustand s eines **STS** $\langle S, S_0, \text{Next} \rangle$ ist erreichbar:

- $s \in S_0$ oder
- $\exists s^* \in S: s^* \text{ erreichbar} \wedge \langle s^*, s \rangle \in \text{Next}$



$S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$
 $S_0 = \{1\}$
 $\text{Next} = \{ \langle 1, 2 \rangle, \langle 3, 5 \rangle, \langle 6, 8 \rangle, \langle 2, 4 \rangle, \langle 5, 7 \rangle, \langle 8, 9 \rangle, \langle 1, 3 \rangle, \langle 2, 5 \rangle, \langle 4, 7 \rangle, \langle 3, 6 \rangle, \langle 5, 8 \rangle, \langle 7, 9 \rangle, \langle 10, 11 \rangle, \langle 11, 12 \rangle, \langle 13, 14 \rangle, \langle 14, 9 \rangle \}$

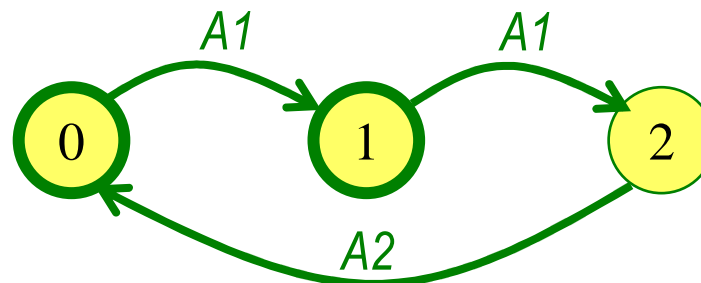
F6: STS – Definition über Variablen und Aktionen

STS $\langle \mathbf{S}, \mathbf{S}_0, \mathbf{Next} \rangle$ kann definiert werden durch

- Menge von Datenvariablen V_1, V_2, \dots, V_n
mit den Wertbereichen W_1, W_2, \dots, W_n
- Initialisierungsprädikat **init** über Variablen
- Menge von Aktionsprädikaten A_1, A_2, \dots, A_m über Variablen und Folgevariablen
 - » wenn V eine Variable ist, steht V für den Wert der Variablen im Momentanzustand und V' für den Wert von V im Folgezustand der Transitionen

mit den Festlegungen

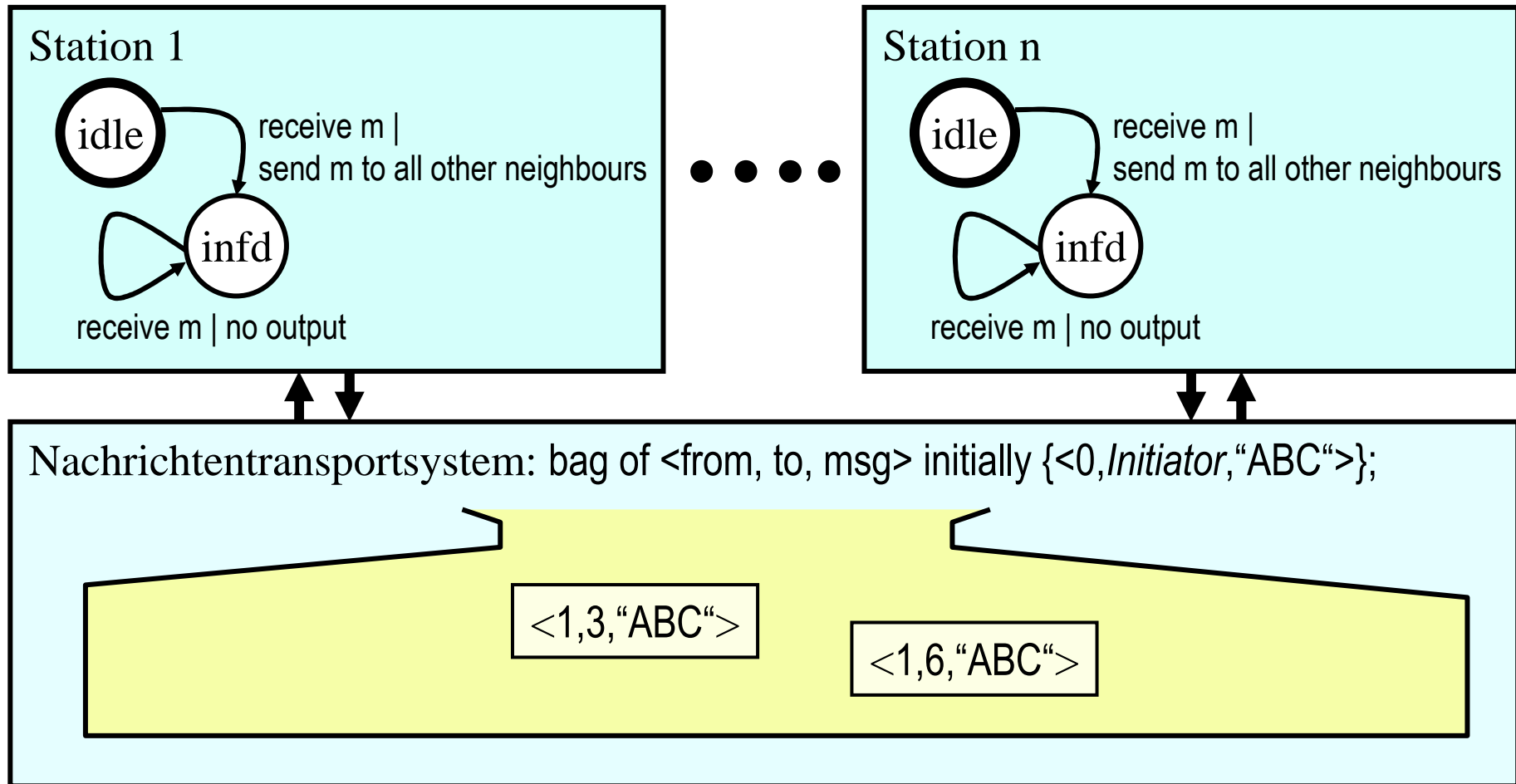
- $\mathbf{S} = W_1 \times W_2 \times \dots \times W_n$
- $\mathbf{S}_0 = \{ s : \mathbf{init} \text{ ist wahr in } s \}$
- $\mathbf{Next} = \{ \langle s, s' \rangle : A_1 \vee A_2 \vee \dots \vee A_m \text{ ist wahr für } \langle s, s' \rangle \}$



```
var V : (0, 1, 2);  
init V=0 ∨ V=1;  
act A1: V<2 ∧ V'=V+1;  
    A2: V=2 ∧ V'=0;
```

F6: STS – Beispiel

- ◆ System aus gekoppelten Zustandsmaschinen (Mealy)
(initial sei 1 Nachricht, adressiert an die Station Initiator, im Transportsystem)



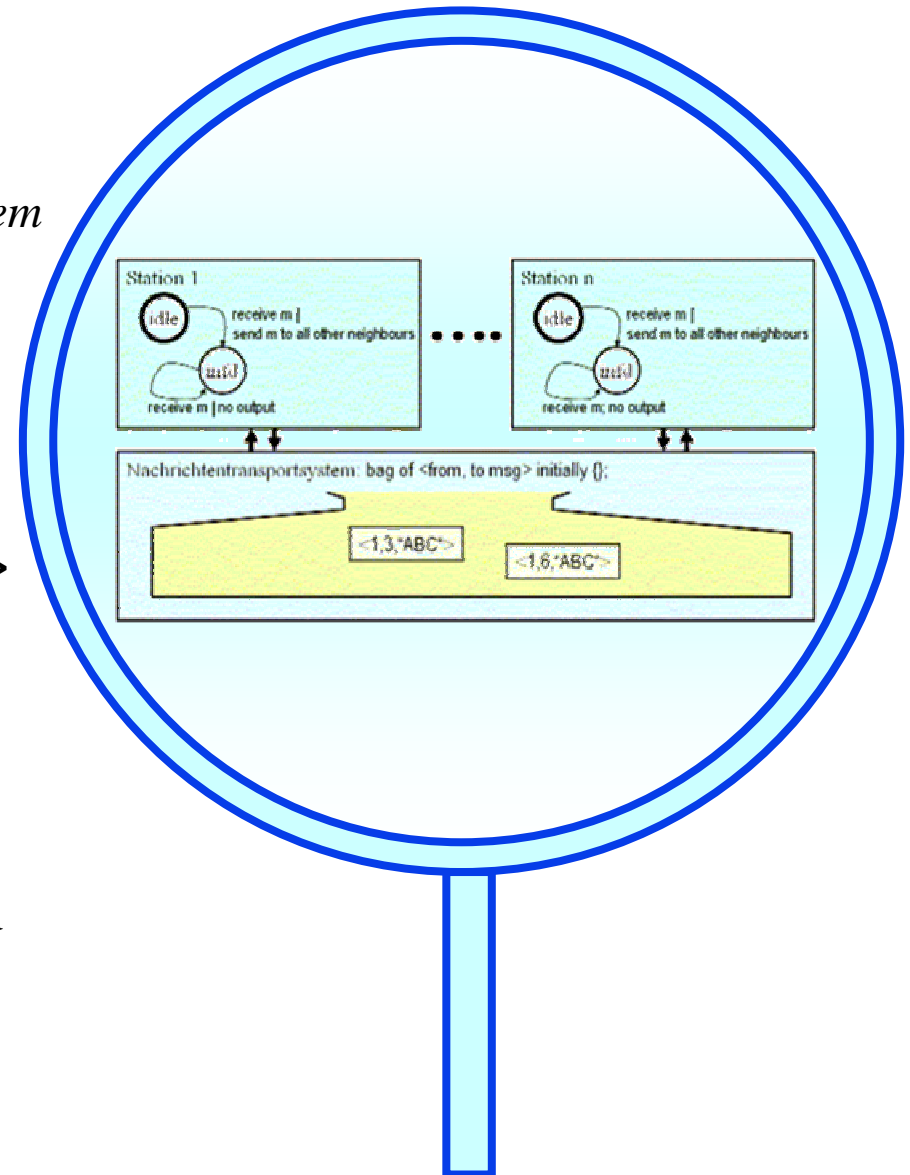
F6: STS – Beispiel

- ◆ Eine einzige (globale) Zustandsmaschine
 - Zustandsraum
cs: array [1..n] of (idle, infd) ! Stationen
nts: bag of < from, to, msg > ! Transportsystem

- Zustand
Vektor mit n+1 Stellen:
z.B. (es gelte n=3, d.h. es gibt 3 Stationen)

<infd, idle, idle, {<1,2,“ABC“>, <1,3,“ABC“>} >

In diesem Zustand hat Station 1 den Ablauf gestartet und 2 Nachrichten mit dem Inhalt „ABC“, je eine an 2 und an 3, gesendet. Die beiden Stationen 2 und 3 haben diese Nachrichten noch nicht empfangen. Sie sind noch idle und die Nachrichten sind noch im Transportsystem.



F6: STS – Beispiel

- ◆ **S** = Menge aller möglichen Wertbelegungen der Variablen
 - **cs**: array [1..n] of (idle, infd) ! Stationen
 - **nts**: bag of <from, to, msg> ! Transportsystem
- ◆ **S₀** = Menge aller Zustände, welche die folgende Initialisierungsbedingung erfüllen
 - **cs** = < idle, idle, ..., idle > \wedge
nts = {<0, Initiator, Text>}
- ◆ **Next** = Menge aller Transitionen, welche „**Forward v Skip**“ erfüllen.
 - **Forward**

$$\exists i, m: cs[i]=idle \wedge m \in nts \wedge m.to=i \wedge$$

$$cs[i]'=infd \wedge \forall j \neq i: cs[j]'=cs[j] \wedge$$

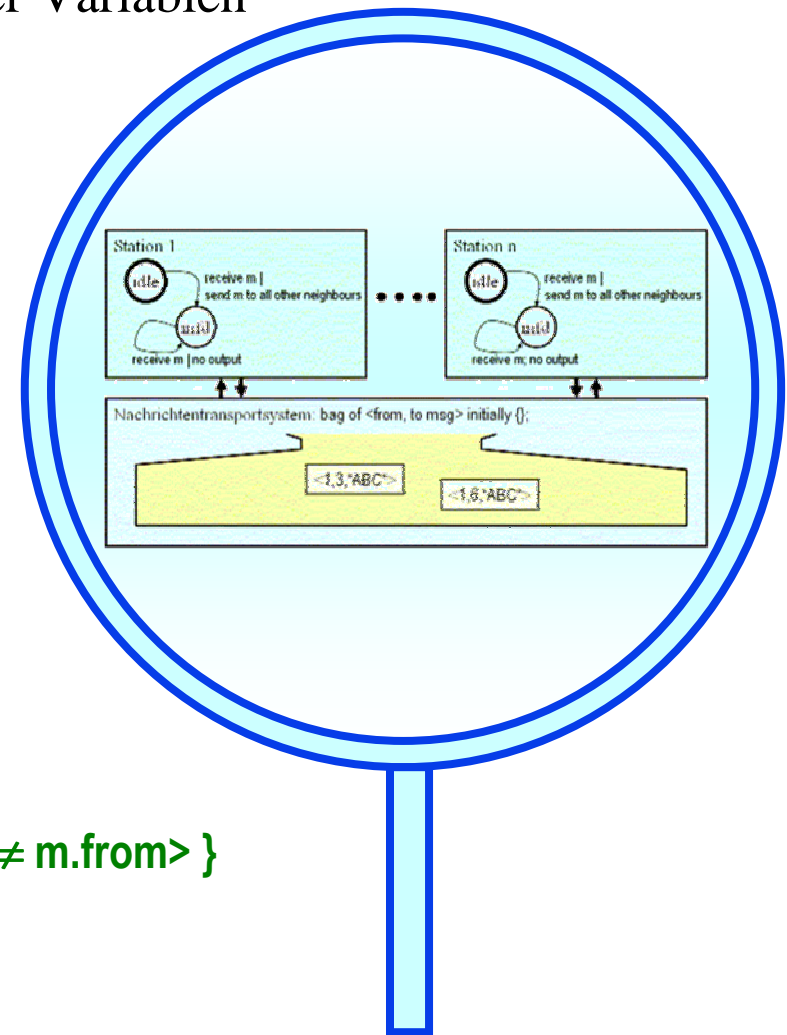
$$nts' = nts \cup$$

$$\{ \langle i, k, m.msg \rangle : istNachbar(k,i) \wedge k \neq m.from \}$$

$$\setminus \{m\}$$
 - **Skip**

$$\exists i, m: cs[i]=infd \wedge m \in nts \wedge m.to=i \wedge$$

$$\forall j: cs[j]'=cs[j] \wedge nts'=nts \setminus \{m\}$$



F6: STS – Beispiel

Variablen

cs: array [1..n] of (idle, infd) ! *Stationen*

nts: bag of < from, to, msg > ! *Transportsystem*

Init

cs = < idle, idle, ..., idle > \wedge

nts = {<0, Initiator, Text>}

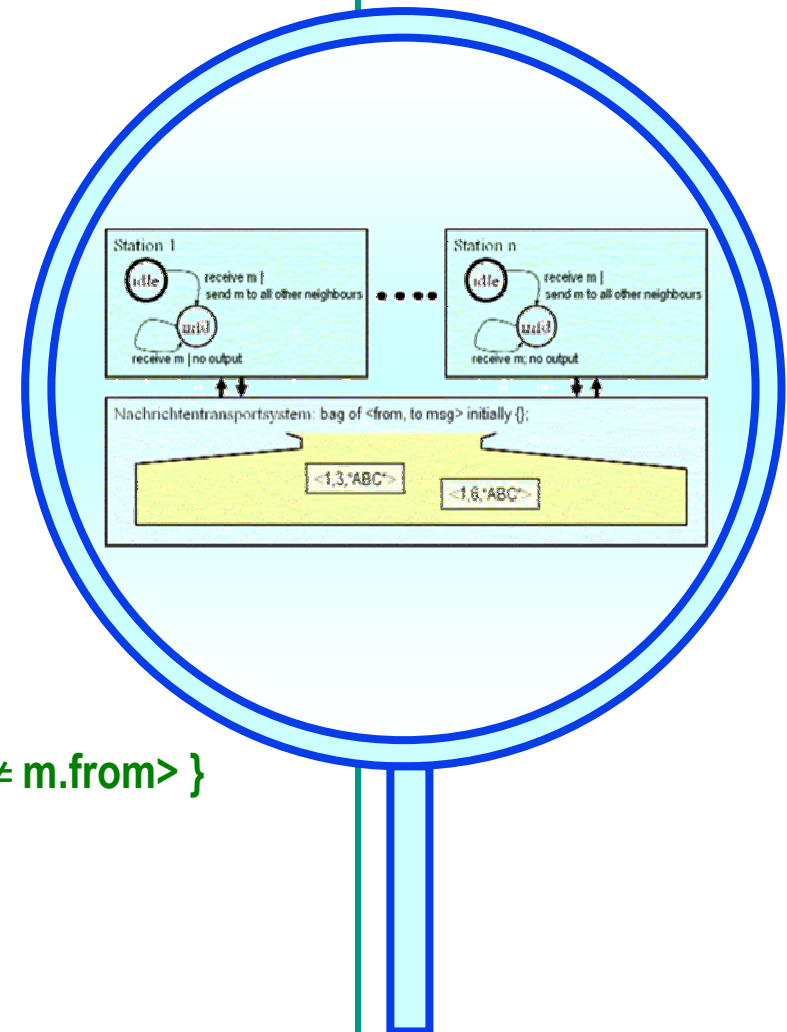
Aktionen

Forward

$\exists i, m$: **cs**[i]=idle \wedge $m \in$ nts \wedge m.to=i \wedge
cs[i]'=infd \wedge $\forall j \neq i$: **cs**[j]'=**cs**[j] \wedge
 nts'= nts \cup
 { <i, k, m.msg>: istNachbar(k,i) \wedge k \neq m.from }
 \setminus {m}

Skip

$\exists i, m$: **cs**[i]=infd \wedge $m \in$ nts \wedge m.to=i \wedge
 $\forall j$: **cs**[j]'=**cs**[j] \wedge nts'=nts \setminus {m}



F6: STS – Aktionsprädikate

Allgemein ist ein Aktionsprädikat eine Bedingung über Variablen und Folgevariablen V, V'

Nach Möglichkeit wird die spezielle Form **<Guard, Effekt>** verwendet

- **Aktion = Guard \wedge Effekt**
- **Guard** ist ein Prädikat, in welchem keine Folgevariablen vorkommen, d.h. **Guard** ist eine Bedingung allein für den Momentanzustand
- **Effekt** ist eine Konjunktion von „Zuweisungen“ der Form **Folgevariable = <Ausdruck über Variablen>**

z.B.

$$V1' = (V3-5)*2$$

Jede Variable darf höchstens einmal vorkommen.

Forward

```

$$\exists i, m: cs[i]=idle \wedge m \in nts \wedge m.to=i \wedge$$

$$cs[i]'=infd \wedge \forall j \neq i: cs[j]'=cs[j] \wedge$$

$$nts' = nts \cup$$

$$\{ \langle i, k, m.msg \rangle : istNachbar(k,i) \wedge k \neq m.from \}$$

$$\setminus \{m\}$$

```

Skip

```

$$\exists i, m: cs[i]=infd \wedge m \in nts \wedge m.to=i \wedge$$

$$\forall j : cs[j]'=cs[j] \wedge nts'=nts \setminus \{m\}$$

```

F6: STS – Hilfsvariablen

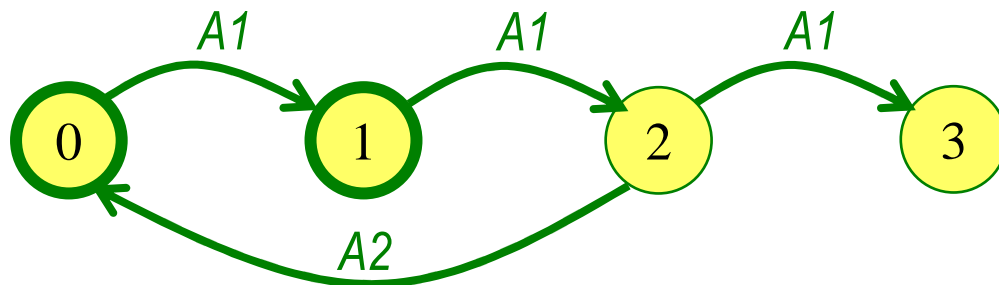
- ◆ Durch Einführung geeigneter Hilfsvariablen können alle interessierenden Safety-Eigenschaften eines Systems ausgedrückt werden.
- ◆ Hilfsvariable
 - neue Variable
 - darf System nicht beeinflussen
 - darf nur Informationen zur Historie aus dem System speichern
- ◆ Beispiel aus der Programmierung
 - Anreichern eines Programms mit Testausgaben
 - Dazu Einführen neuer Variablen
(z.B. Einrückzähler, Kontext-Speicher)

*S. Owicki, D. Gries:
'Verifying properties of parallel programs:
An axiomatic approach',
CACM 19(5), 279–285, 1976.*

F6: STS – Hilfsvariablen

- ◆ Hilfsvariable einführen, so dass das System nicht beeinflusst wird.
- ◆ Wie?
 - neue Variable **H** einführen
 - Initialzustandsbedingung **B** für neue Variable einbringen, welche die Startwerte anderer Variablen nicht einschränken darf.
 - » Init: $\text{BisherigeBedingung} \wedge \mathbf{B}$ so dass gilt
 $\text{BisherigeBedingung} \Rightarrow \exists \mathbf{H}: \text{BisherigeBedingung} \wedge \mathbf{B}$
- ◆ In Aktionen, bei spezieller **<Guard, Effekt>**-Form
 - die Hilfsvariable darf im **Guard** überhaupt nicht vorkommen
 - die Hilfsvariable darf im **Effekt** nur vorkommen
 - » als **Folgevariable** auf der linken Seite einer „Zuweisung“
 - » als **Variable** auf der rechten Seite bei einer Hilfsvariablen-Zuweisung, aber nicht bei einer Zuweisung an eine „richtige“ Zustandsvariable des Systems

F6: STS – Hilfsvariablen



```
var V : (0, 1, 2, 3) ;  
init V=0  $\vee$  V=1 ;  
act A1: V<3  $\wedge$  V'=V+1 ;  
    A2: V=2  $\wedge$  V'=0 ;
```

Wie oft wurde die Schleife schon durchlaufen?

- ◆ Offensichtlich verändert **H** das Systemverhalten nicht. **H** wird nur beschrieben.
- ◆ Offensichtlich wird **H** genau mit jeder A2-Transition 1-mal erhöht.
- ◆ **H** erweitert den Zustand. In der Zustandskomponente **H** findet sich immer die Anzahl der schon durchgeführten Schleifenrücksprünge.

```
var V : (0, 1, 2, 3) ;  
    H : integer ; ! Hilfsvariable  
init ( V=0  $\vee$  V=1 )  $\wedge$  ( H=1 ) ;  
act A1: V<3  $\wedge$  V'=V+1  $\wedge$  H'=H ;  
    A2: V=2  $\wedge$  V'=0  $\wedge$  H'=H+1 ;
```

F6: STS – Hilfsvariablen

- Safety-Aussage informal:
 „Es werden nicht mehr als n^2 Nachrichten gesendet.“
- Wie formalisieren?
 Die Anzahl der gesendeten Nachrichten ist im Zustand nicht erkennbar!
- Wie würde man vorgehen, wenn das ein zu debuggendes Programm wäre:
 Testausgaben einer Testhilfevariable, die mitzählt

Variablen

cs: array [1..n] of (idle, infd) ! *Stationen*

nts: bag of < from, to, msg > ! *Transportsystem*

Init

cs = < idle, idle, ..., idle > \wedge

nts = {<0, Initiator, Text>}

Aktionen

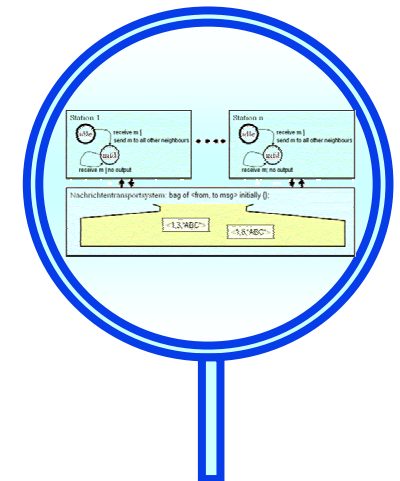
Forward

$\exists i, m: cs[i]=idle \wedge m \in nts \wedge m.to=i \wedge$
 $cs[i]'=infd \wedge \forall j \neq i: cs[j]'=cs[j] \wedge$
 $nts' = nts \cup$

$\{ \langle i, k, m.msg \rangle : istNachbar(k,i) \wedge k \neq m.from \}$
 $\setminus \{m\}$

Skip

$\exists i, m: cs[i]=infd \wedge m \in nts \wedge m.to=i \wedge$
 $\forall j: cs[j]'=cs[j] \wedge nts' = nts \setminus \{m\}$



F6: STS – Hilfsvariablen

- Safety-Aussage informal:
 „Es werden nicht mehr als n^2 Nachrichten gesendet.“
- Wie formalisieren?
 Die Anzahl der gesendeten Nachrichten ist im Zustand nicht erkennbar!
- Lösung: Hilfsvariable **MC** erweitert den Zustandsraum um einen Nachrichtenzähler
- Aussage formal:
 $MC \leq n^2$

Variablen

cs: array [1..n] of (idle, infd) ! Stationen

nts: bag of < from, to, msg > ! Transportsystem

MC : integer ! Hilfsvariable – Anzahl gesendeter Nachrichten

Init

cs = < idle, idle, ..., idle > \wedge

nts = {<0, Initiator, Text>} \wedge **MC**=0

Aktionen

Forward

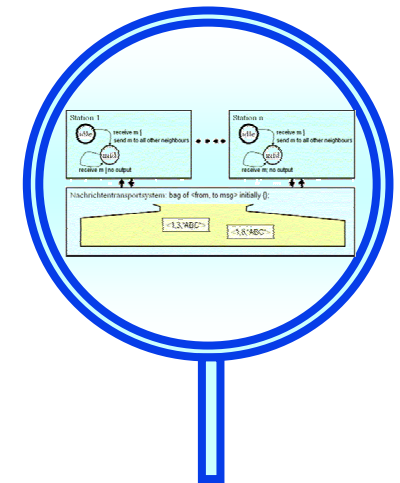
$\exists i, m$: **cs**[i]=idle \wedge $m \in$ nts \wedge m.to=i \wedge
cs[i]'=infd \wedge $\forall j \neq i$: **cs**[j]'=**cs**[j] \wedge
 nts' = nts \cup

{<i, k, m.msg>: istNachbar(k,i) \wedge k \neq m.from} \setminus {m} \wedge

MC'=**MC** + AnzNachbVon(i) - 1

Skip

$\exists i, m$: **cs**[i]=infd \wedge $m \in$ nts \wedge m.to=i \wedge **MC**'=**MC** \wedge
 $\forall j$: **cs**[j]'=**cs**[j] \wedge nts' = nts \setminus {m}



F6: STS – Abläufe als Zustandsfolgen

Die **Ausführung** eines **STS $\langle S, S_0, \text{Next} \rangle$** kann in Form der dabei auftretenden Folge eingenommener Zustände repräsentiert werden.

Def: **Mögliche Zustandsfolge σ** eines **STS $\langle S, S_0, \text{Next} \rangle$**

- $\sigma \in S^* \cup S^\infty, \sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$
- $s_0 \in S_0$
- $\forall i: \langle s_i, s_{i+1} \rangle \in \text{Next} \vee s_i = s_{i+1}$

Jedes aufeinanderfolgende Zustandspaar einer Zustandsfolge entspricht einem Schritt des **STS**

- $s_i = s_{i+1}$: **Stottersschritt**, das System führt keine Transition aus,
- $\langle s_i, s_{i+1} \rangle \in \text{Next} \wedge s_i \neq s_{i+1}$: echter Schritt, das System führt eine **Transition** aus.

Def: **Menge Σ** aller möglichen Zustandsfolgen eines **STS**

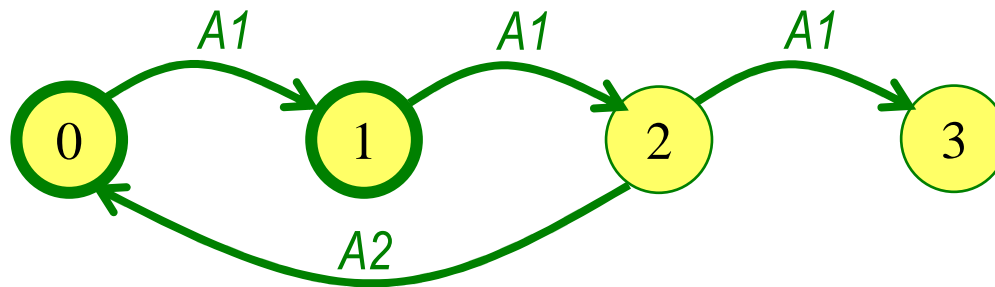
- $\Sigma = \{ \sigma : \sigma \text{ ist mögliche Zustandsfolge des STS} \}$

F6: STS – Unendliche Zustandsfolgen und Stotterschrte

Warum Stotterschrte? *1. am Ende einer Folge 2. in der Mitte einer Folge*

1. **Terminierende Systeme** (sie können nur endlich viele Transitionen ausführen und erreichen schließlich einen Zustand, von dem aus keine weitere Transition möglich ist) können in der Form ihrer Zustandsfolgen gleichbehandelt werden mit nicht-terminierenden Systemen.
 - Ein terminierendes System führt am Ende unendlich viele Stotterschrte aus und hat darum ebenfalls unendliche Zustandsfolgen.
Es genügt deshalb, generell nur unendliche Folgen zu betrachten.
 2. Erleichterte Modellierung **zusammengesetzter Systeme**
 - Wenn ein System A (z.B. als Subsystem) im Zusammenhang mit einem anderen System B betrachtet wird, kann es sein, dass B eine Transition ausführt, A aber nicht (Interleaving).
Bei Betrachtung des Gesamtsystems AB erscheint eine Gesamtsystem-Transition, welche aus einem A-Stotterschrte und einer B-Transition besteht.
- Wir betrachten im Folgenden unendliche Zustandsfolgen. Es gilt $\sigma \in S^\infty$.
 - Wenn σ eine mögliche Zustandsfolge eines **STS** ist, und σ^+ aus σ durch Einfügen von Stotterschrten entsteht, ist σ^+ auch eine mögliche Zustandsfolge des **STS**.
 - Wieviele Stotterschrte dürfen eingefügt werden?
Bei unendlich vielen stoppt das System \rightarrow Liveness-Aspekte

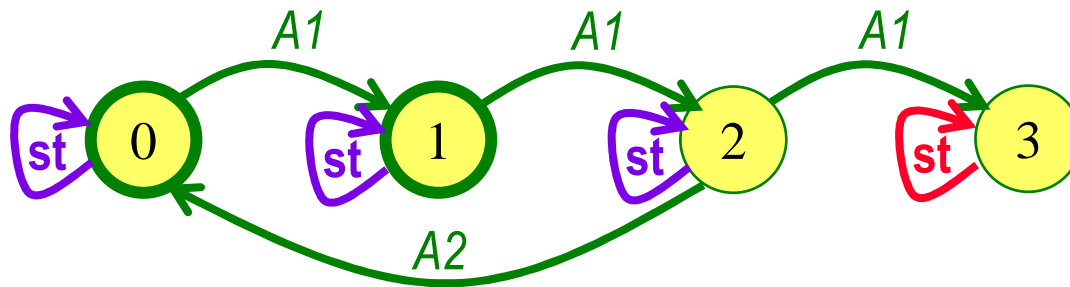
F6: STS – Unendliche Zustandsfolgen und Stotterschritte



```
var V : (0, 1, 2, 3);  
init V=0 ∨ V=1;  
act  A1: V<3 ∧ V'=V+1;  
      A2: V=2 ∧ V'=0;
```

- ◆ $\langle 0, 1, 2, 3 \rangle$
- ◆ $\langle 0, 1, 2, 0, 1, 2, 3 \rangle$
- ◆ $\langle 0, 1, 2, 0, 1, 2, 0, 1, 2, 3 \rangle$
- ◆ $\langle 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 3 \rangle$
- ◆ ...

F6: STS – Unendliche Zustandsfolgen und Stottersschritte



```

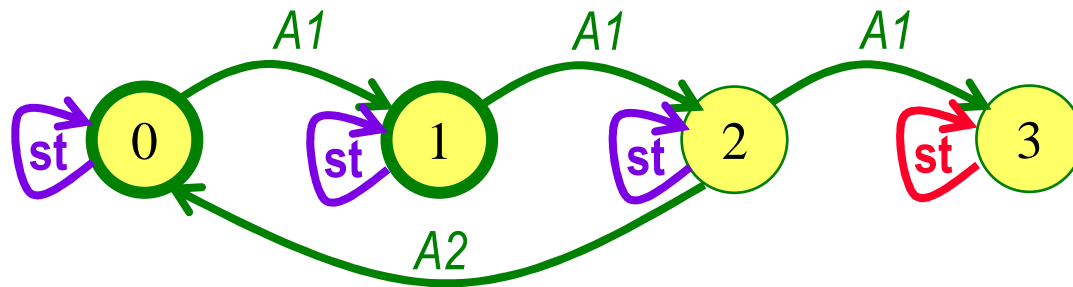
var V : (0, 1, 2, 3);
init V=0 ∨ V=1;
act A1: V<3 ∧ V'=V+1;
    A2: V=2 ∧ V'=0;
  
```

- ◆ $\langle 0, 1, 2, 3 \rangle$
- ◆ $\langle 0, 1, 2, 0, 1, 2, 3 \rangle$
- ◆ $\langle 0, 1, 2, 0, 1, 2, 0, 1, 2, 3 \rangle$
- ◆ $\langle 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 3 \rangle$
- ◆ ...

*Einstreuen endlich vieler Stottersschritte
Unendliches Stottern nach Terminierung*

- ◆ $\langle 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 3, 3, 3, \dots \rangle$
- ◆ $\langle 0, 0, 1, 1, 1, 2, 2, 2, 2, 0, 0, 1, 1, 1, 1, 2, 2, 3, 3, 3, \dots \rangle$
- ◆ $\langle 0, 0, 1, 2, 0, 0, 0, 1, 2, 0, 1, 2, 2, 2, 2, 2, 2, 3, 3, 3, \dots \rangle$
- ◆ $\langle 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 3, 3, 3, \dots \rangle$
- ◆ ...

F6: STS – Unendliche Zustandsfolgen und Stotterschritte



```

var V : (0, 1, 2, 3) ;
init V=0 ∨ V=1 ;
act A1: V<3 ∧ V'=V+1 ;
    A2: V=2 ∧ V'= 0 ;
    
```

- ◆ $\langle 0, 1, 2, 3 \rangle$
- ◆ $\langle 0, 1, 2, 0, 1, 2, 3 \rangle$
- ◆ $\langle 0, 1, 2, 0, 1, 2, 0, 1, 2, 3 \rangle$
- ◆ $\langle 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 3 \rangle$
- ◆ ...

