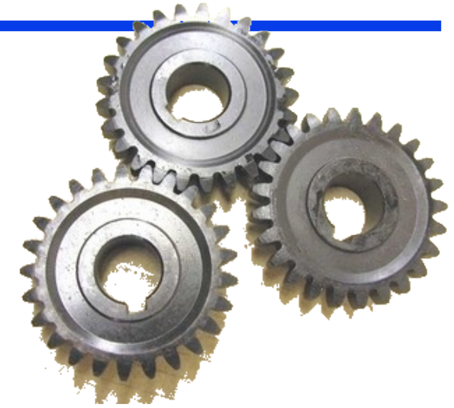
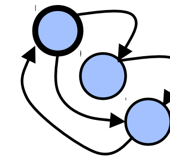
Modellierung und Analyse eingebetteter und verteilter Systeme

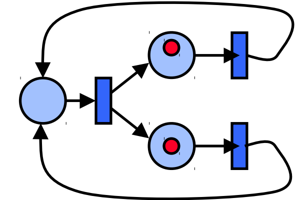


—

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) | X

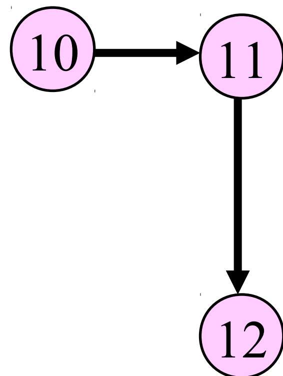
E((EX.P)U(AG.Q))

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



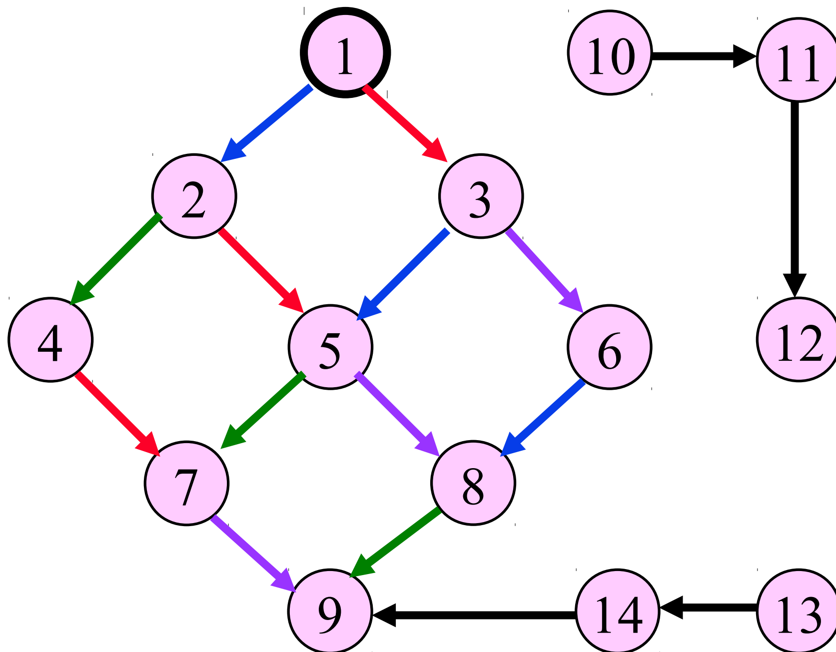
<u>Literatur</u>
Leslie Lamport: The temporal logic of actions (TLA). ACM TOPLAS, 16(3) 872-923, May 1994.

F6: Definition STS

Zustandstransitionssystem (State Transition System STS)

STS = < S, S_0, Next >

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



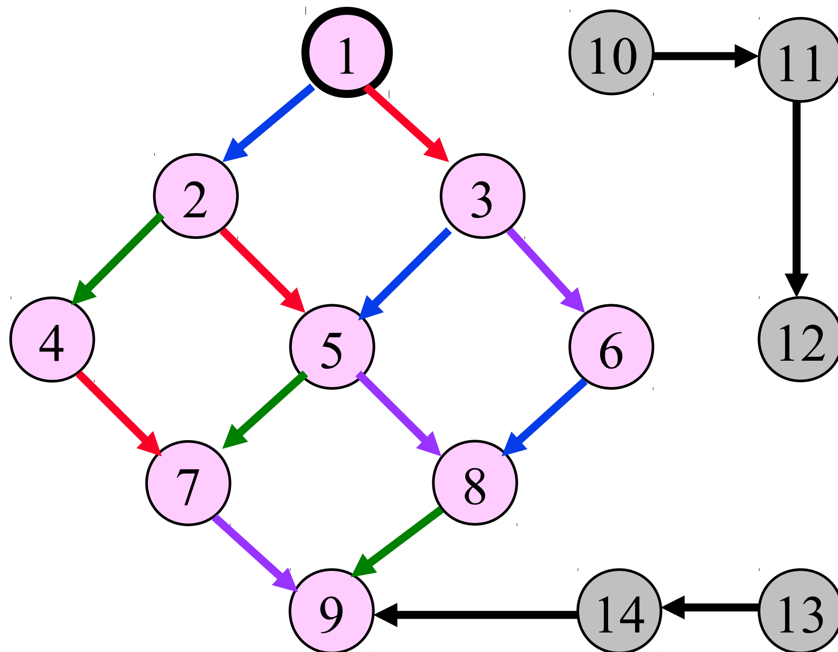
S = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
S_0 = {1}
Next = {<1, 2>, <3, 5>, <6, 8>,
<2, 4>, <5, 7>, <8, 9>,
<1, 3>, <2, 5>, <4, 7>,
<3, 6>, <5, 8>, <7, 9>,
<10, 11>, <11, 12>,
<13, 14>, <14, 9>}

F6: STS – Erreichbare Zustände

Ein Zustand **s** eines **STS <S, S_0, Next>** ist erreichbar:

- **s** $\in$ **S_0** oder

  $\exists$ **s*** $\in$ **S: s* erreichbar** $\wedge$ **<s*, s>** $\in$ **Next**



S = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}

S_0 = {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>}

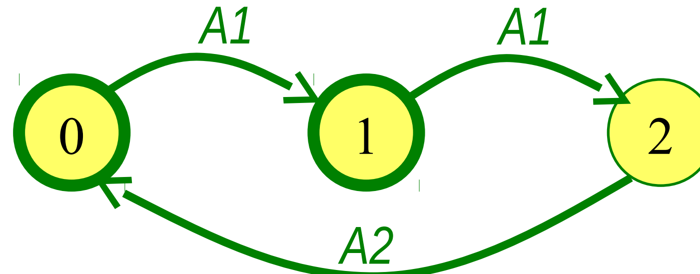
F6: STS – Definition über Variablen und Aktionen

STS <S, S_0, Next> kann definiert werden durch

- Menge von Datenvariablen $V_1, V_2, .., V_n$
  mit den Wertbereichen $W_1, W_2, .., W_n$
- Initialisierungsprädikat **Init** über Variablen
- Menge von Aktionsprädikaten $A_1, A_2, .. A_m$ 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

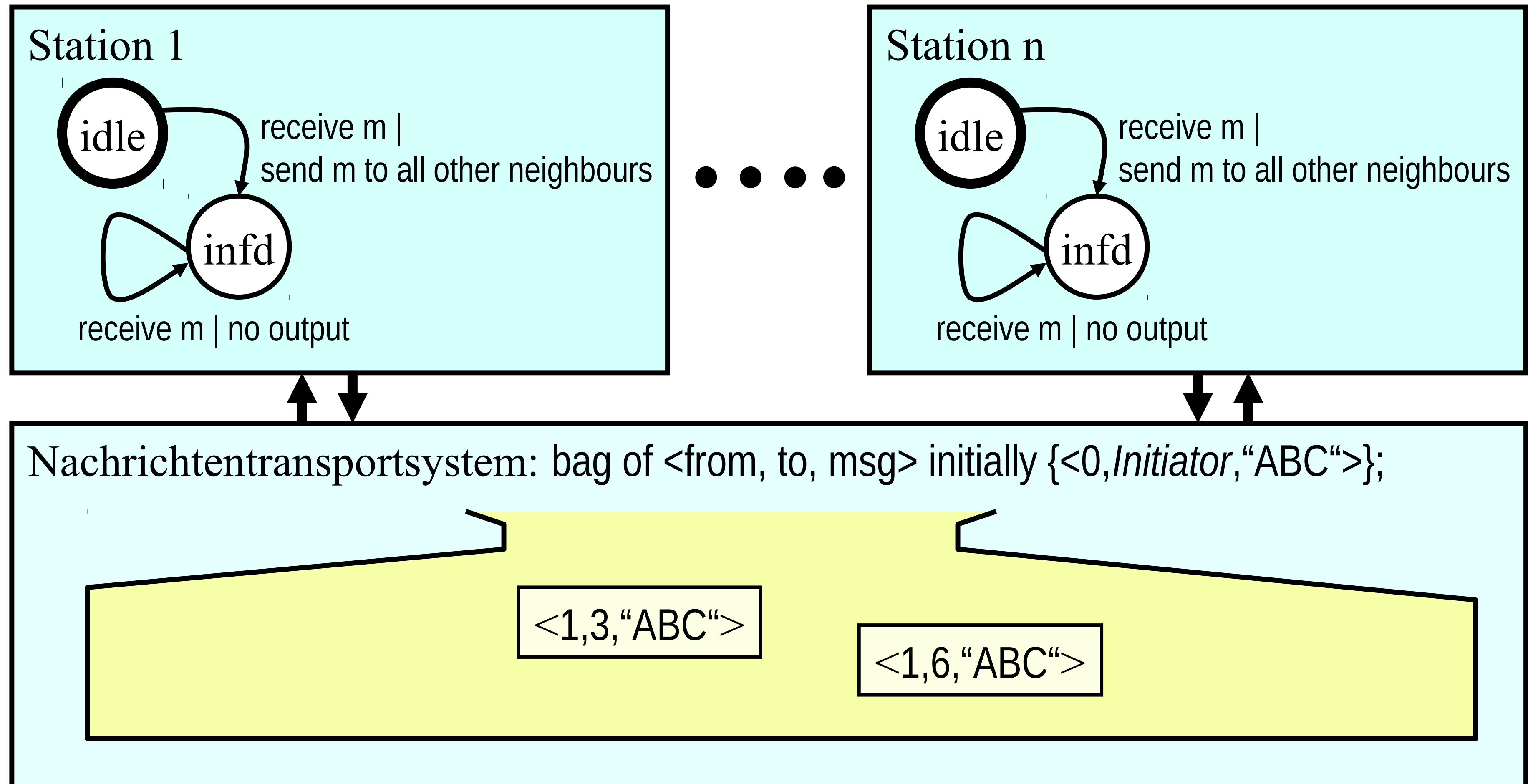
- $S = W_1 \times W_2 \times \ldots \times W_n$
- S_0 = { s : **Init** ist wahr in s}
- **Next** = { <s, s'> : $A_1 \vee A_2 \vee \ldots \vee A_m$ ist wahr für <s, s'> }



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)



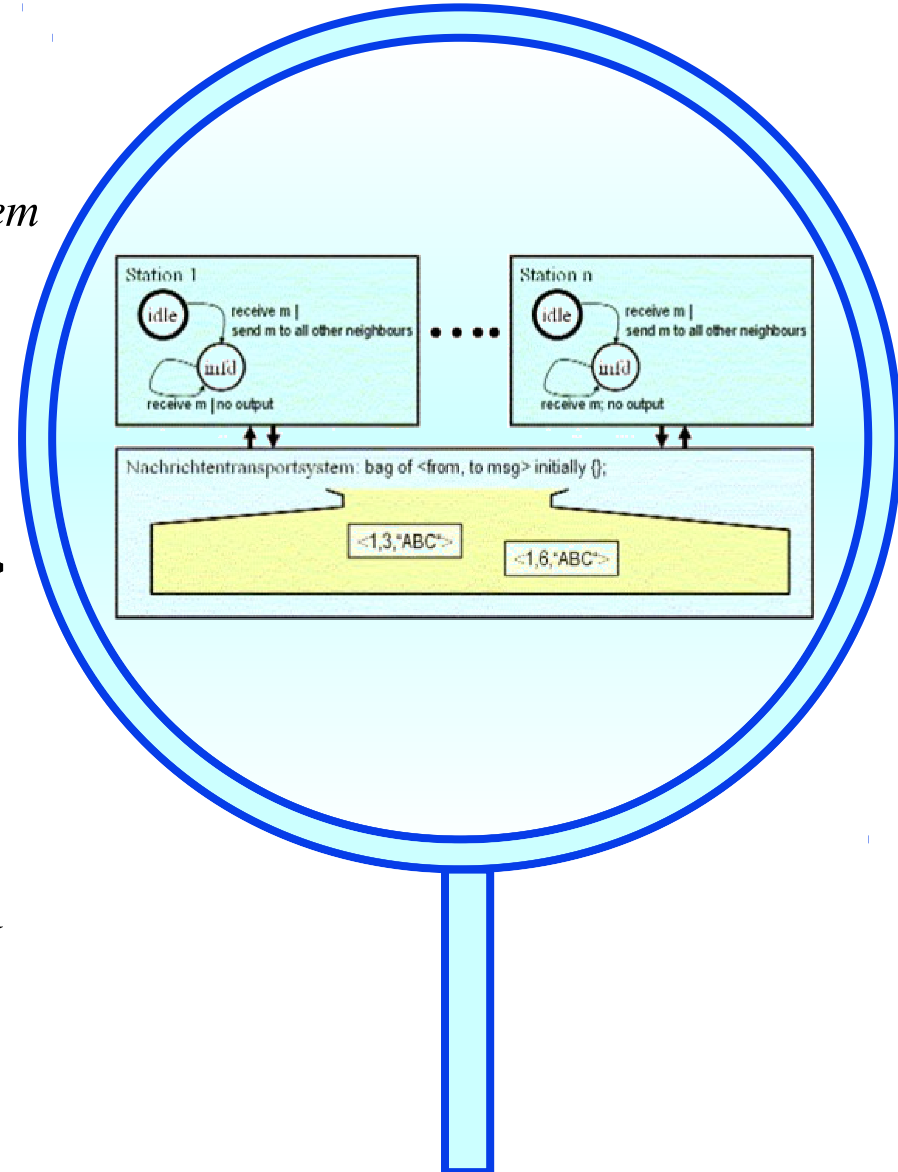
F6: STS – Beispiel

- 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*)

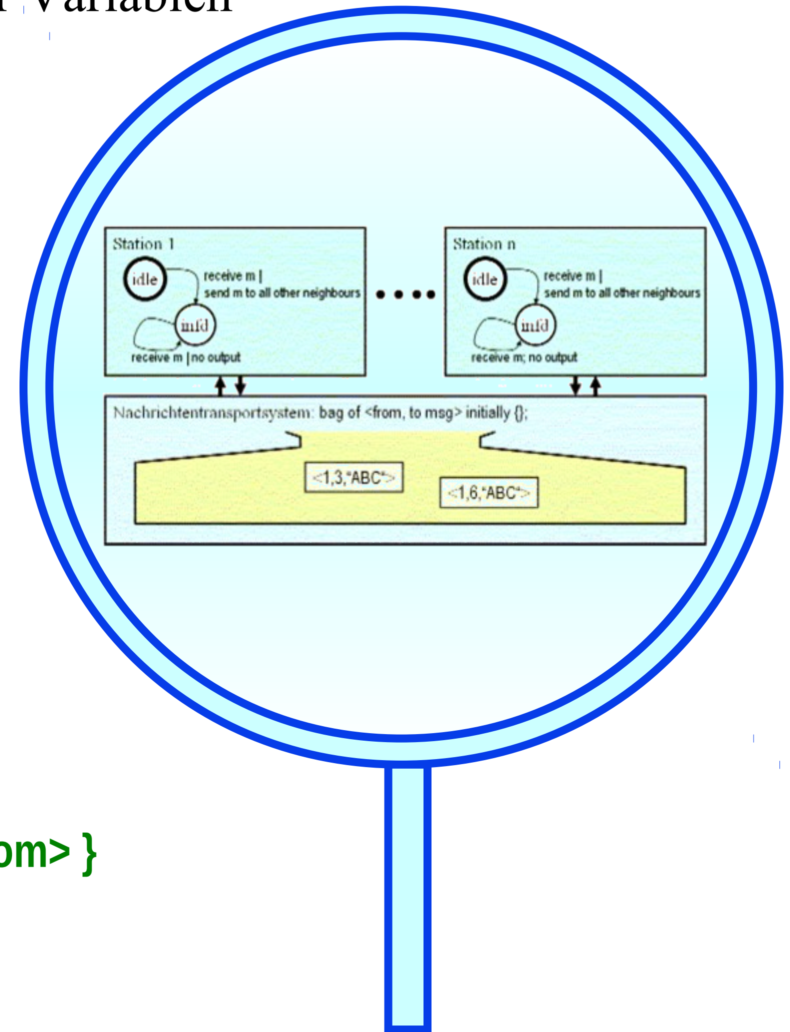
    <infd, idle, idle, {<1,2,“ABC“>, <1,3,“ABC“>} >