Einstreuen unendlich vieler Stotterschritte
 → *Vorzeitiger Stop,*
System terminiert nicht wie gewünscht.
Implizite versus explizite Liveness-Angaben

- ◆ $\langle 0, 0, 0, 0, \dots \rangle$
- ◆ $\langle 0, 1, 1, 1, 1, \dots \rangle$
- ◆ $\langle 0, 1, 2, 2, 2, 2, \dots \rangle$

F6: STS – Unendliche Zustandsfolgen und Stotterschritte

- Variante „**Implizit**“
 System muss generell,
 wenn in einem Zustand
 eine echte Transition
 möglich ist, nach
 endlicher Zeit
 irgendeine echte
 Transition ausführen.

In vielen Fällen kann
 damit eine
 gewünschte
 Liveness nicht
 differenziert genug
 ausgedrückt
 werden.

- Soll im Beispiel
Loss auch „*richtig
 fleißig*“ schalten??

Variablen

cs: array [1..n] of (idle, infd) ! *Stationen*

nts: bag of < from, to, msg > ! *Transportsystem*

Init

cs = < idle, idle, ..., idle > \wedge

nts = {<0, Initiator, Text>}

Aktionen

Forward

$\exists i, m: cs[i]=idle \wedge m \in nts \wedge m.to=i \wedge$
 $cs[i]'=infd \wedge \forall j \neq i: cs[j]'=cs[j] \wedge$

$nts' = nts \cup$

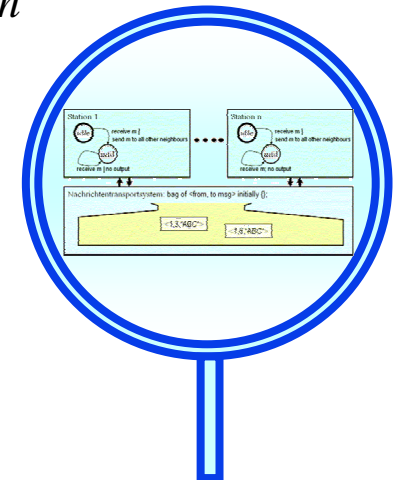
$\{ \langle i, k, m.msg \rangle : istNachbar(k,i) \wedge k \neq m.from \}$
 $\setminus \{m\}$

Skip

$\exists i, m: cs[i]=infd \wedge m \in nts \wedge m.to=i \wedge$
 $\forall j: cs[j]'=cs[j] \wedge nts'=nts \setminus \{m\}$

Loss

$\exists m: m \in nts \wedge \forall j: cs[j]'=cs[j] \wedge nts'=nts \setminus \{m\}$



F6: STS – Unendliche Zustandsfolgen und Stotters Schritte

- Variante „**Explizit**“
 Es werden zusätzliche ausdrückliche Liveness-Anforderungen formuliert.
 - z.B. wichtige Schritte erzwingen
 - störende Schritte nur tolerieren
 - Schritte nur unter bestimmten Umständen erzwingen
- Dazu später mehr unter **Aktionen-Fairness**

Variablen

cs: array [1..n] of (idle, infd) ! *Stationen*

nts: bag of < from, to, msg > ! *Transportsystem*

Init

cs = < idle, idle, ..., idle > \wedge

nts = { < 0, Initiator, Text > }

Aktionen

Forward

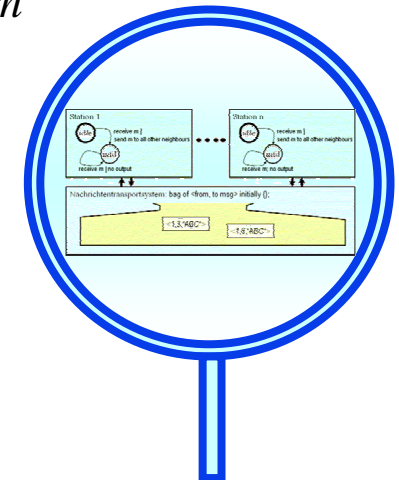
$\exists i, m: cs[i]=idle \wedge m \in nts \wedge m.to=i \wedge$
 $cs[i]'=infd \wedge \forall j \neq i: cs[j]'=cs[j] \wedge$

$nts' = nts \cup$

$\{ \langle i, k, m.msg \rangle : istNachbar(k,i) \wedge k \neq m.from \}$
 $\setminus \{m\}$

Skip

$\exists i, m: cs[i]=infd \wedge m \in nts \wedge m.to=i \wedge$
 $\forall j: cs[j]'=cs[j] \wedge nts'=nts \setminus \{m\}$



F7: Safety und Liveness im STS

- ◆ Systemeigenschaft als Zustandsfolgenmenge
- ◆ Safety- und Liveness-Eigenschaften
- ◆ Safety per STS
- ◆ Safety per Zustandsinvariante
- ◆ Aktionen-Fairness
- ◆ Liveness per Aktionen-Fairness

Literatur

B. Alpern, F. Schneider:

„Defining liveness“.

Information Processing Letters 21, 4 (1985).

F7: Systemeigenschaft als Zustandsfolgenmenge

◆ Erinnerung

Def: **Mögliche Zustandsfolge** σ eines **STS** $\langle S, S_0, \text{Next} \rangle$

- $\sigma \in S^\infty$, $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$, $s_0 \in S_0$, $\forall i: \langle s_i, s_{i+1} \rangle \in \text{Next} \vee s_i = s_{i+1}$
- Jeder Ablauf / jede Ausführung eines Systems kann als unendliche Folge von Zuständen dargestellt werden (bei Terminierung: unendliches Stottern am Ende)
- Σ sei die Menge aller möglichen Zustandsfolgen eines Systems **STS** $\langle S, S_0, \text{Next} \rangle$

◆ Def: **Eigenschaft** Π

$\Pi \subset 2^{(S^\infty)}$, d.h. Π ist Menge von Zustandsfolgen

Anmerkung: Wie kann man eine **Eigenschaft** definieren?

A] **Intensional**

Charakteristika der Eigenschaft darstellen.

B] **Extensional**

Menge der Dinge aufzählen, welche die Eigenschaft besitzen.

Beispiel:

*„Farbe **Lila** haben“*

*A] **Lichtfrequenz lila***

*B] **Menge der lila Dinge***

F7: Systemeigenschaft als Zustandsfolgenmenge

◆ Def: **Mögliche Zustandsfolge** σ eines **STS** $\langle S, S_0, \text{Next} \rangle$

- $\sigma \in S^\infty$, $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$, $s_0 \in S_0$, $\forall i: \langle s_i, s_{i+1} \rangle \in \text{Next} \vee s_i = s_{i+1}$
- Jeder Ablauf / jede Ausführung eines Systems kann als unendliche Folge von Zuständen dargestellt werden (bei Terminierung: unendliches Stottern am Ende)
- Σ sei die Menge aller möglicher Zustandsfolgen eines Systems **STS** $\langle S, S_0, \text{Next} \rangle$
 $\Sigma = \{ \sigma: \sigma \text{ ist mögliche Zustandsfolge des STS} \}$

◆ Def: **Eigenschaft** Π

$\Pi \subset 2^{(S^\infty)}$, d.h. Π ist Menge von Zustandsfolgen

Was ist eine **Eigenschaft von Systemabläufen**?

B] Extensional

Menge der Abläufe, welche die Eigenschaft besitzen.

→ Eigenschaft ist Zustandsfolgenmenge

F7: Systemeigenschaft als Zustandsfolgenmenge

◆ Def: **Mögliche Zustandsfolge** σ eines **STS** $\langle S, S_0, \text{Next} \rangle$

- $\sigma \in S^\infty$, $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$, $s_0 \in S_0$, $\forall i: \langle s_i, s_{i+1} \rangle \in \text{Next} \vee s_i = s_{i+1}$
- Jeder Ablauf / jede Ausführung eines Systems kann als unendliche Folge von Zuständen dargestellt werden (bei Terminierung: unendliches Stottern am Ende)
- Σ sei die Menge aller möglicher Zustandsfolgen eines Systems **STS** $\langle S, S_0, \text{Next} \rangle$

◆ Def: **Eigenschaft** Π

$\Pi \subset 2^{(S^\infty)}$, d.h. Π ist Menge von Zustandsfolgen

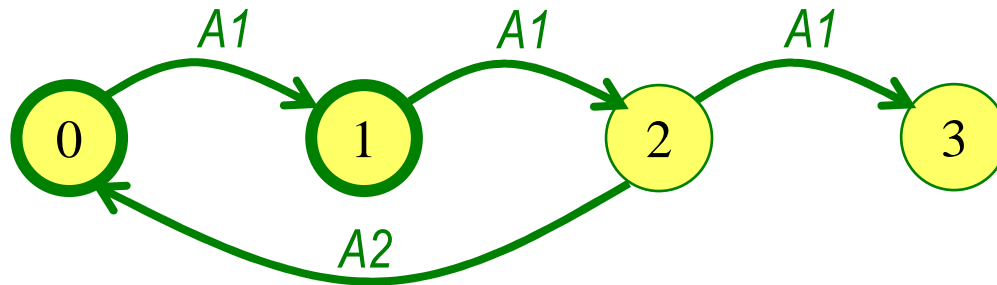
◆ Def: **STS** $\langle S, S_0, \text{Next} \rangle$ **besitzt Eigenschaft** Π

Jede mögliche Systemausführung ist in Π enthalten

$$\Sigma \subset \Pi$$

F7: Systemeigenschaft als Zustandsfolgenmenge

- ◆ Beispiel:
Eigenschaft „Fängt nicht mit 2 an“



```
var V : (0, 1, 2, 3) ;  
init V=0 ∨ V=1 ;  
act A1: V<3 ∧ V'=V+1 ;  
    A2: V=2 ∧ V'= 0 ;
```

$\Sigma = \{$
 $\langle 0, 1, 2, 3, 3, 3, \dots \rangle,$
 $\langle 0, 1, 2, 0, 1, 2, 3, 3, 3, \dots \rangle,$
 $\langle 0, 1, 2, 0, 1, 2, 0, 1, 2, 3, 3, 3, \dots \rangle,$
 $\langle 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 3, 3, 3, \dots \rangle,$
 $\dots \}$

$\supseteq \Pi = \{0,1,3\} \times \{0,1,2,3\}^\infty$

F7: Safety-Eigenschaft

- ◆ „Etwas Schlechtes“ passiert nicht.
 - *Der Aufzug öffnet die Etagentür nur, wenn die Kabine da ist.*
 - *Das Programm liefert nur die laut Spezifikation vorgesehenen Ausgabewerte.*
 - *Die Ausführungen der im gegenseitigen Ausschluss auszuführenden Operationen überlappen sich zeitlich nicht.*
 - *Die Aufträge werden in der FIFO-Reihenfolge bearbeitet.*
- ◆ Wenn eine Systemausführung σ eine Safety-Eigenschaft Π verletzt, wird das nach endlicher Zeit erkennbar.
 - Es gibt einen Index i in der Systemausführung
$$\sigma = \langle s_0, s_1, s_2, \dots, s_i, s_{i+1}, \dots \rangle,$$
so dass an s_i erkennbar wird, dass σ die Safety-Eigenschaft Π verletzt.
 - Wenn $\sigma = \langle s_0, s_1, s_2, \dots, s_i, s_{i+1}, \dots \rangle$ die Eigenschaft Π mit s_i verletzt, so verletzt jede Ausführung $\sigma' = \langle s_0, s_1, s_2, \dots, s_i, x_1, x_2, \dots \rangle$ die Eigenschaft Π ebenfalls.
 - Wenn eine Safety-Eigenschaft in einem Ablauf einmal verletzt ist, kann keine Fortsetzung des Ablaufs dies heilen.

F7: Safety-Eigenschaft

◆ Defs:

- $\sigma \in \mathbf{S}^\infty$, $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$, unendliche Zustandsfolge
- $\zeta \in \mathbf{S}^\infty$, $\zeta = \langle z_0, z_1, z_2, z_3, \dots \rangle$, unendliche Zustandsfolge
- $\theta \in \mathbf{S}^*$, $\theta = \langle t_0, t_1, t_2, t_3, \dots, t_n \rangle$, endliche Zustandsfolge
- $\Pi \subset 2^{(\mathbf{S}^\infty)}$, Eigenschaft
- $\text{pre}(\sigma, i) = \langle s_0, s_1, s_2, \dots, s_i \rangle$, $\text{pre}(s, i) \in \mathbf{S}^*$, Teilausführung von σ bis i
- $\sigma \in \Pi$, σ hat Eigenschaft Π
- $\theta \bullet \zeta = \langle t_0, t_1, t_2, t_3, \dots, t_n, z_0, z_1, z_2, z_3, \dots \rangle$, $\theta \bullet \zeta \in \mathbf{S}^\infty$, θ fortgesetzt mit ζ

◆ Def: Eigenschaft $\Pi \subset 2^{(\mathbf{S}^\infty)}$ ist Safety-Eigenschaft

$$\forall \sigma \in \mathbf{S}^\infty : \sigma \notin \Pi \Rightarrow (\exists i \forall \zeta \in \mathbf{S}^\infty : \text{pre}(\sigma, i) \bullet \zeta \notin \Pi)$$

Eine Safety-Eigenschaft liegt genau dann vor, wenn die Eigenschaft, wenn sie in einem Ablauf einmal verletzt ist, durch keine Fortsetzung wieder hergestellt werden kann.

F7: Liveness-Eigenschaft

- ◆ „Etwas Gutes“ lässt nicht unendlich lange auf sich warten.
 - *Wenn der Rufknopf an der Etagentür gedrückt wurde, wird die Aufzugkabine schließlich auch die Etage erreichen.*
 - *Das Programm terminiert.*
 - *Jeder Server-Prozess wird immer wieder in der Lage sein, einen neuen Auftrag zu bearbeiten.*
- ◆ Ob eine Systemausführung σ eine Liveness-Eigenschaft Π nicht erfüllt, kann nie anhand einer endlichen Teilausführung entschieden werden.
 - Jede System-Teilausführung $\text{pre}(\sigma, i)$, welche eine Liveness-Eigenschaft Π nicht erfüllt, kann so fortgesetzt werden, dass sie Π erfüllt.
 - Eine Liveness-Eigenschaft kann aus einer beliebigen Teilausführung heraus hergestellt werden.

F7: Liveness-Eigenschaft

◆ Defs:

- $\sigma \in \mathbf{S}^\infty$, $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$, unendliche Zustandsfolge
- $\zeta \in \mathbf{S}^\infty$, $\zeta = \langle z_0, z_1, z_2, z_3, \dots \rangle$, unendliche Zustandsfolge
- $\theta \in \mathbf{S}^*$, $\theta = \langle t_0, t_1, t_2, t_3, \dots, t_n \rangle$, endliche Zustandsfolge
- $\Pi \subset 2^{\mathbf{S}^\infty}$, Eigenschaft
- $\text{pre}(\sigma, i) = \langle s_0, s_1, s_2, \dots, s_i \rangle$, $\text{pre}(s, i) \in \mathbf{S}^*$, Teilausführung von σ bis i
- $\sigma \in \Pi$, σ hat Eigenschaft Π
- $\theta \bullet \zeta = \langle t_0, t_1, t_2, t_3, \dots, t_n, z_0, z_1, z_2, z_3, \dots \rangle$, $\theta \bullet \zeta \in \mathbf{S}^\infty$, θ fortgesetzt mit ζ

◆ Def: Eigenschaft $\Pi \subset 2^{\mathbf{S}^\infty}$ ist Liveness-Eigenschaft

$$\forall \theta \in \mathbf{S}^* : (\exists \zeta \in \mathbf{S}^\infty : \theta \bullet \zeta \in \Pi)$$

Eine Liveness-Eigenschaft liegt genau dann vor, wenn die Eigenschaft aus einer beliebigen Teilausführung heraus durch eine geeignete Fortsetzung hergestellt werden kann.

F7: Liveness-Eigenschaft

- ◆ Wichtige Liveness-Eigenschaften sind so genannte „Leads-to“-Eigenschaften der Form „ $P \leadsto Q$ “
 - *Immer wenn ein Zustand entsprechend P auftritt, soll ihm nach endlicher Zeit ein Zustand folgen, der Q erfüllt.*
 - Unter der Annahme, dass Q erfüllbar ist, ist eine Leadsto-Eigenschaft eine Liveness-Eigenschaft.
- ◆ Die Beispiele als Leads-to-Eigenschaften
 - *Wenn der Rufknopf an der Etagentür gedrückt wurde, wird die Aufzugkabine schließlich auch die Etage erreichen.*
Rufknopf_gedrückt \leadsto Kabine_in_Etage
 - *Das Programm terminiert.*
true \leadsto Terminierungszustand_liegt_vor
 - *Jeder Server-Prozess wird immer wieder in der Lage sein, einen neuen Auftrag zu bearbeiten.*
Server_tätig \leadsto Server_frei

F7: Gemischte Eigenschaften

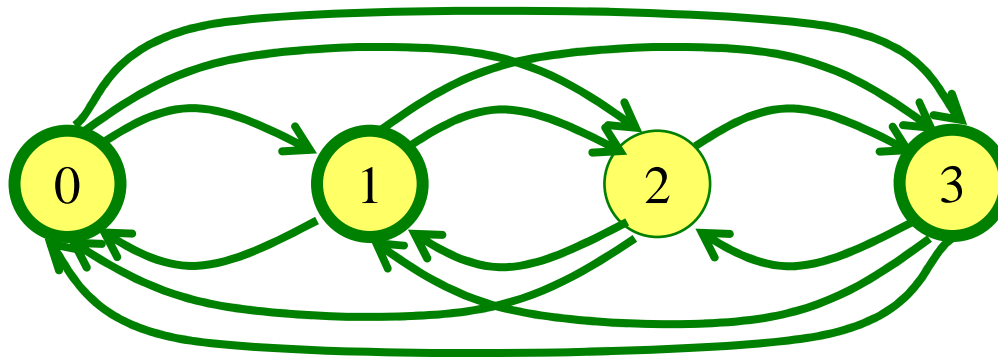
- ◆ Es gibt Eigenschaften, die im Schnitt einer Safety- und einer Liveness-Eigenschaft liegen.
 - Beispiel Safety-Eigenschaft **S**
Es kommt nicht vor, dass E2 gilt und vorher E1 irgendwann nicht zutraf.
 - Beispiel Liveness-Eigenschaft **L**
E2 gilt schließlich nach endlicher Zeit.
 - Eigenschaft **$S \cap L$**
E2 wird schließlich auftreten und vorher wird durchgehend E1 gelten.
(„E1 until E2“)
- ◆ Theorem: Jede Eigenschaft kann als Schnitt einer Safety- mit einer Liveness-Eigenschaft dargestellt werden.
 - *(hier ohne Beweis)*
- ◆ Es gibt Liveness- und Safety-Eigenschaften, welche sich widersprechen (deren Schnitt deshalb leer ist)
 - Beispiel Safety-Eigenschaft **T**
E2 darf nie gelten.
 - Beispiel Liveness-Eigenschaft **L**
E2 gilt schließlich nach endlicher Zeit.
 - Eigenschaft **$T \cap L = \{\}$**
Keine Zustandsfolge kann sowohl T als auch L erfüllen.

F7: Gemischte Eigenschaften

- ◆ Die für ein System gewünschte Eigenschaft ist in der Regel eine gemischte Eigenschaft, d.h. ein Schnitt (eine logische Konjunktion) aus Safety- und Liveness-Eigenschaft.
 - Problem (*Lösung folgt später*)
Häufig entsteht der Fehler, dass eine gewünschte Liveness-Eigenschaft zu ungenau definiert wird, so dass sie im Widerspruch zur Safety-Eigenschaft steht.
 - Beispiel
 - » **Safety**
Eine Etagentür eines Fahrstuhls darf sich nur öffnen, wenn sich die Kabine in der entsprechenden Etage befindet.
Wenn ein Motordefekt aufgetreten ist, darf die Kabinenfahrt nicht mehr gestartet werden.
 - » **Liveness**
Wenn der Rufknopf einer Etage gedrückt wurde, wird sich schließlich irgendwann die Etagentür öffnen.
 - » **Widerspruchssituation**
Der Motor ist defekt und der Rufknopf einer Etage ist gedrückt, in welcher die Kabine sich momentan nicht befindet.

F7: Safety per STS

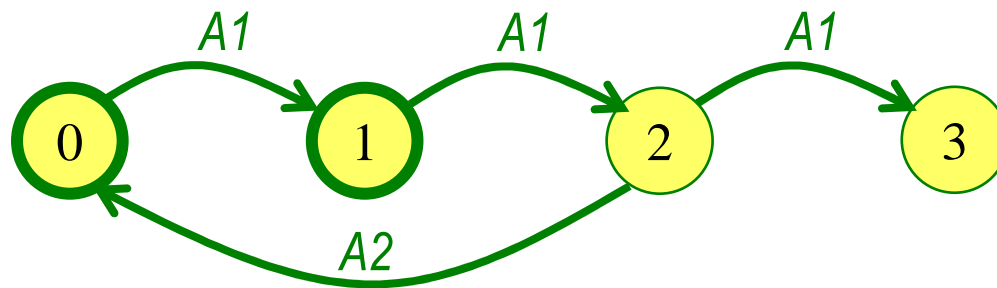
- ◆ Eine Safety-Eigenschaft kann durch ein **STS** $\langle S, S_0, \text{Next} \rangle$ definiert werden.
 - **Voraussetzung**
Es gibt keine impliziten Liveness-Annahmen zur STS-Ausführung, d.h. das **STS** darf in jedem Zustand auf Dauer verharren (d.h. unendlich viele Stotter Schritte ausführen)
 - Die Menge Σ aller möglichen Zustandsfolgen des **STS** ist eine Safety-Eigenschaft.
- ◆ Beispiel: Eigenschaft „*Fängt nicht mit 2 an*“
(im Zustandsraum $\{0, 1, 2, 3\}$)



```
var V : (0, 1, 2, 3) ;  
init V=0 ∨ V=1 ∨ V=3 ;  
act A: V ∈ {0,1,2,3} ∧  
      V' ∈ {0,1,2,3} ;
```

F7: Safety per Invarianten

- ◆ Zustandsinvariante I
 - I ist Bedingung über Zustandsvariablen
 - I gilt in jedem erreichbaren Zustand
- ◆ Die Menge der Zustandsfolgen, bei welchen I in jedem Zustand gilt, ist eine Safety-Eigenschaft



```
var V : (0, 1, 2, 3) ;  
init V=0 ∨ V=1 ;  
act A1: V<3 ∧ V'=V+1 ;  
    A2: V=2 ∧ V'=0 ;
```

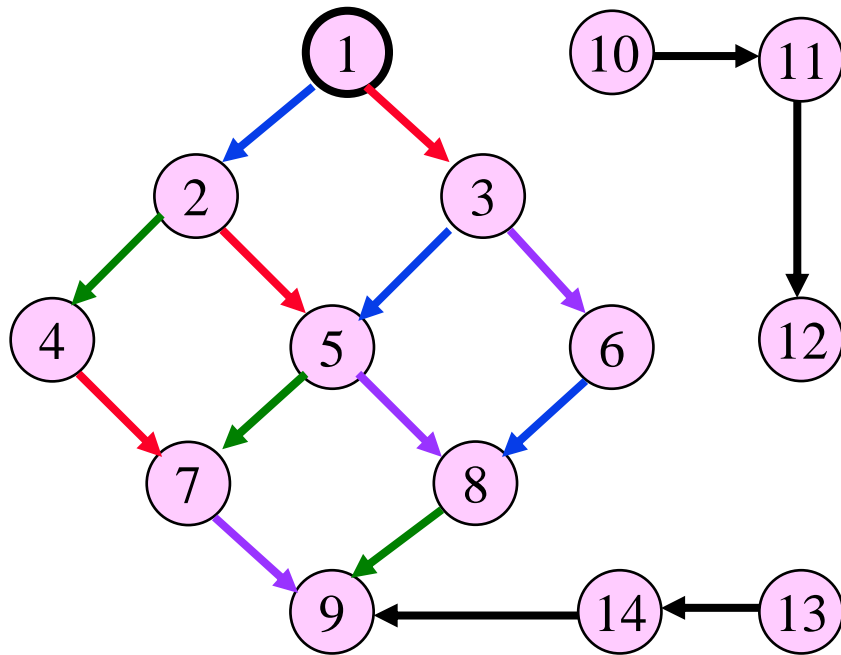
Beispiel:

I1 $V=0 \vee V=1 \vee V=2 \vee V=3$

F7: Safety per Invarianten

◆ Zustandsinvariante I

- I ist Bedingung über Zustandsvariablen
- I gilt in jedem erreichbaren Zustand



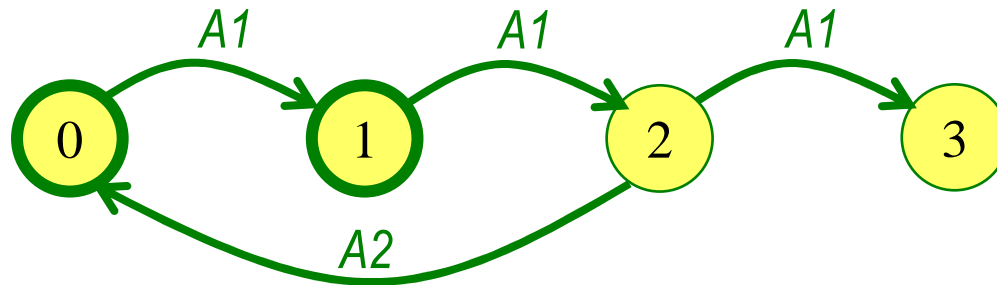
Beispiel:
I2 cs < 10

```
S = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
S0 = {1}
Next = {<1, 2>, <3, 5>, <6, 8>,
        <2, 4>, <5, 7>, <8, 9>,
        <1, 3>, <2, 5>, <4, 7>,
        <3, 6>, <5, 8>, <7, 9>,
        <10, 11>, <11, 12>,
        <13, 14>, <14, 9>}
```

```
var cs : (1..14);
init cs=1;
act .....
```

F7: Safety per Invarianten

- ◆ Safety-Eigenschaften eines STS können durch eine Zustands-Invariante definiert werden.
 - Unter Umständen sind vorher geeignete Hilfsvariablen einzuführen.



Beispiel

„Dem Zustand 3 geht immer Zustand 2 voraus“

$$I3 \quad V=3 \Rightarrow H=2$$

```
var V : (0, 1, 2, 3);
    H : (0, 1, 2, 3);
init (V=0  $\vee$  V=1)
     $\wedge$  H=V;
act A1: V<3  $\wedge$  V'=V+1
     $\wedge$  H'=V;
    A2: V=2  $\wedge$  V'=0
     $\wedge$  H'=V;
```

F7: Safety per Invarianten

- Bei Terminierung wurden alle Stationen informiert.
 - $nts = \{\} \Rightarrow \forall i: cs[i] = infd$
- Es werden höchstens $2e$ Nachrichten ausgetauscht.
 - AnzSend** $< 2e$
(Hilfsvariable AnzSend geeignet eingeführt)
- Wenn eine Station im Zustand *infd* ist, verlässt sie ihn nie wieder.
 - H=0**
(Hilfsvariable *H* so eingeführt, dass sie nie von 1 auf 0 kippt und immer dann auf 1 kippt, wenn eine Station von *infd* auf *idle* wechselt)

Variablen

cs: array [1..n] of (idle, infd) ! Stationen

nts: bag of $\langle \text{from}, \text{to}, \text{msg} \rangle$! Transportsystem

Init

cs = $\langle \text{idle}, \text{idle}, \dots, \text{idle} \rangle \wedge$

nts = $\{\langle 0, \text{Initiator}, \text{Text} \rangle\}$

Aktionen

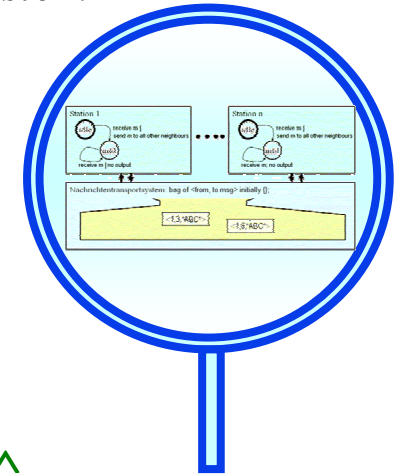
Forward

$\exists i, m: cs[i] = \text{idle} \wedge m \in nts \wedge m.\text{to} = i \wedge$
 $cs[i]' = \text{infd} \wedge \forall j \neq i: cs[j]' = cs[j] \wedge$
 $nts' = nts \cup$

$\{ \langle i, k, m.\text{msg} \rangle : \text{istNachbar}(k, i) \wedge k \neq m.\text{from} \} \setminus \{m\}$

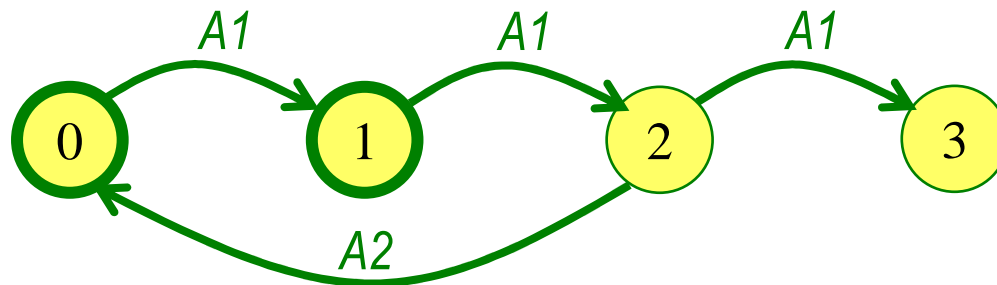
Skip

$\exists i, m: cs[i] = \text{infd} \wedge m \in nts \wedge m.\text{to} = i \wedge$
 $\forall j: cs[j]' = cs[j] \wedge nts' = nts \setminus \{m\}$



F7: Safety per Invarianten

- ◆ Die **kanonische Historienhilfsvariable** speichert die gesamte Historie eines Momentanzustands.
 - Seien V_1, V_2, \dots, V_n die Variablen des **STS**
 - **HH** : list of $\langle v_1, v_2, \dots, v_n \rangle$! Hilfsvariable
 - Initial: **HH=empty**
 - bei jeder Aktion: **HH'=append(HH, $\langle V_1, V_2, \dots, V_n \rangle$)**



Die in HH enthaltene Liste gibt die gesamte Vorgeschichte des aktuellen Zustands wieder.

```
var V : (0, 1, 2, 3) ;  
    HH : list of (0, 1, 2, 3) ;  
init (V=0  $\vee$  V=1 )  
     $\wedge$  HH=empty ;  
act A1: V<3  $\wedge$  V'=V+1  
     $\wedge$  HH'=append(HH,V);  
    A2: V=2  $\wedge$  V'= 0  
     $\wedge$  HH'=append(HH,V);
```

F7: Aktionen-Fairness

◆ Gegeben **STS** $\langle S, S_0, \text{Next} \rangle$

mit **Next** = $a_1 \cup a_2 \cup \dots \cup a_n$, gegliedert in Subrelationen / Aktionen
sowie eine (unendliche) Ausführung $\sigma \in S^\infty$, $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$, .

- Def: **Aktion** a_i tritt in Ausführung σ auf $\Leftrightarrow \exists j: \langle s_j, s_{j+1} \rangle \in a_i$
- Def: Aktion a_i ist schaltbereit in einem Zustand $s \in S$:
Enabled(a_i, s) $\Leftrightarrow \exists x \in S: \langle s, x \rangle \in a_i$

◆ Fairness bzgl. des Auftretens einer Aktion in einer Ausführung

Frage:

- Aktion a_i tritt in σ nie auf.
- Aktion a_i tritt in σ nur endlich oft auf.
- Aktion a_i tritt in σ immer wieder auf (also unendlich oft).

Unter welchen Umständen wird die Aktion a_i vom Scheduler / vom Zufall fair behandelt.

Wie fair wird sie behandelt:

- **Stark fair**
- **Schwach fair**

F7: Aktionen-Fairness

Beispiele

◆ Aktion A

Student_Müller_bekommt_Essen_in_Mensa

- *Wir erwarten, dass Student Müller in der Mensa bei der Essensausgabe fair behandelt wird, und wenn er hingeht, auch Essen bekommt.*
- *Hier soll ein Verfahren (Schlangenbildung und Essensausgabe-Algorithmus) die Fairness garantieren.*
- *Wir setzen beim hungrigen Müller auch eine gewisse Mitwirkung voraus. Er soll sich in die Schlange stellen und dort bleiben, bis er Essen bekommt.
Wenn er vorher die Schlange verlässt, oder gar immer wieder mal sich anstellt und die Schlange wieder verlässt, soll er sich damit abfinden, u.U. auch kein Essen zu bekommen.*

◆ Aktion B

Student_Müllers_Notebook_sendet_WLAN_Frame_erfolgreich

- *Wir erwarten, dass Student Müllers Notebook immer wieder mal einen WLAN-Frame ungestört übertragen kann.*
- *Hier soll der Zufall dazu führen, dass Müller nicht über Gebühr benachteiligt wird.*
- *Müllers Notebook ist nicht stur. Wenn ein Frame nicht gesendet werden konnte, versucht es erst mal eine Weile nicht, wieder einen Frame zu senden.
Trotzdem sollen immer wieder Frames erfolgreich übertragen werden.*

◆ Hier gibt es Unterschiede

A] Schwache Fairness (Weak Fairness WF)

B] Starke Fairness (Strong Fairness SF)

F7: Aktionen-Fairness

Unterschiede

◆ Aktion A

Student_Müller_bekommt_Essen_in_Mensa

- *Wir setzen beim hungrigen Müller auch eine gewisse Mitwirkung voraus. Er soll sich in die Schlange stellen und dort bleiben, bis er Essen bekommt.*
Wenn er vorher die Schlange verlässt, oder gar immer wieder mal sich anstellt und die Schlange wieder verlässt, soll er sich damit abfinden, u.U. auch kein Essen zu bekommen.
- *Ein Ablauf, in welchem Müller kein Essen bekommt und immer wieder ein Zustand „Müller nicht in Schlange“ auftritt, wird nicht als „unfair zu Müller“ bewertet.*

◆ Aktion B

Student_Müllers_Notebook_sendet_WLAN_Frame_erfolgreich

- *Müllers Notebook ist nicht stur. Wenn ein Frame nicht gesendet werden konnte, versucht es erst mal eine Weile nicht, wieder einen Frame zu senden.*
Trotzdem sollen immer wieder Frames erfolgreich übertragen werden.
- *Ein Ablauf, in welchem Müllers Frames nie erfolgreich übertragen werden, obwohl immer wieder ein Zustand „Müllers Notebook sendet“ auftritt, wird als „unfair zu Müller“ bewertet, auch wenn das Notebook nicht pausenlos sendet.*

◆ Hier gibt es

A] Schwache Fairness (**Weak Fairness WF**)

B] Starke Fairness (**Strong Fairness SF**)

F7: Aktionen-Fairness

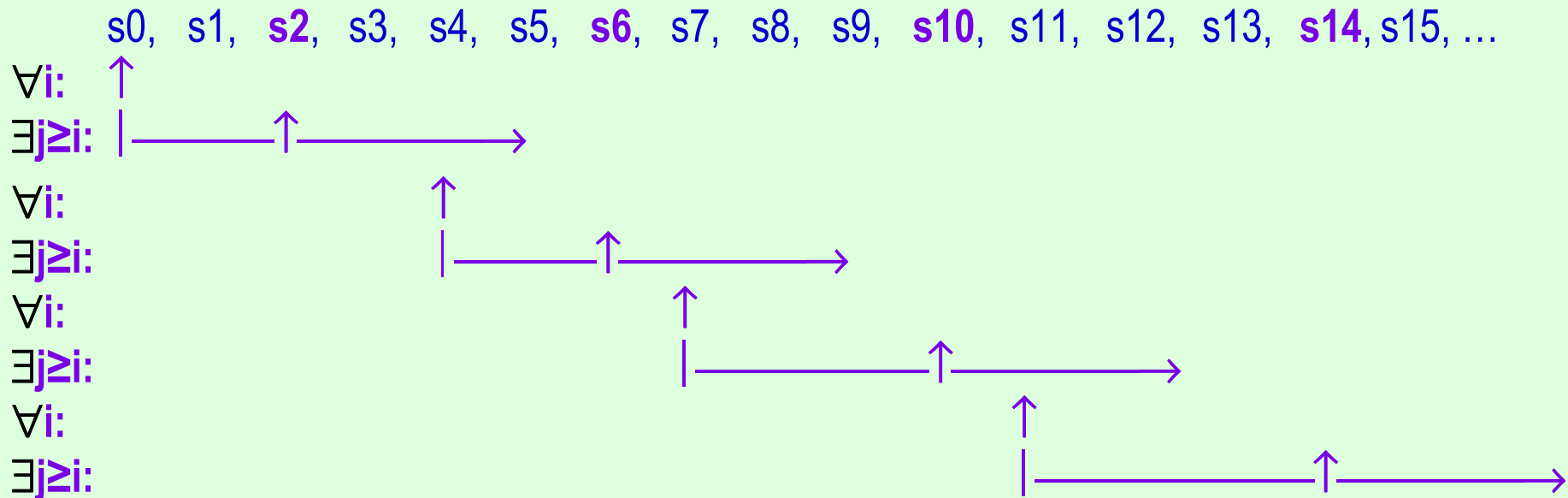
◆ Def: **WF(a)** in $\sigma \in \mathbf{S}^\infty$, $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$

$\forall i \exists j \geq i: \langle s_j, s_{j+1} \rangle \in a$! immer wieder a

\vee ! oder

$\forall i \exists j \geq i: \neg \text{Enabled}(a, s_j)$! immer wieder ist a disabled

„Immer wieder“ $\forall i \exists j \geq i$

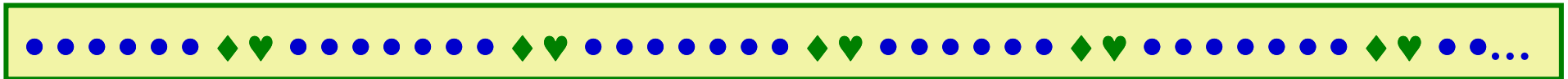


F7: Aktionen-Fairness

◆ Def: **WF(a)** in $\sigma \in S^\infty$, $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$

$$\forall i \exists j \geq i: \langle s_j, s_{j+1} \rangle \in a$$

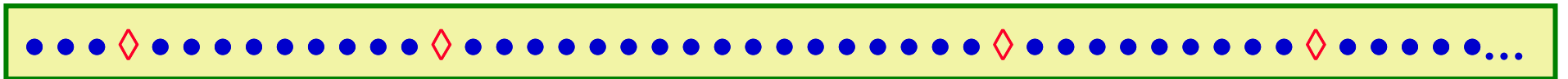
! immer wieder a



$$\vee \forall i \exists j \geq i: \neg \text{Enabled}(a, s_j)$$

! oder

! immer wieder ist a disabled



Legende

- Zustand ohne Besonderheit
- ◆♥ Zustandspaar, so dass $\text{Enabled}(a, \blacklozenge)$ und $\langle \blacklozenge, \heartsuit \rangle \in a$
- ◇ Zustand, so dass $\neg \text{Enabled}(a, \blacklozenge)$

F7: Aktionen-Fairness

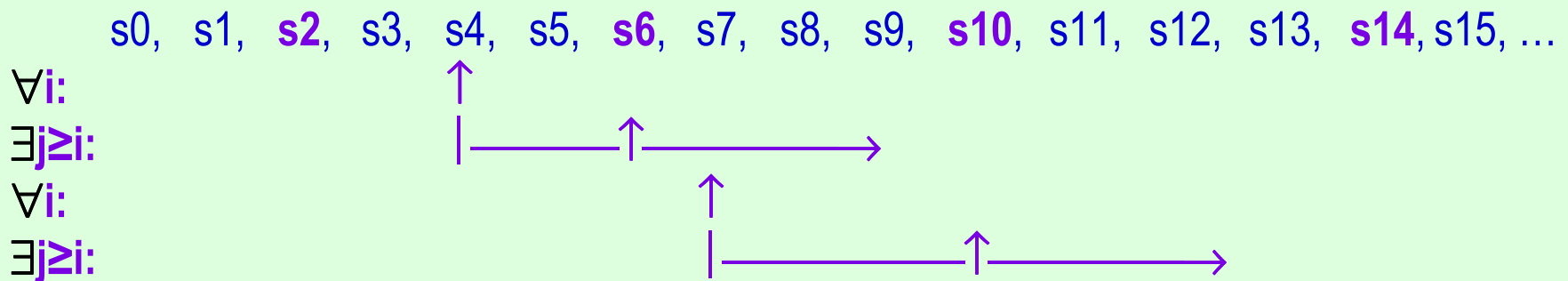
◆ Def: **SF(a)** in $\sigma \in \mathbf{S}^\infty$, $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$

$\forall i \exists j \geq i: \langle s_j, s_{j+1} \rangle \in a$! immer wieder a

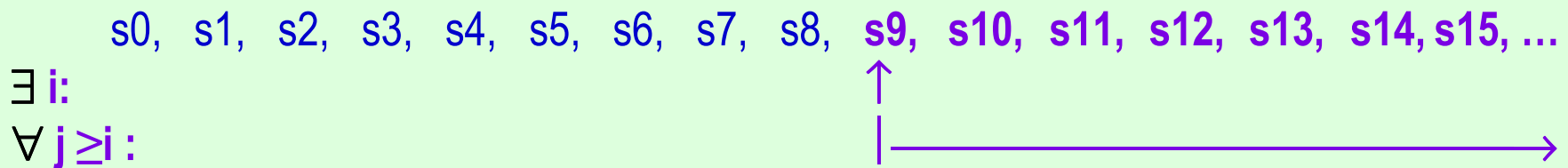
\vee ! oder

$\exists i \forall j \geq i: \neg \text{Enabled}(a, s_j)$! schließlich immer ist a disabled

„Immer wieder“ $\forall i \exists j \geq i$



„Schließlich immer“ $\exists i \forall j \geq i$

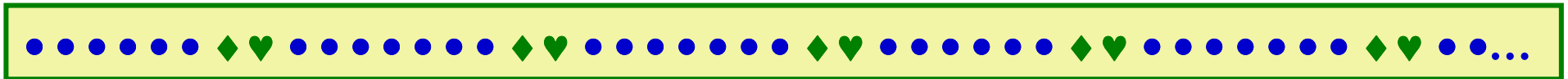


F7: Aktionen-Fairness

◆ Def: **SF(a)** in $\sigma \in \mathbf{S}^\infty$, $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$

$$\forall i \exists j \geq i: \langle s_j, s_{j+1} \rangle \in a$$

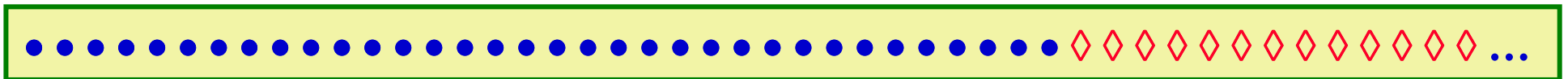
! immer wieder a



$$\vee \exists i \forall j \geq i: \neg \text{Enabled}(a, s_j)$$

! oder

! schließlich immer ist a disabled



Legende

- Zustand ohne Besonderheit
- ◆♥ Zustandspaar, so dass $\text{Enabled}(a, \blacklozenge)$ und $\langle \blacklozenge, \heartsuit \rangle \in a$
- ◇ Zustand, so dass $\neg \text{Enabled}(a, \blacklozenge)$

F7: Liveness per Aktionen-Fairness-Annahmen

- ◆ Gegeben **STS** $\langle S, S_0, \text{Next} \rangle$
mit **Next** = $a_1 \cup a_2 \cup \dots \cup a_n$, gegliedert in Subrelationen / Aktionen a_i
 - Es gibt keine impliziten Liveness-Annahmen zur STS-Ausführung, d.h. das **STS** darf in jedem Zustand auf Dauer verharren (d.h. unendlich viele Stottersschritte ausführen)
 - Die Menge Σ aller möglichen Zustandsfolgen des **STS** ist eine Safety-Eigenschaft
- ◆ Liveness-Anforderungen werden durch zusätzliche explizite Fairness-Annahmen definiert:
 - WF für bestimmte Aktionen
 - SF für bestimmte Aktionen
 - keine Fairness-Annahmen für übrige Aktionen
 - das STS darf nur noch dann in Zuständen auf Dauer verharren, wenn dadurch die gegebenen Fairness-Annahmen nicht verletzt werden.
- ◆ Die durch Aktionen-Fairness-Annahmen eines **STS** definierten Liveness-Eigenschaften stehen nicht im Widerspruch zu der durch das **STS** definierten Safety-Eigenschaft.
 - Sie erzwingen höchstens dann eine Transition, wenn die entsprechende Aktion **Enabled** ist.
 - Sie erzwingen nur Transitionen, die laut **STS**-Safety erlaubt sind.

F7: Liveness per Aktionen-Fairness-Annahmen

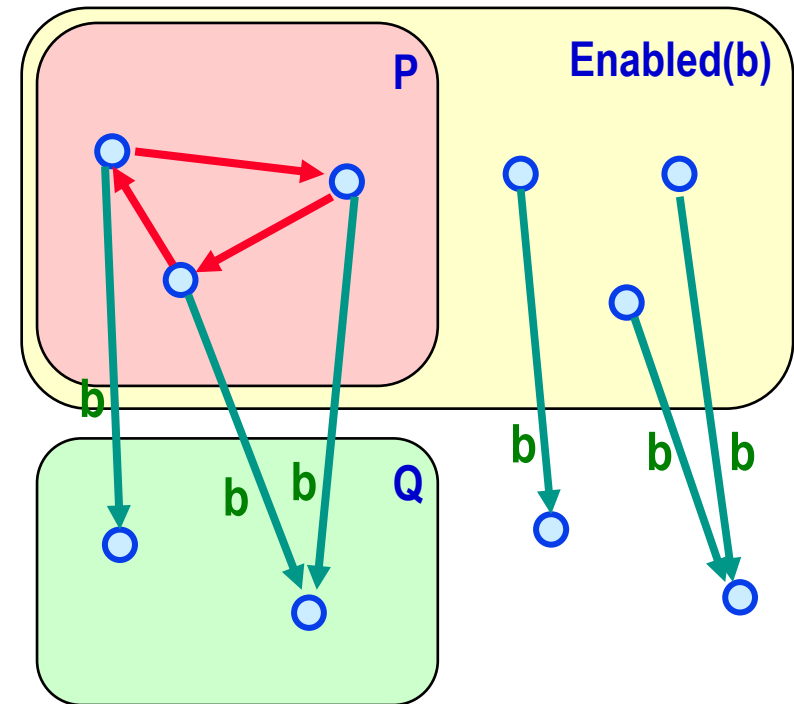
Liveness: $P \rightsquigarrow Q$

- ◆ Gegeben **STS** $\langle S, S_0, \text{Next} \rangle$
mit $\text{Next} = a_1 \cup a_2 \cup \dots \cup a_n$, gegliedert in Subrelationen / Aktionen a_i
- ◆ Einfacher Fall
Es gibt Aktion $b \subseteq \text{Next}$, mit folgenden Eigenschaften
 - **P impliziert** $\text{Enabled}(b)$
 - wenn **P** zutrifft, bleibt es gültig, bis **b** schaltet
 - wenn **P** gilt und **b** schaltet, gilt **Q** im Folgezustand

In diesem Zusammenhang bringt

WF(b)

die Eigenschaft $P \rightsquigarrow Q$ zum Ausdruck

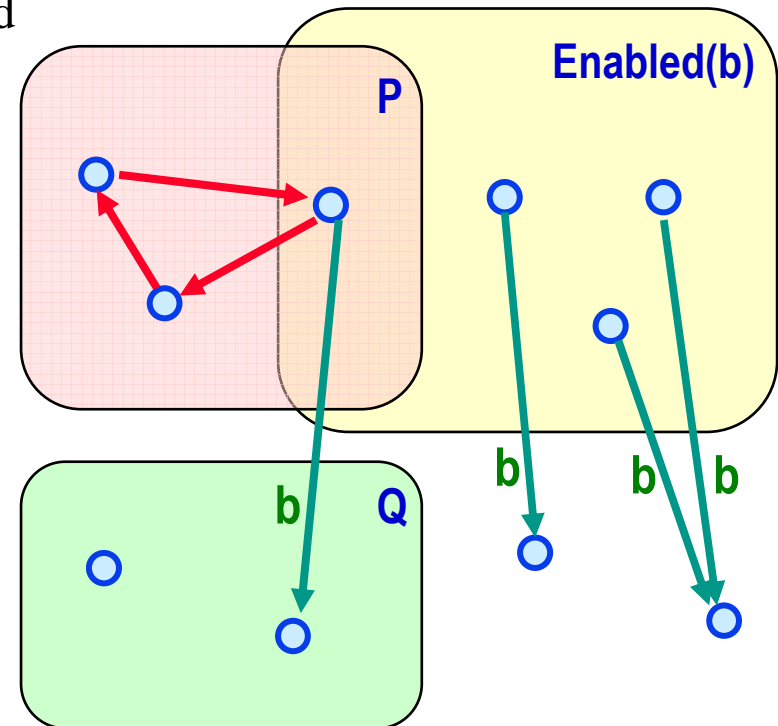


F7: Liveness per Aktionen-Fairness-Annahmen

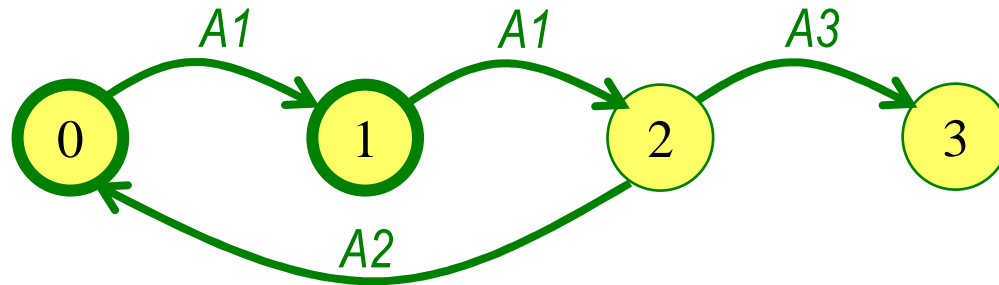
Liveness: $P \rightsquigarrow Q$

- ◆ Gegeben **STS** $\langle S, S_0, \text{Next} \rangle$
mit $\text{Next} = a_1 \cup a_2 \cup \dots \cup a_n$, gegliedert in Subrelationen / Aktionen a_i
- ◆ Etwas weniger einfacher Fall
Es gibt Aktion $b \subseteq \text{Next}$, mit folgenden Eigenschaften
 - P **leadsto** **Enabled(b)**
 - wenn P zutrifft, bleibt es gültig, bis b schaltet
 - wenn P gilt und b schaltet, gilt Q im Folgezustand

In diesem Zusammenhang bringt
SF(b)
die Eigenschaft $P \rightsquigarrow Q$ zum Ausdruck



F7: Liveness per Aktionen-Fairness-Annahmen



```

var V : (0, 1, 2, 3) ;
init V=0 ∨ V=1 ;
act A1: V<3 ∧ V'=V+1 ;
    A2: V=2 ∧ V'= 0 ;
    A3: V=2 ∧ V'= 3 ;
  
```

Beispiele

$V=0 \rightsquigarrow V=1$:

$WF(A1)$

$V=2 \rightsquigarrow V=3$:

$WF(A1) \wedge WF(A2) \wedge SF(A3)$

Zyklus 0, 1, 2 wird lebendig durchlaufen:
Immer wieder gilt $Enabled(A3)$

A3 muss schließlich einmal schalten

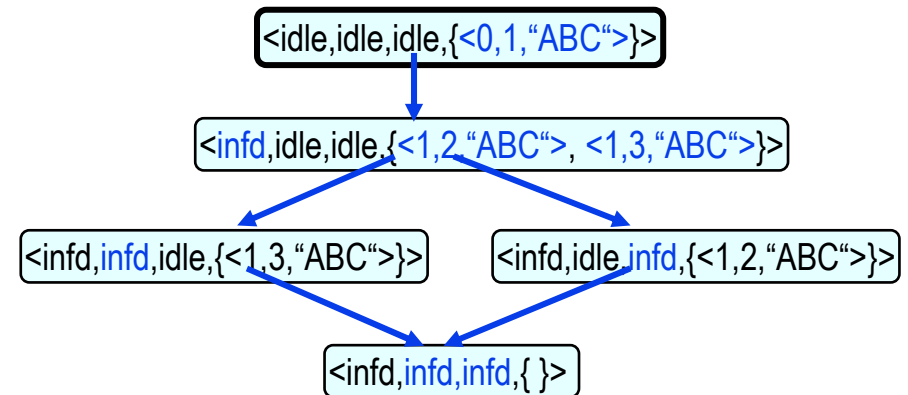
Anmerkung

Die Aktionen müssen nicht disjunkt sein.

F8: Erreichbarkeitsanalyse

- ◆ geg. Systemdefinition, z.B. als
 - STS
 - Petri Netz
 - LTS
 - Erweiterter Mealy-Automat
 - System aus gekoppelten ...

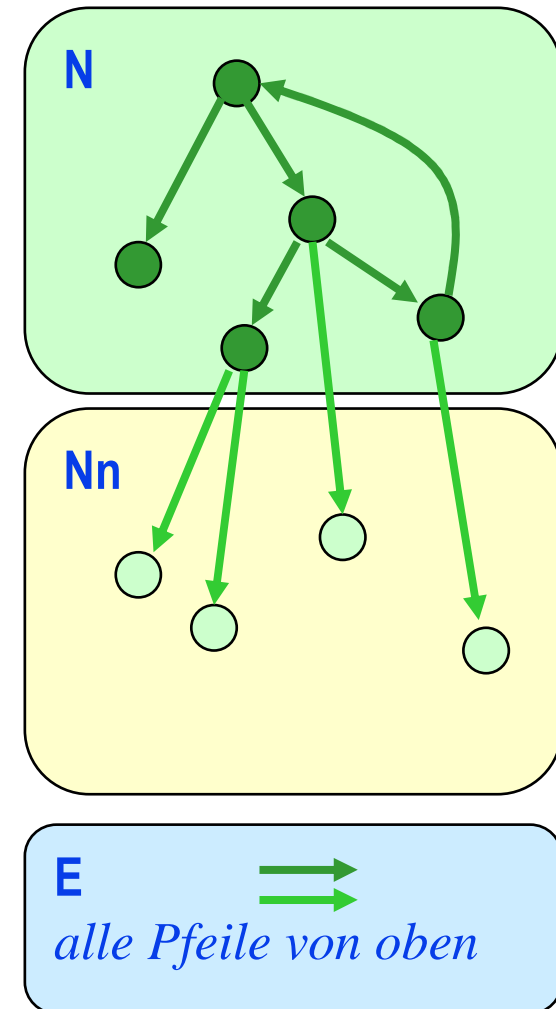
- ◆ ges. Erreichbarkeitsgraph
 - Graph
 - » Knoten: erreichbare Systemzustände
 - » Kanten: System-Transitionen
 - Analyse des Graphen
 - » Safety: erreichbare Systemzustände
 - » Liveness: Zusammenhangskomponenten, Pfade, Zyklen



- *Algorithmus*
 - *Varianten Tiefe / Breite / Random*
- *STS*
- *Petri Netz Markierungsgraph*
- *Gekoppelte LTS*
- *Bezug zu CCS*
- *Gekoppelte erweiterte Mealy-Automaten*
- *Werkzeug: SPIN*

F8: Erreichbarkeitsanalyse – Algorithmus

- ◆ Variablen
 - **N** : set of States *! Bearbeitete Knoten des Graphen*
 - **Nn** : set of States *! Anhängige Knoten des Graphen*
 - **E** : set of State-Pairs *! Kantenmenge des Graphen*
- ◆ Initialzustand
 - **N** = {}
 - **Nn** = {x : x ist Startzustand}
 - **E** = {}
- ◆ Anweisung
 - solange** **Nn** ≠ {}
 - wähle $s \in Nn$;
 - **N** := $N \cup \{s\}$; **Nn** := $Nn \setminus \{s\}$;
 - für jedes s' , das von s aus durch eine Transition als Folgezustand erreicht werden kann:
 - **E** := $E \cup \{ \langle s, s' \rangle \}$;
 - Falls $s' \notin N \cup Nn$
 - **Nn** := $Nn \cup \{s'\}$;
- ◆ **Problem: Zustandsexplosion (State Explosion)**
 - Anzahl der Zustände in Zustandsraum wächst
 - » exponentiell mit der Anzahl der Zustandsvariablen
 - » bei Systemen aus nebenläufigen Komponenten: exponentiell mit der Anzahl der Komponenten



F8: Erreichbarkeitsanalyse – Algorithmus-Varianten

◆ Variablen

- **N** : set of States ! Bearbeitete Knoten des Graphen
- **Nn** : set of States ! Anhängige Knoten des Graphen
- **E** : set of State-Pairs ! Kantenmenge des Graphen

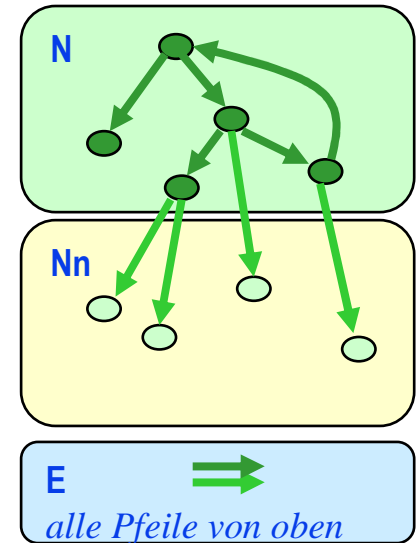
◆ Initialzustand

- **N** = {}
- **Nn** = {x : x ist Startzustand}
- **E** = {}

◆ Anweisung

solange $Nn \neq \{\}$

- wähle $s \in Nn$;
- $N := N \cup \{s\}$; $Nn := Nn \setminus \{s\}$;
- für jedes s' , das von s aus durch eine Transition als Folgezustand erreicht werden kann:
 - $E := E \cup \{<s, s'>\}$;
 - Falls $s' \notin N \cup Nn$
 - $Nn := Nn \cup \{s'\}$;



erschöpfend:

- **Breitendurchlauf**
 - **Tiefendurchlauf**
- nicht erschöpfend:
- **Random**

Suche in Menge:

- **exakt**
- **Hash ohne Kollisionsauflösung**

F8: Erreichbarkeitsanalyse – STS

Variablen

cs: array [1..n] of (idle, infd) ! Stationen
nts: bag of < from, to, msg > ! Transportsys.

Init

cs = < idle, idle, ..., idle > \wedge
nts = {<0, Initiator, Text>}

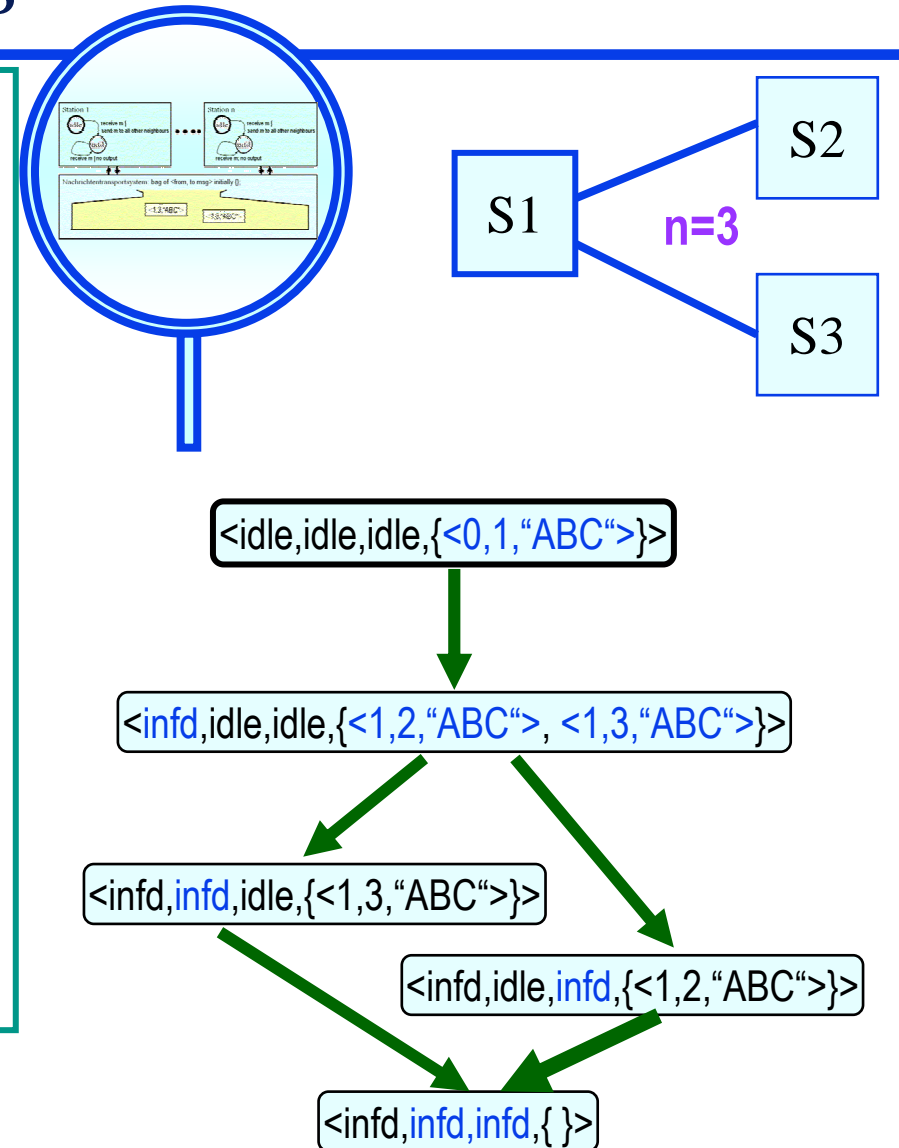
Aktionen

Forward

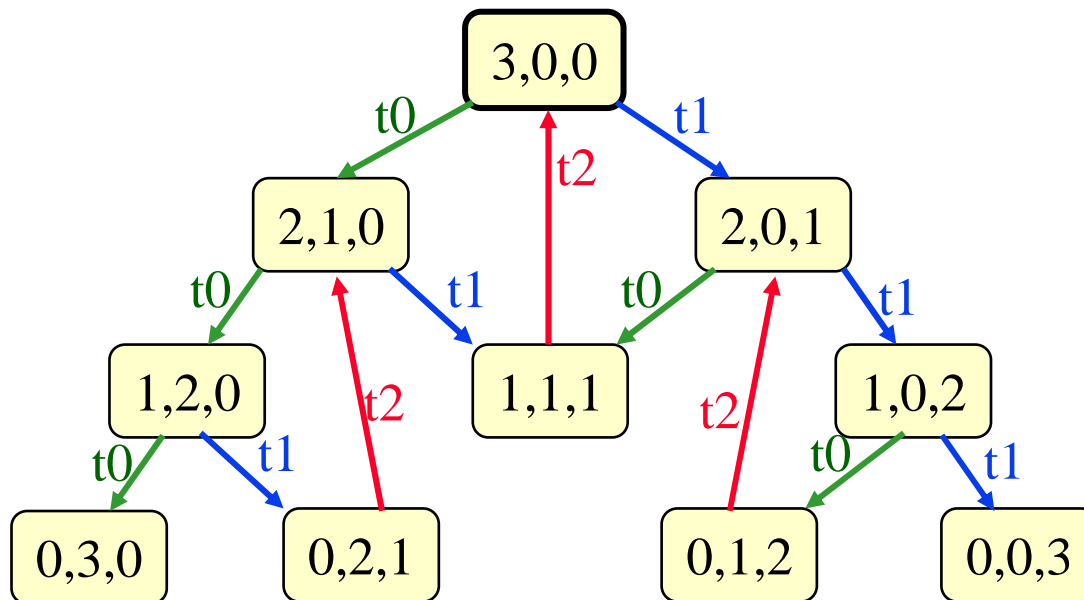
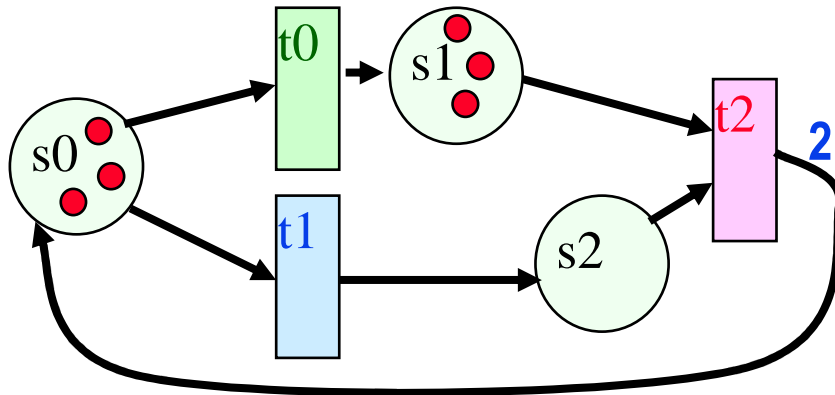
$\exists i, m: cs[i]=idle \wedge m \in nts \wedge m.to=i \wedge$
 $cs[i]'=infd \wedge \forall j \neq i: cs[j]'=cs[j] \wedge$
 $nts' = nts \cup$
 $\{ \langle i, k, m.msg \rangle : istNachbar(k,i) \wedge$
 $k \neq m.from \} \setminus \{m\}$

Skip

$\exists i, m: cs[i]=infd \wedge m \in nts \wedge m.to=i \wedge$
 $\forall j: cs[j]'=cs[j] \wedge nts'=nts \setminus \{m\}$



F8: Erreichbarkeitsanalyse – Markierungsgraph zum Petri Netz

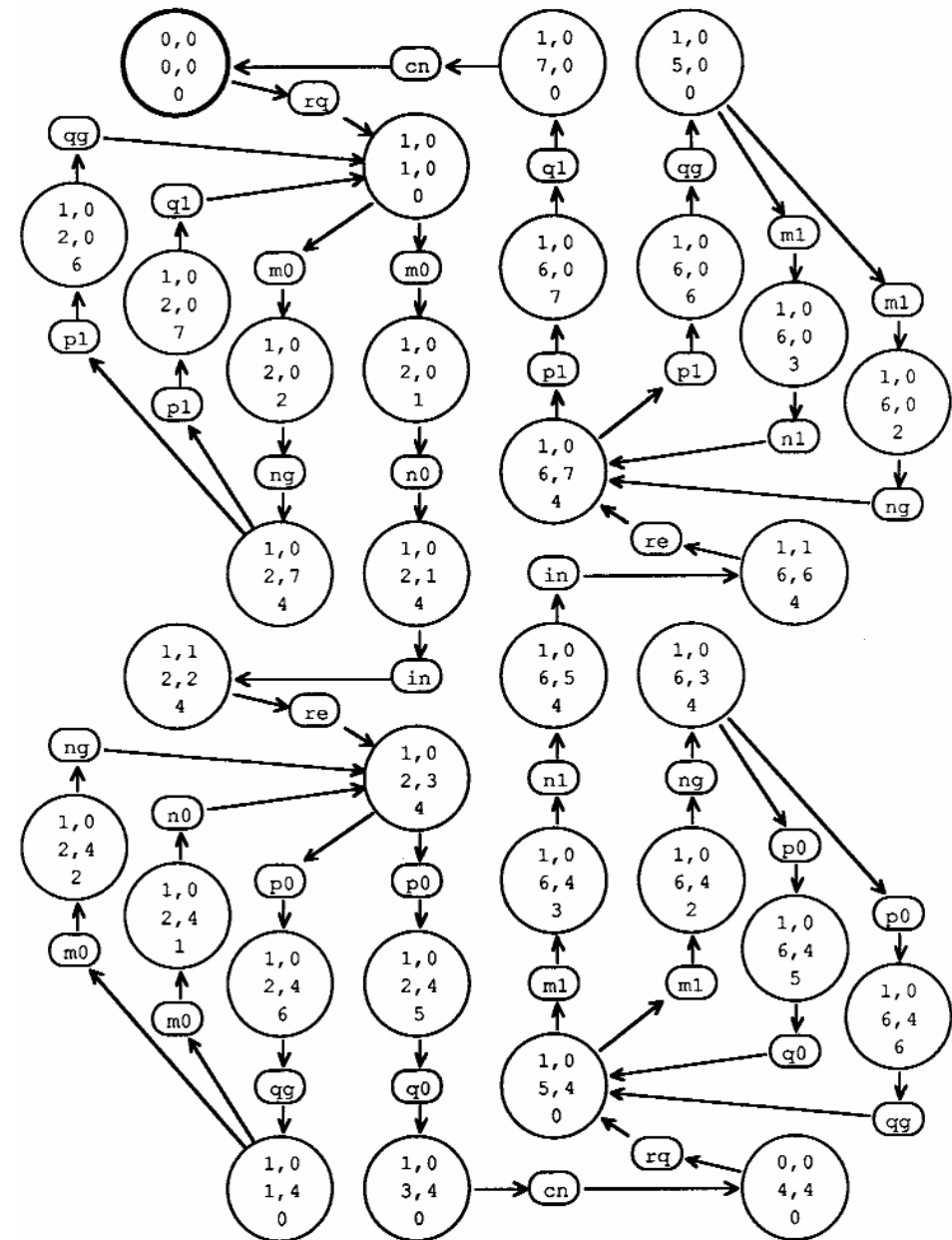
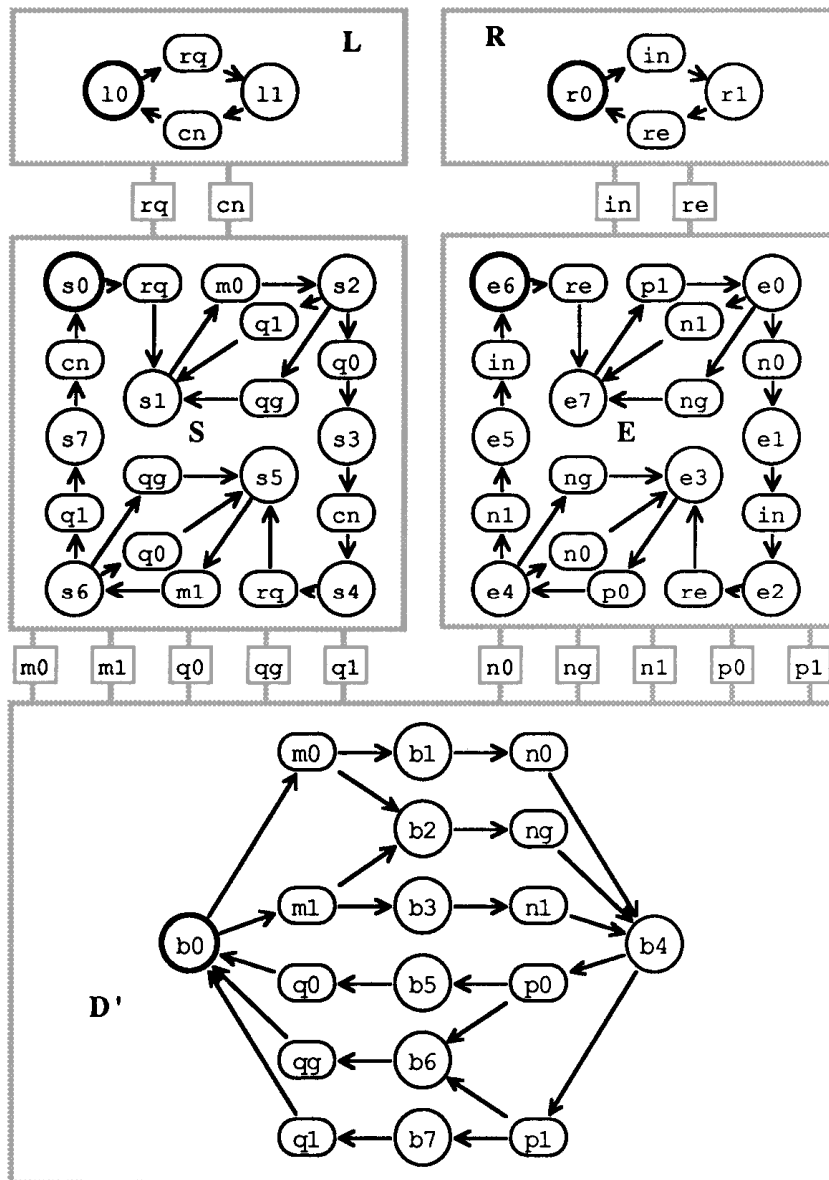