    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*
- S_0 = Menge aller Zustände, welche die folgende Initialisierungsbedingung erfüllen
  - cs = < idle, idle, ..., idle> ∧
    nts = {<0, *Initiator*, *Text*>}
- **Next** = Menge aller Transitionen, welche „Forward v Skip" erfüllen.
  - 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}



F6: STS – Beispiel

Variablen

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

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

Init

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

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

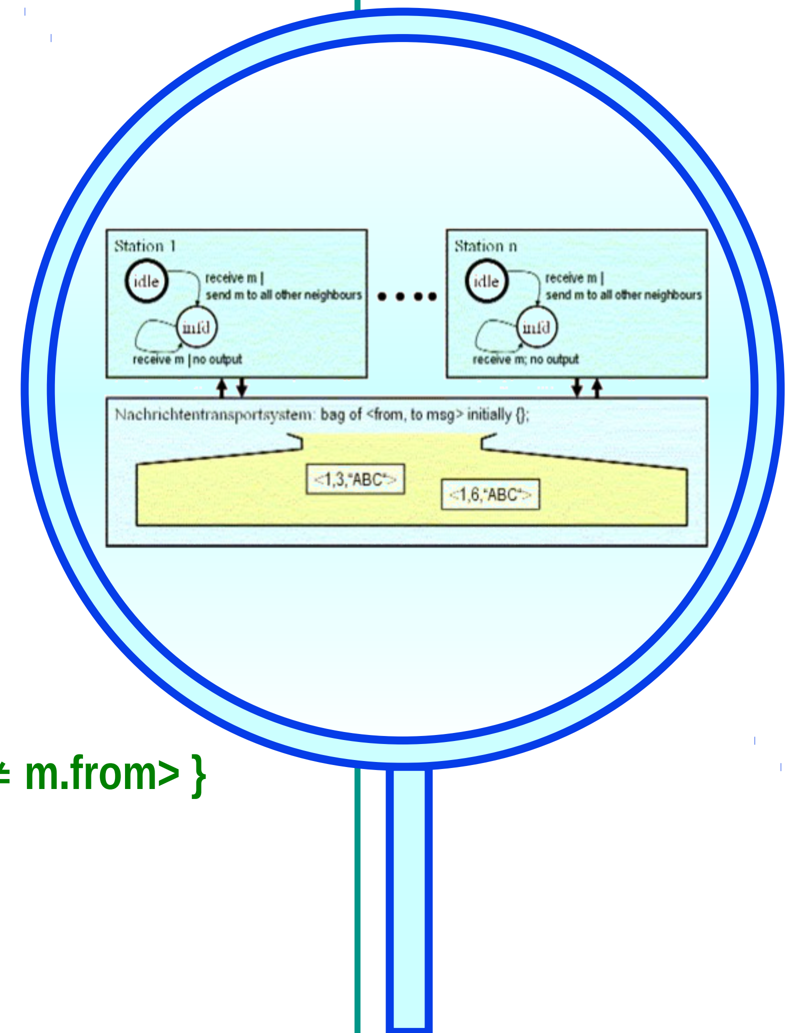
Aktionen

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 – Aktionsprädikate

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

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

- **Aktion = Guard ∧ 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

∃ 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}

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)

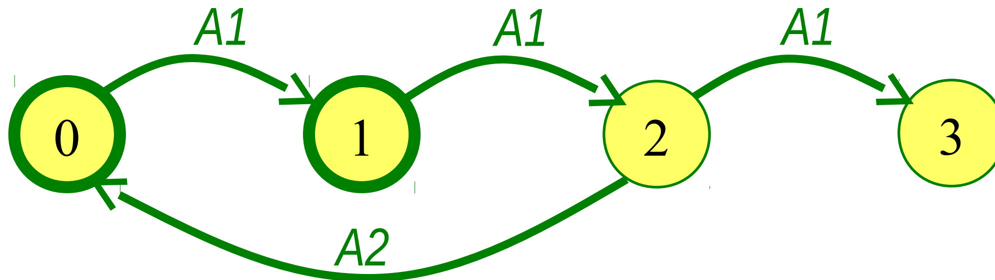
S. Owicki, D. Gries:
'Verifying properties of parallel programs: An axiomatic approach',
CACM 19(5), 279–285, 1976.

F6: STS – Hilfsvariablen

- Hilfsvariable einführen, so dass das System nicht beeinflusst wird.
- Wie?
  - neue Variable **H** einführen
  - Initalzustandsbedingung **B** für neue Variable einbringen, welche die Startwerte anderer Variablen nicht einschränken darf.
    - Init: BisherigeBedingung $\wedge$ **B** so dass gilt

      BisherigeBedingung $\Rightarrow \exists$ **H**: BisherigeBedingung $\wedge$ **B**
- In Aktionen, bei spezieller <Guard, Effekt>-Form
  - die Hilfsvariable darf im **Guard** überhaupt nicht vorkommen
  - die Hilfvariable 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 ∨ V=1 ;*
act	*A1: V<3 ∧ V'=V+1 ;*
	A2: V=2 ∧ 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 ∨ V=1) ∧ (H=1) ;*
act	*A1: V<3 ∧ V'=V+1 ∧ H'=H;*
	A2: V=2 ∧ V'= 0 ∧ 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

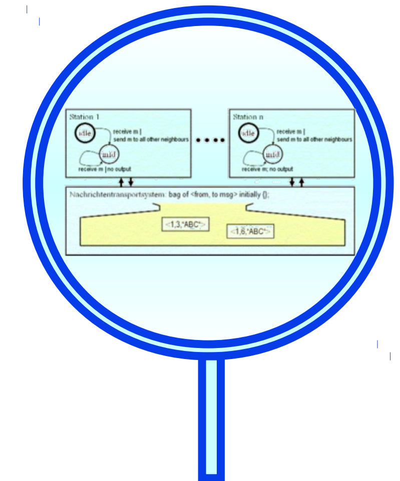
Variablen

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

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

Init

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



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

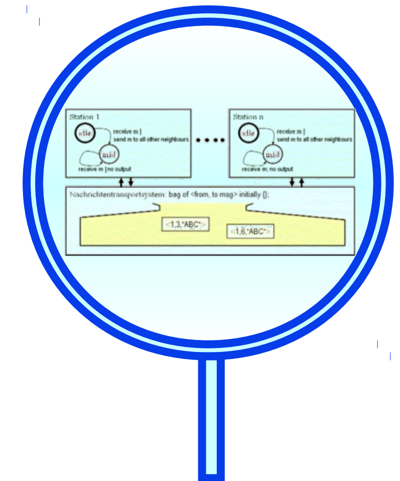
cs: array [1..n] of (idle, infd) ! *Stationen*

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

MC : integer ! *Hilfsvariable – Anzahl gesendeter Nachrichten*

Init

cs = < idle, idle, ..., idle> ∧
nts = {<0, *Initiator*,*Text*>} ∧ MC=0



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} ∧
MC‘=MC + AnzNachbVon(i) - 1

Skip

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

F6: STS – Abläufe als Zustandsfolgen

Die Ausführung eines **STS <S, S_0, Next>** kann in Form der dabei auftretenden Folge eingenommener Zustände repräsentiert werden.

Def: **Mögliche Zustandsfolge σ** eines **STS <S, S_0, Next>**

– $\sigma \in S^* \cup S^\infty$, $\sigma = \langle s_0, s_1, s_2, s_3, \ldots \rangle$

– $s_0 \in S_0$

$\forall i: \langle s_i, s_{i+1} \rangle \in Next \vee s_i = s_{i+1}$

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

– $s_i = s_{i+1}$: Stotterschritt, das System führt keine Transition aus,

– $\langle s_i, s_{i+1} \rangle \in Next \wedge s_i \neq s_{i+1}$: echter Schritt, das System führt eine Transition aus.

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

– $\Sigma = \{\sigma : \sigma$ **ist mögliche Zustandsfolge des STS** $\}$

F6: STS – Unendliche Zustandsfolgen und Stotterschritte

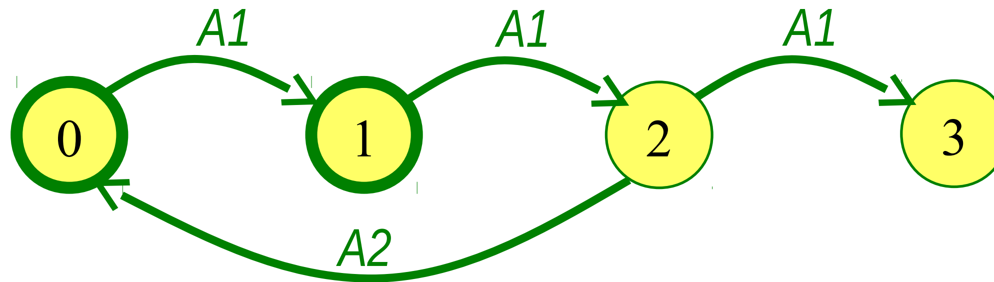
Warum Stotterschritte? *1. am Ende einer Folge 2. in der Mitte einer Folge*

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 Stotterschritte aus und hat darum ebenfalls unendliche Zustandsfolgen.
     Es genügt deshalb, generell nur unendliche Folgen zu betrachten.
1. 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-Stotterschritt 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 Stotterschritten entsteht, ist σ^{+} auch eine mögliche Zustandsfolge des **STS**.
  1. Wieviele Stotterschritte dürfen eingefügt werden?
     Bei unendlich vielen stoppt das System → Liveness-Aspekte

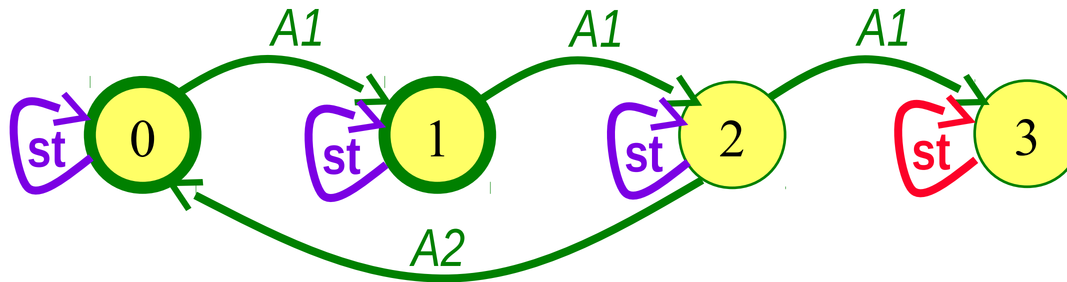
F6: STS – Unendliche Zustandsfolgen und Stotterschritte



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

- ☒ < 0, 1, 2, 3 >
- < 0, 1, 2, 0, 1, 2, 3 >
- < 0, 1, 2, 0, 1, 2, 0, 1, 2, 3 >
- < 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 3 >
- …

F6: STS – Unendliche Zustandsfolgen und Stotterschritte



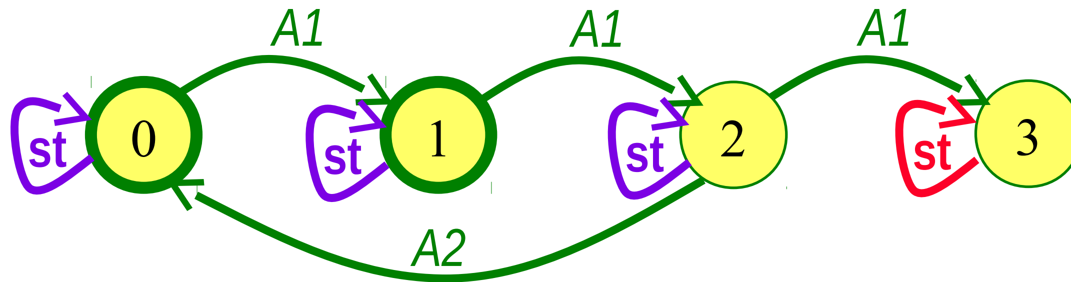
var V : (0, 1, 2, 3) ;
init V=0 ∨ V=1 ;
act A1: V<3 ∧ V'=V+1 ;
A2: V=2 ∧ V'= 0 ;

- < 0, 1, 2, 3 >
- < 0, 1, 2, 0, 1, 2, 3 >
- < 0, 1, 2, 0, 1, 2, 0, 1, 2, 3 >
- < 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 3 >
- …

Einstreuen endlich vieler Stotterschritte
Unendliches Stottern nach Terminierung

- < 0, **0, 0,** 1, **1, 1, 1, 1, 1, 1, 1,** 2, **2,** 3, **3, 3, …** >
- < 0, **0,** 1, **1, 1,** 2, **2, 2, 2,** 0, **0,** 1, **1, 1, 1,** 2, **2,** 3, **3, 3, …** >
- < 0, **0,** 1, 2, 0, **0, 0,** 1, 2, 0, 1, 2, **2, 2, 2, 2, 2, 2,** 3, **3, 3, …** >
- < 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 3, **3, 3, …** >
- …

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 ;

- < 0, 1, 2, 3 >
- < 0, 1, 2, 0, 1, 2, 3 >
- < 0, 1, 2, 0, 1, 2, 0, 1, 2, 3 >
- < 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 3 >
- …

Einstreuen unendlich vieler Stotterschritte

➔ *Vorzeitiger Stop,*
System terminiert nicht wie gewünscht.