- ◆ Variablen
 - N : set of States !
Bearbeitete Knoten des Graphen
 - N_n : set of States !
Anhängige Knoten des Graphen
 - E : set of State-Pairs !
Kantenmenge des Graphen
- ◆ Initialzustand
 - $N = \{\}$
 - $N_n = \{x : x \text{ ist Startzustand}\}$
 - $E = \{\}$
- ◆ Anweisung

solange $N_n \neq \{\}$

 - wähle $s \in N_n$;
 - $N := N \cup \{s\}$; $N_n := N_n \setminus \{s\}$;
 - für jedes s' , das von s aus durch eine Transition als Folgezustand erreicht werden kann:
 - $E := E \cup \{<s, s'>\}$;
 - Falls $s' \notin N \cup N_n$
 - $N_n := N_n \cup \{s'\}$;

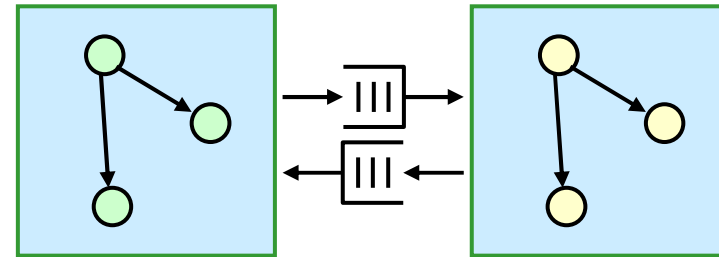
F8: Erreichbarkeitsanalyse – Gekoppelte LTS



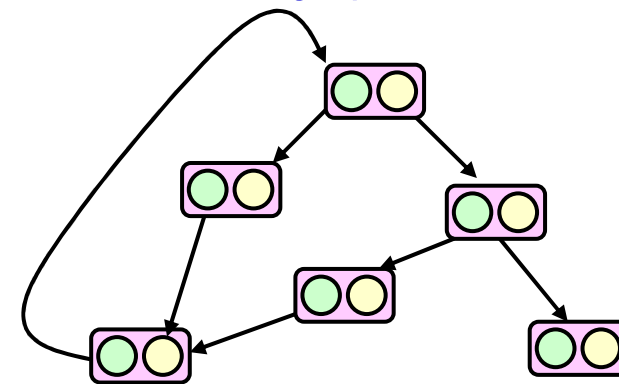
F8: Erreichbarkeitsanalyse – Kriterien

- ◆ Safety allgemein
 - Stop-Zustände
 - Tote Instanzenzustände
 - Tote Instanzentransitionen
 - Tote Interaktionen / Zeichen
- ◆ Safety speziell
 - Verletzt erreichbarer Zustand
gewünschte Invariante
- ◆ Liveness allgemein
 - Zyklen
 - Zusammenhangskomponenten
- ◆ Liveness speziell
 - unproduktive Zyklen
 - Zyklen über Erfolgzuständen

System gekoppelter Instanzen



Erreichbarkeitsgraph des Gesamtsystems



Safety speziell, Liveness speziell:

Vorstufe des Model Checkings

F8: Erreichbarkeitsanalyse – Tool SPIN

SPIN (Simple Promela Interpreter)

- Modelchecker von G. Holzmann
- Jährliche SPIN Workshops seit 1995
 - Teil der ETAPS (European Joint Conferences on Theory and Practice of Software)
- ACM Software System Award (2001)
- In Forschung & Industrie eingesetzt
 - NASA/JPL Laboratory for Reliable Software
 - Bell Labs/Lucent Technologies
 - Microsoft Research
 - ...
- Frei verfügbar für Unix, Windows etc
 - Web <http://spinroot.com>

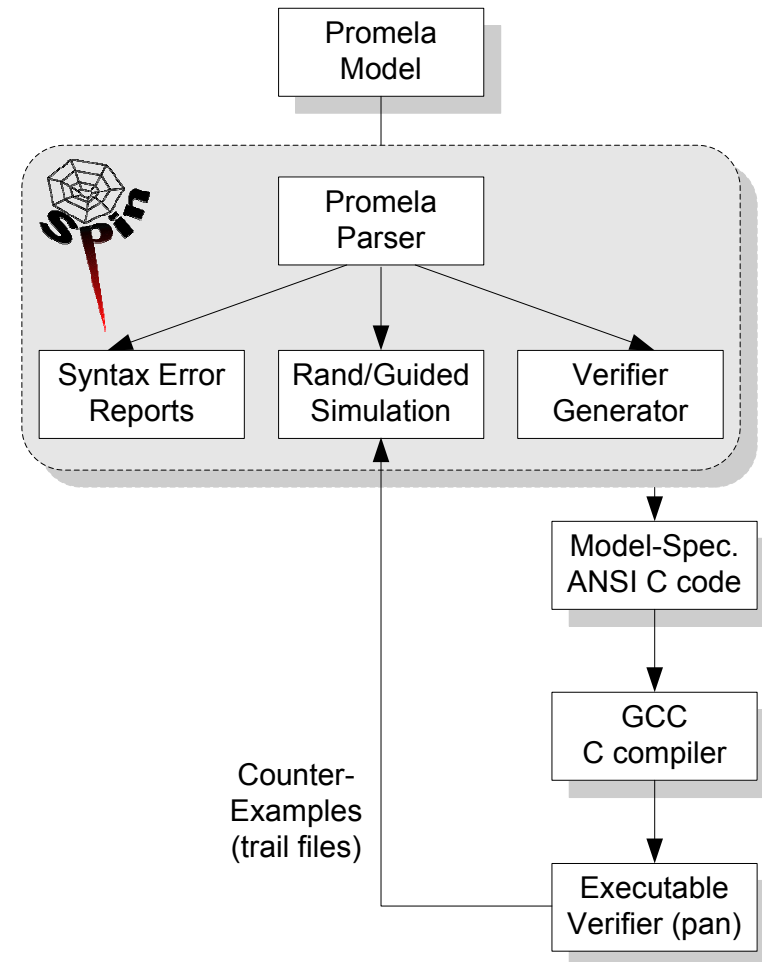


G. Holzmann:
The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, 2003.
M. Ben-Ari:
Principles of Spin, Springer Verlag, 2008.

F8: Erreichbarkeitsanalyse – Tool SPIN

Modellchecker Spin

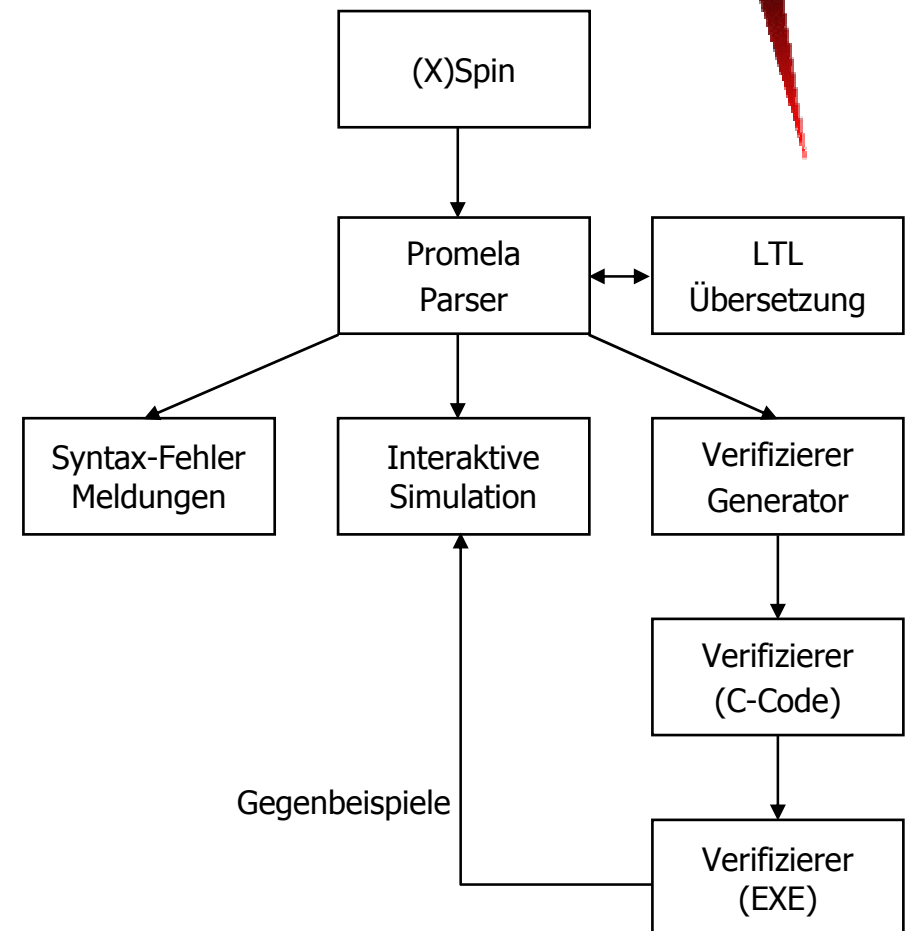
- ◆ Spin: Finite-State Model Checker
 - Kontinuierlich weiterentwickelt seit 1985 von Gerard Holzmann (Bell Labs / NASA JPL)
 - unterstützt verschiedene Simulations- und Verifikationsmodi
- ◆ Promela: Modellierungssprache von Spin
 - Reduziertes C plus Prozesse und Channels
 - Keine Aktionen in Prozessen, keine Prozesstypen, keine Komposition
- ◆ Besonderheit: Executable Verifier
 - Spin erzeugt ANSI C Code eines modellspezifischen Verifizierers
 - Übersetzung mit GCC ergibt ausführbaren Verifizierer (benötigt das Modell nicht mehr)



F8: SPIN Funktionsweise, Vorgehensweise



- Basis:
 - Erreichbarkeitsanalyse
 - Modelchecking
- Diverse Optimierungen (Partial Order Reduction) und Verifikationsmodi (Exhaustive, Supertrace, Bitstate...)
- Angabe von Invarianten und Lebendigkeitsanforderungen durch temporallogische Formeln (LTL)
- Grafisches Front-End XSpin



F8: Promela

- Basis-Typen: `bit`, `byte`, `bool`, `short`, `unsigned`
- Kanäle: `chan name = [Puffergröße] of {TYP, ...}`
- Aufzählungstyp: `mtype = {elem1, elem2, ...}` (max. 256 Elemente)
- Array: `TYP name[Größe]`
- Zusammengesetzter Typ:

```
typedef name {  
    TYP feld1;  
    TYP feld2;  
}
```
- Kontrollfluss:

```
if  
:: guard -> effect;  
:: ...  
fi
```

 Schleifen:

```
do  
:: guard -> effect;  
:: ...  
od
```
- Invarianten: `assert (Bedingung)` und `never-claim` (LTL Formel)

F8: Promela – Aufbau einer Spezifikation

```
// Generated on Sat Jan 24 13:32:56 CET 2004 by cTLA2Promela compiler.
// Version: ($Id: CTLA2PC.java,v 1.93 2004/01/02 14:58:55 cvs Exp $)
...
#define BVSET(bv, esz, idx, val) bv = ((bv & (~(((1<<esz)-1) << (idx*esz)))) |
    (val<<(idx*esz)))    /* Makros */
...
#define UNKNOWN_NODE 0    /* Konstanten */
...
typedef PacketT {        /* Typen */
    unsigned scn : 2;
    ...
    unsigned dat : 15;
}
...
active proctype IpArpExampleInstance() {    /* Prozesse */
    PacketBufT med_buf[MAXZONES];
    ...
    d_step {
        med_buf[0].usd = false;
        bnA_ifs[0].usd = true;
        bnA_ifs[0].rpa.usd = false;
    }
    ...
}
```


F9: Lineare temporale Logik LTL – Syntax

- ◆ Betrachtet werden unendliche Zustandsfolgen: $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$
- ◆ Syntax (φ, ψ seien Formeln)
 - **Aussage**: $a \in AP$, AP Menge der atomaren Aussagen
 - true
 - a
 - **Aussagenlogische Verknüpfungen**:
 - $\neg \varphi$
 - $\varphi \wedge \psi$
 - $\varphi \vee \psi$
 - $\varphi \Leftrightarrow \psi$
 - $\varphi \Rightarrow \psi$
 - **Grundlegende temporale Operatoren**:
 - X** φ Ne**X**t, ab nächstem Schritt gilt φ
 - φ **U** ψ **U**ntil, φ hält bis schließlich ψ
 - **Definierbare temporale Operatoren**
 - F** $\varphi \equiv \diamond \varphi \equiv \text{true U } \varphi$ **F**uture, **E**ventually, **S**chließlich,
 - G** $\varphi \equiv \square \varphi \equiv \neg \diamond \neg \varphi$ **G**lobally, **A**lways, **I**mmmer
 - φ **R** $\psi \equiv \neg(\neg \varphi \text{ U } \neg \psi)$ **R**elease, φ entlässt ψ
 - $\varphi \leadsto \psi \equiv \square(\varphi \Rightarrow \diamond \psi)$ **l**eadsto, φ führt zu ψ

F9: Lineare temporale Logik LTL – Semantik

◆ Betrachtet werden unendliche Zustandsfolgen: $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$

◆ Semantik („ \models “ steht für „erfüllt“)

– **Aussage**: $a \in AP$, AP Menge der atomaren Aussagen

$\sigma \models \text{true}$

$\sigma \models a \Leftrightarrow s_0 \models a$

– **Aussagenlogische Verknüpfungen**:

$\neg \varphi, \varphi \wedge \psi, \varphi \vee \psi, \varphi \Leftrightarrow \psi, \varphi \Rightarrow \psi$ *wie üblich*

– **Grundlegende temporale Operatoren**:

$\sigma \models X \varphi \Leftrightarrow \langle s_1, s_2, s_3, s_4, \dots \rangle \models \varphi$ **N**ext, ab nächstem Schritt gilt φ

$\sigma \models \varphi U \psi \Leftrightarrow \wedge$

$\exists j: 0 \leq j \forall i: 0 \leq i < j$

$\langle s_j, s_{j+1}, s_{j+2}, s_{j+3}, \dots \rangle \models \varphi \wedge$

$\langle s_j, s_{j+1}, s_{j+2}, s_{j+3}, \dots \rangle \models \psi$

Until, φ hält bis schließlich ψ

F9: Lineare temporale Logik LTL – Semantik

◆ Betrachtet werden unendliche Zustandsfolgen: $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$

◆ Semantik („ \models “ steht für „erfüllt“)

– Grundlegende temporale Operatoren:

$$\sigma \models X \varphi \Leftrightarrow \langle s_1, s_2, s_3, s_4, \dots \rangle \models \varphi$$

Next, ab nächstem Schritt gilt φ

$$\sigma \models \varphi U \psi \Leftrightarrow \wedge$$

$$\exists j: 0 \leq j \forall i: 0 \leq i < j$$

$$\langle s_i, s_{i+1}, s_{i+2}, s_{i+3}, \dots \rangle \models \varphi \wedge$$

$$\langle s_j, s_{j+1}, s_{j+2}, s_{j+3}, \dots \rangle \models \psi$$

Until, φ hält bis schließlich ψ

– Definierbare temporale Operatoren

$$\sigma \models F \varphi \equiv \Diamond \varphi \equiv \text{true } U \varphi \Leftrightarrow$$

$$\exists i: \langle s_i, s_{i+1}, s_{i+2}, s_{i+3}, \dots \rangle \models \varphi$$

Future, Eventually, Schließlich

$$\sigma \models G \varphi \equiv \Box \varphi \equiv \neg \Diamond \neg \varphi \Leftrightarrow$$

$$\forall i: \langle s_i, s_{i+1}, s_{i+2}, s_{i+3}, \dots \rangle \models \varphi$$

Globally, Always, Immer

$$\sigma \models \varphi R \psi \equiv \neg(\neg \varphi U \neg \psi) \Leftrightarrow$$

$$\forall j: 0 \leq j \exists i: 0 \leq i < j$$

Release, φ entlässt ψ

$$\langle s_i, s_{i+1}, s_{i+2}, s_{i+3}, \dots \rangle \models \varphi \vee$$

$$\langle s_j, s_{j+1}, s_{j+2}, s_{j+3}, \dots \rangle \models \psi$$

$$\sigma \models \varphi \leadsto \psi \equiv \Box(\varphi \Rightarrow \Diamond \psi) \Leftrightarrow$$

leadsto, φ führt zu ψ

$$\forall i: (\langle s_i, s_{i+1}, s_{i+2}, s_{i+3}, \dots \rangle \models \varphi \Rightarrow$$

$$\exists j \ i \leq j: \langle s_j, s_{j+1}, s_{j+2}, s_{j+3}, \dots \rangle \models \psi)$$

F9: Lineare temporale Logik LTL – Formulierungen

- ◆ Betrachtet werden unendliche Zustandsfolgen: $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$
- ◆ Einige Formulierungen (mittels \square , \diamond und $\sim \rightarrow$)
 - Immer wieder a
 - $\square \diamond a$
 - Schließlich immer a (a ist schließlich stabil)
 - $\diamond \square a$
 - a folgt nach endlicher Zeit b (Auftrag a terminiert mit b)
 - $a \sim \rightarrow b$
 - a ist Invariante, a gilt immer
 - $\square a$
 - a gilt nie (a ist verboten)
 - $\square \neg a$

F9: Lineare temporale Logik LTL – Beispiel

◆ Betrachtet werden unendliche Zustandsfolgen: $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$

◆ Beispiel *Verkehrsampel*

– \square rot \Rightarrow - \times grün

! auf Rot folgt nicht gleich Grün

– \square \diamond grün

! Es wird immer wieder grün

– rot \leadsto grün

! Rot folgt nach endlicher Zeit Grün

– \square (rot \Rightarrow (rot \cup rotgelb) \cup grün)

! auf Rot folgt Rotgelb und dann Grün



F9: Lineare temporale Logik LTL – Beispiel

Safety

- ◆ Bei Terminierung wurden alle Stationen informiert.
 - (nts={ } ⇒ cs=<infd, infd, ..., infd>)
- ◆ Wenn eine Station im Zustand infd ist, verlässt sie ihn nie wieder.
 - (cs[i]=infd ⇒ ¬ ◇ cs[i] ≠ infd)

Liveness

- ◆ Nach dem Start tritt nach endlicher Zeit Terminierung ein.
 - Init \rightsquigarrow nts={ }
- ◆ Jede Station wird schließlich informiert.
 - ◇ cs = <infd, infd, ..., infd>
- ◆ Jede gesendete Nachricht wird schließlich empfangen.
 - x ∈ nts \rightsquigarrow x ∉ nts

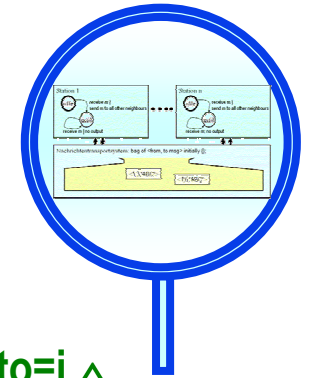
Variablen

cs: array [1..n] of (idle, infd) ! Stationen

nts: bag of < from, to, msg > ! Transportsystem

Init

cs = < idle, idle, ..., idle > ∧
nts = {<0, ini, Text>}



Aktionen

Forward

∃ i, m: cs[i]=idle ∧ m ∈ nts ∧ m.to=i ∧
cs[i]'=infd ∧ ∀ j ≠ i: cs[j]'=cs[j] ∧
nts' = nts ∪
{ <i, k, m.msg>: istNachbar(k,i) ∧
k ≠ m.from } \ {m}

Skip

∃ i, m: cs[i]=infd ∧ m ∈ nts ∧ m.to=i ∧
∀ j: cs[j]'=cs[j] ∧ nts'=nts \ {m}

F9: Lineare temporale Logik LTL – Aktionen

STS $\langle \mathbf{S}, \mathbf{S}_0, \mathbf{Next} \rangle$ kann definiert werden durch

- Menge von Datenvariablen $\mathbf{V}_1, \mathbf{V}_2, \dots, \mathbf{V}_n$
mit den Wertbereichen $\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_n$
- Initialisierungsprädikat \mathbf{Init} über Variablen
- Menge von Aktionsprädikaten $\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_m$ über Variablen und Folgevariablen

◆ Operatoren für Aktionen

◆ Aktion: Prädikat \mathbf{A} über \mathbf{V} und \mathbf{V}' notiert Teilmenge von \mathbf{Next}

- $\langle \mathbf{A} \rangle$ Aktion \mathbf{A} schaltet und ist kein Stotter Schritt
- $\mathbf{Enabled}(\mathbf{A})$ Aktion \mathbf{A} ist schaltbereit

◆ Betrachtet werden unendliche Zustandsfolgen: $\sigma = \langle \mathbf{S}_0, \mathbf{S}_1, \mathbf{S}_2, \mathbf{S}_3, \dots \rangle$

◆ Semantik

- $\sigma \models \langle \mathbf{A} \rangle \Leftrightarrow \mathbf{s}_1 \neq \mathbf{s}_2 \wedge \langle \mathbf{s}_0, \mathbf{s}_1 \rangle \models \mathbf{A}$
- $\sigma \models \mathbf{Enabled}(\mathbf{A}) \Leftrightarrow \exists x \in \mathbf{S}: \langle \mathbf{s}_0, x \rangle \models \mathbf{A}$

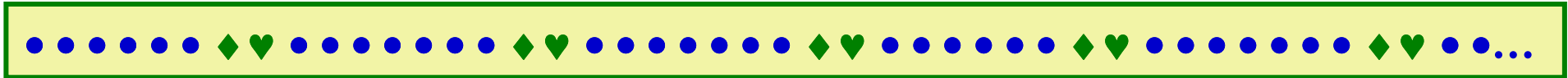
F9: LTL – Aktionenfairness

◆ Aktionenfairness in LTL

◆ $WF(A) \equiv \Box \Diamond \langle A \rangle \vee \Box \Diamond \neg \text{Enabled}(A)$

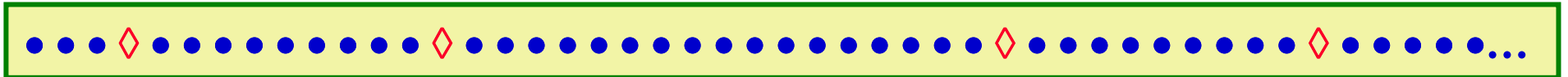
– A schaltet immer wieder

! unbedingte Fairness



– A ist immer wieder nicht schaltbereit

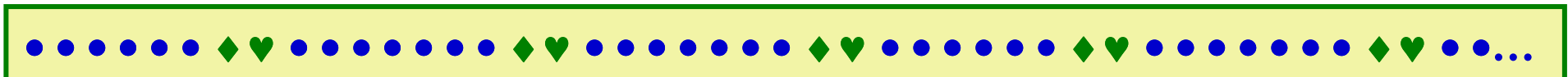
! schwache Fairness dann nicht im Zwang



◆ $SF(A) \equiv \Box \Diamond \langle A \rangle \vee \Diamond \Box \neg \text{Enabled}(A)$

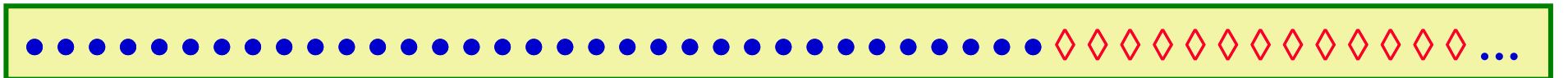
– A schaltet immer wieder

! unbedingte Fairness



– A ist schließlich überhaupt nicht mehr schaltbereit

! starke Fairness dann nicht im Zwang

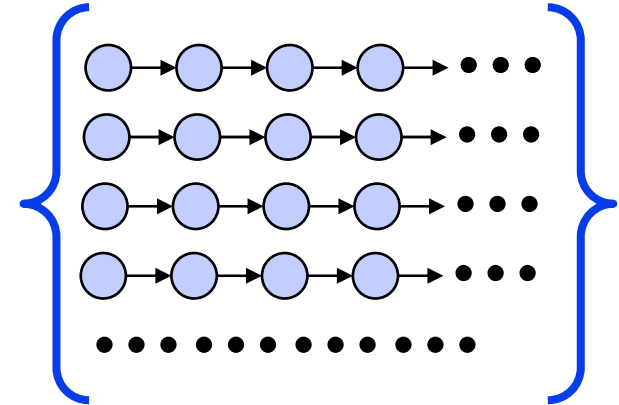


F9: Lineare temporale Logik LTL – STS erfüllt Formel

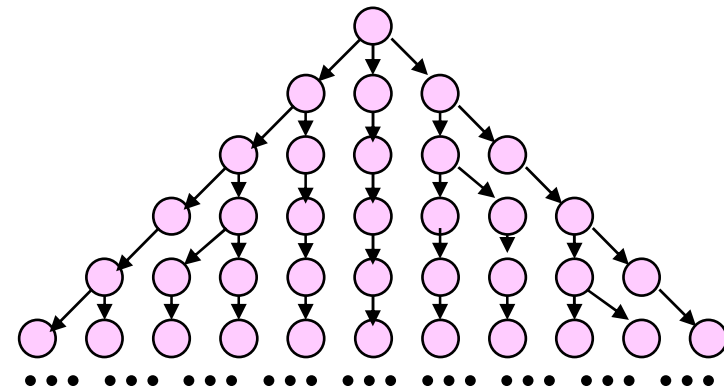
- ◆ Betrachtet werden unendliche Zustandsfolgen: $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$
- ◆ Def: **Eigenschaft Π**
 $\Pi \subset 2^{(S^\infty)}$, d.h. Π ist Menge von Zustandsfolgen
- ◆ Def: **STS $\langle S, S_0, \text{Next} \rangle$ besitzt Eigenschaft Π**
Jede mögliche Systemausführung ist in Π enthalten: $\Sigma \subset \Pi$
- ◆ Def: Von Formel **F** spezifizierte Eigenschaft
 - Zustandsfolgenmenge $\Pi = \{ \sigma : \sigma \models F \}$
- ◆ Def: **STS erfüllt Formel F, STS $\models F$**
 - $\Sigma \subset \Pi$

F9: Computation Tree Logik – CTL

- ◆ **LTL:** Betrachtet werden **Mengen von Zustandsfolgen**
- ◆ **Linear Time**
- ◆ Allerdings: Die Dynamik eines Systems kann weiter durch den Erreichbarkeitsgraph dargestellt werden:
 - Die Menge der möglichen Zustandsfolgen ist die Menge der möglichen Pfade im Graph.
 - Darüber hinaus ist im Graph erkennbar, ob einzelne Pfade mit bestimmten Eigenschaften vorhanden sind.
 - ➔ LTL quantifiziert implizit über alle Pfade des Graphen.
 - ➔ Interessant ist ergänzend eine Logik, die über die Existenz von Pfaden sprechen kann.



- ◆ **CTL:** Betrachtet werden **Bäume**
 - Ein Baum ist ein abgewickelter Erreichbarkeitsgraph
- ◆ **Branching Time**
- ◆ CTL unterstützt die explizite Quantifikation über Pfaden
 - **E:** es existiert ein Pfad, so dass
 - **G:** für alle Pfade gilt



F9: Computation Tree Logik – Syntax

◆ Betrachtet werden unendliche Bäume: $\rho =$

◆ Syntax (φ, ψ seien CTL-Formeln)

– **Aussage:** $a \in AP$, AP Menge der atomaren Aussagen
true, false, a

– **Aussagenlogische Verknüpfungen:**

$\neg \varphi, \varphi \wedge \psi, \varphi \vee \psi, \varphi \Leftrightarrow \psi, \varphi \Rightarrow \psi$

– **Pfadquantifizierte Formeln**

A X φ all next, für alle Pfade gilt im nächsten Schritt φ

E X φ exist next, es existiert Pfad, so dass im nächsten Schritt φ

A φ U ψ all until, für alle Pfade gilt φ bis ψ

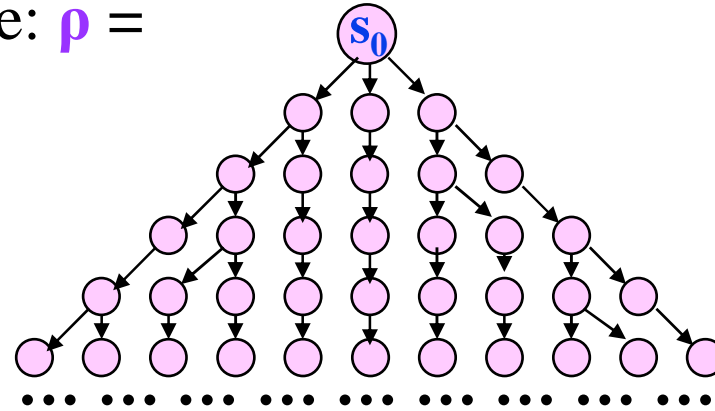
E φ U ψ exist next, es existiert Pfad, in welchem φ bis ψ gilt

A G φ all globally, für alle Pfade gilt immer φ

E G φ exist globally, es existiert Pfad, so dass dort immer φ

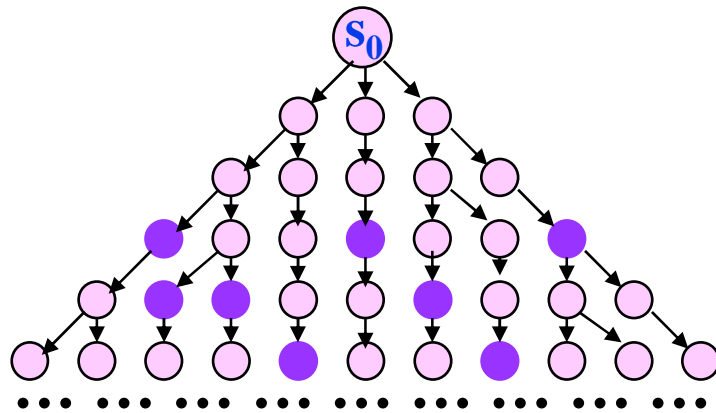
A F φ all futures, für alle Pfade gilt schließlich φ

E F φ exist future, es existiert Pfad, so dass dort schließlich φ

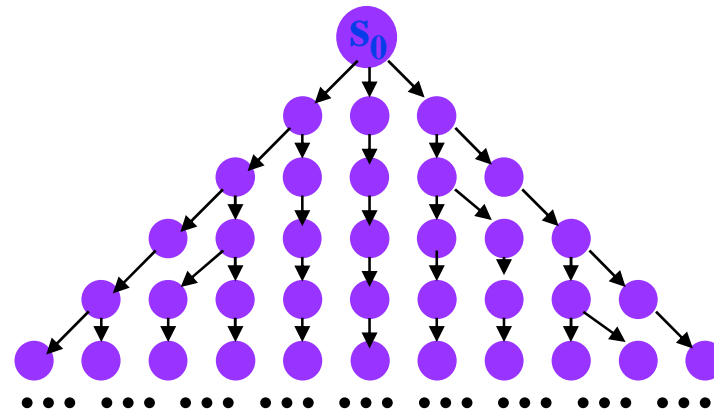


F9: Computation Tree Logik – Formulierungen

Alle schließlich: **AF** φ

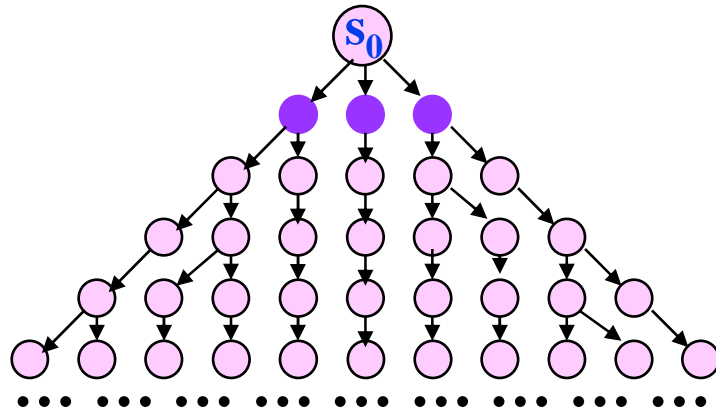


Alle immer: **AG** φ

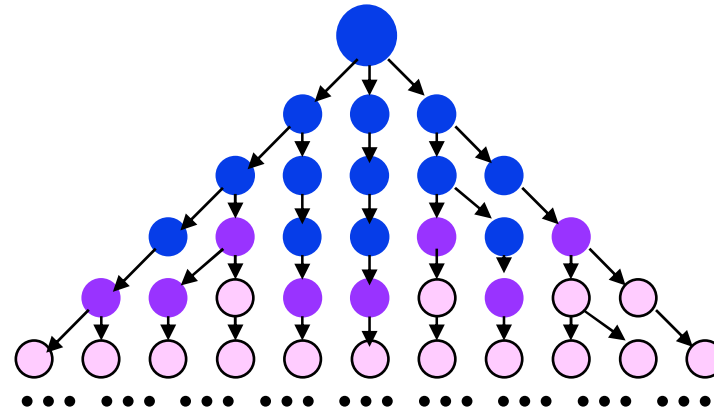


F9: Computation Tree Logik – Formulierungen

Alle next: $AX \varphi$

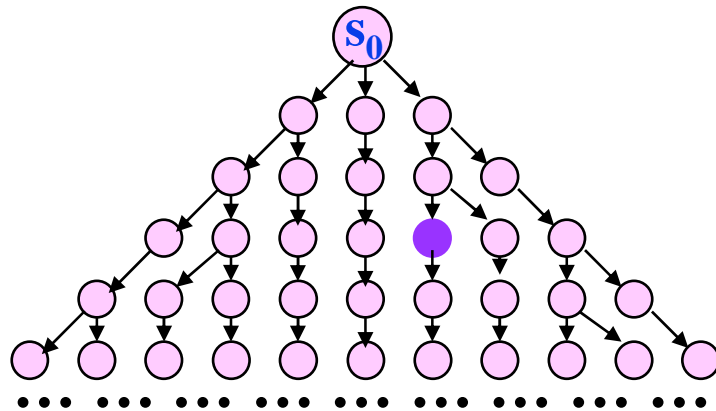


Alle bis: $A \varphi U \psi$

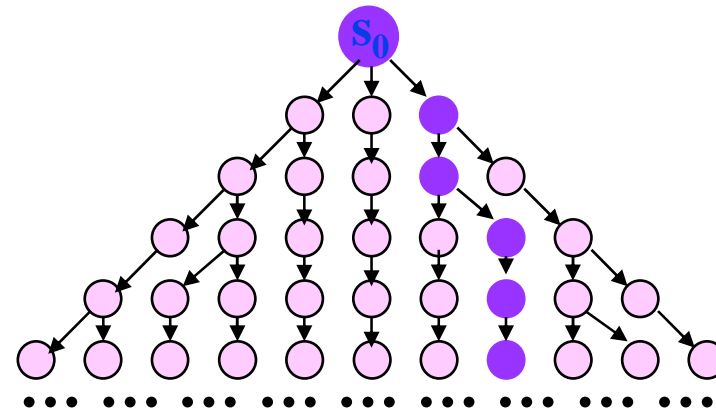


F9: Computation Tree Logik – Formulierungen

Einer schließlich: $EF \varphi$

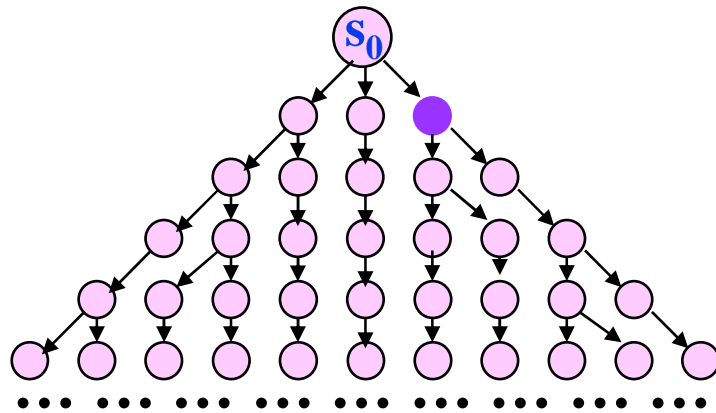


Einer immer: $EG \varphi$

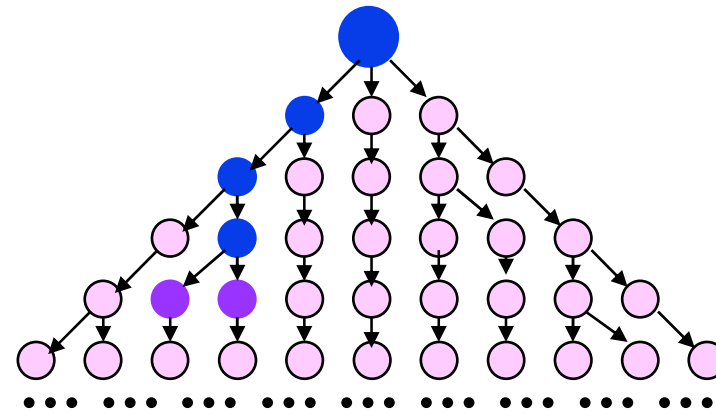


F9: Computation Tree Logik – Formulierungen

Einer next: $EX \varphi$



Einer bis: $E \varphi U \psi$



F9: Computation Tree Logik – Semantik

- ◆ Betrachtet werden unendliche Bäume ρ mit Wurzel s_0 :
(bzw. Unterbäume ρ, s_i davon)

- ◆ $\Sigma(s_0)$ sei die Menge der in s_0 startenden Pfade der Form $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$.