Implizite versus explizite Liveness-Angaben

- < 0, **0, 0, 0, …**>
- < 0, 1, **1, 1, 1, …** >
- < 0, 1, 2, **2, 2, 2, …**>

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> ∧

nts = {<0, *Initiator*,*Text*>}

Aktionen

Forward

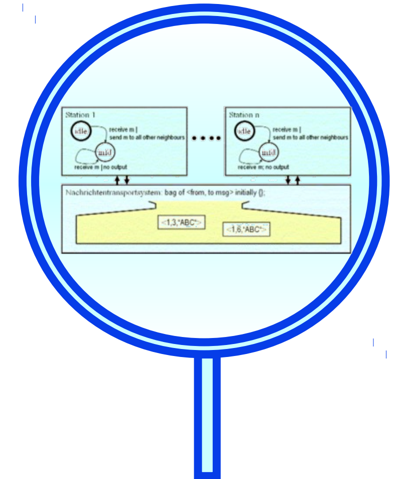
∃ i, m: cs[i]=idle ∧ m∈ nts ∧ m.to=i ∧
cs[i]‘=infd ∧ ∀ j ≠ i: cs[j]‘=cs[j] ∧
nts‘= nts ∪
{ <i, k, m.msg>: istNachbar(k,i) ∧ k ≠ m.from> }
\ {m}

Skip

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

Loss

∃ m: m∈ nts ∧ ∀ j : cs[j]‘=cs[j] ∧ nts‘=nts \ {m}



F6: STS – Unendliche Zustandsfolgen und Stotterschritte

- 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

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

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

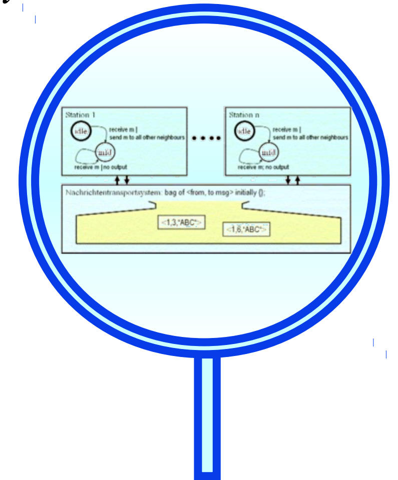
Aktionen

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 <S, S_0, Next>**

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

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

    Σ = {σ: σ 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 <S, S_0, Next>**
  - $\sigma \in S^\infty$, $\sigma = \langle s_0, s_1, s_2, s_3, \ldots \rangle$, $s_0 \in S_0$, $\forall$ i: $\langle s_i, s_{i+1} \rangle \in$ 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 <S, S_0, Next>**
- Def: **Eigenschaft Π**

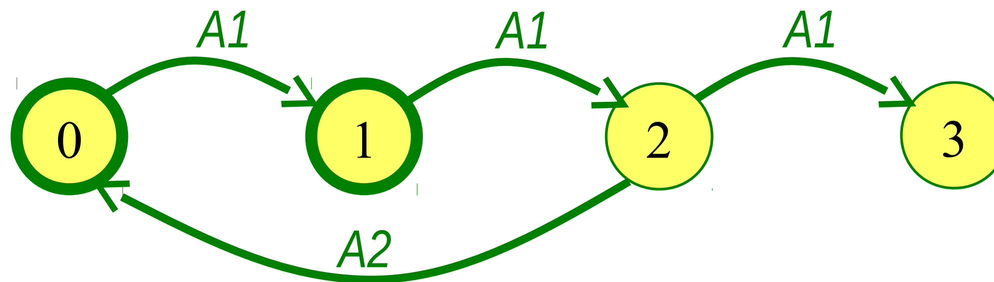
  $\Pi \subset 2^{(S^\infty)}$, d.h. **Π** ist Menge von Zustandsfolgen
- Def: **STS <S, S_0, Next> besitzt Eigenschaft Π**

  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 ;

Σ = {
< 0, 1, 2, 3, 3, 3, … >,
< 0, 1, 2, 0, 1, 2, 3, 3, 3, … >,
< 0, 1, 2, 0, 1, 2, 0, 1, 2, 3, 3, 3, … >,
< 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 3, 3, 3, … >,
…}

$\subset$

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

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 $\sigma = <s_0, s_1, s_2, .., s_i, s_{i+1}, \ldots >$, so dass an s_i erkennbar wird, dass σ die Safety-Eigenschaft Π verletzt.
  - Wenn $\sigma = <s_0, s_1, s_2, .., s_i, s_{i+1}, \ldots >$ die Eigenschaft Π mit s_i verletzt, so verletzt jede Ausführung $\sigma' = <s_0, s_1, s_2, .., s_i, x_1, x_2, \ldots >$ 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:
  - $\sigma \in S^\infty$, $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$, unendliche Zustandsfolge
  - $\zeta \in S^\infty$, $\zeta = \langle z_0, z_1, z_2, z_3, \dots \rangle$, unendliche Zustandsfolge
  - $\theta \in S^*$, $\theta = \langle t_0, t_1, t_2, t_3, \dots, t_n \rangle$, endliche Zustandsfolge
  - $\Pi \subset 2^{(S^\infty)}$, Eigenschaft
  - $pre(\sigma,i) = \langle s_0, s_1, s_2, \dots, s_i \rangle$, $pre(s,i) \in S^*$, Teilausführung von σ bis i
  - $\sigma \in \Pi$, σ hat Eigenschaft Π
  - $\theta \bullet \zeta = \langle t_0, t_1, t_2, t_3, \dots, t_n, z_0, z_1, z_2, z_3, \dots \rangle$, $\theta \bullet \zeta \in S^\infty$, θ fortgesetzt mit ζ

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

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

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

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

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

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 ~> 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 ~> Kabine_in_Etage
  - *Das Programm terminiert.*
    true ~> Terminierungszustand_liegt_vor
  - *Jeder Server-Prozess wird immer wieder in der Lage sein, einen neuen Auftrag zu bearbeiten.*
    Server_tätig ~> 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∩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∩L={}**
    Keine Zustandsfolge kann sowohl T als auch L erfüllen.

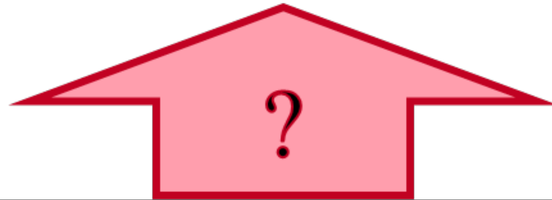
F7: Gemischte Eigenschaften

- 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.

Abstrakte Eigenschaften versus Modell

Abstrakte Eigenschaften

Sie sollen unsere Anforderungen wiedergeben.



Analyse:
Hat Modell gewünschte abstrakte Eigenschaften?

Modell
(und dort dargestellte konkrete Eigenschaften des realen Systems)

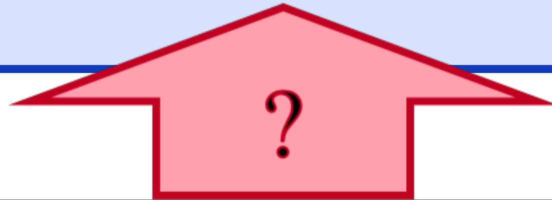
Es soll unseren Entwurf für ein reales System adäquat und überzeugend darstellen,
d.h. es soll die relevanten konkreten Eigenschaften des später implementierten realen Systems auch besitzen.

Abstrakte Eigenschaften versus Modell: Safety

Abstrakte Eigenschaften

Safety,
z.B. „*Etagentür darf nur öffnen, wenn Kabine in Etage*“

Zustandsinvariante über den Zustandsvariablen
z.B.
„Tür[i]=offen ⇒ Kabinenpos=i“



Analyse:
Hat Modell gewünschte abstrakte Eigenschaften?

Modell

(und dort dargestellte konkrete Eigenschaften des realen Systems)

Safety
z.B. Modell eines Gebäude-Aufzugs, so dass im Modell die Safety-Eigenschaften des Systems offensichtlich vorhanden sind.

STS = < S, S0, Next >

Abstrakte Eigenschaften versus Modell: Liveness

Abstrakte Eigenschaften

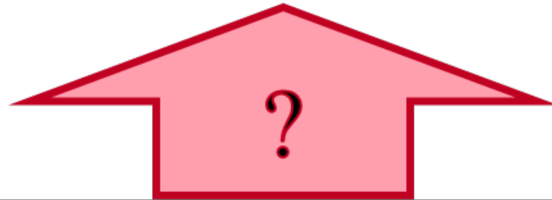
Liveness

z.B. „Wenn ein Rufknopf gedrückt wird, wird sich nach endlicher Zeit die Tür öffnen, oder es wird eine Fehleranzeige gegeben“

Leads-To-Formel über Zustandsbedingungen

z.B.

„FW[i]=1 ~> (Tür[i]=offen ∨ Fehler=1)“



Analyse:

Hat Modell gewünschte abstrakte Eigenschaften?

Modell

(und dort dargestellte konkrete Eigenschaften des realen Systems)

Liveness

z.B. Anreicherung des Safety-Modells eines Gebäude-Aufzugs um Liveness-Annahmen

STS = < S, S0, Next >

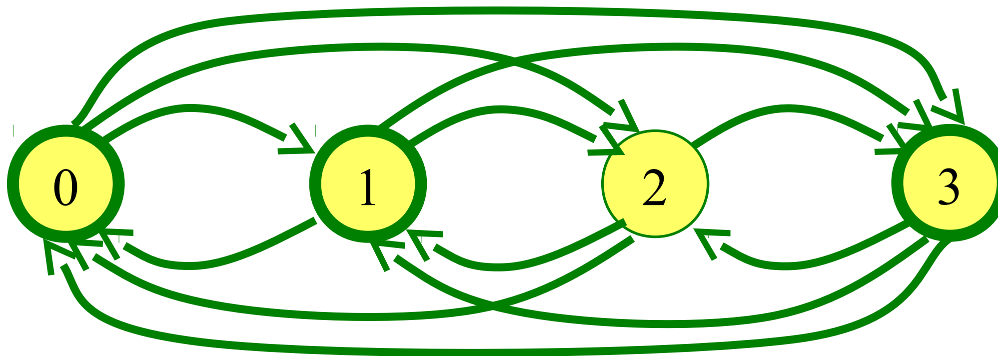
Fairnessannahmen zu den Aktionen des STS,

z.B. WF: go, indicate ;

F7: Safety per STS

- Eine Safety-Eigenschaft kann durch ein **STS <S, S_0, Next>** 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 Stotterschritte ausführen)
  - Die Menge **Σ** aller möglichen Zustandsfolgen des **STS** ist eine Safety-Eigenschaft.
- Beispiel: Eigenschaft „***Fängt nicht mit 2 an***“

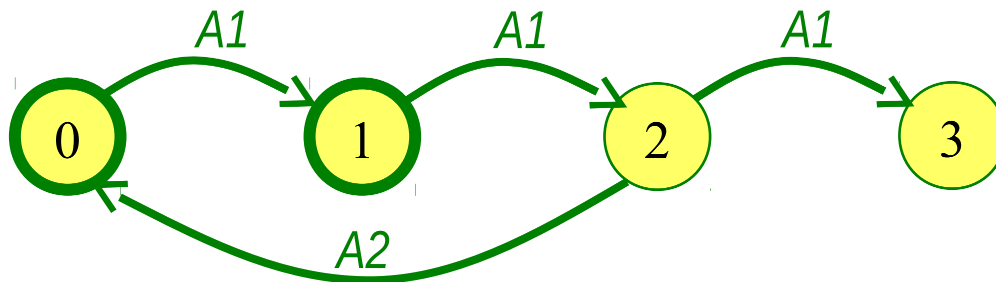
  (im Zustandsraum {0, 1, 2, 3})




```
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



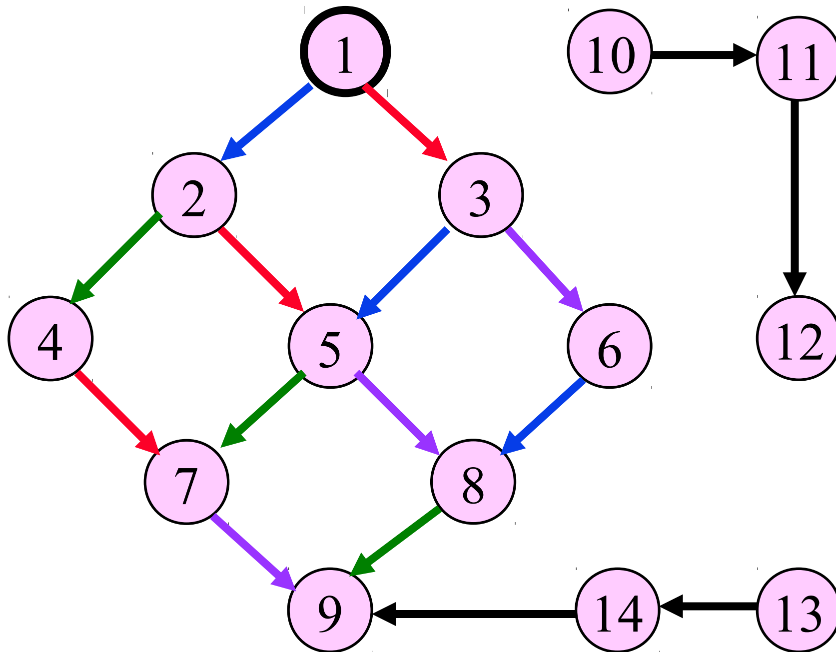
var $V : (0, 1, 2, 3)$;
init $V=0 \vee V=1$;
act A1: $V<3 \wedge V'=V+1$;
A2: $V=2 \wedge V'= 0$;

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



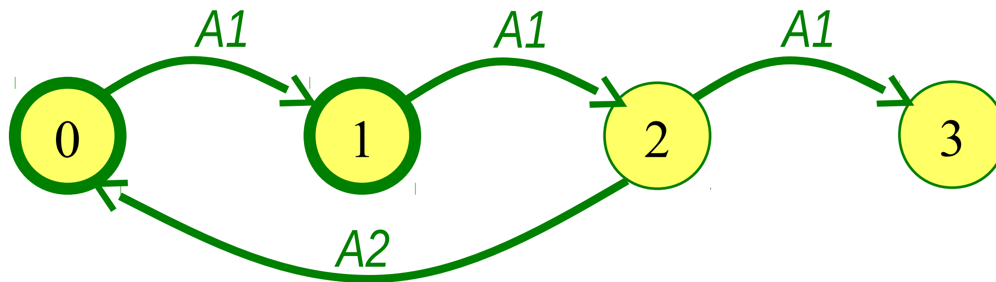
S = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
S_0 = {1}
Next = {<1, 2>, <3, 5>, <6, 8>,
<2, 4>, <5, 7>, <8, 9>,
<1, 3>, <2, 5>, <4, 7>,
<3, 6>, <5, 8>, <7, 9>,
<10, 11>, <11, 12>,
<13, 14>, <14, 9>}

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

Beispiel:
I2 **cs<10**

F7: Safety per Invarianten

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




```
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 ;
```

Beispiel

„Dem Zustand 3 geht immer Zustand 2 voraus“

I3 $V=3 \Rightarrow H=2$

F7: Safety per Invarianten

- Bei Terminierung wurden alle Stationen informiert.
  - nts={} ⇒ ∀ 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 < from, to, msg > ! *Transportsystem*

Init

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

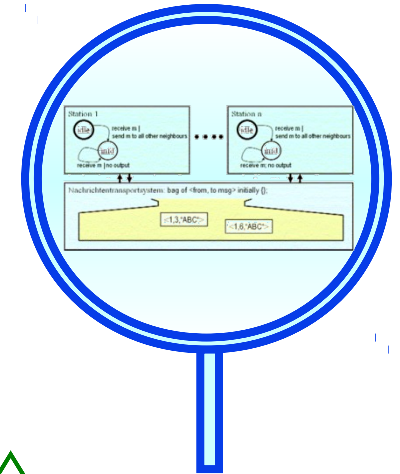
Aktionen

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}

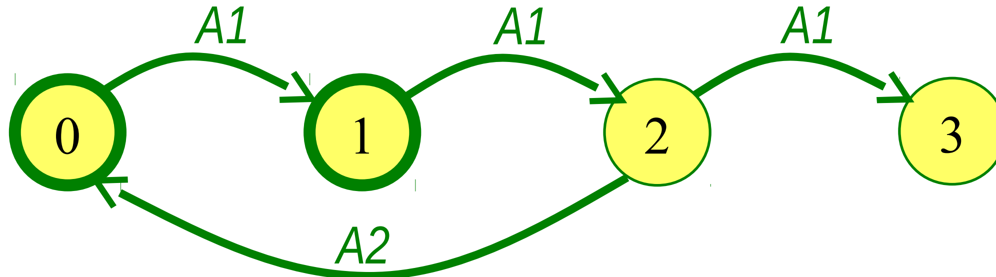
Skip

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



F7: Safety per Invarianten

- Die kanonische Historienhilfsvariable speichert die gesamte Historie eines Momentanzustands.
  - Seien **V_1, V_2, …, V_n** die Variablen des **STS**
  - **HH : list of < v_1, v_2, …, v_n >** ! Hilfsvariable
  - Initial: **HH=empty**
  - bei jeder Aktion: **HH‘=append(HH, <V_1, V_2, …, V_n>)**



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 ∨ V=1 )
      ∧ HH=empty ;
act   A1: V<3 ∧ V‘=V+1
       ∧ HH‘=append(HH,V);
      A2: V=2 ∧ V‘= 0
       ∧ HH‘=append(HH,V);
```

F7: Aktionen-Fairness

- Gegeben **STS <S, S_0, Next>**
  mit **Next** = $a_1 \cup a_2 \cup \ldots \cup a_n$, gegliedert in Subrelationen / Aktionen
  sowie eine (unendliche) Ausführung $\sigma \in S^\infty$, $\sigma = \langle s_0, s_1, s_2, s_3, \ldots \rangle$, .
  - 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**)