- ◆ Semantik („ \models “ steht für „erfüllt“)

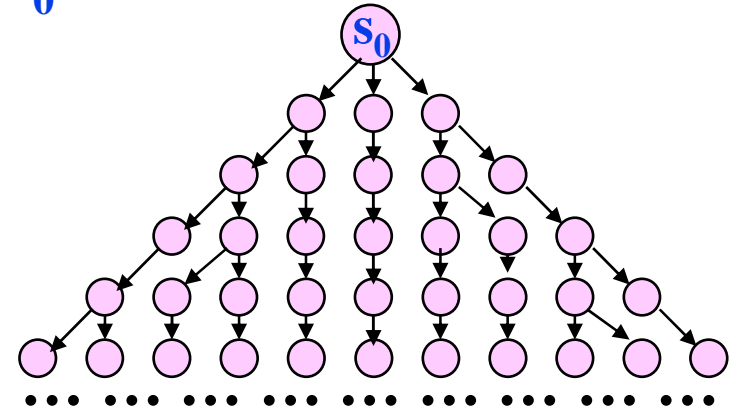
- **Aussage:** $a \in AP$,
AP Menge der atomaren Aussagen
 $\rho, s_0 \models \text{true}$
 $\rho, s_0 \models a \Leftrightarrow s_0 \models a$
- **Aussagenlogische Verknüpfungen:**
 $\neg \varphi, \varphi \wedge \psi, \varphi \vee \psi, \varphi \Leftrightarrow \psi, \varphi \Rightarrow \psi$ wie üblich
- **Pfadquantifizierte Formeln**

$$\rho, s_0 \models \mathbf{A X} \varphi \Leftrightarrow \forall \sigma \in \Sigma(s_0): \rho, s_1 \models \varphi$$

$$\rho, s_0 \models \mathbf{E X} \varphi \Leftrightarrow \exists \sigma \in \Sigma(s_0): \rho, s_1 \models \varphi$$

$$\rho, s_0 \models \mathbf{A} \varphi \mathbf{U} \psi \Leftrightarrow \forall \sigma \in \Sigma(s_0): \exists j: \rho, s_j \models \psi \wedge \forall i, 0 \leq i < j: \rho, s_i \models \varphi$$

$$\rho, s_0 \models \mathbf{E} \varphi \mathbf{U} \psi \Leftrightarrow \exists \sigma \in \Sigma(s_0): \exists j: \rho, s_j \models \psi \wedge \forall i, 0 \leq i < j: \rho, s_i \models \varphi$$

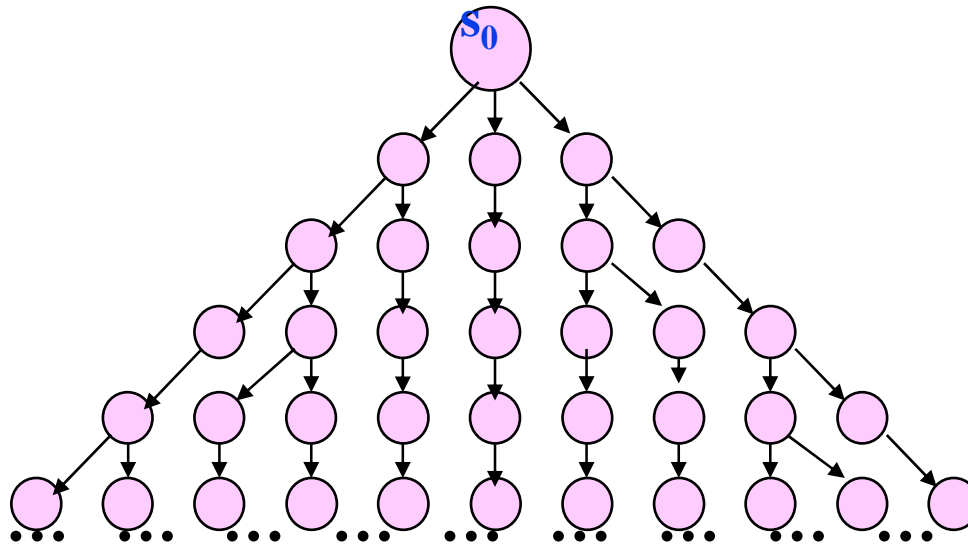


F9: Computation Tree Logik – Beispielaussagen

Safety

„*Etwas Schlechtes wird nicht passieren*“

- ◆ $\mathbf{A G} \neg (\text{TürOffen} \wedge \neg \text{KabineDa})$
- ◆ $\mathbf{A G} \neg (\text{RichtungAGrün} \wedge \mathbf{A X} \text{RichtungBGrün})$
- ◆ $\mathbf{A G} \neg (x=0 \wedge \mathbf{A X} y=z/x)$

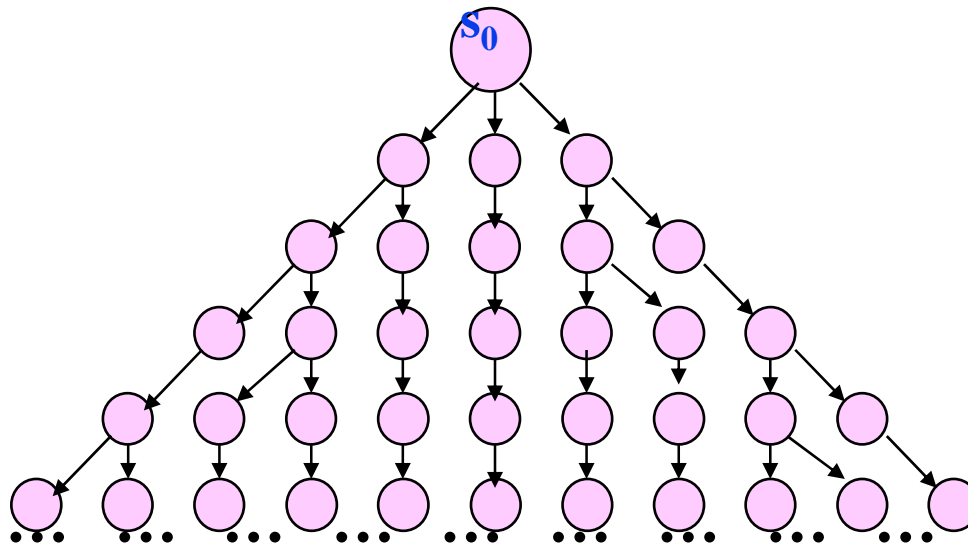


F9: Computation Tree Logik – Beispielaussagen

Liveness

„*Etwas Gutes wird nach endlicher Zeit passieren*“

- ◆ **A F** (TürOffen \wedge KabineDa)
- ◆ **A F** RichtungAGrün
- ◆ **A G** (start \Rightarrow **A F** terminiert)

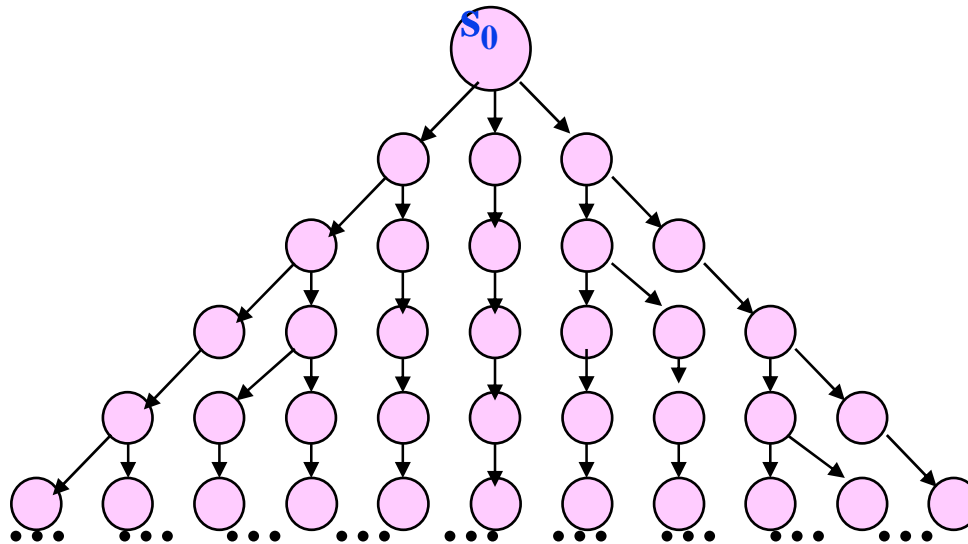


F9: Computation Tree Logik – Beispielaussagen

Scheduling Fairness

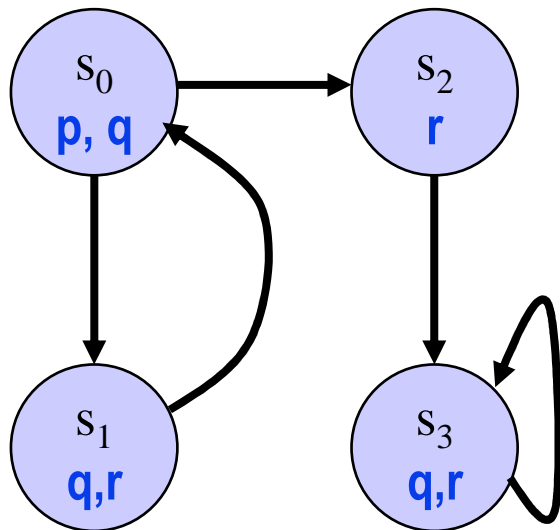
„*Etwas Gutes wird unendlich oft ermöglicht*“

◆ **AG (AF AufzugKannNeuenAuftragAnnehmen)**



F9: Computation Tree Logik – Beispielaussagen

Wir betrachten folgendes Transitionssystem
(in den Zuständen sind die dort jeweils gültigen
Aussagen notiert)

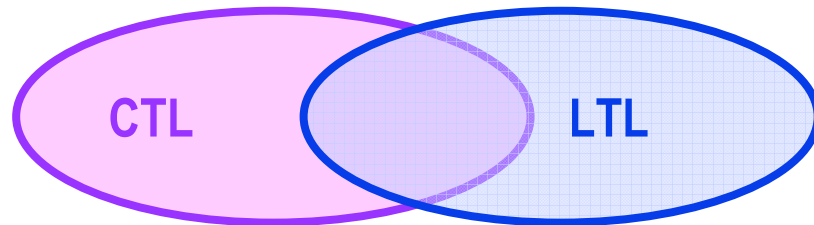
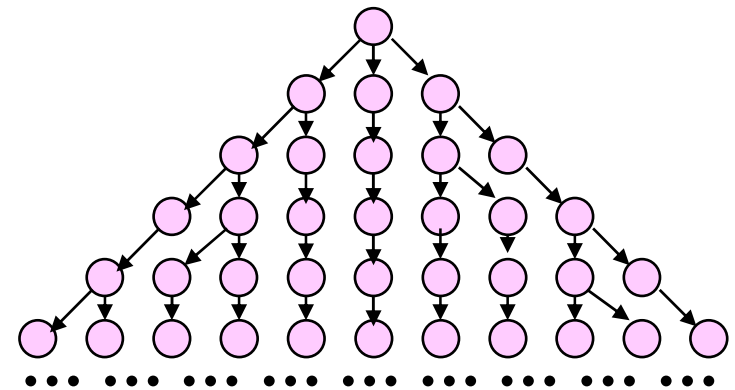
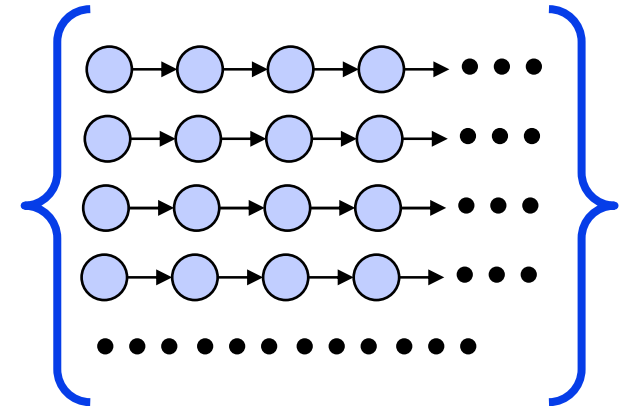


Gelten folgende Formeln für Abläufe mit
bestimmtem Startzustand ?

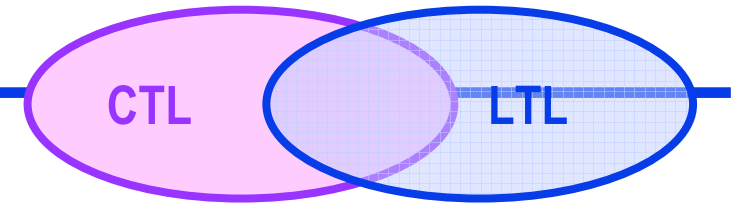
1. s_0 : $EX(\neg p)$
2. s_0 : $EXEG(r)$
3. s_1 : $AG(q \vee r)$
4. s_2 : $A(r U q)$
5. s_1 : $A(q U AG(r))$
6. s_1 : $E(q U EG(r))$
7. s_0 : $\neg EG(q)$
8. s_1 : $EFAG(q)$

F9: LTL – CTL

- ◆ **LTL:** Betrachtet werden **Mengen von Zustandsfolgen**
 - ◆ **Linear Time**
 - ◆ **CTL:** Betrachtet werden **Bäume**
 - ◆ **Branching Time**
-
- ◆ Es gibt LTL Formeln, die nicht in CTL ausgedrückt werden können
 - ◆ Es gibt CTL Formeln, die nicht in LTL ausgedrückt werden können



F9: LTL – CTL



- ◆ Es gibt LTL Formeln, die nicht in CTL ausgedrückt werden können (z.B. Formeln, die sich auf auf spezielle Teilpfade beziehen)

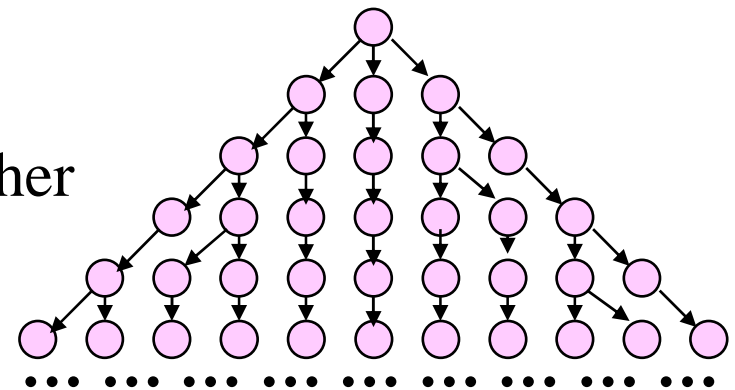
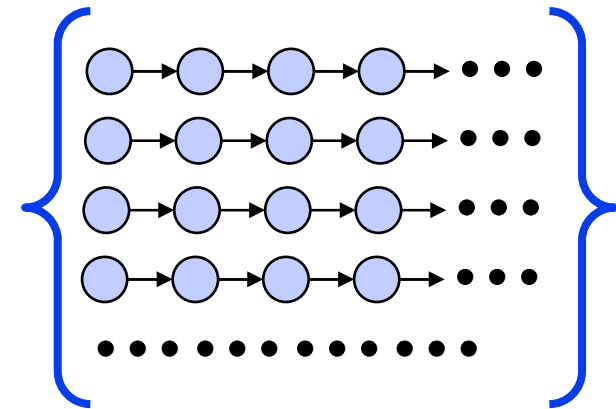
- $\Box \Diamond \langle a \rangle \vee \Diamond \Box \text{Disabled}(a)$

- ◆ Es gibt CTL Formeln, die nicht in LTL ausgedrückt werden können (z.B. Formeln mit existenzquantifizierten Pfaden)

- $AG(\varphi \Rightarrow EF\psi)$

- ◆ Es gibt LTL- CTL-Formeln, die sich entsprechen (z.B. LTL-Formeln mit einfacher Schachtelung temporaler Operatoren)

- $\Box \neg \varphi$ $AG \neg \varphi$
 - $\Diamond \varphi$ $AF \varphi$
 - $\Box (\varphi \Rightarrow \Diamond \psi)$ $AG(\varphi \Rightarrow AF \psi)$
 - $\Box \Diamond \varphi$ $AGF \varphi$



F9: LTL+CTL: CTL*

◆ CTL* kombiniert LTL und CTL

- die temporalen Operatoren können in CTL* annähernd beliebig eingesetzt werden
- Beispiele

» $A (X \varphi \vee X X \varphi)$

In allen Pfaden gilt φ im zweiten oder dritten Zustand

» $E (\Box \Diamond \varphi)$

Es gibt einen Pfad, in welchem φ immer wieder gilt

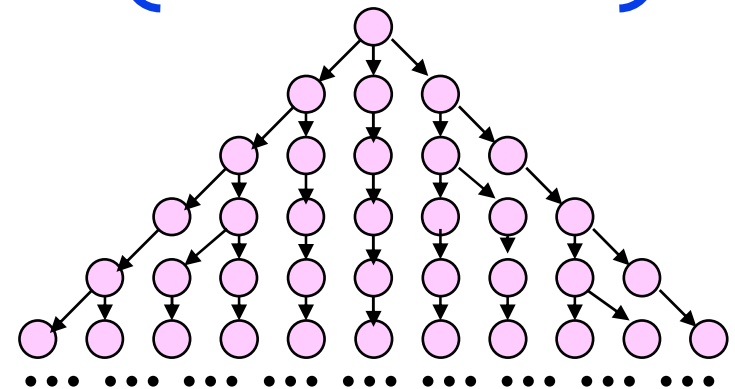
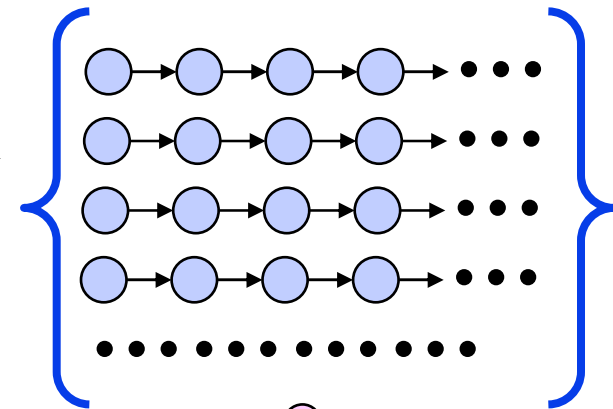
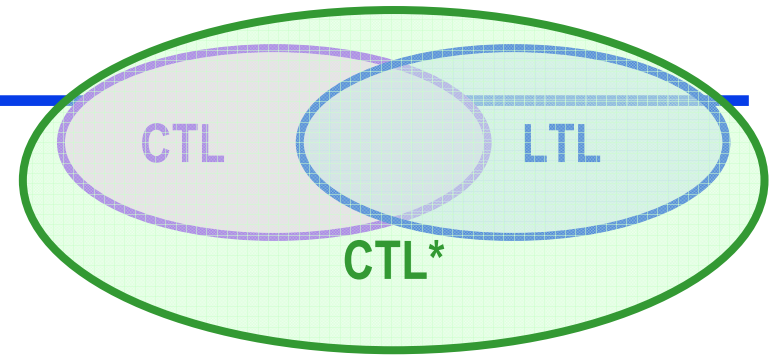
– φ ist Formel in CTL

φ ist Formel in CTL*

– φ ist Formel in LTL

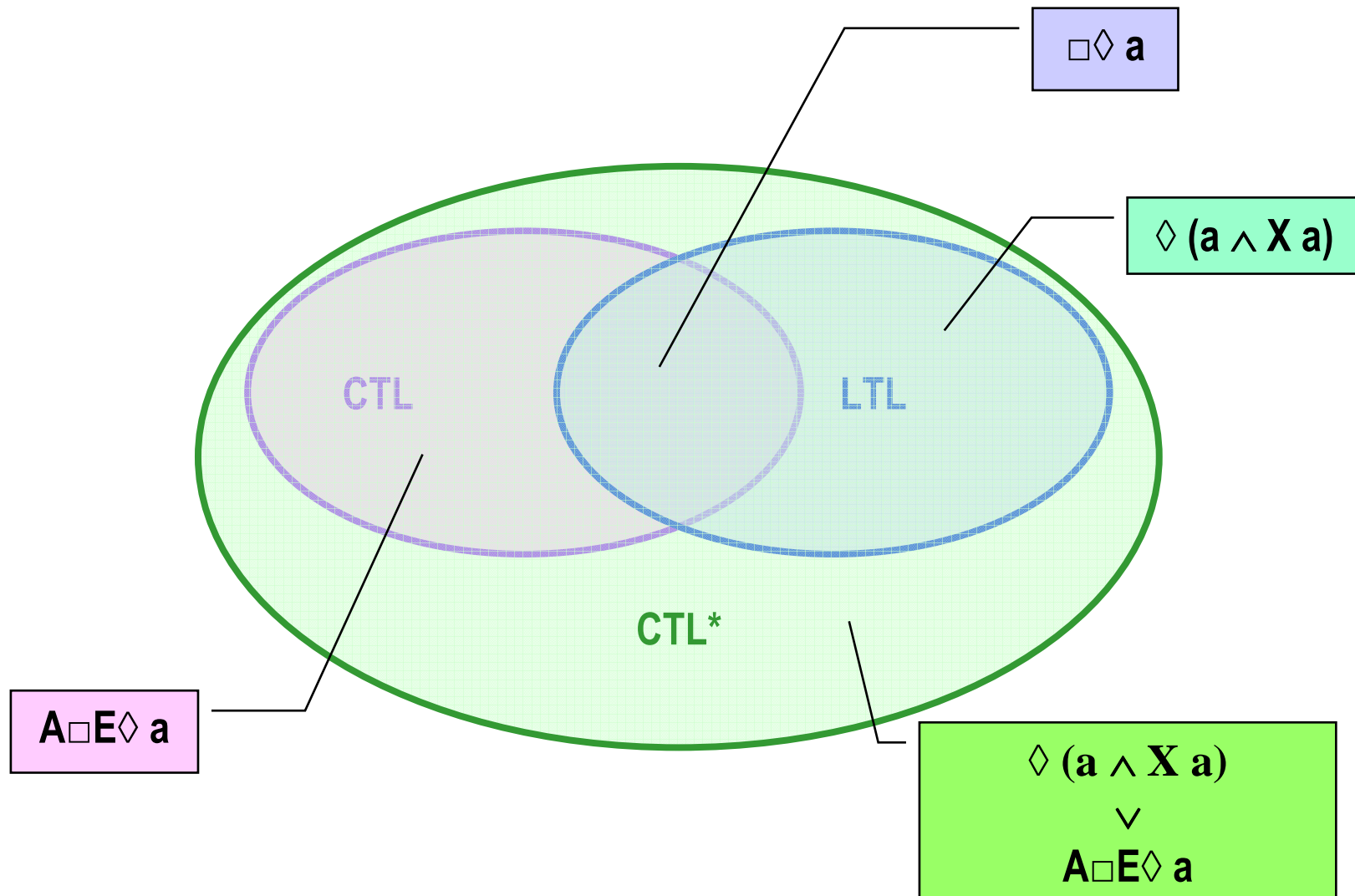
$A \varphi$ ist Formel in CTL*

- darüber hinaus gibt es in CTL* Formeln, die weder CTL- noch LTL-Formeln sind



F9: LTL+CTL: CTL*

- ◆ CTL* kombiniert LTL und CTL



F9: CTL* – Syntax

- ◆ Es gibt zwei Klassen von Formeln
 - **Zustandsformeln**, sie werden anhand von Zuständen ausgewertet
 - **Pfadformeln**, sie werden anhand von Pfaden ausgewertet
- ◆ Syntax (φ, ψ seien Zustandsformeln; Φ, Ψ seien Pfadformeln; a sei eine atomare Aussage)
 - **Zustandsformeln**:
 - $a, \text{true}, \text{false}, \neg \varphi, \varphi \wedge \psi, \varphi \vee \psi, \varphi \Leftrightarrow \psi, \varphi \Rightarrow \psi$
 - $A \Phi, E \Phi$
 - **Pfadformeln**
 - φ
 - $\neg \Phi, \Phi \wedge \Psi, \Phi \vee \Psi, \Phi \Leftrightarrow \Psi, \Phi \Rightarrow \Psi$
 - $X \Phi, \square \Phi, \diamond \Phi, \Phi U \Psi$

F9: CTL* – Semantik: Zustandsformeln

- ◆ Betrachtet werden unendliche Bäume ρ mit Wurzel s_0 :
(bzw. Unterbäume ρ, s_i davon)

- ◆ $\Sigma(s_0)$ sei die Menge der in s_0 startenden Pfade
der Form $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$.

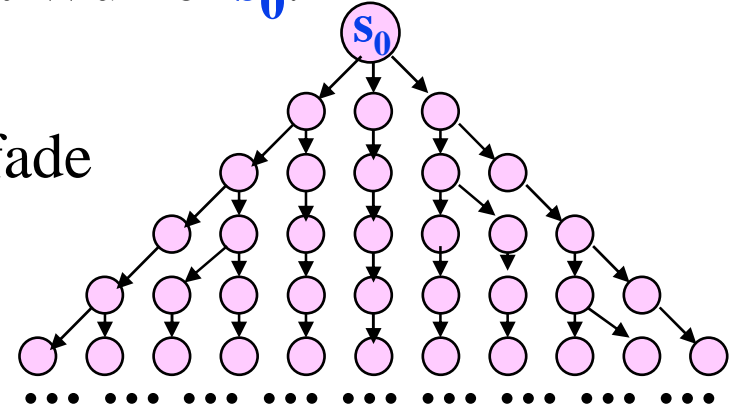
- ◆ φ, ψ seien Zustandsformeln
 Φ, Ψ seien Pfadformeln
 a sei eine atomare Aussage

- ◆ Semantik („ \models “ steht für „erfüllt“)

- $\rho, s_0 \models \text{true}$
 $\rho, s_0 \models a \Leftrightarrow s_0 \models a$
- Aussagenlogische Verknüpfungen:
 $\neg \varphi, \varphi \wedge \psi, \varphi \vee \psi, \varphi \Leftrightarrow \psi, \varphi \Rightarrow \psi$ wie üblich
- Pfadquantifizierte Formeln

$$\rho, s_0 \models \mathbf{A} \Phi \Leftrightarrow \forall \sigma \in \Sigma(s_0): \rho, \sigma \models \Phi$$

$$\rho, s_0 \models \mathbf{E} \Phi \Leftrightarrow \exists \sigma \in \Sigma(s_0): \rho, \sigma \models \Phi$$



F9: CTL* – Semantik: Pfadformeln

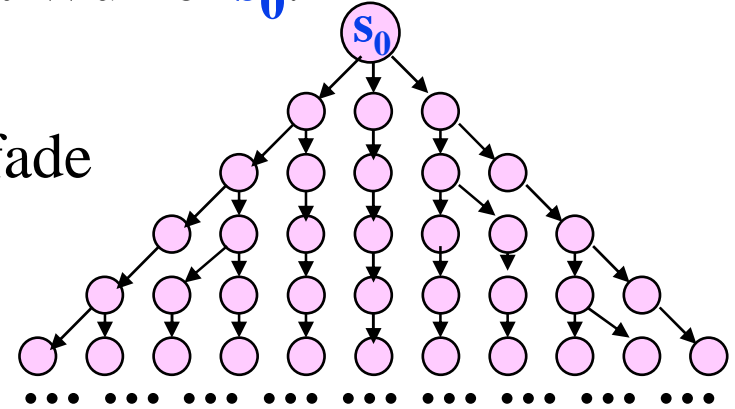
- ◆ Betrachtet werden unendliche Bäume ρ mit Wurzel s_0 :
(bzw. Unterbäume ρ, s_i davon)

- ◆ $\Sigma(s_0)$ sei die Menge der in s_0 startenden Pfade
der Form $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$.

- ◆ φ, ψ seien Zustandsformeln
 Φ, Ψ seien Pfadformeln
 a sei eine atomare Aussage

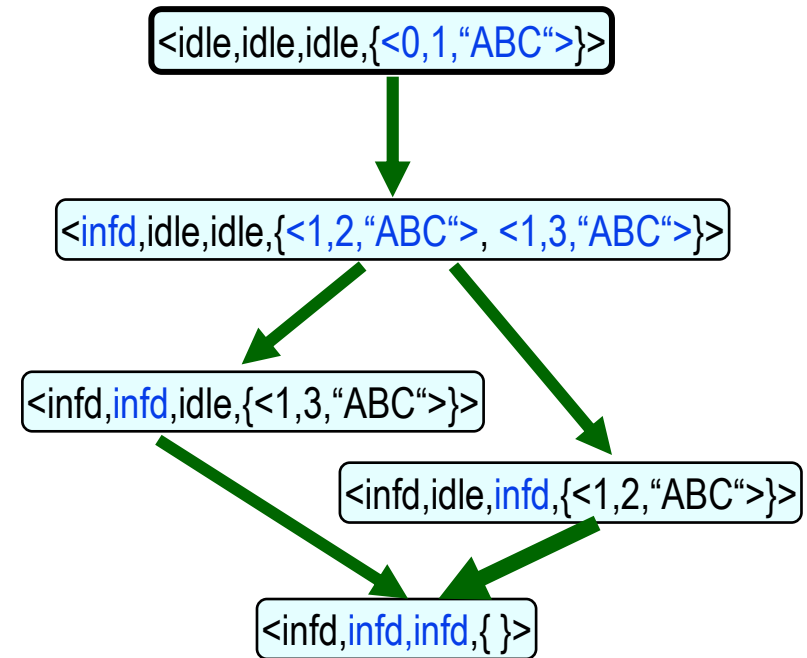
- ◆ Semantik („ \models “ steht für „erfüllt“)

- $\sigma \models \varphi \Leftrightarrow \rho, s_0 \models \varphi$
- Aussagenlogische Verknüpfungen:
 $\neg \Phi, \Phi \wedge \Psi, \Phi \vee \Psi, \Phi \Leftrightarrow \Psi, \Phi \Rightarrow \Psi$ wie üblich
- $\sigma \models \diamond \varphi \Leftrightarrow$
 $\exists i: \langle s_i, s_{i+1}, s_{i+2}, s_{i+3}, \dots \rangle \models \varphi$
- $\sigma \models \square \varphi \Leftrightarrow$
 $\forall i: \langle s_i, s_{i+1}, s_{i+2}, s_{i+3}, \dots \rangle \models \varphi$



F10: Model Checking

- ◆ Vorstufe: **Erreichbarkeitsanalyse**
 - Berechne und inspiziere den Erreichbarkeitsgraphen
 - » allgemeine Kriterien
 - » spezielle Kriterien
- ◆ Spezielles Kriterium: Ist φ erfüllt?
d.h. ist Erreichbarkeitsgraph Modell der Formel φ ?
→ **Model Checking**
- ◆ Weitere Stufe:
On-the-Fly-Model-Checking
Erreichbarkeitsgraph wird nicht vollständig gespeichert.