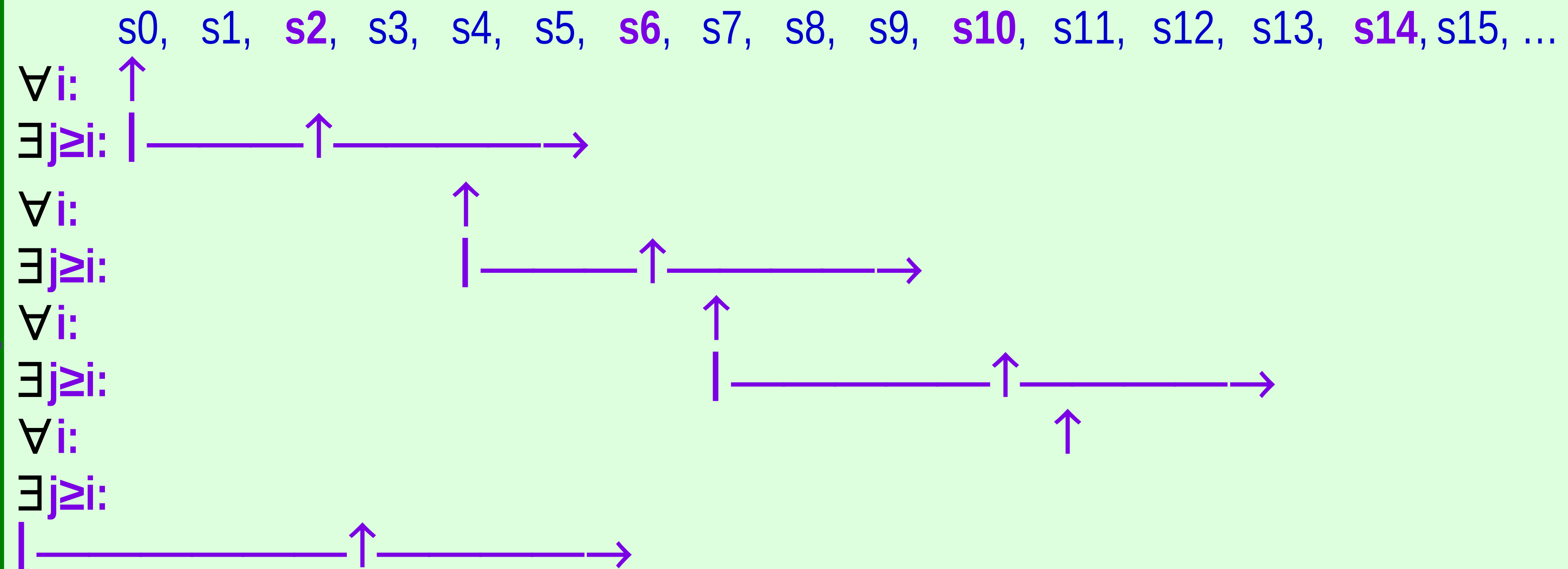
F7: Aktionen-Fairness

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

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

  $\vee$ *! oder*

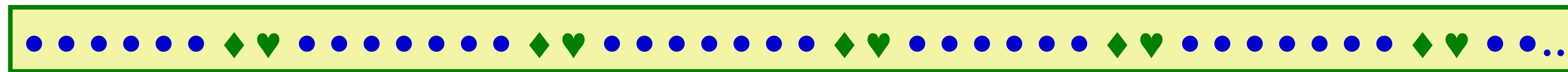
  $\forall i \; \exists j \geq i: \neg \mathrm{Enabled}(a, s_j)$ *! immer wieder ist a disabled*



F7: Aktionen-Fairness

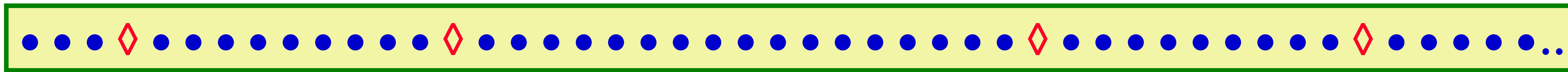
☒ Def: **WF**(**a**) in $\sigma \in S^{\infty}$, $\sigma = \langle s_0, s_1, s_2, s_3, \ldots \rangle$

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



$\vee$ *! oder*

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



Legende

●   Zustand ohne Besonderheit

♦ ♥   Zustandspaar, so dass Enabled(**a**,♦) und <♦,♥>∈ **a**

◊   Zustand, so dass ¬Enabled(**a**,◊)

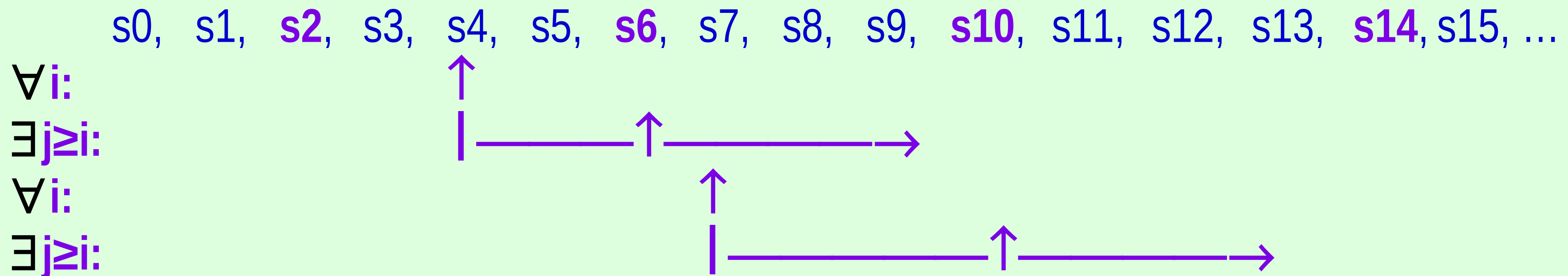
F7: Aktionen-Fairness

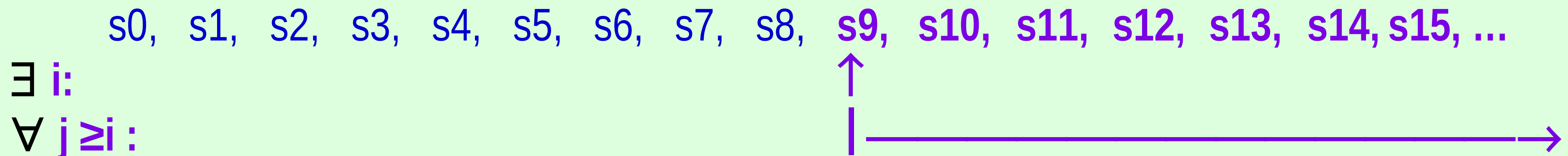
☒ Def: **SF**(a) in $\sigma \in S^\infty$, $\sigma = \langle s_0, s_1, s_2, s_3, \dots \rangle$

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

$\vee$ *! oder*

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





F7: Aktionen-Fairness

- Def: SF(a) in $\sigma \in S^\infty$, $\sigma = <s_0, s_1, s_2, s_3, \ldots>$

$\forall$ i $\exists$ j≥i: $<s_j, s_{j+1}> \in$ a *! immer wieder a*

● ● ● ● ● ● ♦ ♥ ● ● ● ● ● ● ● ♦ ♥ ● ● ● ● ● ● ● ♦ ♥ ● ● ● ● ● ● ♦ ♥ ● ● ● ● ● ● ● ♦ ♥ ● ●..

$\vee$ *! oder*

$\exists$ i $\forall$ j≥i: ¬Enabled(a, s_j) *! schließlich immer ist a disabled*

● ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ .

Legende

● Zustand ohne Besonderheit

♦ ♥ Zustandspaar, so dass Enabled(a,♦) und <♦,♥>∈ a

◊ Zustand, so dass ¬Enabled(a,◊)

F7: Liveness per Aktionen-Fairness-Annahmen

- Gegeben **STS <S, S_0, Next>**
  mit **Next** = $a_1 \cup a_2 \cup \ldots \cup a_n$, 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 Stotterschritte 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.

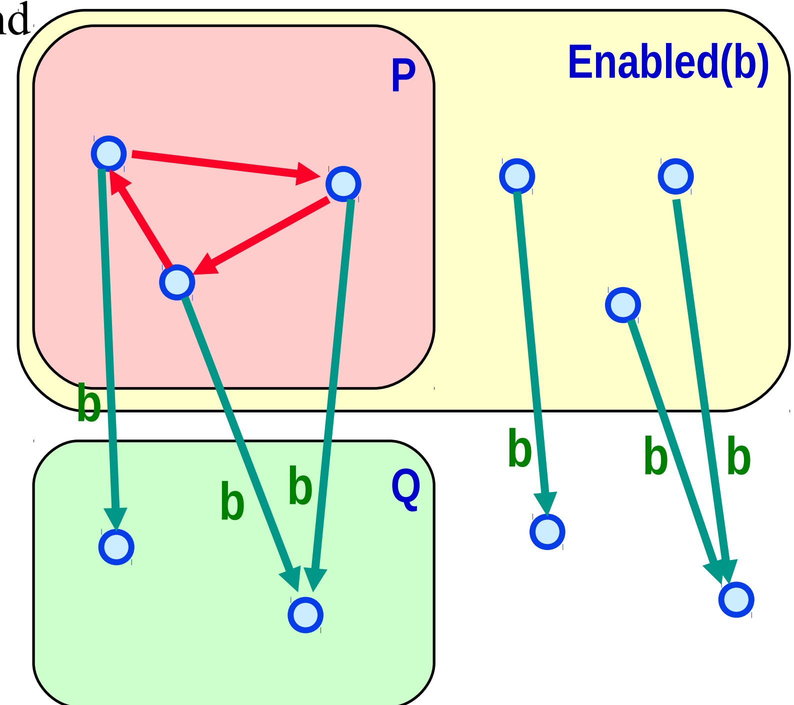
F7: Liveness per Aktionen-Fairness-Annahmen

Liveness: P ~> Q

- Gegeben **STS <S, S_0, Next>**
  mit **Next** = $a_1 \cup a_2 \cup \ldots \cup a_n$, gegliedert in Subrelationen / Aktionen a_i

- Einfacher Fall
  Es gibt Aktion **b⊂Next**, mit folgenden Eigenschaften
  - **P impliziert** 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 ~> Q** zum Ausdruck



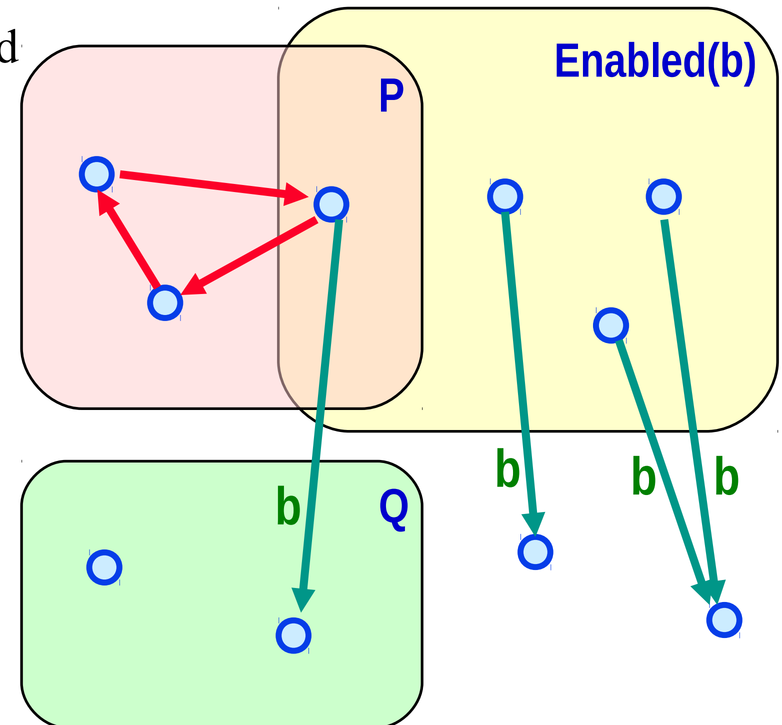
F7: Liveness per Aktionen-Fairness-Annahmen

Liveness: P ~> Q

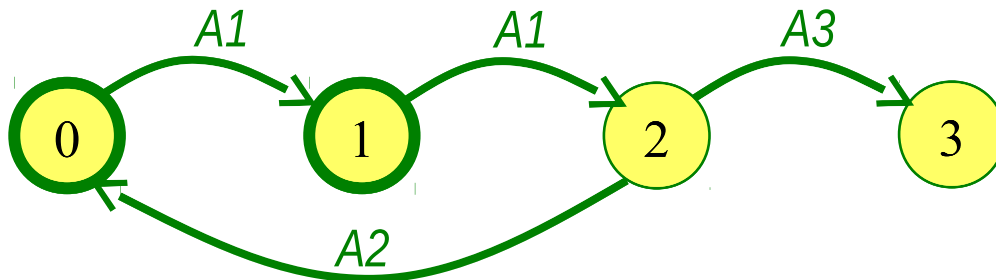
- Gegeben **STS <S, S_0, Next>**
  mit **Next** = $a_1 \cup a_2 \cup \ldots \cup a_n$, gegliedert in Subrelationen / Aktionen a_i

- Etwas weniger einfacher Fall
  Es gibt Aktion **b⊂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 ~> 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 ~> V=2: WF(A1)

V=2 ~> V=3: WF(A1) ∧ SF(A3)

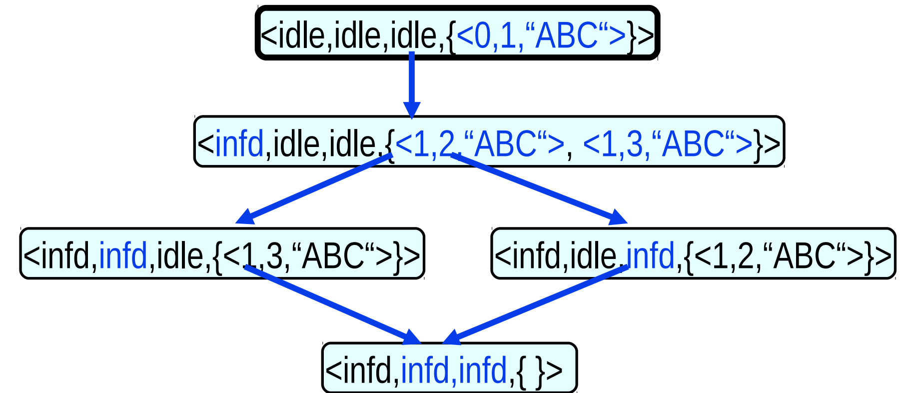
Zyklus 0, 1, 2 wird lebendig durchlaufen:
Immer wieder gilt Enabled(A3)

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

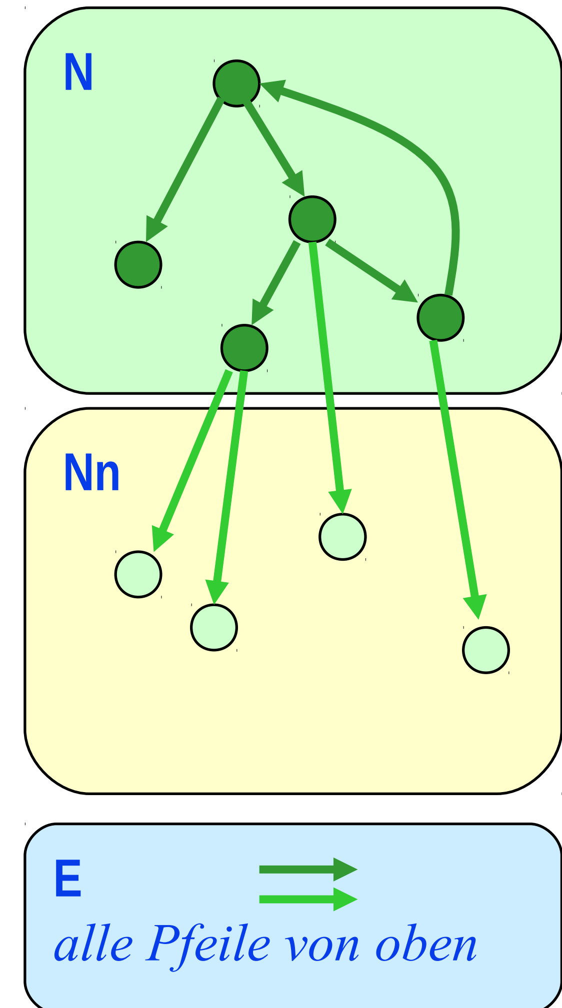


- *Algorithmus*
  - *Varianten Tiefe / Breite / Random*
- *STS*
- *Petri Netz Markierungsgraph*
- *Gekoppelte LTS*
- *Bezug zu CCS*
- *Gekoppelte erweiterte Mealy-Automaten*
- *Werkzeug: SPIN*

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

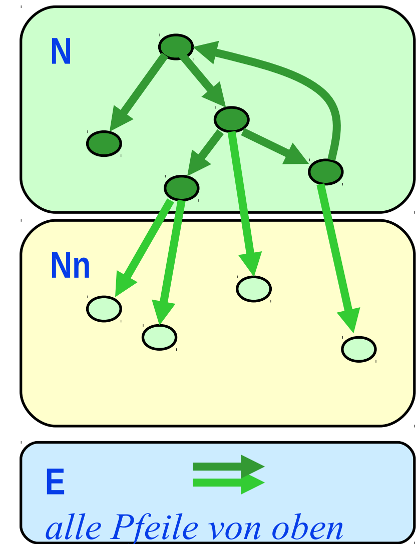
  solange Nn ≠ {}
  - wähle s∈ Nn ;
  - N := N∪{s} ; Nn := Nn\{s} ;
  - für jedes s‘, das von s aus durch eine Transition als Folgezustand erreicht werden kann:
    - E:=E∪{<s, s‘>} ;
    - Falls s‘∉ N∪Nn
      - Nn:=Nn ∪{s‘} ;
- **Problem: Zustandsexplosion (State Explosion)**
  - Anzahl der Zustände 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 ≠ {}
  - wähle s∈ Nn ;
  - N := N∪{s} ; Nn := Nn\{s} ;
  - für jedes s‘, das von s aus durch eine Transition als Folgezustand erreicht werden kann:
    - E:=E∪{<s, s‘>} ;
    - Falls s‘∉ N∪Nn
      - Nn:=Nn ∪{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

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

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

Aktionen

Forward

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

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

$nts' = nts \cup$

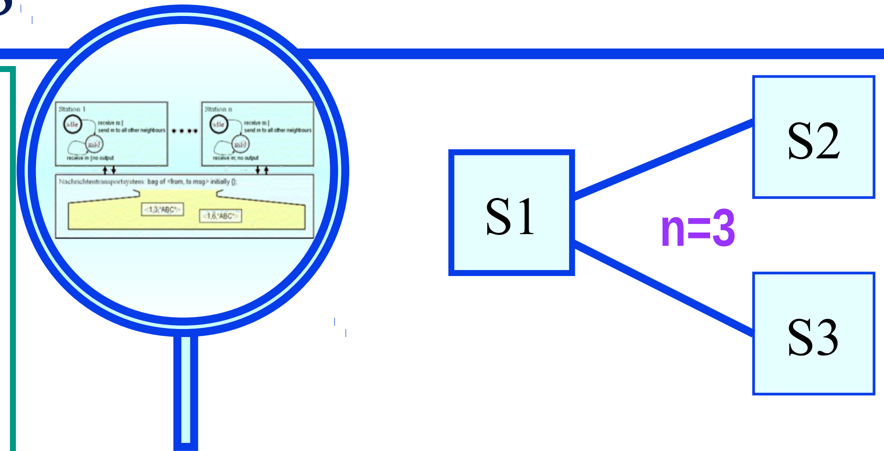
$\{ \langle i, k, m.msg \rangle : istNachbar(k,i) \wedge$

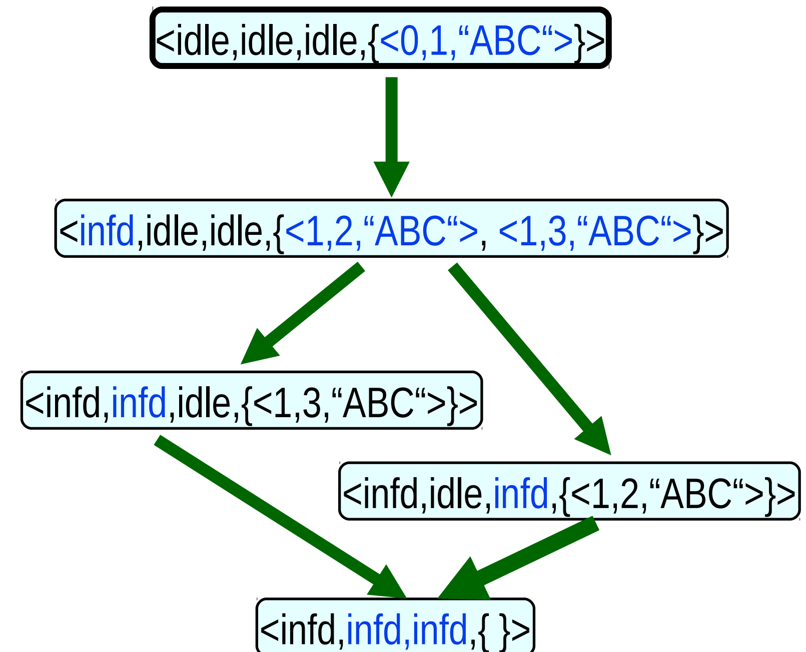
$k \neq m.from \} \setminus \{m\}$

Skip

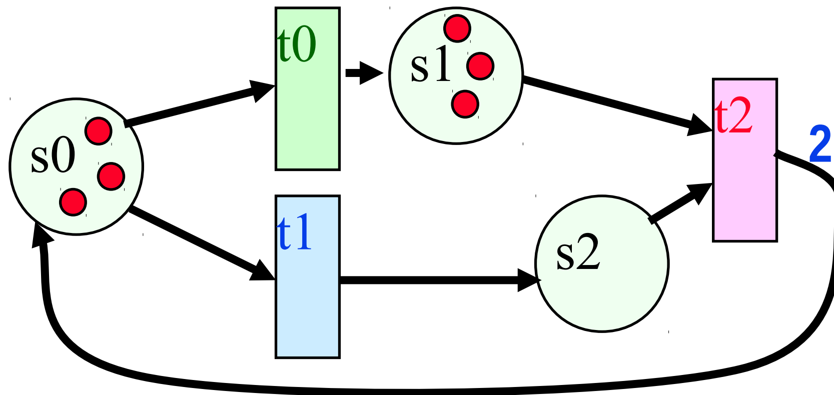
$\exists\, i, m: cs[i]=infd \wedge m \in nts \wedge m.to=i \wedge$

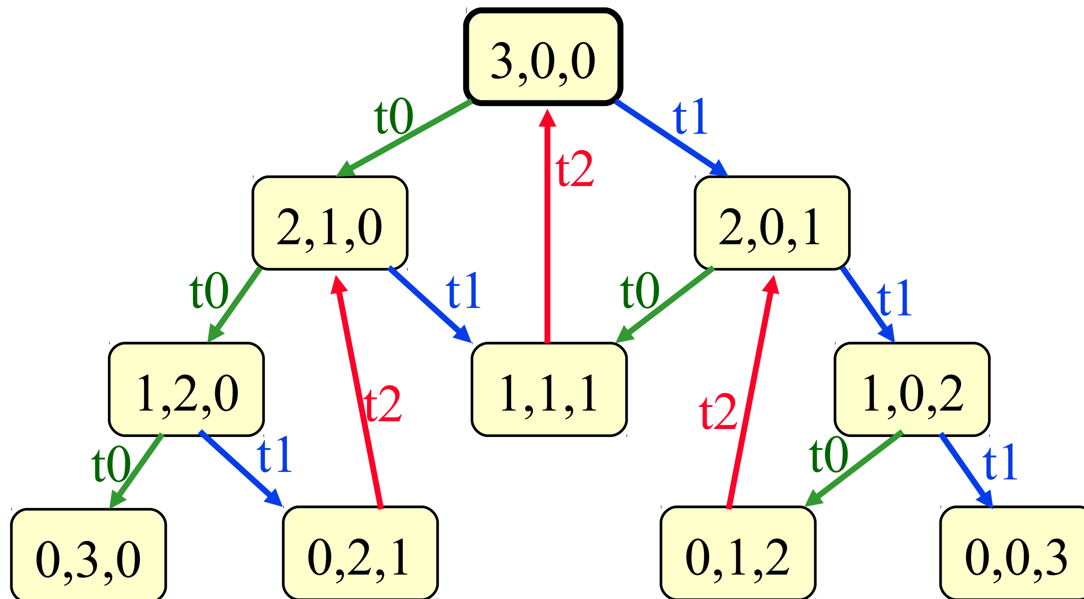
$\forall\, j : cs[j]'=cs[j] \wedge nts'=nts \setminus \{m\}$





F8: Erreichbarkeitsanalyse – Markierungsgraph zum Petri Netz

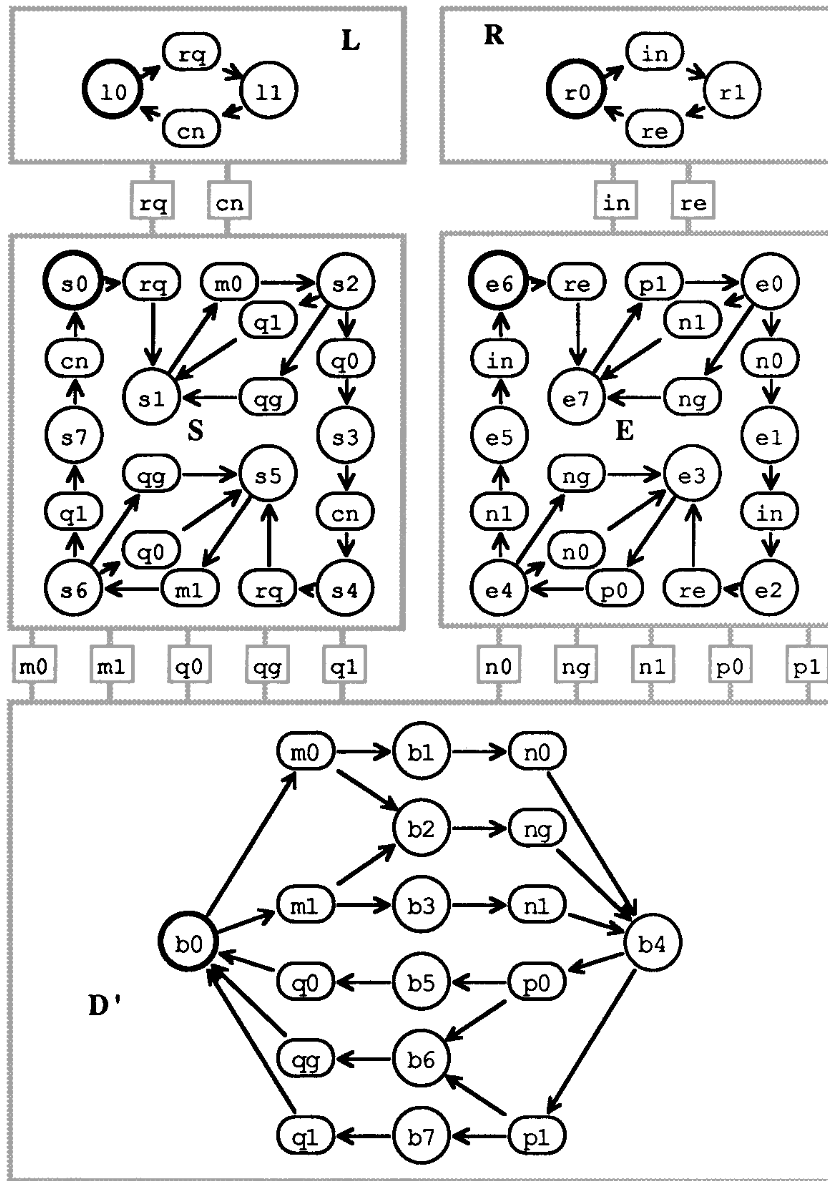




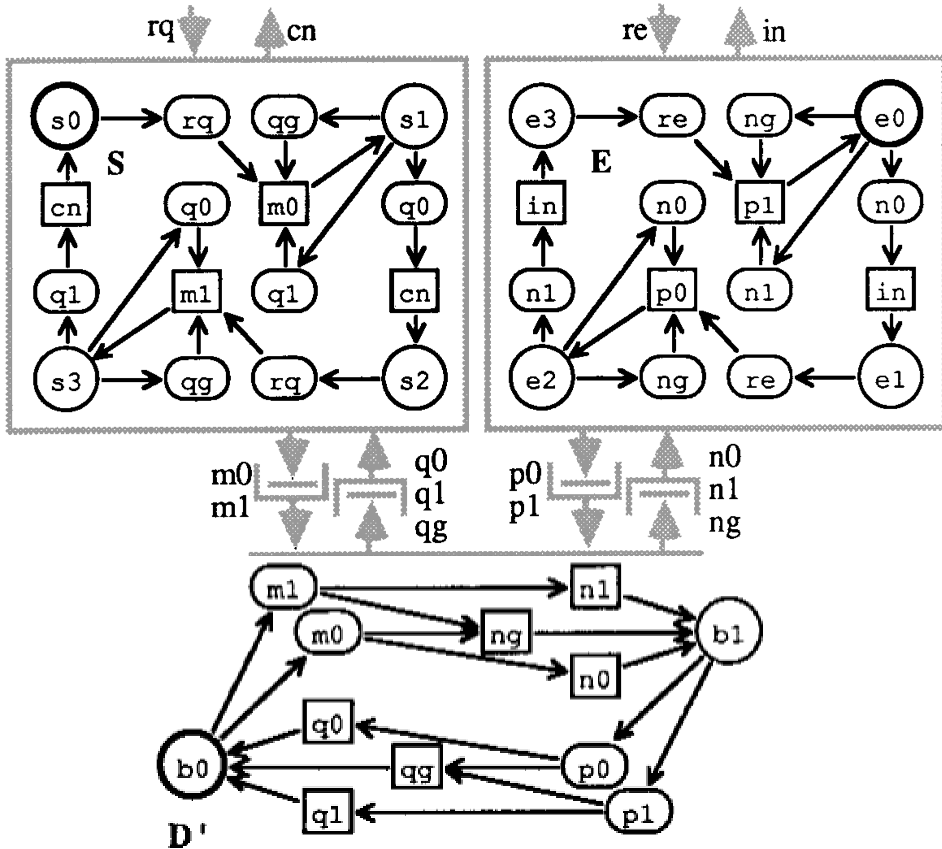
- Variablen
  - **N : set of States** !
    Bearbeitete Knoten des Graphen
  - **Nn : set of States** !
    Anhängige Knoten des Graphen
  - **E : set of State-Pairs** !
    Kantenmenge des Graphen
- Initialzustand
  - **N = {}**
  - **Nn = {x : x ist Startzustand}**
  - **E = {}**
- Anweisung

  solange Nn ≠ {}
  - **wähle s∈ Nn ;**
  - **N := N∪{s} ; Nn := Nn\{s} ;**
  - **für jedes s‘, das von s aus durch eine Transition als Folgezustand erreicht werden kann:**
    - **E:=E∪{<s, s‘>} ;**
    - **Falls s‘∉ N∪Nn**
      - **Nn:=Nn ∪{s‘} ;**

F8: Erreichbarkeitsanalyse – Gekoppelte LTS



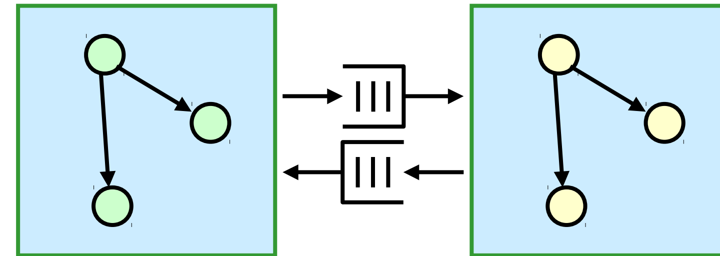
F8: Erreichbarkeitsanalyse – Gekoppelte EFSM



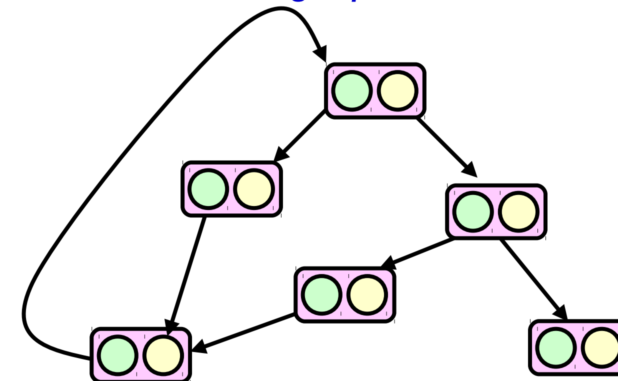
F8: Erreichbarkeitsanalyse – Kriterien

- 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 Erfolgszuständen

System gekoppelter Instanzen



Erreichbarkeitsgraph des Gesamtsystems



Safety speziell, Liveness speziell:

Vorstufe des Model Checkings

SPIN (Simple Promela Interpreter)

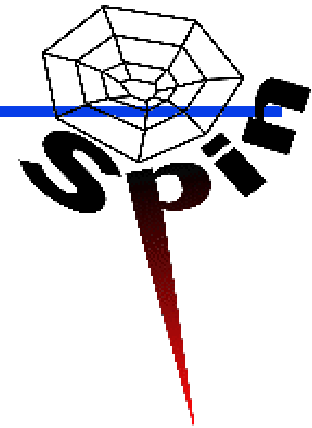
- Modelchecker von G. Holzmann
- Jährliche SPIN Workshops seit 1995
  - Teil der ETAPS (European Joint Conferences on Theory and Practice of Software)
- ACM Software System Award (2001)
- In Forschung & Industrie eingesetzt
  - NASA/JPL Laboratory for Reliable Software
  - Bell Labs/Lucent Technologies
  - 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.