E. Clarke, O. Grumberg, D. Peled:
Model Checking, MIT-Press, 2000.
Ch. Baier, J.P. Katoen, K. Larsen:
Principles of Model Checking, MIT Press, 2008.

CTL Model Checking

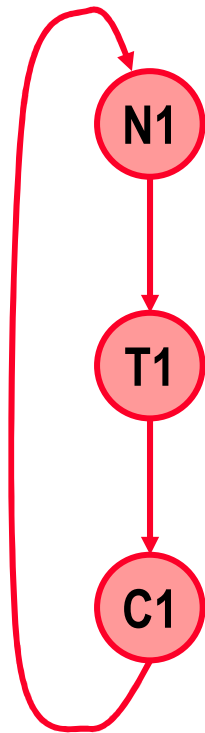
- ◆ Gegeben:
Repräsentation eines (unendlichen) Baums ρ mit Wurzel s_0
- ◆ Gegeben: CTL-Formel φ
- ◆ Gesucht: Gilt $\rho, s_0 \models \varphi$?

- ◆ Baum als Graph repräsentiert
 - Knoten: Erreichbare Zustände
 - Kanten: Transitionen

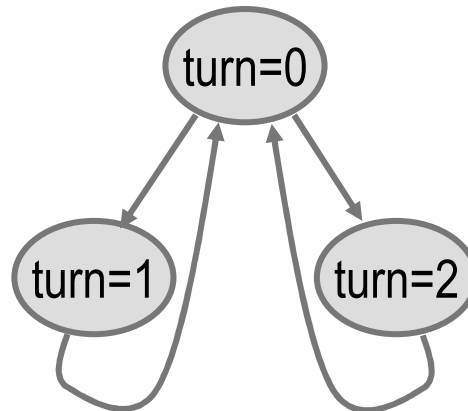
- ◆ Safety: **AG** φ
 - Erfüllen alle erreichbaren Zustände φ
- ◆ Liveness: **AG**($\varphi \Rightarrow$ **AF** ψ)
 - Erfüllen alle Pfade die Bedingung $\varphi \rightsquigarrow \psi$

Model Checking: Beispiel „Gegenseitiger Ausschluss“

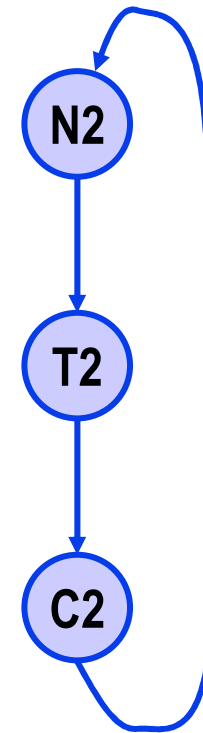
Prozess 1



Gegenseitiger
Ausschluss:
Verwalter



Prozess 2

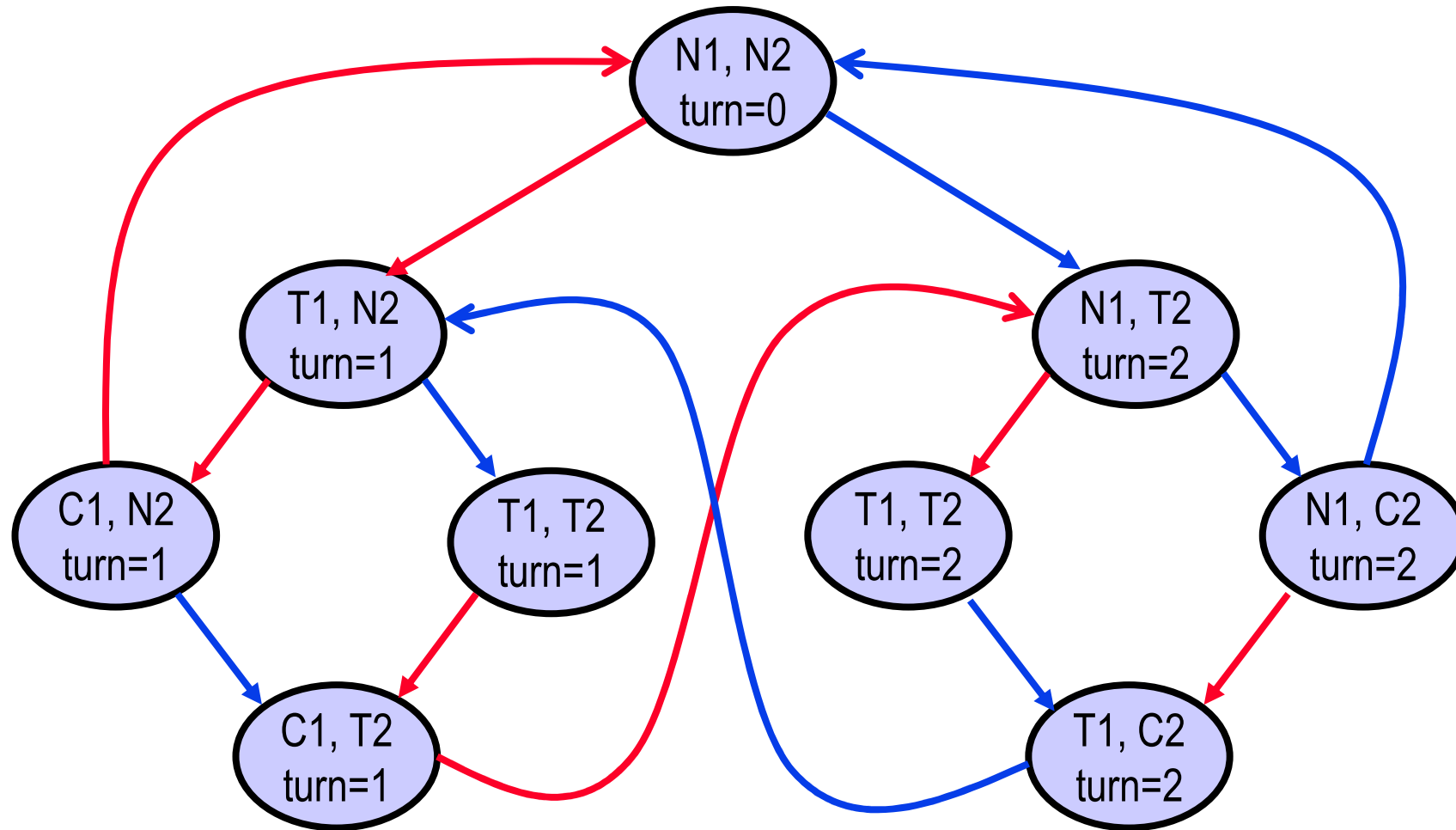


N: Nicht kritisch

T: Anfordernd

C: Kritisch

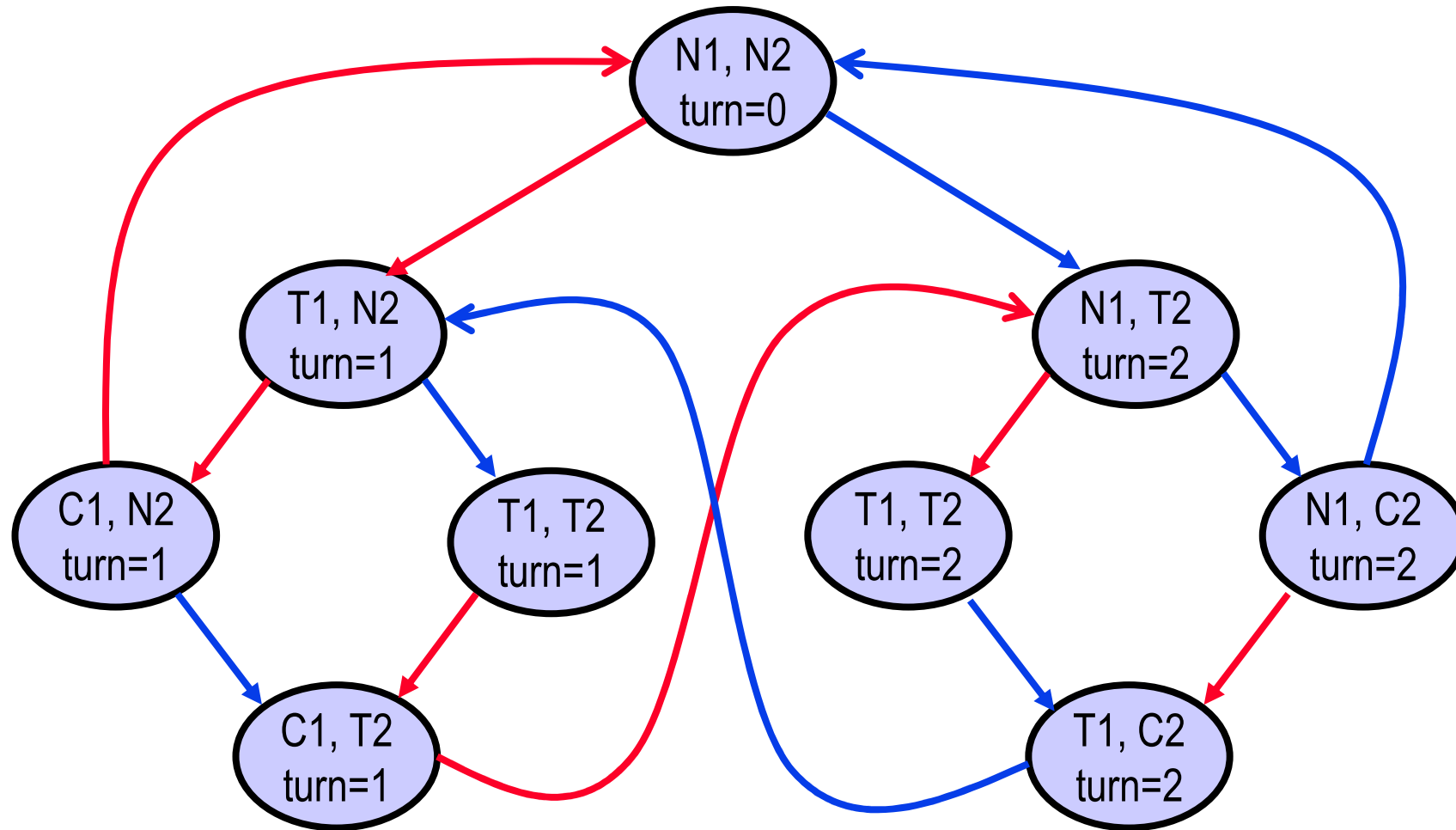
LTL Model Checking: Safety am Beispiel



Gilt: $\square \neg (C1 \wedge C2)$?

Ja: Alle erreichbaren Zustände erfüllen „ $\neg (C1 \wedge C2)$ “

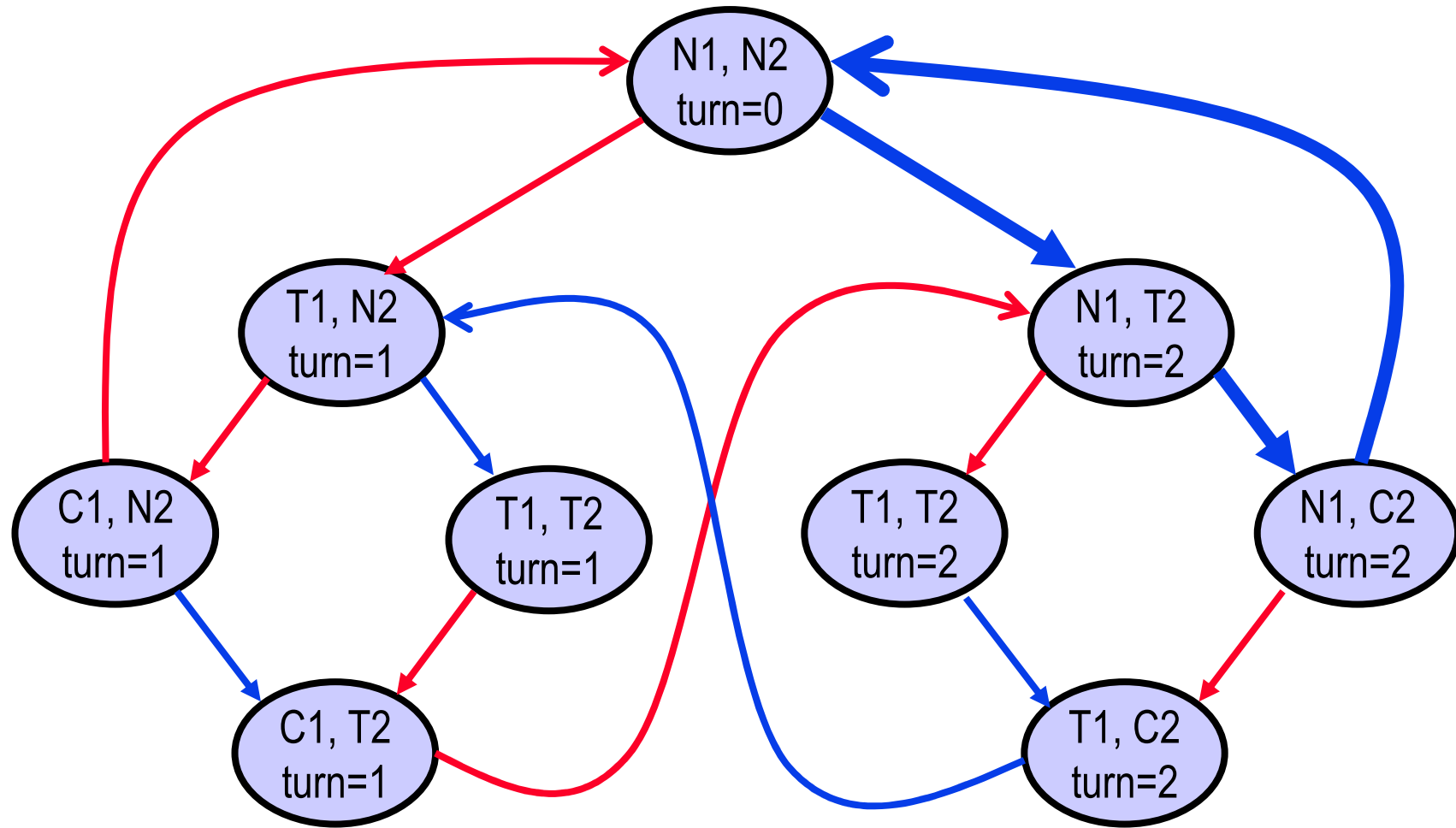
LTL Model Checking: Liveness am Beispiel



Gilt: $T1 \leadsto C1$
„ $T1$ leadsto $C1$ “ ?

Ja: Jeder Pfad, der in einem Zustand mit $T1$ startet, führt bestimmt zu einem Zustand mit $C1$

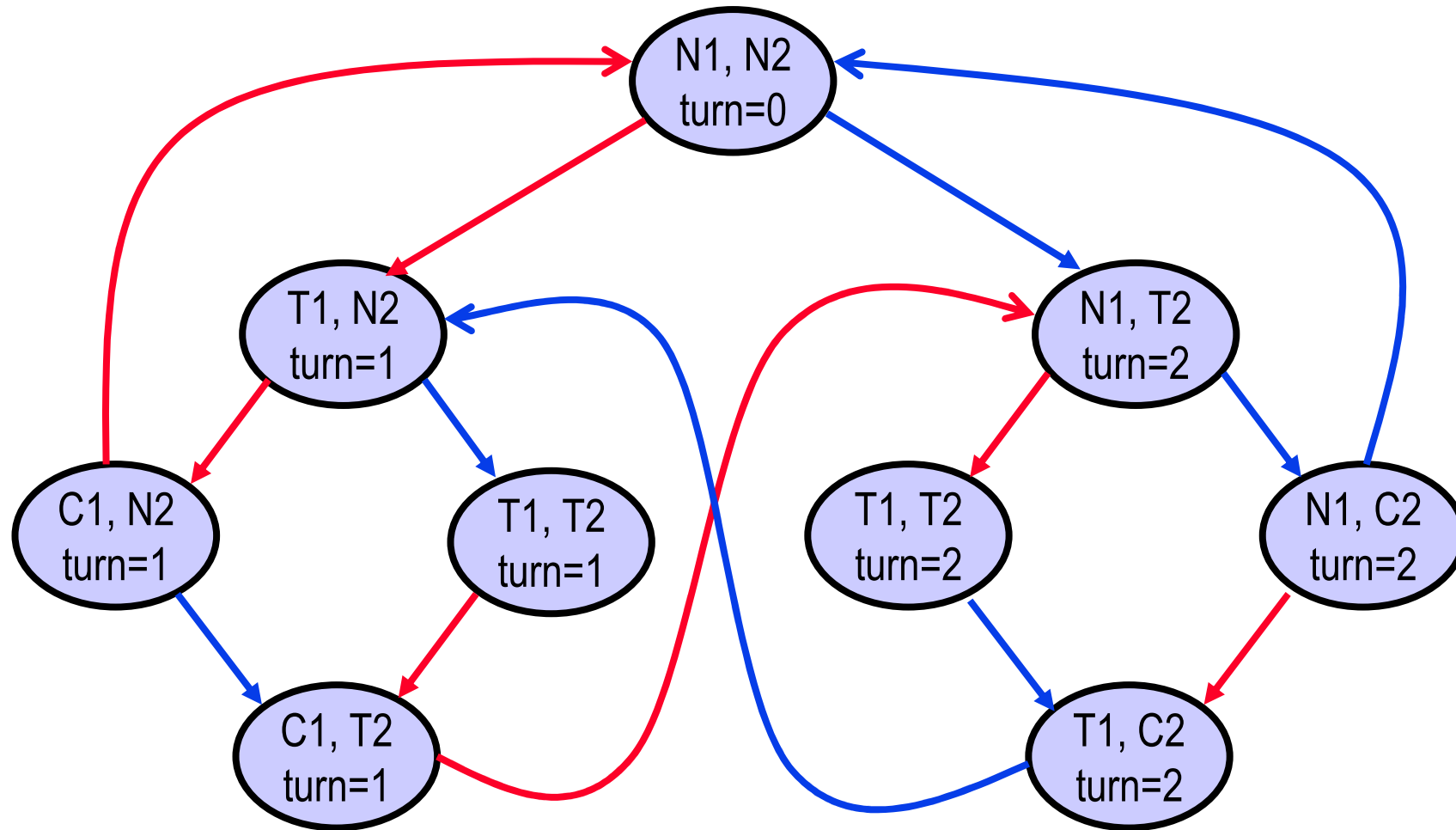
LTL Model Checking: Fairness am Beispiel



Gilt: $\square \diamond C1$
„immer wieder $C1$ “ ?

Nein: Denn es gibt z.B. den dick-blauen zyklischen Pfad, der nie einen Zustand mit $C1$ durchläuft.

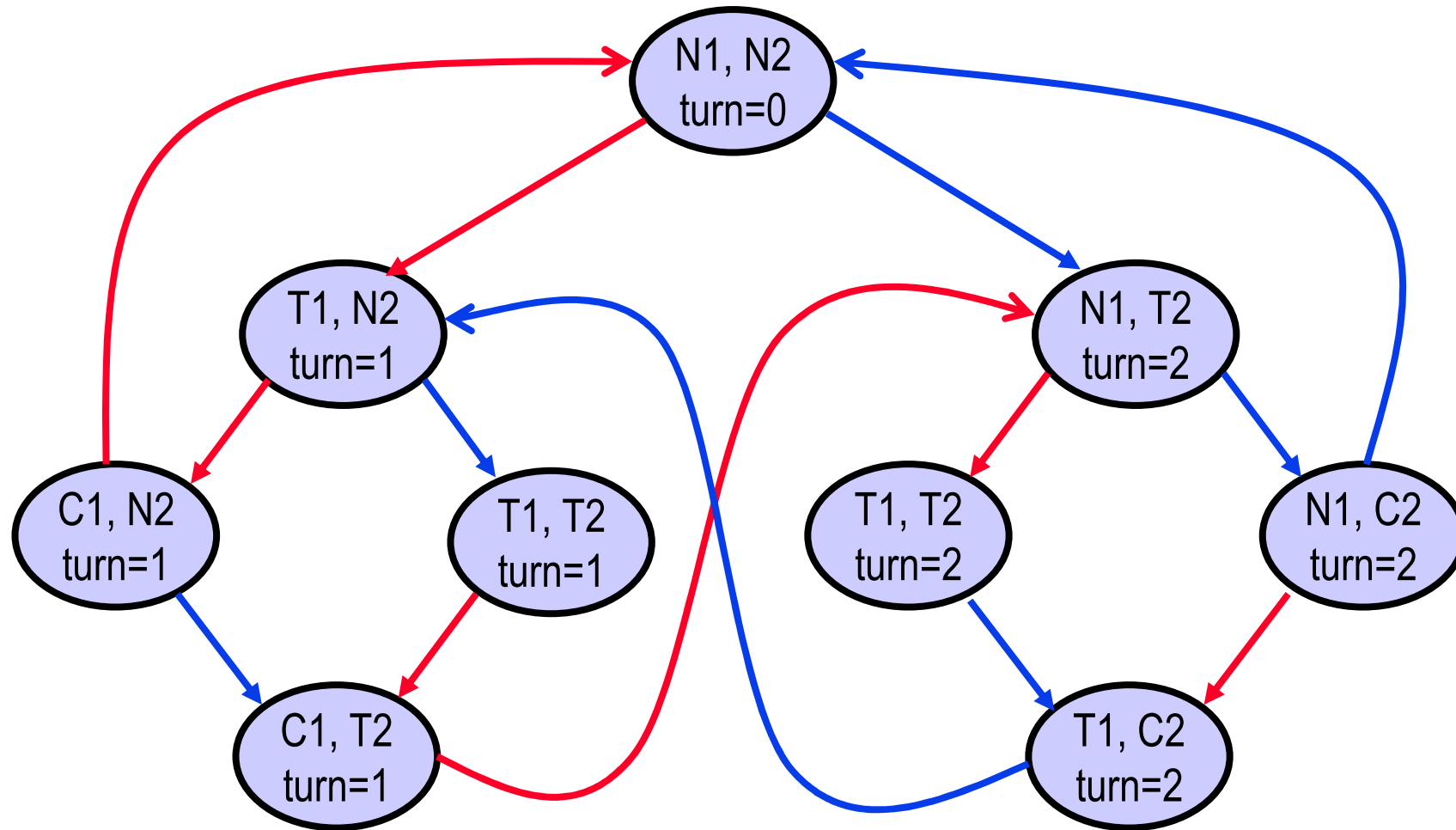
CTL Model Checking: Safety am Beispiel



Gilt: $AG \neg(C1 \wedge C2)$?

Ja: Alle erreichbaren Zustände erfüllen „ $\neg(C1 \wedge C2)$ “

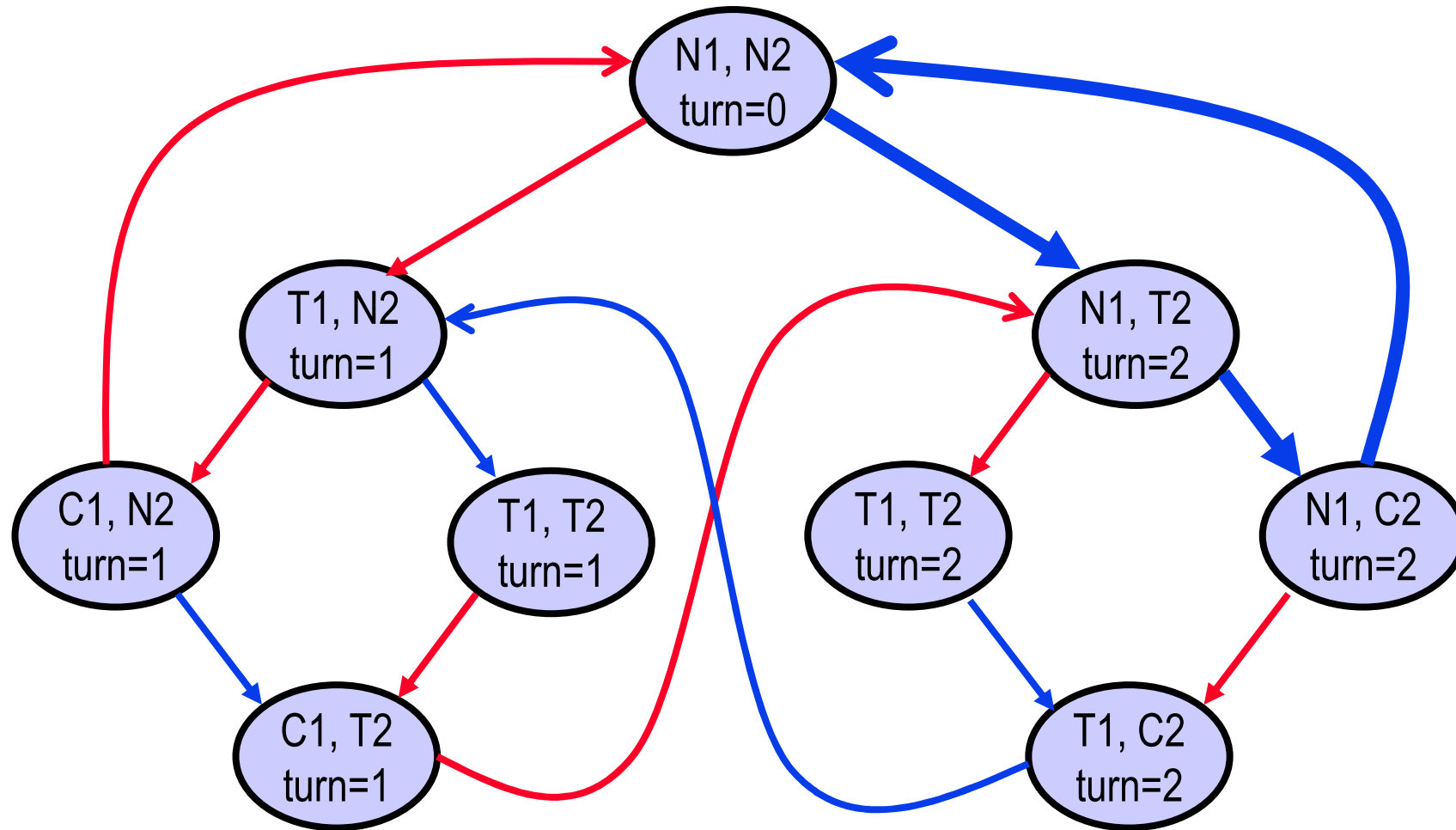
CTL Model Checking: Liveness am Beispiel



Gilt: $AG(T1 \Rightarrow AF C1)$
„*T1 leadsto C1*“ ?

Ja: Jeder Pfad, der in einem Zustand mit **T1** startet, führt bestimmt zu einem Zustand mit **C1**

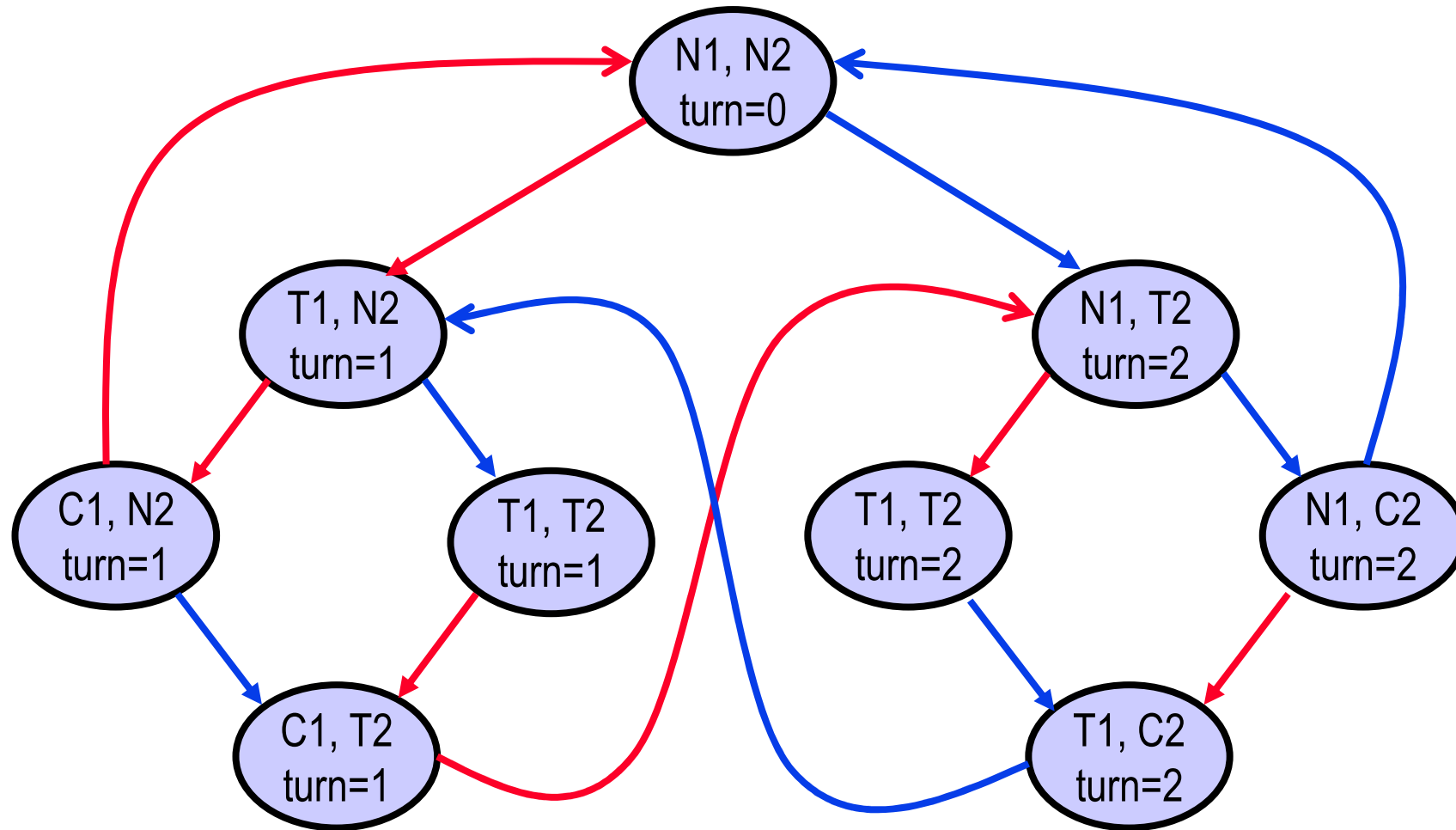
CTL Model Checking: Fairness am Beispiel



Gilt: AGAF C1
„immer wieder C1“ ?

Nein: Denn es gibt z.B. den dick-blauen zyklischen Pfad, der nie einen Zustand mit **C1** durchläuft.

CTL Model Checking: Non-Blocking

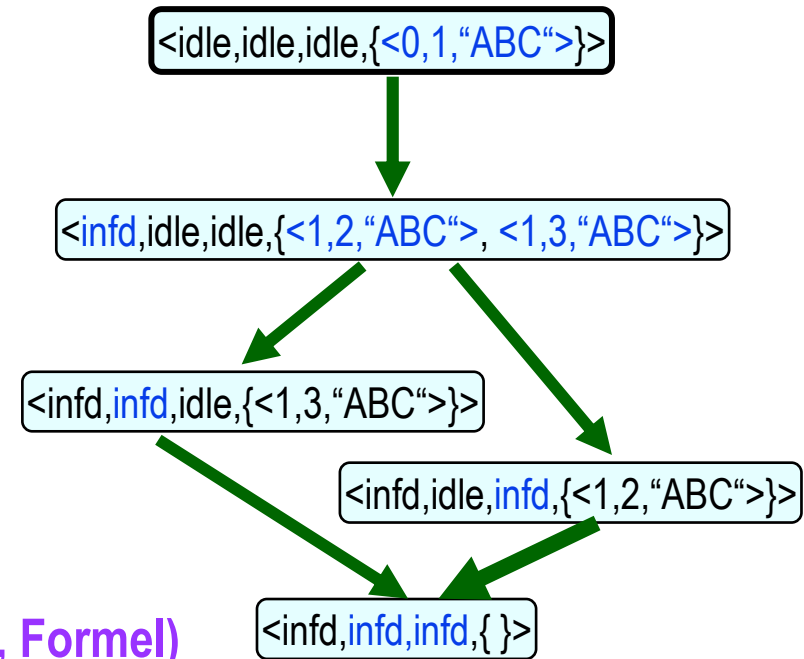


Gilt: $AG(N1 \Rightarrow EF T1)$
*„P1 kann immer
wieder versuchen“ ?*

Ja: Von jedem Zustand mit N1 aus gibt es einen Pfad,
der einen Zustand mit T1 durchläuft.

F10: Model Checking – Schlussbemerkungen

- ◆ Model Checking ist inzwischen breites Feld der theoretischen Informatik
- ◆ Logiken
 - LTL, TLA
 - Hennessy-Milner, CTL, CTL*
 - modaler μ -Kalkül, μ -Kalkül
 - dynamische Logik
 - ...
- ◆ (Erweiterte) Normalformen
- ◆ Algorithmen für das Problem **IstModell(System, Formel)**
 - Komplexität der Eingaben: System, Formel
 - Komplexität der Algorithmen: Zeit, Speicher
- ◆ Weitere Stufe:
On-the-Fly-Model-Checking
Erreichbarkeitsgraph wird nicht vollständig gespeichert.



E. Clarke, O. Grumberg, D. Peled:
Model Checking, MIT-Press, 2000.
Ch. Baier, J.P. Katoen, K. Larsen:
Principles of Model Checking, MIT Press, 2008.

F10: Model Checking – Schlussbemerkungen

◆ Komplexität

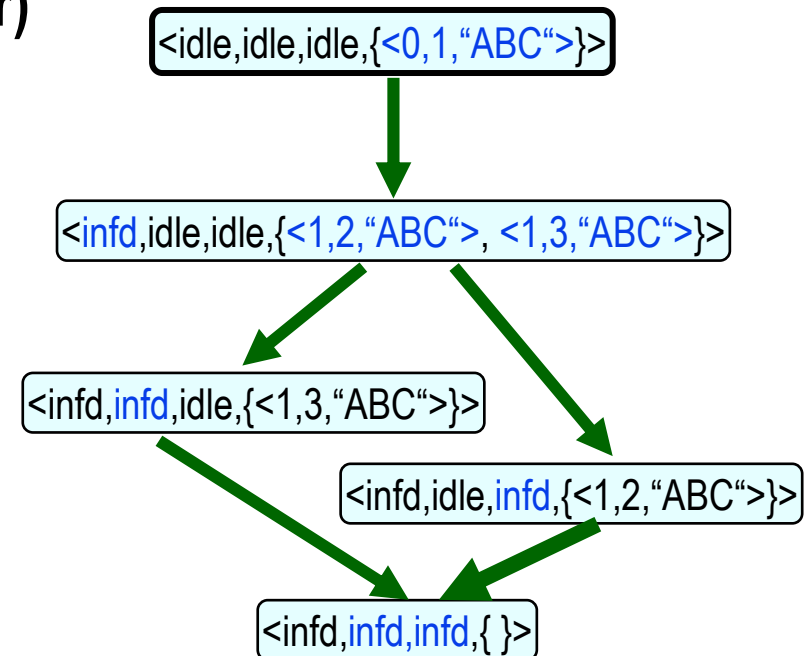
Logik	bzgl. $ \varphi $	bzgl. Modell
LTL	PSpace-Complete	P (linear)
CTL	P-Complete	P (linear)
CTL*	PSpace-Complete	P (linear)

Anmerkung

P (linear) bezieht sich auf die Anzahl der erreichbaren Systemzustände.