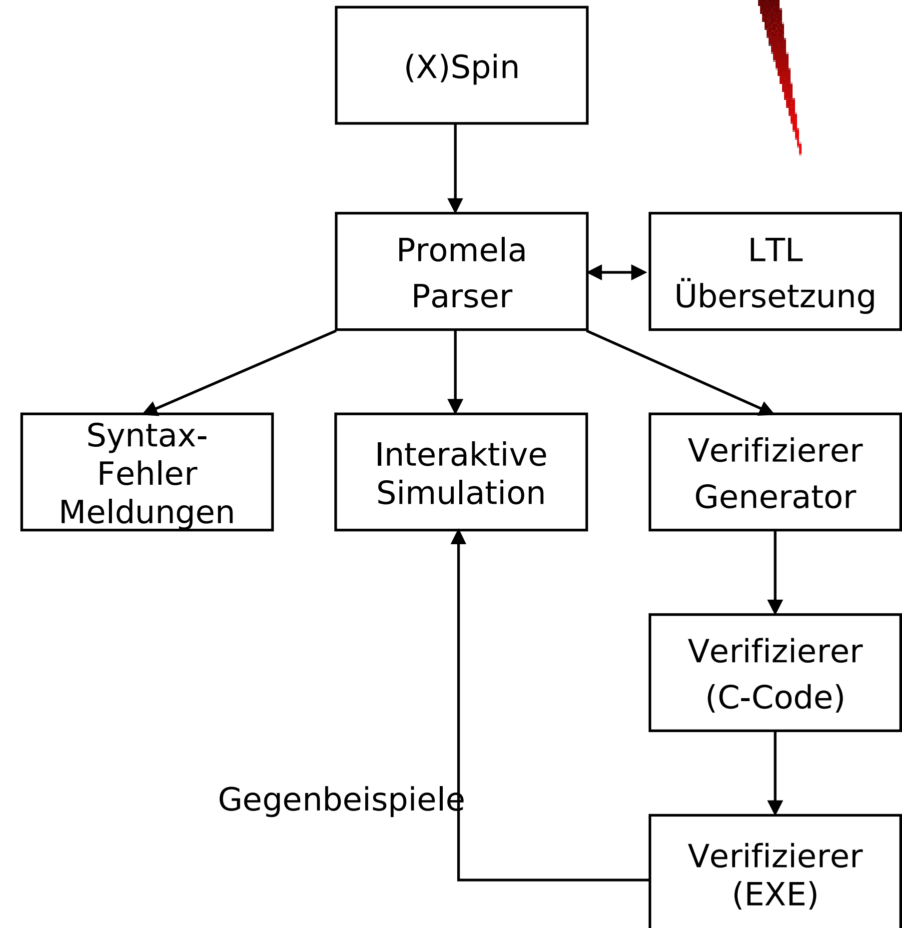
Modellchecker Spin

- 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

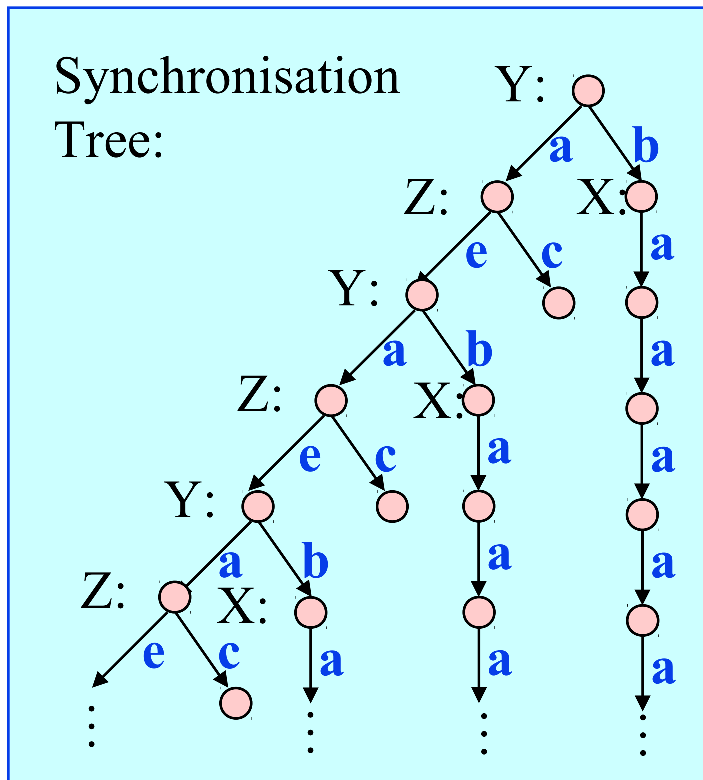
```
// Generated on Sat Jan 24 13:32:56 CET 2004 by cTLA2Promela compiler.
// Version: ($Id: CTLA2PC.java,v 1.93 2004/01/02 14:58:55 cvs Exp $)
...
#define BVSET(bv, esz, idx, val) bv = ((bv & (~(((1<<esz)-1) << (idx*esz)))) |
    (val<<(idx*esz)))      /* Makros */
...
#define UNKNOWN_NODE 0     /* 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: Logiken (LTL, CTL, CTL*)

- ☒ Lineare temporale Logik LTL
- Computation Tree Logik CTL
- Kombination CTL*

L. Lamport: „Sometimes is sometimes not never“

$$\Diamond a \Leftrightarrow \neg \Box \neg a$$



Ausführung: $\sigma \in S^\infty$, $\sigma = <s0, s1, s2, s3, \ldots>$

<u>Literatur</u>

M. Huth and M. Ryan:
Logic in Computer Science – Modelling and reasoning about systems,
Cambridge University Press, 2000.

F9: Lineare temporale Logik LTL – Syntax

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

$\mathbf{X}\ \varphi$	NeXt, ab nächstem Schritt gilt φ
$\varphi\ \mathbf{U}\ \psi$	Until, φ hält bis schließlich ψ

  - Definierbare temporale Operatoren

$\mathbf{F}\ \varphi \equiv \Diamond\ \varphi \equiv \text{true}\ \mathbf{U}\ \varphi$	Future, Eventually, Schließlich,
$\mathbf{G}\ \varphi \equiv \Box\ \varphi \equiv \neg \Diamond \neg \varphi$	Globally, Always, Immer
$\varphi\ \mathbf{R}\ \psi \equiv \neg(\neg \varphi\ \mathbf{U} \neg \psi)$	Release, φ entlässt ψ
$\varphi \leadsto \psi \equiv \Box(\varphi \Rightarrow \Diamond\ \psi)$	leadsto, φ führt zu ψ

F9: Lineare temporale Logik LTL – Semantik

- Betrachtet werden unendliche Zustandsfolgen: $\sigma = \langle s_0, s_1, s_2, s_3, \ldots \rangle$
- Semantik („⊨“ 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:

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

    $\sigma \models \varphi\, \mathbf{U}\, \psi \Leftrightarrow \wedge$

    $\exists\, j: 0 \le j\ \forall\, i: 0 \le i < j$

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

    $\langle s_j, s_{j+1}, s_{j+2}, s_{j+3}, \ldots \rangle \models \psi$ **U**ntil, φ hält bis schließlich ψ

F9: Lineare temporale Logik LTL – Semantik

- Betrachtet werden unendliche Zustandsfolgen: $\sigma = \langle s_0, s_1, s_2, s_3, \ldots \rangle$
- Semantik („$\models$“ steht für „erfüllt“)
  - Grundlegende temporale Operatoren:

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

$\sigma \models \varphi\ U\ \psi \Leftrightarrow \wedge$
$\exists\ j: 0 \le j\ \forall\ i: 0 \le i < j$
$\langle s_i, s_{i+1}, s_{i+2}, s_{i+3}, \ldots \rangle \models \varphi \wedge$
$\langle s_j, s_{j+1}, s_{j+2}, s_{j+3}, \ldots \rangle \models \psi$ — **U**ntil, φ hält bis schließlich ψ

  - Definierbare temporale Operatoren

$\sigma \models F\ \varphi \equiv \lozenge\ \varphi \equiv \text{true}\ U\ \varphi \Leftrightarrow$ — **F**uture, Eventually, Schließlich
$\exists\ i: \langle s_i, s_{i+1}, s_{i+2}, s_{i+3}, \ldots \rangle \models \varphi$

$\sigma \models G\ \varphi \equiv \square\ \varphi \equiv \neg\ \lozenge\ \neg\ \varphi \Leftrightarrow$ — **G**lobally, Always, Immer
$\forall\ i: \langle s_i, s_{i+1}, s_{i+2}, s_{i+3}, \ldots \rangle \models \varphi$

$\sigma \models \varphi\ R\ \psi \equiv \neg(\neg\ \varphi\ U\ \neg\ \psi) \Leftrightarrow$ — **R**elease, φ entlässt ψ
$\forall\ j: 0 \le j\ \exists\ i: 0 \le i < j$
$\langle s_i, s_{i+1}, s_{i+2}, s_{i+3}, \ldots \rangle \models \varphi \vee$
$\langle s_j, s_{j+1}, s_{j+2}, s_{j+3}, \ldots \rangle \models \psi$

$\sigma \models \varphi \rightsquigarrow \psi \equiv \square(\varphi \Rightarrow \lozenge\ \psi) \Leftrightarrow$ — leadsto, φ führt zu ψ
$\forall\ i: (\langle s_i, s_{i+1}, s_{i+2}, s_{i+3}, \ldots \rangle \models \varphi \Rightarrow$
$\exists\ j\ i \le j: \langle s_j, s_{j+1}, s_{j+2}, s_{j+3}, \ldots \rangle \models \psi)$

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 □, ◊ und ~>)
  - Immer wieder a

    □ ◊ a
  - Schließlich immer a (a ist schließlich stabil)

    ◊ □ a
  - a folgt nach endlicher Zeit b (Auftrag a terminiert mit b)

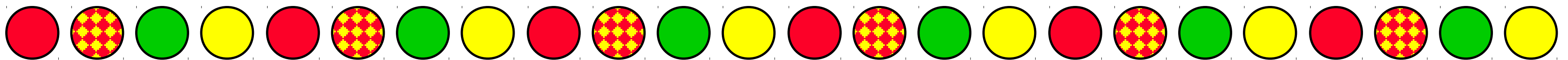
    a ~> b
  - a ist Invariante, a gilt immer

    □ a
  - a gilt nie (a ist verboten)

    □ ¬ a

F9: Lineare temporale Logik LTL – Beispiel

- Betrachtet werden unendliche Zustandsfolgen: $\sigma = \langle s_0, s_1, s_2, s_3, \ldots \rangle$
- Beispiel *Verkehrsampel*
  - □ rot => - X grün *! auf Rot folgt nicht gleich Grün*
  - □ ◊ grün *! Es wird immer wieder grün*
  - rot ~> grün *! Rot folgt nach endlicher Zeit Grün*
  - □ (rot =>(rot U rotgelb) U grün) *!auf Rot folgt Rotgelb und dann Grün*



F9: Lineare temporale Logik LTL – Beispiel

Safety

- Bei Terminierung wurden alle Stationen informiert.
  $\square(nts=\{\} \Rightarrow cs=\langle infd, infd, .., infd \rangle)$
- Wenn eine Station im Zustand infd ist, verlässt sie ihn nie wieder.
  $\square(cs[i]=infd \Rightarrow \neg \Diamond\ cs[i] \neq infd)$

Liveness

- Nach dem Start tritt nach endlicher Zeit Terminierung ein.
  Init ~> nts={}
- Jede Station wird schließlich informiert.
  $\Diamond\ cs = \langle infd, infd, .., infd \rangle$
- Jede gesendete Nachricht wird schließlich empfangen.
  $x \in nts$ ~> $x \notin nts$

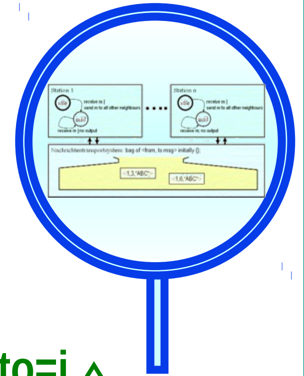
Variablen

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 <S, S_0, Next> kann definiert werden durch

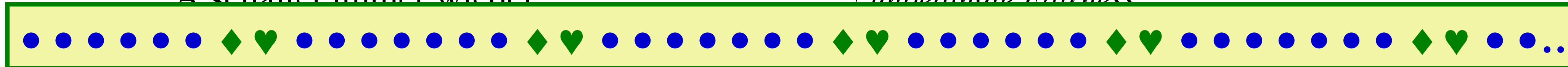
- Menge von Datenvariablen $V_1, V_2, .., V_n$ mit den Wertbereichen $W_1, W_2, .., W_n$
- Initialisierungsprädikat Init über Variablen
- Menge von Aktionsprädikaten $A_1, A_2, .. A_m$ über Variablen und Folgevariablen

- Operatoren für Aktionen
- Aktion: Prädikat A über V und V' notiert Teilmenge von Next
  - <A> — Aktion A schaltet und ist kein Stotterschritt
  - Enabled(A) — Aktion A ist schaltbereit
- Betrachtet werden unendliche Zustandsfolgen: $\sigma = \langle s_0, s_1, s_2, s_3, \ldots \rangle$
- Semantik
  - $\sigma \models \langle A \rangle \Leftrightarrow s_1 \neq s_2 \ \wedge \langle s_0, s_1 \rangle \models A$
  - $\sigma \models Enabled(A) \Leftrightarrow \exists\, x \in S: \langle s_0, x \rangle \models A$

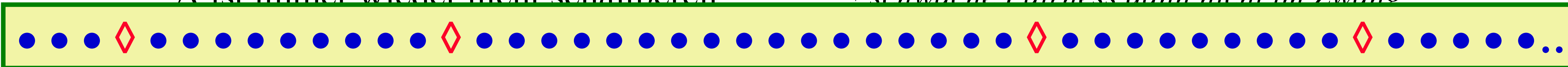
F9: LTL – Aktionenfairness

- Aktionenfairness in LTL

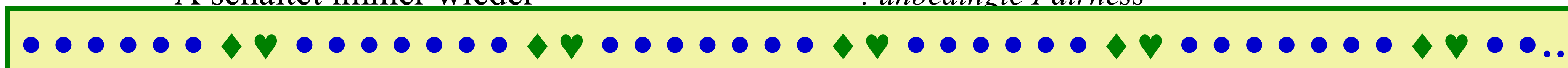
- WF(A) ≡ □◊ <A> ∨ □◊ ¬Enabled(A)
  - A schaltet immer wieder *! unbedingte Fairness*



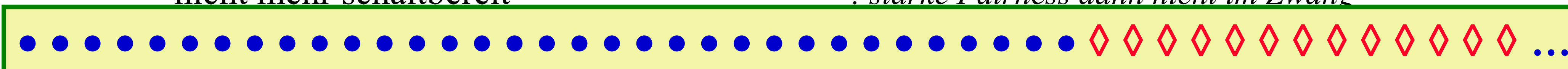
  - A ist immer wieder nicht schaltbereit *! schwache Fairness dann nicht im Zwang*



- SF(A) ≡ □◊ <A> ∨ ◊□ ¬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: $\sigma = < s_0, s_1, s_2, s_3, \ldots >$
- Def: **Eigenschaft Π**

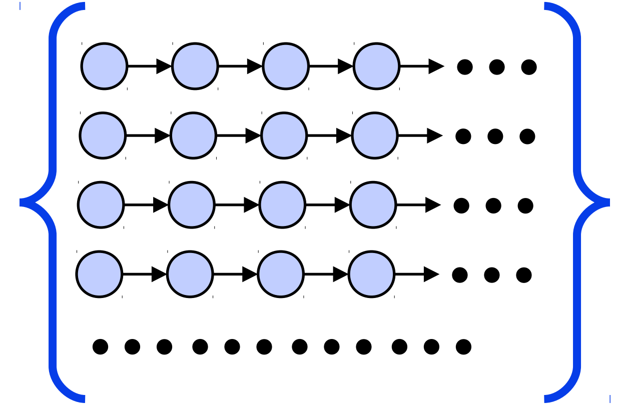
  $\Pi \subset 2^{(S^\infty)}$, d.h. Π ist Menge von Zustandsfolgen
- Def: **STS <S, S₀, Next> besitzt Eigenschaft Π**

  Jede mögliche Systemausführung ist in Π enthalten: $\Sigma \subset \Pi$

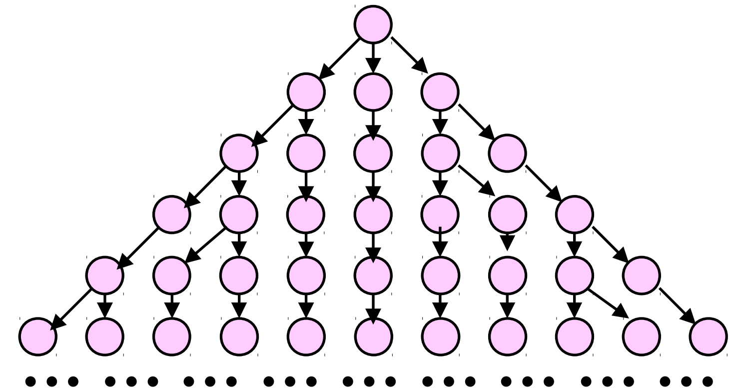
- Def: Von Formel F spezifizierte Eigenschaft
  - Zustandsfolgenmenge $\Pi = \{\sigma : \sigma \models F\}$
- Def: **STS** erfüllt Formel F, $STS \models F$
  - $\Sigma \subset \Pi$

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.

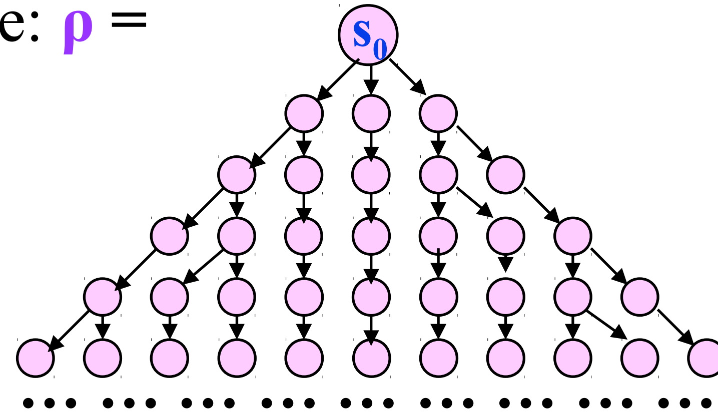


- **CTL:** Betrachtet werden **Bäume**
  - Ein Baum ist ein abgewickelter Erreichbarkeitsgraph
- Branching Time
- CTL unterstützt die explizite Quantifikation über Pfaden
  - **E**: es existiert ein Pfad, so dass
  - **G**: für alle Pfade gilt



F9: Computation Tree Logik – Syntax

- Betrachtet werden unendliche Bäume: ρ =

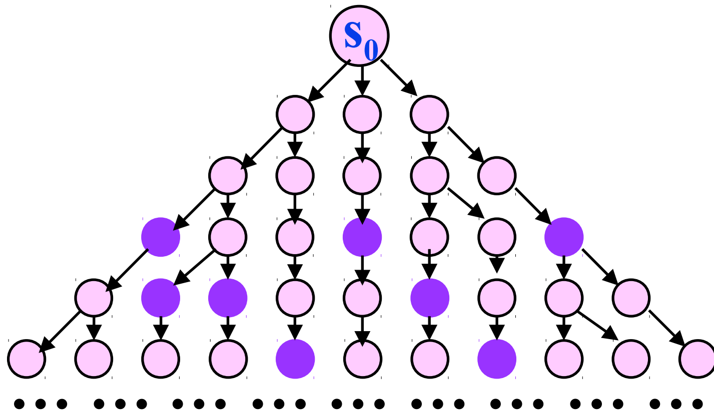


- Syntax (φ, ψ seien CTL–Formeln)
  - Aussage: $a \in AP$, AP Menge der atomaren Aussagen
    true, false, a
  - Aussagenlogische Verknüpfungen:
    $\neg \varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Leftrightarrow \psi$, $\varphi \Rightarrow \psi$
  - Pfadquantifizierte Formeln

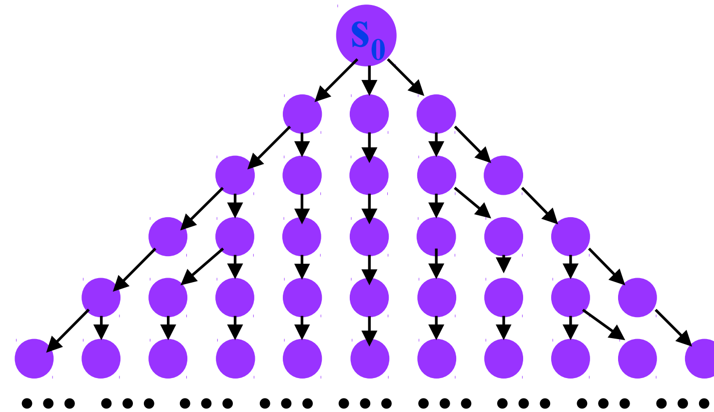
A X φ	all next, für alle Pfade gilt im nächsten Schritt φ
E X φ	exist next, es existiert Pfad, so dass im nächsten Schritt φ
A φ **U** ψ	all until, für alle Pfade gilt φ bis ψ
E φ **U** ψ	exist next, es existiert Pfad, in welchem φ bis ψ gilt
A G φ	all globally, für alle Pfade gilt immer φ
E G φ	exist 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 schießlich: **AF φ**

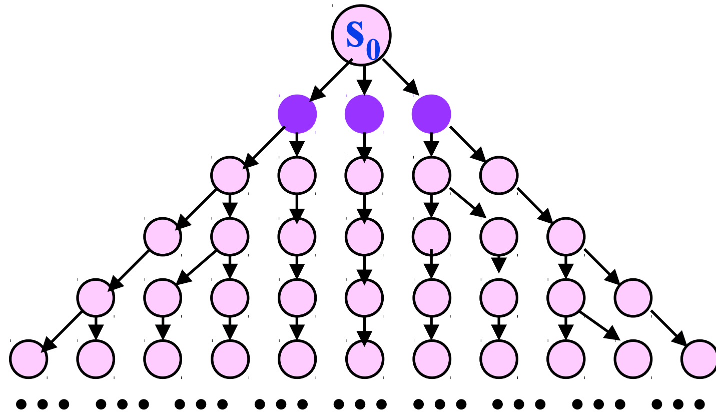