Sie wächst exponentiell mit der Anzahl der Komponenten.

➔ *auch Model Checking unterliegt der Zustandsexplosion*



F11: Safety- und Livenessbeweise

Basis:

**STS, definiert über Variablen und Aktionen (siehe F6),
bzgl. Liveness ferner Annahmen zu Aktionenfairness (WF, SF)**

- ◆ Safety per Invarianten, Hilfsvariablen
- ◆ Safetybeweis per Induktion über Systemablauf
- ◆ Induktive Invarianten und Invariantenverschärfung

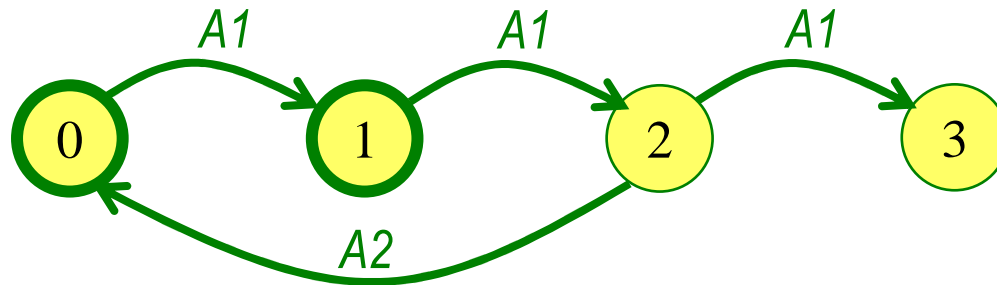
- ◆ Livenessbeweise, Regeln
 - Schon da
 - Ein WF-Schritt
 - Ein SF-Schritt
 - Transitivität
 - Lattice-Regel

Literatur

Leslie Lamport: *The temporal logic of actions (TLA)*, ACM TOPLAS, 16(3) 872-923, 1994.

F11: Safetybeweise – Safety per Invarianten (siehe F7)

- ◆ Safety-Eigenschaften eines STS können durch eine Zustands-Invariante definiert werden.
 - Unter Umständen sind vorher geeignete Hilfsvariablen einzuführen.



Beispiel

„Dem Zustand 3 geht immer Zustand 2 voraus“

$$I3 \quad V=3 \Rightarrow H=2$$

```
var V : (0, 1, 2, 3);
    H : (0, 1, 2, 3);
init (V=0  $\vee$  V=1)
     $\wedge$  H=V;
act A1: V<3  $\wedge$  V'=V+1
     $\wedge$  H'=V;
    A2: V=2  $\wedge$  V'=0
     $\wedge$  H'=V;
```

F11: Safetybeweise – Induktion über Ablauf

- ◆ gegeben: **STS**, φ
(**STS** per Variablen, Init und Aktionen $\alpha_1, \dots, \alpha_n$ definiert)
- ◆ gesucht: Beweis für „ φ ist Zustandsinvariante (es soll gelten $\Box\varphi$)“
- ◆ Verfahren: Induktion über Ablauf des Systems
 - $n=0$
Init $\Rightarrow \varphi$
 - $n \rightarrow n+1$
 $\varphi \wedge (\alpha_1 \vee \dots \vee \alpha_n) \Rightarrow \varphi'$
 - » lässt sich in n Fälle splitten, d.h. die Aktionen lassen sich einzeln betrachten
 - $\varphi \wedge \alpha_1 \Rightarrow \varphi'$
 - $\varphi \wedge \alpha_2 \Rightarrow \varphi'$
 - ...
 - $\varphi \wedge \alpha_n \Rightarrow \varphi'$

F11: Safetybeweise – Induktion über Ablauf

- ◆ gegeben: **STS**, φ
(**STS** per Variablen, Init und Aktionen $\alpha_1, \dots, \alpha_n$ definiert)
- ◆ gesucht: Beweis für „ φ ist Zustandsinvariante (es soll gelten $\square\varphi$)“
- ◆ Verfahren: Induktion über Ablauf des Systems

– $n=0$

$\varphi \Rightarrow \text{Init}$

– $n \rightarrow n+1$

$\varphi \wedge (\alpha_1 \vee \dots \vee \alpha_n) \Rightarrow \varphi'$

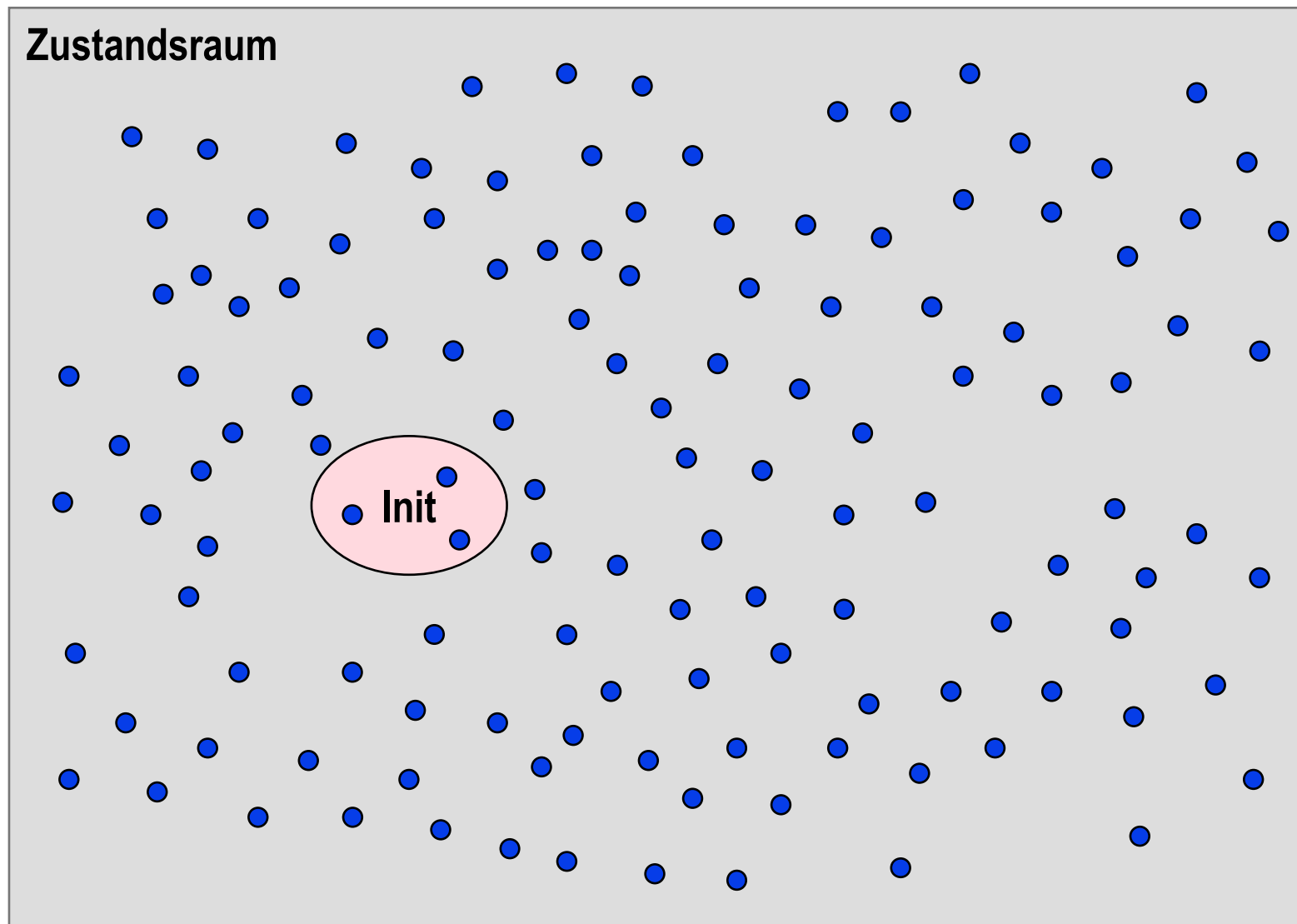
Beweis glückt, wenn die Invariante **induktiv** ist, d.h. wenn sie als Induktionsvoraussetzung stark genug ist, um sie als Induktionsfolgerung abzuleiten

Nicht-**induktive** Invarianten müssen verschärft werden

Induktionsfolgerung

Induktionsvoraussetzung

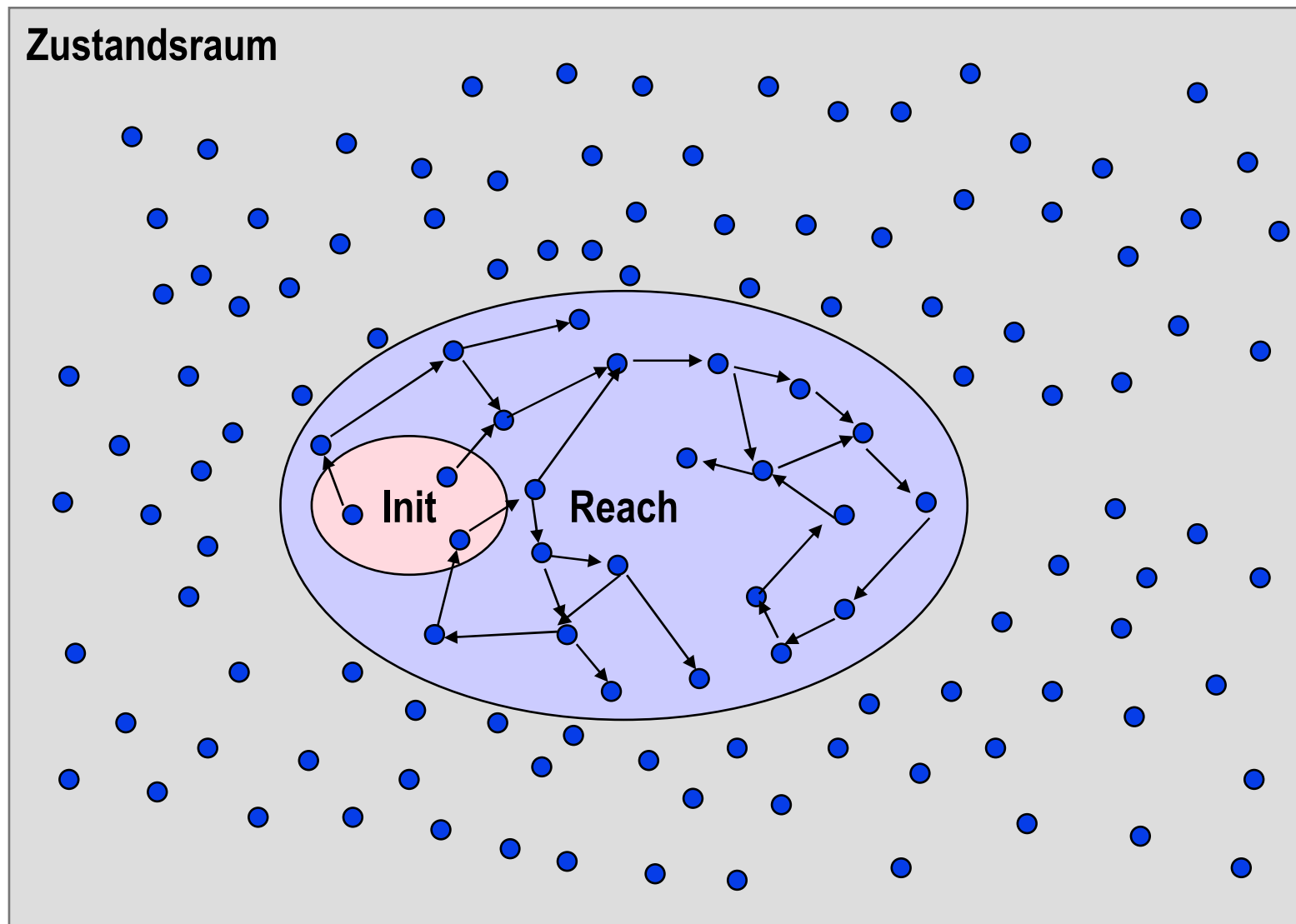
F11: Safetybeweise – Induktion über Ablauf



Zustandsraum
enthält die
kombinatorisch
möglichen
Zustände

davon sind die
Initialzustände
unbedingt
direkt erreichbar

F11: Safetybeweise – Induktion über Ablauf



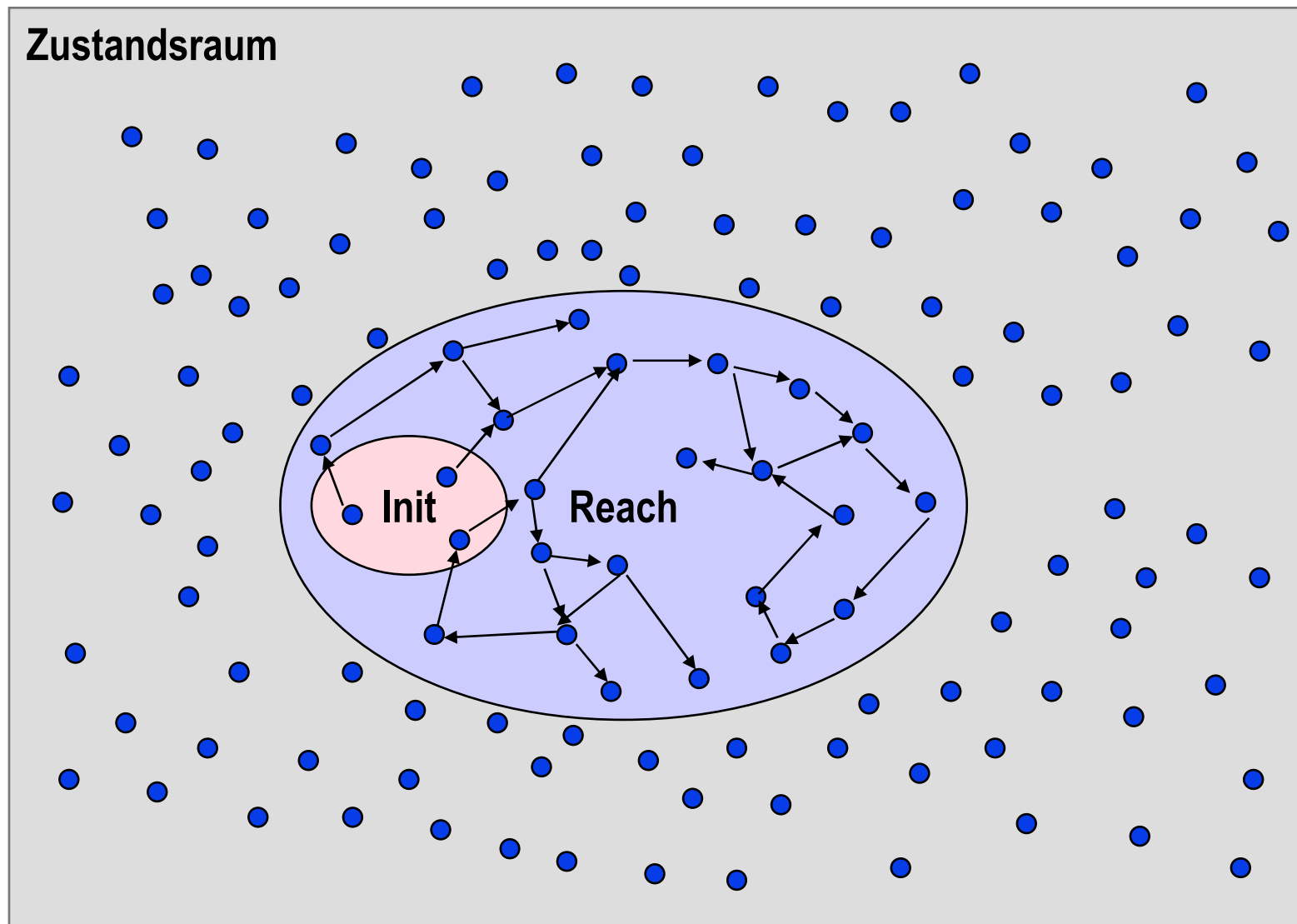
Zustandsraum
enthält die
kombinatorisch
möglichen
Zustände

davon sind die
Initialzustände
unbedingt
direkt erreichbar

davon aus die
per Transitionen
erreichbaren
etc.

→ **Reach:**
erreichbare
Zustände

F11: Safetybeweise – Induktion über Ablauf



Beachte:

Während eines beliebigen Systemablaufs gilt immer $s \in \text{Reach}$.

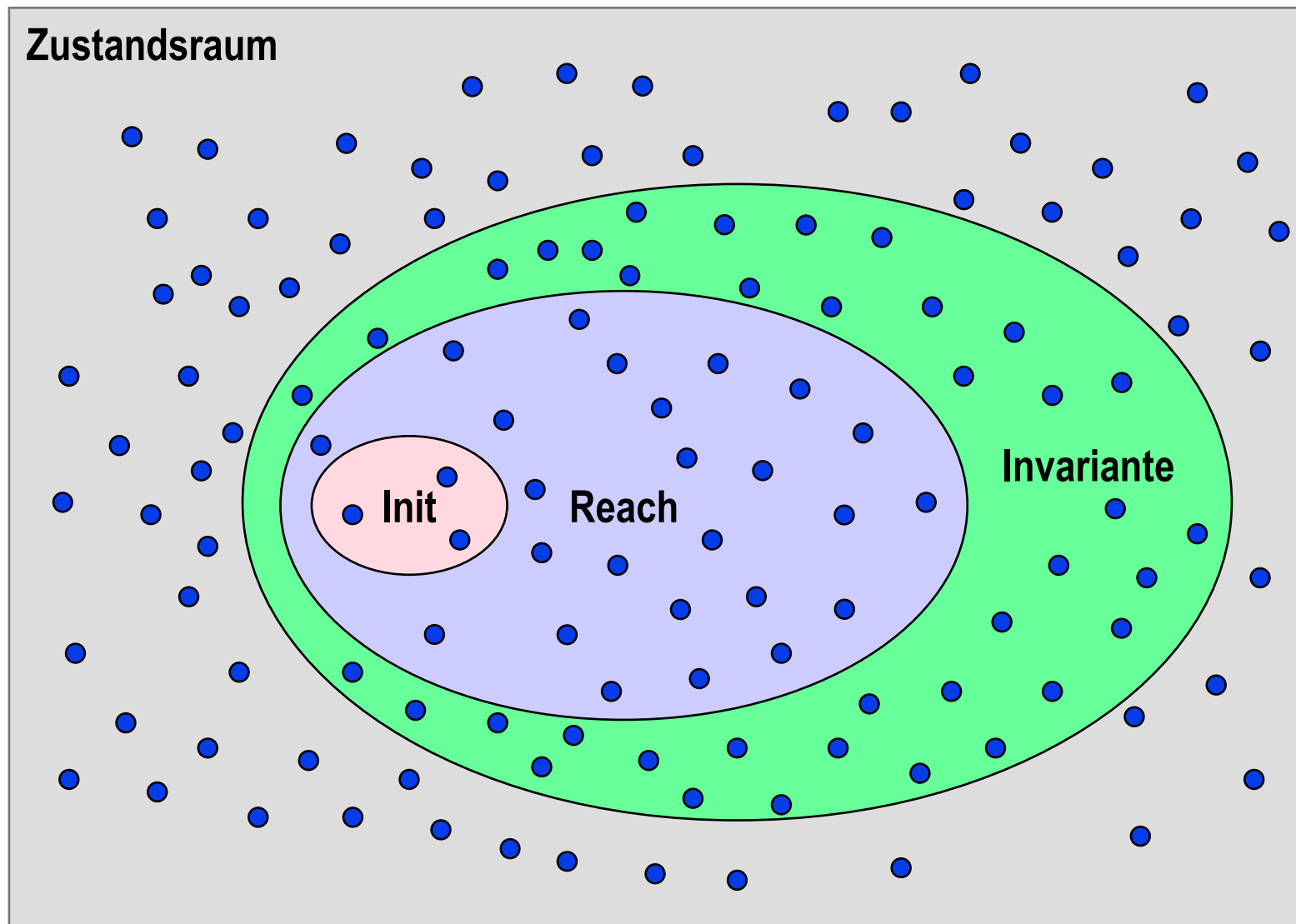
Die Bedingung $s \in \text{Reach}$ ist **Invariante**.

Beachte:

Es gibt keine Transitionen, die aus **Reach** hinausführen!

Die Bedingung $s \in \text{Reach}$ ist **induktiv**.

F11: Safetybeweise – Induktion über Ablauf

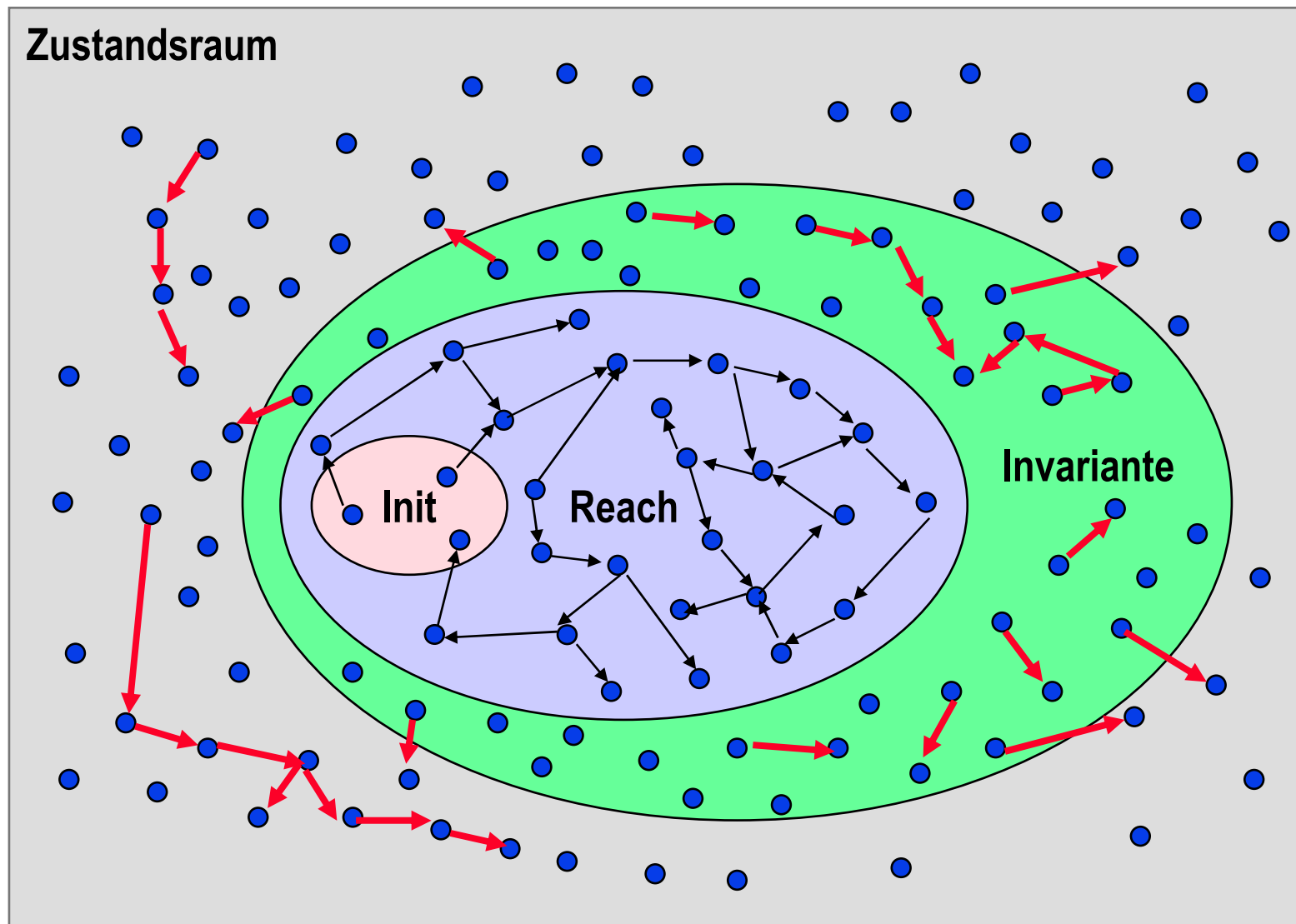


Jede Zustandsmenge, die **Reach** ganz umfasst, ist eine **Invariante**.

Jede Zustandsmenge, die **Reach** nicht ganz umfasst, ist keine Invariante.

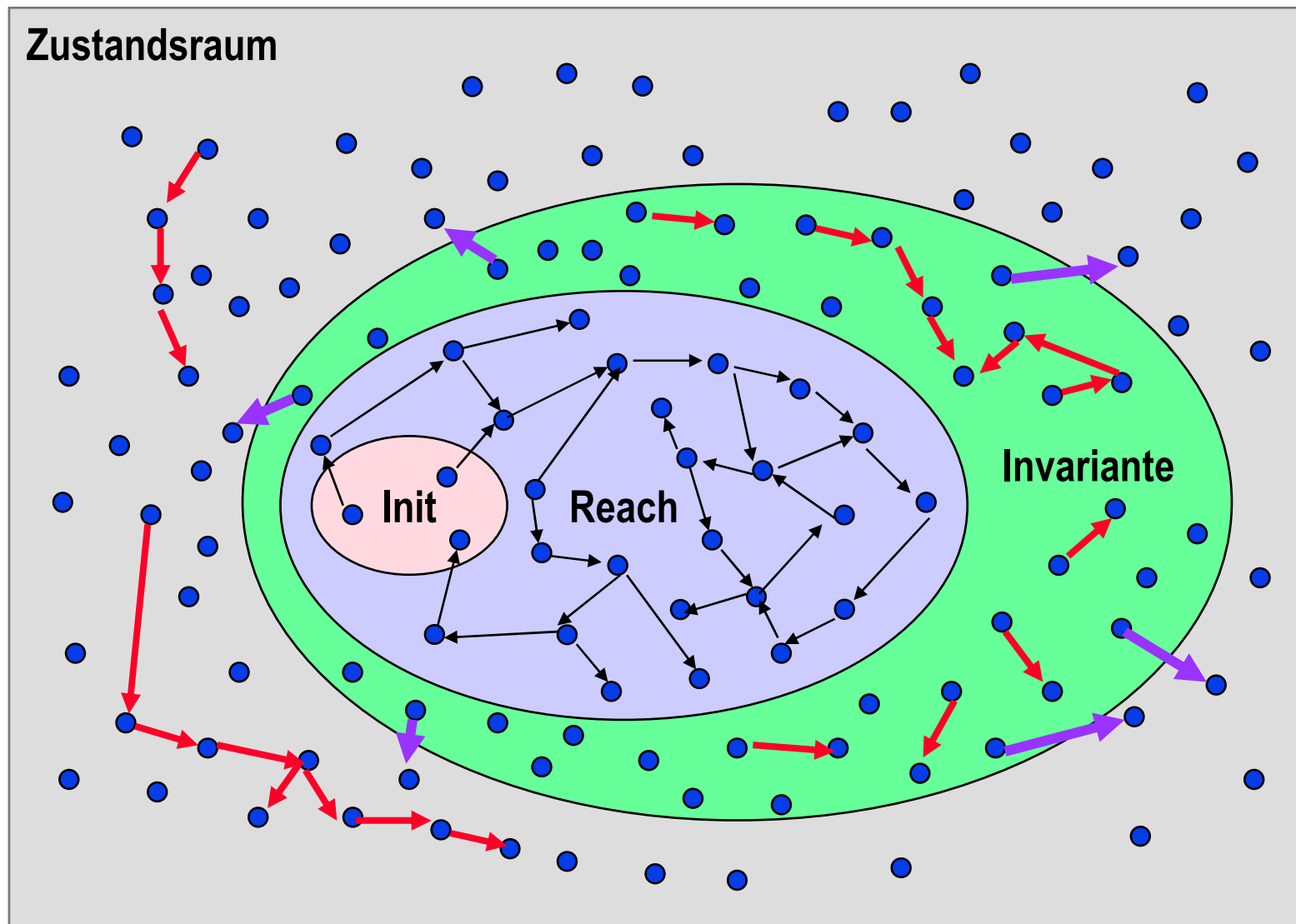
Reach ist die **schärfste Invariante** des STS.

F11: Safetybeweise – Induktion über Ablauf



Transitionen sind nicht nur über erreichbaren Zuständen definiert. Es gibt Transitionen, die nie gefeuert werden!

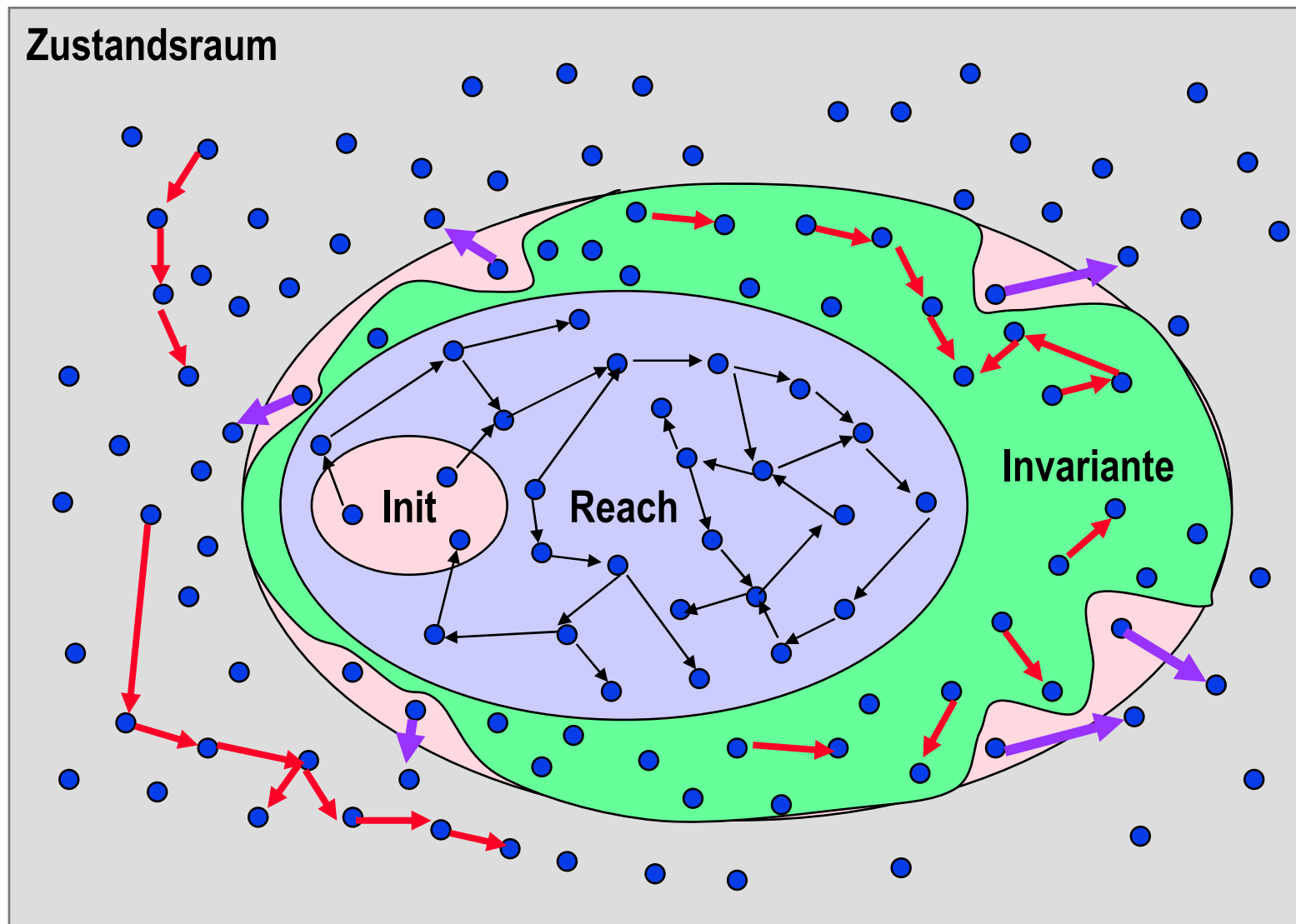
F11: Safetybeweise – Induktion über Ablauf



Transitionen sind nicht nur über erreichbaren Zuständen definiert. Es gibt Transitionen, die nie gefeuert werden!

Es kann Transitionen geben, die aus einer Invarianten herausführen!
Die Invariante ist dann nicht induktiv.

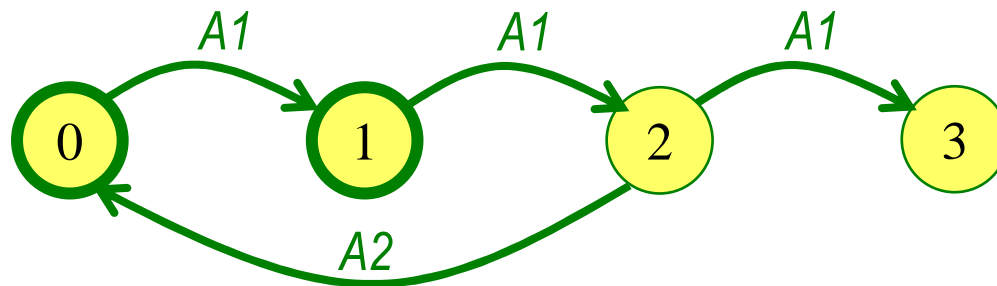
F11: Safetybeweise – Induktion über Ablauf



Eine induktive Invariante enthält keine Zustände, von welchen Transitionen nach Zuständen außerhalb der Invarianten möglich sind.

Eine nicht-induktive Invariante kann durch Verschärfung in eine induktive überführt werden.

F11: Safetybeweise – Beispiel



Beispiel
 „Dem Zustand 3 geht immer Zustand 2 voraus“
 $I_3 \quad V=3 \Rightarrow H=2$

```

var  V : (0, 1, 2, 3) ;
     H : (0, 1, 2, 3) ;
init  (V=0 ∨ V=1)
      ∧ H=V ;
act   A1: V<3 ∧ V'=V+1
      ∧ H'=V ;
      A2: V=2 ∧ V'=0
      ∧ H'=V ;
  
```

- ◆ $n=0: \textit{init} \Rightarrow I_3$
 $((V=0 \vee V=1) \wedge H=V) \Rightarrow (V=3 \Rightarrow H=2)$
- ◆ $n \rightarrow n+1: I_3 \wedge A1 \Rightarrow I_3'$
 $(V=3 \Rightarrow H=2) \wedge (V<3 \wedge V'=V+1 \wedge H'=V) \Rightarrow (V'=3 \Rightarrow H'=2)$
- ◆ $n \rightarrow n+1: I_3 \wedge A2 \Rightarrow I_3'$
 $(V=3 \Rightarrow H=2) \wedge (V=2 \wedge V'=0 \wedge H'=V) \Rightarrow (V'=3 \Rightarrow H'=2)$

F11: Safetybeweise – Beispiel

1. $n=0: \mathit{init} \Rightarrow I3$
 $(V=0 \vee V=1) \wedge H=V \Rightarrow (V=3 \Rightarrow H=2)$
2. $n \rightarrow n+1: I3 \wedge A1 \Rightarrow I3'$
 $(V=3 \Rightarrow H=2) \wedge (V < 3 \wedge V'=V+1 \wedge H'=V) \Rightarrow (V'=3 \Rightarrow H'=2)$
3. $n \rightarrow n+1: I3 \wedge A2 \Rightarrow I3'$
 $(V=3 \Rightarrow H=2) \wedge (V=2 \wedge V'=0 \wedge H'=V) \Rightarrow (V'=3 \Rightarrow H'=2)$

Die Invariante $I3$ ist induktiv.

Die Schritte 2 und 3 sind erfolgreich.

$I3$ umfasst alle Zustände, bei welchen $V \neq 3$ ist, sowie alle, bei welchen $V=3$ und $H=2$ ist:

$(0,0), (0,1), (0,2), (0,3),$
 $(1,0), (1,1), (1,2), (1,3),$
 $(2,0), (2,1), (2,2), (2,3),$
 $(3,0), (3,1), (3,2), (3,3)$

Es gibt keine Transitionen, welche in einem Zustand $(3,*)$ schalten können.

F11: Safetybeweise – Beispiel „Broadcast“

- Bei Terminierung wurden alle Stationen informiert.
 - $nts = \{\} \Rightarrow \forall i: cs[i] = \text{infd}$
- Es werden höchstens $2e$ Nachrichten ausgetauscht.
 - **AnzSend** $< 2e$
(Hilfsvariable AnzSend geeignet eingeführt)
- Wenn eine Station im Zustand *infd* ist, verlässt sie ihn nie wieder.
 - **H=0**
(Hilfsvariable *H* so eingeführt, dass sie nie von 1 auf 0 kippt und immer dann auf 1 kippt, wenn eine Station von *infd* auf *idle* wechselt)

Variablen

cs: array [1..n] of (idle, infd) ! Stationen

nts: bag of $\langle \text{from}, \text{to}, \text{msg} \rangle$! Transportsystem

Init

cs = $\langle \text{idle}, \text{idle}, \dots, \text{idle} \rangle \wedge$

nts = $\{\langle 0, \text{Initiator}, \text{Text} \rangle\}$

Aktionen

Forward

$\exists i, m: cs[i] = \text{idle} \wedge m \in nts \wedge m.\text{to} = i \wedge$
 $cs[i]' = \text{infd} \wedge \forall j \neq i: cs[j]' = cs[j] \wedge$
 $nts' = nts \cup$

$\{ \langle i, k, m.\text{msg} \rangle : \text{istNachbar}(k, i) \wedge k \neq m.\text{from} \}$
 $\setminus \{m\}$

Skip

$\exists i, m: cs[i] = \text{infd} \wedge m \in nts \wedge m.\text{to} = i \wedge$
 $\forall j: cs[j]' = cs[j] \wedge nts' = nts \setminus \{m\}$

F11: Safetybeweise – Beispiel „Broadcast“

Bei Terminierung wurden
alle Stationen informiert.

$nts = \{\} \Rightarrow \forall i: cs[i] = \text{infd}$

$n = 0$

$\text{Init} \Rightarrow (nts = \{\} \Rightarrow \forall i: cs[i] = \text{infd})$

$n \rightarrow n+1$: Forward

$((nts = \{\} \Rightarrow \forall i: cs[i] = \text{infd})$

\wedge

Forward) \Rightarrow

$(nts' = \{\} \Rightarrow \forall i: cs'[i] = \text{infd})$

$n \rightarrow n+1$: Skip

$((nts = \{\} \Rightarrow \forall i: cs[i] = \text{infd})$

\wedge

Skip) \Rightarrow

$(nts' = \{\} \Rightarrow \forall i: cs'[i] = \text{infd})$

Bei Forward und Skip lässt sich
die Implikation nicht beweisen:

Invariante verschärfen!

Variablen

cs : array [1..n] of (idle, infd) ! Stationen

nts : bag of < from, to, msg > ! Transportsystem

Init

$cs = \langle \text{idle}, \text{idle}, \dots, \text{idle} \rangle \wedge$

$nts = \langle 0, \text{Initiator}, \text{Text} \rangle$

Aktionen

Forward

$\exists i, m: cs[i] = \text{idle} \wedge m \in nts \wedge m.to = i \wedge$

$cs[i]' = \text{infd} \wedge \forall j \neq i: cs[j]' = cs[j] \wedge$

$nts' = nts \cup$

$\{ \langle i, k, m.msg \rangle : \text{istNachbar}(k, i) \wedge k \neq m.from \}$

$\setminus \{m\}$

Skip

$\exists i, m: cs[i] = \text{infd} \wedge m \in nts \wedge m.to = i \wedge$

$\forall j: cs[j]' = cs[j] \wedge nts' = nts \setminus \{m\}$

F11: Safetybeweise – Beispiel „Broadcast“

Bei Terminierung wurden alle Stationen informiert:

$$\mathit{nts}=\{\} \Rightarrow \forall i: \mathit{cs}[i]=\mathit{infd}$$

Graphentheorie, Graph G zusammenhängend
Wenn es in G sowohl Knoten mit Eigenschaft e als auch welche mit Eigenschaft $\neg e$ gibt, gibt es auch zwei entsprechende Nachbarn.

Idee zum Algorithmus

Eine informierte Station sendet an jeden nicht-informierten Nachbarn eine Nachricht. Wenn er sie empfangen hat, ist er auch informiert. Wenn nicht, ist das nts nicht leer.

Verschärfte Invariante

$$\mathit{nts}=\{\} \Rightarrow \forall i: \mathit{cs}[i]=\mathit{infd}$$

\wedge

$$\forall k,l: \mathit{istNachbar}(k,l) \wedge \mathit{cs}[k]=\mathit{infd} \wedge \mathit{cs}[l] \neq \mathit{infd}$$

\Rightarrow

$$\exists m: \langle k, l, m \rangle \in \mathit{nts}$$

Variablen

cs : array [1..n] of (idle, infd) ! *Stationen*

nts : bag of < from, to, msg > ! *Transportsystem*

Init

$$\mathit{cs} = \langle \mathit{idle}, \mathit{idle}, \dots, \mathit{idle} \rangle \wedge \\ \mathit{nts} = \{ \langle 0, \mathit{Initiator}, \mathit{Text} \rangle \}$$

Aktionen

Forward

$$\exists i, m: \mathit{cs}[i]=\mathit{idle} \wedge m \in \mathit{nts} \wedge m.\mathit{to}=i \wedge \\ \mathit{cs}[i]'=\mathit{infd} \wedge \forall j \neq i: \mathit{cs}[j]'=\mathit{cs}[j] \wedge \\ \mathit{nts}'= \mathit{nts} \cup \\ \{ \langle i, k, m.\mathit{msg} \rangle: \mathit{istNachbar}(k,i) \wedge \\ k \neq m.\mathit{from} \} \\ \setminus \{m\}$$

Skip

$$\exists i, m: \mathit{cs}[i]=\mathit{infd} \wedge m \in \mathit{nts} \wedge m.\mathit{to}=i \wedge \\ \forall j: \mathit{cs}[j]'=\mathit{cs}[j] \wedge \mathit{nts}'=\mathit{nts} \setminus \{m\}$$

F11: Safetybeweise – Beispiel „Broadcast“

Invar **Is**:

$\text{nts}=\{\} \Rightarrow \forall i: \text{cs}[i]=\text{infid} \wedge$

$\forall k,l: \text{istNachbar}(k,l) \wedge \text{cs}[k]=\text{infid} \wedge$

$\text{cs}[l] \neq \text{infid} \Rightarrow \exists m: \langle k, l, m \rangle \in \text{nts}$

$n=0$

Init \Rightarrow **Is**

! nts ist nicht leer, alle Stationen sind idle

$n \rightarrow n+1$: Forward

Is \wedge Forward \Rightarrow **Is'**

! i sendet an jeden nichtinformierten Nachbarn. Falls i nicht sendet und i nicht letzter uninformierter war, gibt es zwei andere an der Grenze zwischen infd und idle, so dass dort mind. eine Nachricht unterwegs ist.

$n \rightarrow n+1$: Skip

Is \wedge Skip \Rightarrow **Is'**

! Wenn m letztes Element in nts war, war i letzte idle Station.

Variablen

cs: array [1..n] of (idle, infd) ! Stationen

nts: bag of < from, to, msg > ! Transportsystem

Init

cs = < idle, idle, ..., idle > \wedge

nts = {<0, Initiator, Text>}

Aktionen

Forward

$\exists i, m: \text{cs}[i]=\text{idle} \wedge m \in \text{nts} \wedge m.\text{to}=i \wedge$

$\text{cs}[i]'=\text{infid} \wedge \forall j \neq i: \text{cs}[j]'=\text{cs}[j] \wedge$

$\text{nts}'= \text{nts} \cup$

$\{ \langle i, k, m.\text{msg} \rangle: \text{istNachbar}(k,i) \wedge$
 $k \neq m.\text{from} \}$

$\setminus \{m\}$

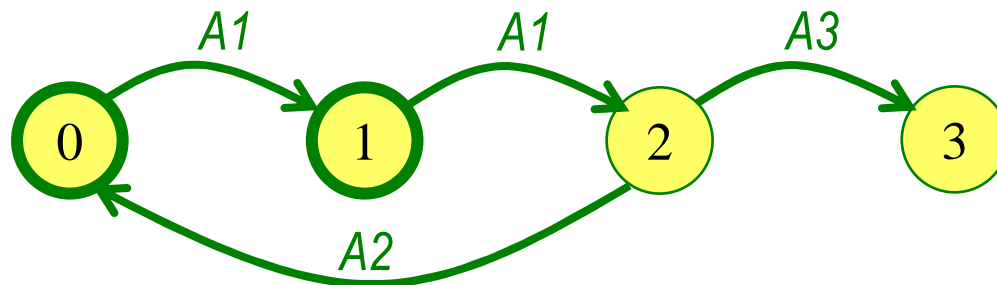
Skip

$\exists i, m: \text{cs}[i]=\text{infid} \wedge m \in \text{nts} \wedge m.\text{to}=i \wedge$

$\forall j: \text{cs}[j]'=\text{cs}[j] \wedge \text{nts}'=\text{nts} \setminus \{m\}$

F11: Livenessbeweise – Leadsto aus Fairnessannahmen

- ◆ Im Rahmen eines STS können Liveness-Voraussetzungen per **Aktionenfairness-Annahmen** definiert werden.
- ◆ Zu beweisende Liveness-Eigenschaften werden durch Formeln der Form **$P \rightsquigarrow Q$** notiert.
- ◆ Mögliche Abläufe des Systems sind in der Safetyeigenschaft des Systems enthalten.
- ◆ Beweis der Liveness vor diesem Hintergrund mithilfe von Regeln



Beispiel

„Das System terminiert in 3“

L $init \rightsquigarrow V=3$

```
var V : (0, 1, 2, 3) ;
init (V=0  $\vee$  V=1) ;
act A1: V<2  $\wedge$  V'=V+1;
    A2: V=2  $\wedge$  V'=0;
    A3: V=2  $\wedge$  V'=3;
WF(A1), SF(A3)
```

F11: Safetybeweise – Beispiel „Broadcast“

Es werden höchstens $2e$ Nachrichten ausgetauscht.

AnzSend $\leq 2e$

(Hilfsvariable *AnzSend* geeignet eingeführt)

n=0

Init \Rightarrow **AnzSend** $\leq 2e$

n \rightarrow **n+1**: **Forward**

$((\text{AnzSend} \leq 2e) \wedge \text{Forward}) \Rightarrow$
 $(\text{AnzSend}' \leq 2e)$

n \rightarrow **n+1**: **Skip**

$((\text{AnzSend} \leq 2e) \wedge \text{Skip}) \Rightarrow$
 $(\text{AnzSend}' \leq 2e)$

Bei Forward lässt sich die Implikation nicht beweisen:

Invariante verschärfen!

- Station *i* wird frisch informiert
- Es kommen $\# \text{Nachbar}(i) - 1$ Nachrichten dazu
 \rightarrow Invariante sollte auf Informierte und deren Nachbar-Zahlen eingehen !

Variablen

cs: array [1..n] of (idle, infd) ! Stationen

nts: bag of < from, to, msg > ! Transportsystem

AnzSend : integer

Init

cs = < idle, idle, ..., idle > \wedge

nts = {<0, Initiator, Text>} \wedge **AnzSend**=0

Aktionen

Forward

$\exists i, m$: **cs**[i]=idle $\wedge m \in \text{nts}$ $\wedge m.\text{to}=i$ \wedge
cs[i]'=infd $\wedge \forall j \neq i$: **cs**[j]'=**cs**[j] \wedge
nts'=

nts \cup { < i, k, m.msg > : istNachbar(k,i) \wedge
k \neq m.from } \ { m }

\wedge **AnzSend**'=**AnzSend** + #Nachbar(i)-1

Skip

$\exists i, m$: **cs**[i]=infd $\wedge m \in \text{nts}$ $\wedge m.\text{to}=i$
 \wedge **AnzSend**'=**AnzSend**

$\wedge \forall j$: **cs**[j]'=**cs**[j] \wedge **nts**'=**nts** \ { m }

F11: Safetybeweise – Beispiel „Broadcast“

Graphentheorie

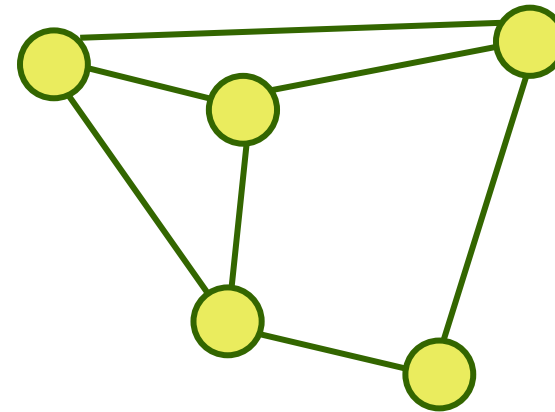
Kante verbindet je 2 Knoten:

$$\sum_i: \#\text{Nachbar}(i) = 2e$$

Verschärfungsidee

Wir achten auf die informierten Stationen.

$$\text{AnzSend} \leq \sum_{i, \text{cs}[i]=\text{inf}}: \#\text{Nachbar}(i)$$



Verschärfte Invariante wäre

$$\text{AnzSend} \leq 2e \wedge \text{AnzSend} \leq \sum_{i, \text{cs}[i]=\text{inf}}: \#\text{Nachbar}(i)$$

Da es nie mehr informierte Knoten als Knoten überhaupt geben kann, gilt:

$$\sum_{i, \text{cs}[i]=\text{inf}}: \#\text{Nachbar}(i) \leq 2e$$

Deshalb wählen wir statt einer Verschärfung per „ \wedge “ einfach die schärfere Invariante:

$$\text{AnzSend} \leq \sum_{i, \text{cs}[i]=\text{inf}}: \#\text{Nachbar}(i)$$

F11: Safetybeweise – Beispiel „Broadcast“

Es wurden jeweils höchstens so viele Nachrichten gesendet, wie die informierten Stationen Nachbarn haben:

$$\text{AnzSend} \leq \sum_{i, cs[i]=\text{infd}}: \#\text{Nachbar}(i)$$

n=0

$$\text{Init} \Rightarrow \text{AnzSend} \leq \sum_{i, cs[i]=\text{infd}}: \#\text{Nachbar}(i)$$

n → n+1: Forward

$$((\text{AnzSend} \leq \sum_{i, cs[i]=\text{infd}}: \#\text{Nachbar}(i))$$

\wedge **Forward**)

\Rightarrow

$$(\text{AnzSend}' \leq \sum_{i, cs[i]'\text{=infd}}: \#\text{Nachbar}(i))$$

n → n+1: Skip

$$((\text{AnzSend} \leq \sum_{i, cs[i]=\text{infd}}: \#\text{Nachbar}(i))$$

\wedge **Skip**)

\Rightarrow

$$(\text{AnzSend}' \leq \sum_{i, cs[i]'\text{=infd}}: \#\text{Nachbar}(i))$$

Variablen

cs: array [1..n] of (idle, infd) ! Stationen

nts: bag of < from, to, msg > ! Transportsystem

AnzSend : integer

Init

cs = < idle, idle, ..., idle > \wedge

nts = {<0, Initiator, Text>} \wedge **AnzSend**=0

Aktionen

Forward

$\exists i, m$: **cs**[i]=idle \wedge $m \in$ nts \wedge m.to=i \wedge
cs[i]'=infd \wedge $\forall j \neq i$: **cs**[j]'=**cs**[j] \wedge
nts' =

nts \cup { <i, k, m.msg>: istNachbar(k,i) \wedge
k \neq m.from } \ {m}

\wedge **AnzSend**'=**AnzSend** + #Nachbar(i)-1

Skip

$\exists i, m$: **cs**[i]=infd \wedge $m \in$ nts \wedge m.to=i

\wedge **AnzSend**'=**AnzSend**

\wedge $\forall j$: **cs**[j]'=**cs**[j] \wedge nts'=nts \ {m}

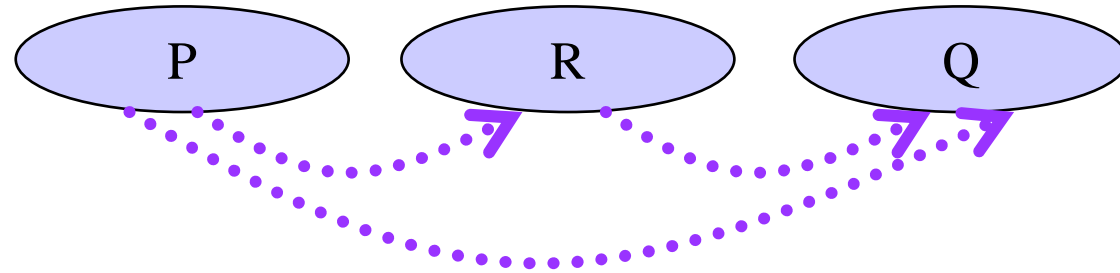
F11: Livenessbeweise – Regeln

1. Transitivität von $\sim>$

$$P \sim> R$$

$$\frac{R \sim> Q}{P \sim> Q}$$

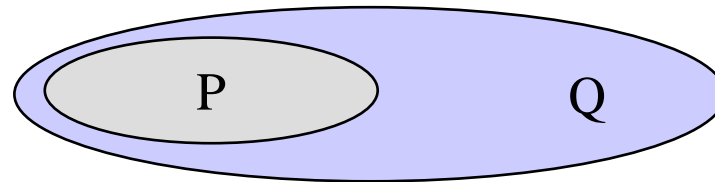
$$P \sim> Q$$



2. Trivialfall „Schon da“

$$\frac{P \Rightarrow Q}{P \sim> Q}$$

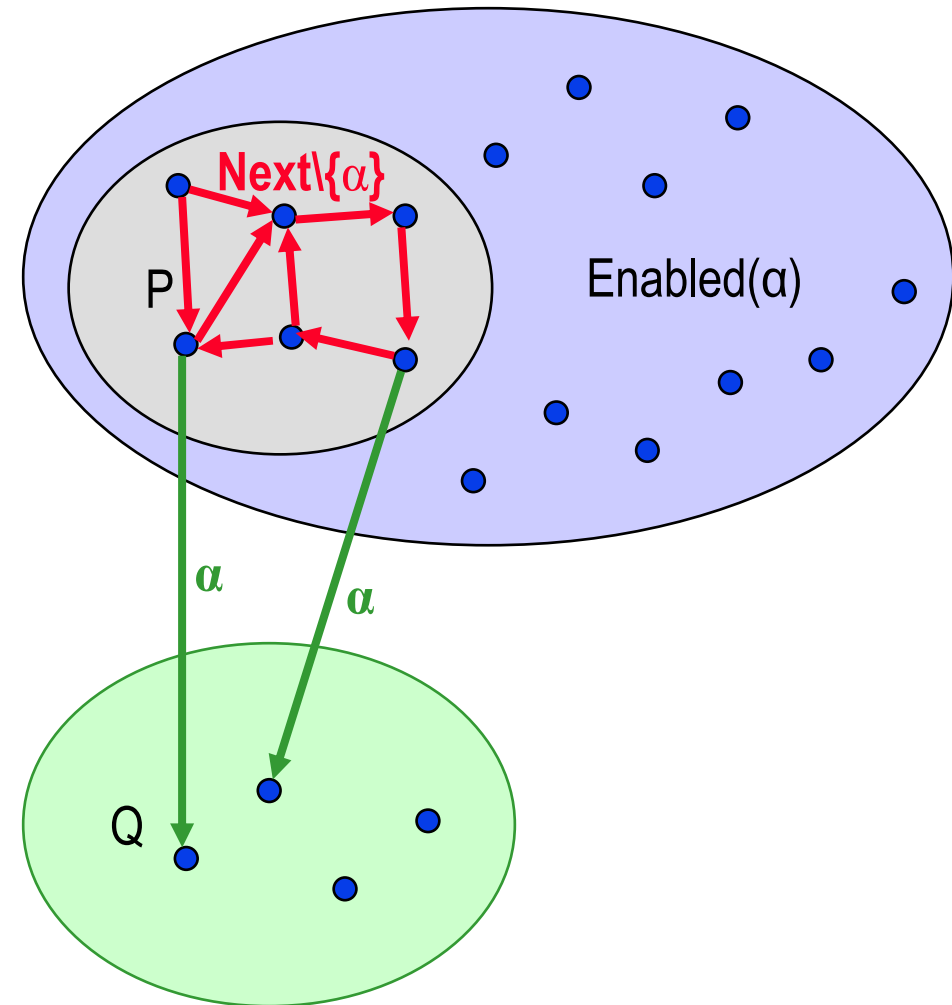
$$P \sim> Q$$



F11: Livenessbeweise – Regeln

3.1 Ein Weak Fair Schritt

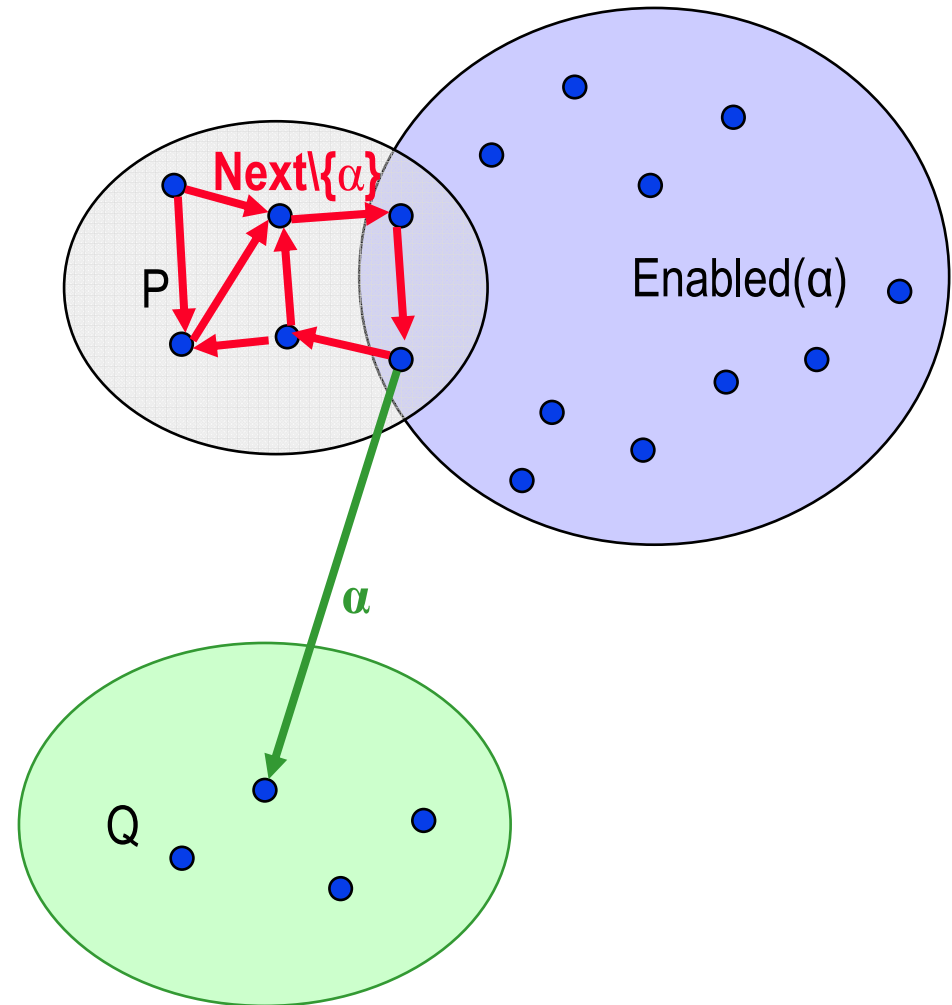
- (1) $WF(\alpha)$
- (2) $P \Rightarrow \text{Enabled}(\langle \alpha \rangle)$
- (3) $P \wedge \text{Next} \wedge \neg \langle \alpha \rangle \Rightarrow P'$
- (4) $\frac{P \wedge \langle \alpha \rangle \Rightarrow Q'}{P \rightsquigarrow Q}$
- (c) $P \rightsquigarrow Q$



F11: Livenessbeweise – Regeln

3.2 Ein Strong Fair Schritt

- (1) $SF(\alpha)$
- (2) $P \rightsquigarrow Enabled(\langle \alpha \rangle)$
- (3) $P \wedge Next \wedge \neg \langle \alpha \rangle \Rightarrow P'$
- (4) $\frac{P \wedge \langle \alpha \rangle \Rightarrow Q'}{P \rightsquigarrow Q}$
- (c) $P \rightsquigarrow Q$



F11: Livenessbeweise – Regeln

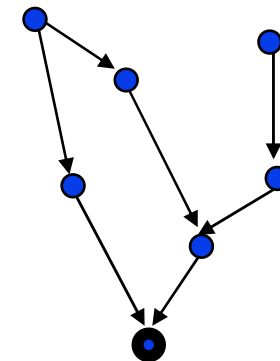
4. Lattice

Man behandelt mit dieser Regel eine Familie von transitiven Leads-To-Schritt-Ketten, die alle in Q münden.

Die Kettenfamilie entspricht einem wohlfundierten Verband $V ::= \langle M, \angle \rangle$, der durch eine Wertemenge M und eine Halbordnung \angle gegeben ist.

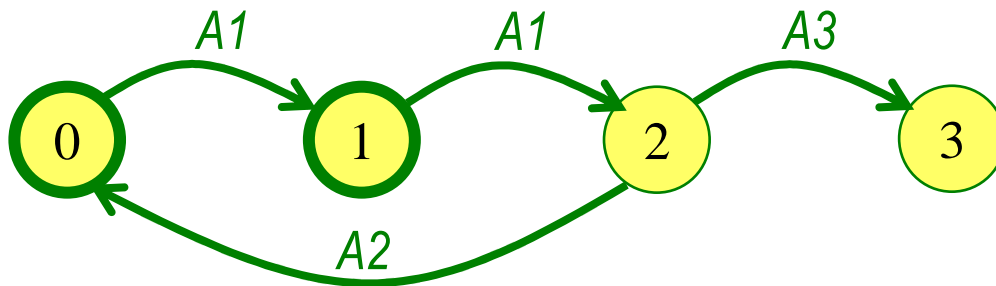
V muss ein Minimum m haben.

Für jeden Wert aus M gibt es nur endlich viele entsprechend \angle kleinere Werte.



1. Wir definieren einen geeigneten Verband V .
2. Wir definieren eine Zustandsabbildung sf , d.h. geben einen Ausdruck $expsf$ über den Zustandsvariablen des Systems an, der zu jedem Zustand einen Wert aus M liefert.
3. Wir beweisen, dass jeder Zustand, der auf das Minimum m abgebildet wird, Q impliziert.
4. Wir beweisen, dass die einzelnen Schritte lebendig sind und im Verband weiterführen:
Für jeden erreichbaren Zustand soll gelten, dass davon ausgehend schließlich ein Zustand erreicht wird, dessen sf -Bild gemäß \angle kleiner ist:
 $expsf = x \rightsquigarrow expsf \angle x$.

F11: Livenessbeweise – Beispiel zu Ein-Schritt-Regeln



Beispiel

„Das System terminiert in 3“

L $init \rightsquigarrow V=3$

1WF: $V=0 \rightsquigarrow V=1$

1WF: $V=1 \rightsquigarrow V=2$

Trans: $V=0 \rightsquigarrow V=2$

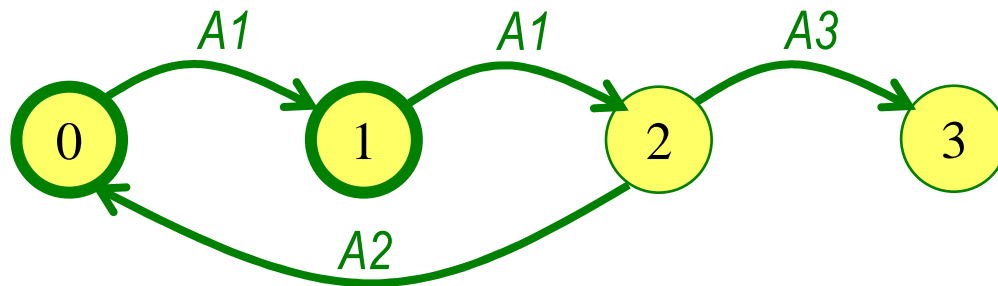
```

var V : (0, 1, 2, 3);
init (V=0 ∨ V=1);
act A1: V<2 ∧ V'=V+1;
    A2: V=2 ∧ V'=0;
    A3: V=2 ∧ V'=3;
WF(A1), WF(A3)
  
```

3.1 Ein Weak Fair Schritt

- (1) $WF(\alpha)$
- (2) $P \Rightarrow Enabled(\langle \alpha \rangle)$
- (3) $P \wedge Next \wedge \neg \langle \alpha \rangle \Rightarrow P'$
- (4) $\frac{P \wedge \langle \alpha \rangle \Rightarrow Q'}{P \rightsquigarrow Q}$
- (c) $P \rightsquigarrow Q$

F11: Livenessbeweise – Beispiel zu Ein-Schritt-Regeln



Beispiel

„Das System terminiert in 3“

L $init \rightsquigarrow V=3$

1WF: $V=0 \rightsquigarrow V=1$

1WF: $V=1 \rightsquigarrow V=2$

Trans: $V=0 \rightsquigarrow V=2$

1SF: $V=2 \rightsquigarrow V=3$

Trans: $V=0 \rightsquigarrow V=3$

```

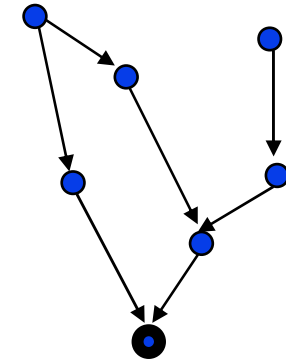
var V : (0, 1, 2, 3);
init (V=0 ∨ V=1);
act A1: V<2 ∧ V'=V+1;
    A2: V=2 ∧ V'=0;
    A3: V=2 ∧ V'=3;
WF(A1), WF(A3)
  
```

3.2 Ein Strong Fair Schritt

- (1) $SF(\alpha)$
- (2) $P \rightsquigarrow Enabled(\langle \alpha \rangle)$
- (3) $P \wedge Next \wedge \neg \langle \alpha \rangle \Rightarrow P'$
- (4) $\frac{P \wedge \langle \alpha \rangle \Rightarrow Q'}{P \rightsquigarrow Q}$
- (c) $P \rightsquigarrow Q$

F11: Livenessbeweise – Beispiel zu Lattice

- ◆ Formalisierung der Eigenschaft als eine Bedingung über den Systemzuständen, welche nach endlich vielen Schritten gelten muss
 - Def.: $\text{Term} \equiv \text{cs}[\text{Initiator}] = \text{infid} \wedge \text{nts} = \{\}$
 - geforderte Liveness-Systemeigenschaft:
Term gilt nach endlich vielen Schritten
- ◆ Beweis durch Konstruktion einer Zustandsabbildung mit „Lattice“-Bildmenge, der Annahme, dass mögliche Systemschritte nicht unendlich lange verzögert werden, dem Nachweis, dass jeder Systemschritt dazu führt, dass das *Lattice*-Bild des aktuellen Zustands sich dem Minimum weiter nähert, und dem Nachweis, dass die Eigenschaft gilt, wenn das Minimum erreicht ist
 - Def.: $\text{lat} \equiv \langle \text{Anzahl Stationen im Zustand idle, Anzahl Nachrichten im nts} \rangle$
 - Lattice: $\text{Nat} \times \text{Nat}$ mit Ordnung $\langle x, y \rangle < \langle x', y' \rangle \Leftrightarrow x < x' \vee (x = x' \wedge y < y')$
 - **$\text{lat} = \langle x, y \rangle \wedge \text{Systemschritt erfolgt} \Rightarrow \text{lat}(\text{im Folgezustand}) < \langle x, y \rangle$!**
 - **$\text{lat} = \langle 0, 0 \rangle \Rightarrow \text{Term}$!**



F11: Livenessbeweise – Beispiel zu Lattice

Term \equiv $cs[\text{Initiator}] = \text{infd} \wedge \text{nts} = \{\}$

Def.: $\text{lat} \equiv$ \langle Anzahl Stationen im Zustand *idle*,
Anzahl Nachrichten im *nts* \rangle

Lattice: $\text{Nat} \times \text{Nat}$ mit Ordnung \langle

$\langle x, y \rangle < \langle x', y' \rangle$

\Leftrightarrow

$x < x' \vee (x = x' \wedge y < y')$

1. $\text{lat} = \langle x, y \rangle \wedge$ **Systemschritt erfolgt**

\Rightarrow

$\text{lat}(\text{im Folgezustand}) < \langle x, y \rangle$

! Forward informiert neue Station

! Skip nimmt Nachr. aus nts

2. $\text{lat} = \langle 0, 0 \rangle \Rightarrow$ **Term**

*! nach Def. von lat sind dann
alle Stationen informiert und
das nts ist leer*

Variablen

cs : array $[1..n]$ of (idle, infd) ! *Stationen*

nts : bag of \langle from, to, msg \rangle ! *Transportsystem*

Init

$cs = \langle$ idle, idle, ..., idle $\rangle \wedge$

$nts = \{ \langle 0, \text{Initiator}, \text{Text} \rangle \}$

Aktionen

Forward

$\exists i, m:$ $cs[i] = \text{idle} \wedge m \in nts \wedge m.\text{to} = i \wedge$
 $cs[i]' = \text{infd} \wedge \forall j \neq i: cs[j]' = cs[j] \wedge$
 $nts' = nts \cup$
 $\{ \langle i, k, m.\text{msg} \rangle : \text{istNachbar}(k, i) \wedge$
 $k \neq m.\text{from} \}$
 $\setminus \{m\}$

Skip

$\exists i, m:$ $cs[i] = \text{infd} \wedge m \in nts \wedge m.\text{to} = i \wedge$
 $\forall j: cs[j]' = cs[j] \wedge nts' = nts \setminus \{m\}$