Alle immer: **AG φ**

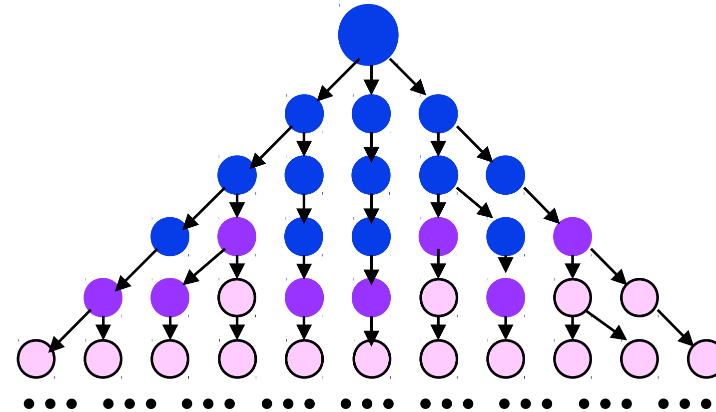


F9: Computation Tree Logik – Formulierungen

Alle next: **AX φ**

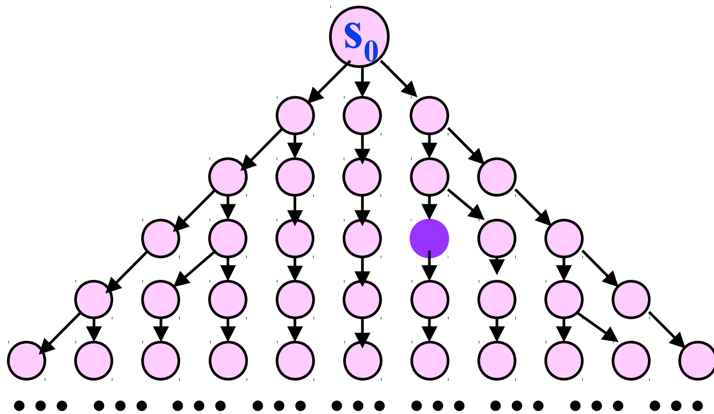


Alle bis: **A φ U ψ**

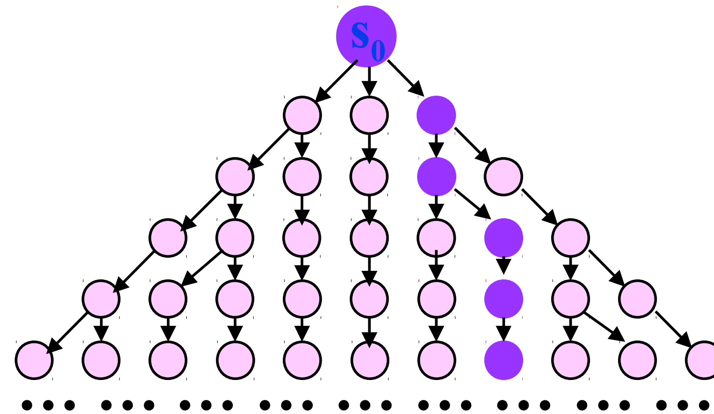


F9: Computation Tree Logik – Formulierungen

Einer schießlich: **EF φ**

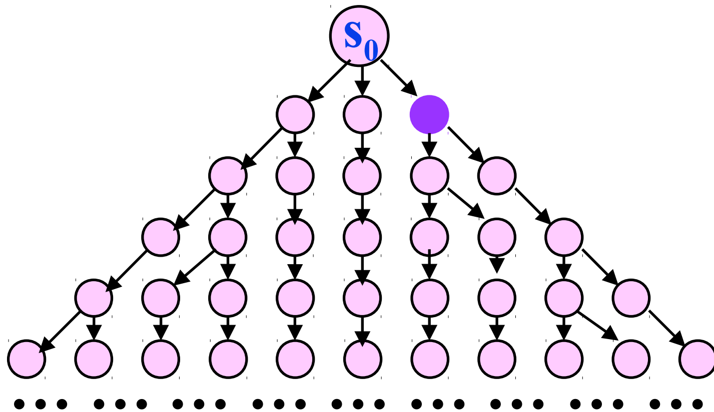


Einer immer: **EG φ**

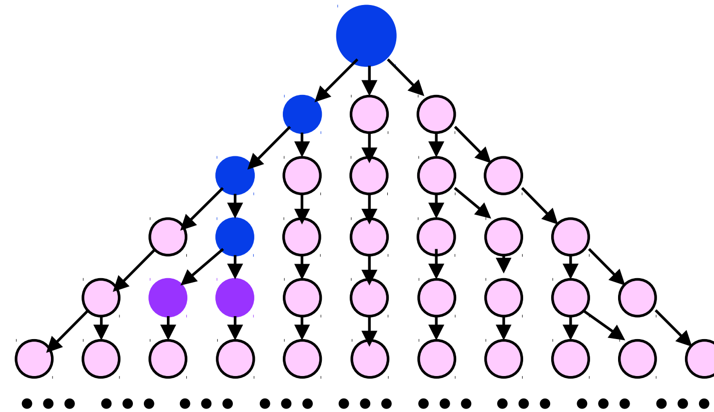


F9: Computation Tree Logik – Formulierungen

Einer next: **EX φ**



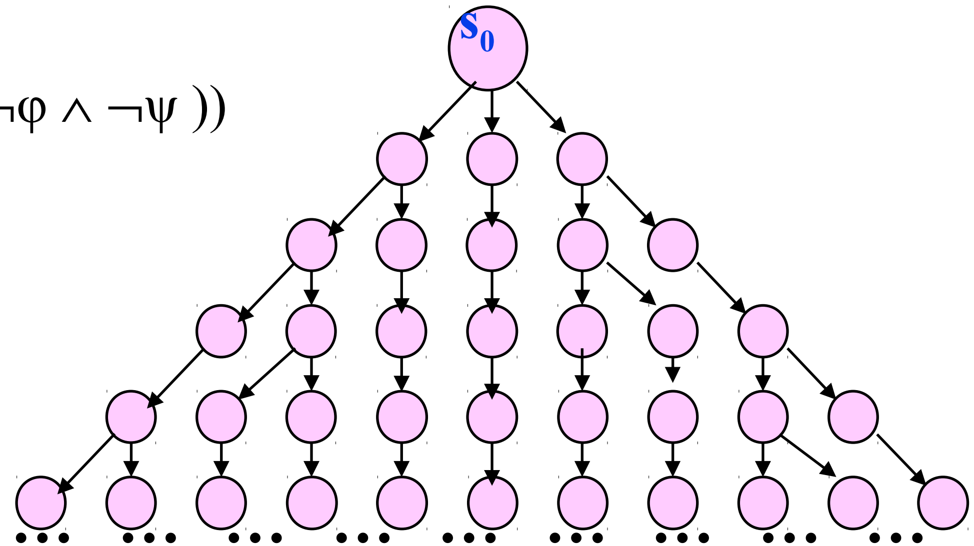
Einer bis: **E φ U ψ**



F9: Computation Tree Logik – Syntax

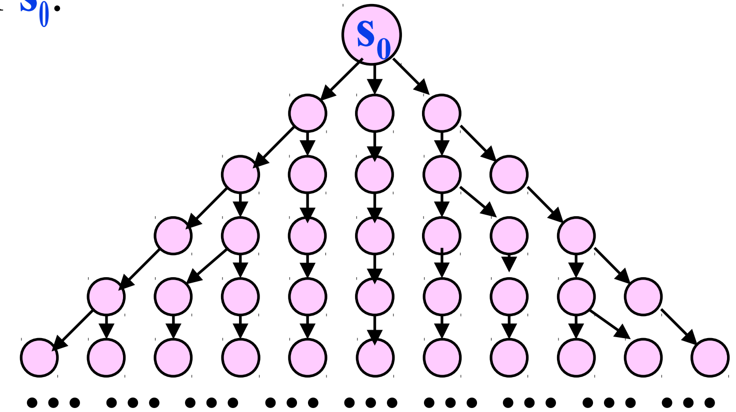
- Alle Pfadquantoren können ausgedrückt werden mittels
  $\mathbf{E\,X}\,\varphi$, $\mathbf{E\,G}\,\varphi$, $\mathbf{E}\,\varphi\,\mathbf{U}\,\psi$

  - $\mathbf{A\,X}\,\varphi \equiv \neg\mathbf{E\,X}\,\neg\varphi$
  - $\mathbf{A\,F}\,\varphi \equiv \neg\mathbf{E\,G}\,\neg\varphi$
  - $\mathbf{E\,F}\,\varphi \equiv \mathbf{E}\,\text{true}\,\mathbf{U}\,\varphi$
  - $\mathbf{A\,G}\,\varphi \equiv \neg\mathbf{E\,F}\,\neg\varphi \equiv \neg\mathbf{E}\,\text{true}\,\mathbf{U}\,\neg\varphi$
  - $\mathbf{A}\,\varphi\,\mathbf{U}\,\psi \equiv \neg\mathbf{E\,G}\,\neg\psi \wedge \neg\mathbf{E}\,(\neg\psi\,\mathbf{U}\,(\neg\varphi \wedge \neg\psi\,))$



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 $\sigma = \langle s_0, s_1, s_2, s_3, \ldots \rangle$.
- 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

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

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

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

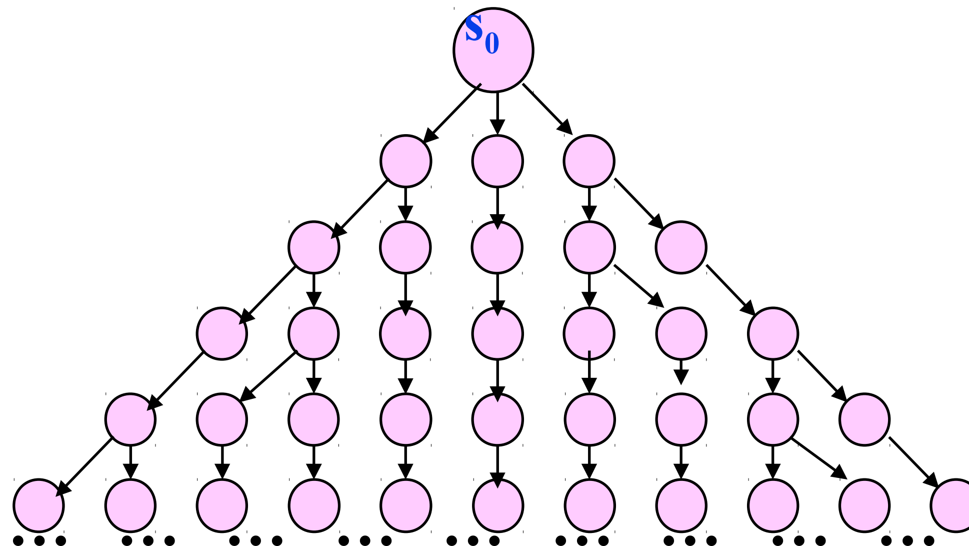
    $\rho, s_0 \models \mathbf{E}\, \varphi\, \mathbf{U}\, \psi \Leftrightarrow \exists \sigma \in \Sigma(s_0): \exists\, j: \rho, s_j \models \psi \wedge \forall i, 0 \leq i < j: \rho, s_i \models \varphi$

F9: Computation Tree Logik – Beispielaussagen

Safety

„*Etwas Schlechtes wird nicht passieren*“

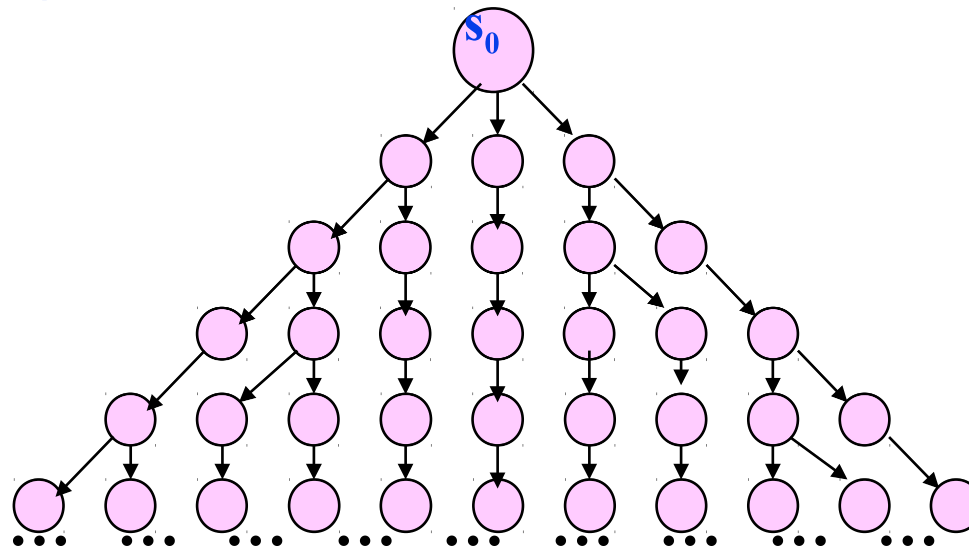
- $\boxtimes$ **A G** ¬ (TürOffen ∧ ¬KabineDa)
- **A G** ¬(RichtungAGrün ∧ **A X** RichtungBGrün)
- **A G** ¬(x=0 ∧ **A X** y=z/x)



Liveness

„*Etwas Gutes wird nach endlicher Zeit passieren*“

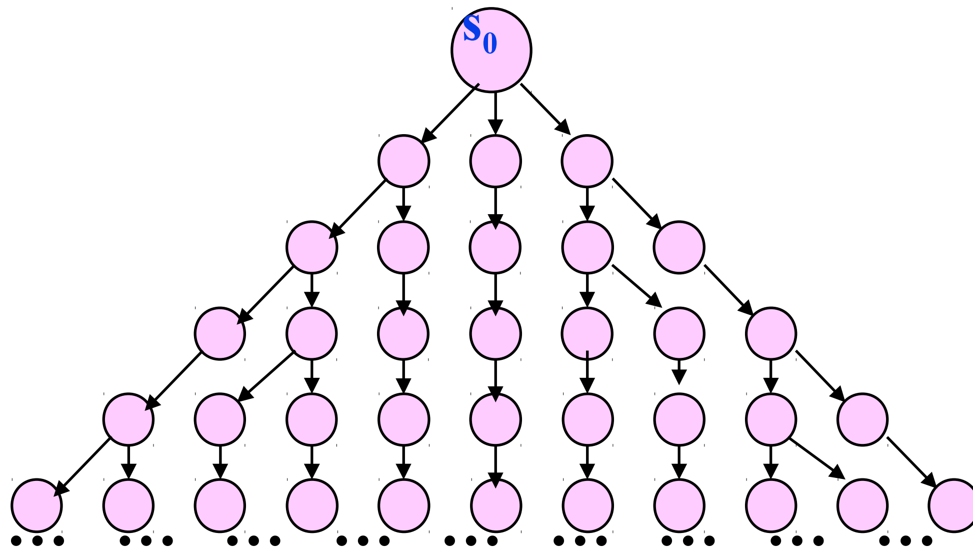
- **A F** (TürOffen ∧ KabineDa)
- **A F** RichtungAGrün
- **A G** (start ⇒ **A F** terminiert)



Scheduling Fairness

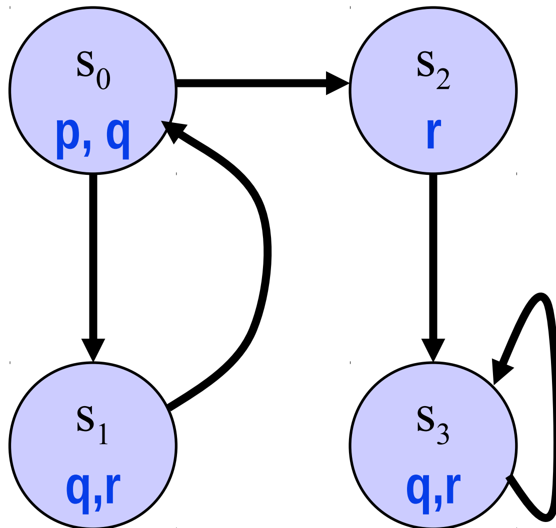
„*Etwas Gutes wird unendlich oft ermöglicht*“

- **A G (A F** AufzugKannNeuenAuftragAnnehmen)



F9: Computation Tree Logik – Beispielaussagen

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

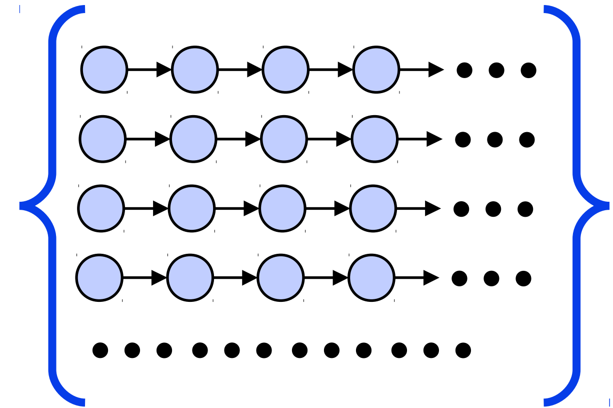


Gelten folgende Formeln für Abläufe mit bestimmtem Startzustand ?

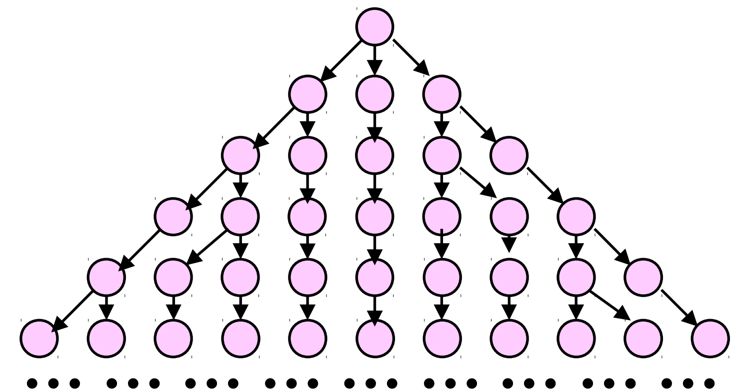
1. s_0: **EX(¬ p)**
2. s_0: **EXEG(r)**
3. s_1: **AG(q ∨ r)**
4. s_2: **A(r U q)**
5. s_1: **A(q U AG(r))**
6. s_1: **E(q U EG(r))**
7. s_0: **¬EG(q)**
8. s_1: **EFAG(q)**

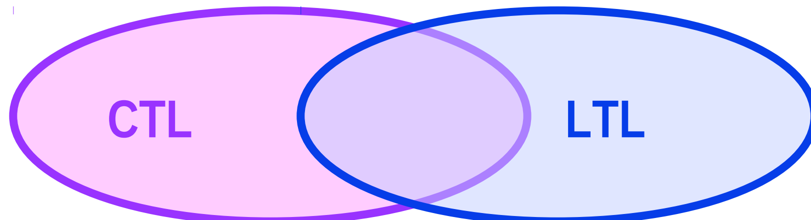
F9: LTL – CTL

- **LTL:** Betrachtet werden **Mengen von Zustandsfolgen**
- **Linear Time**
- **CTL:** Betrachtet werden **Bäume**
- Branching Time

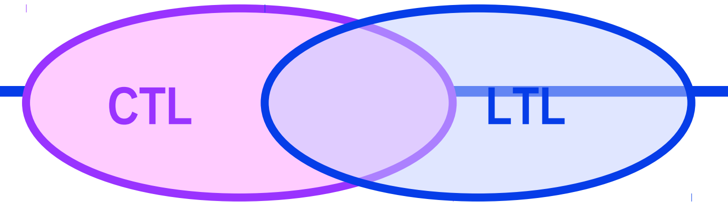


- Es gibt LTL Formeln, die nicht in CTL ausgedrückt werden können
- Es gibt CTL Formeln, die nicht in LTL ausgedrückt werden können

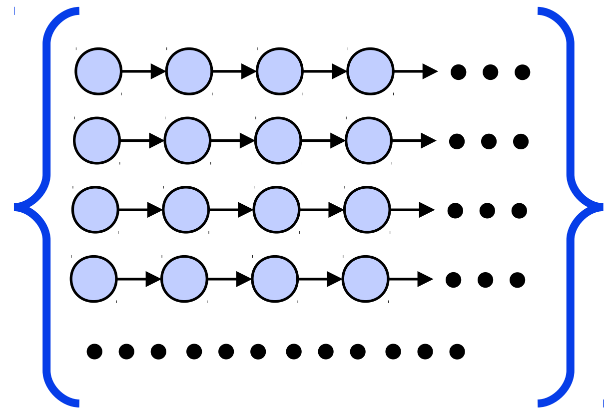




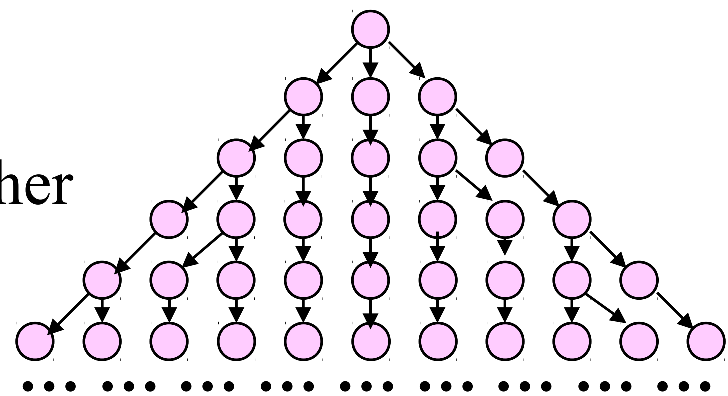
F9: LTL – CTL



- Es gibt LTL Formeln, die nicht in CTL ausgedrückt werden können (z.B. Formeln, die sich auf auf spezielle Teilpfade beziehen)
  - $\Box\Diamond\langle a\rangle \vee \Diamond\Box$ Disabled(a)

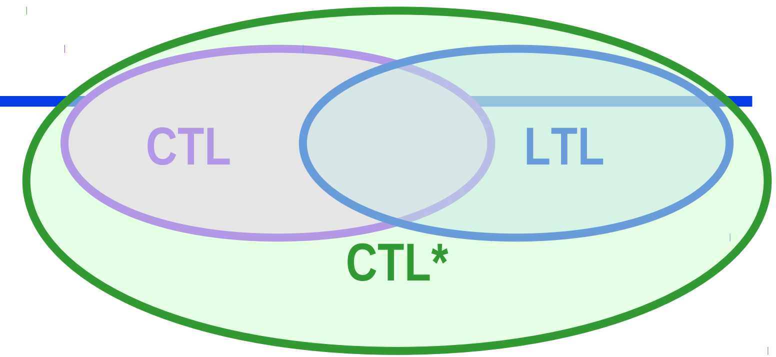


- Es gibt CTL Formeln, die nicht in LTL ausgedrückt werden können (z.B. Formeln mit existenzquantifizierten Pfaden)
  - $A\,G\,(\varphi \Rightarrow E\,F\,\psi)$

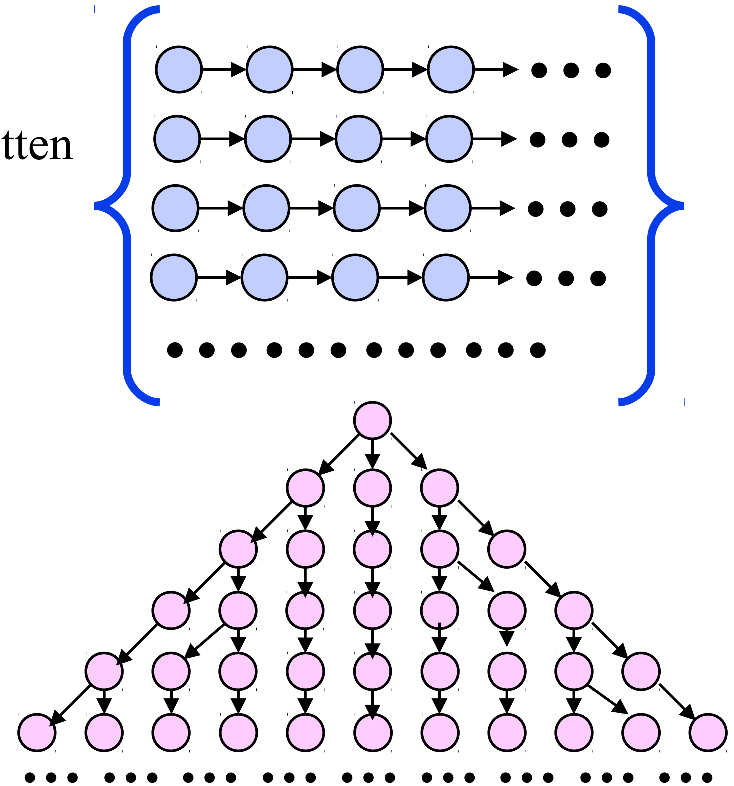


- Es gibt LTL- CTL-Formeln, die sich entsprechen (z.B. LTL-Formeln mit einfacher Schachtelung temporaler Operatoren)
  - $\Box \neg \varphi$ — $AG \neg \varphi$
  - $\Diamond \varphi$ — $AF\,\varphi$
  - $\Box (\varphi \Rightarrow \Diamond \psi)$ — $AG\,(\varphi \Rightarrow AF\,\psi)$
  - $\Box\Diamond \varphi$ — $AGF\,\varphi$

F9: LTL+CTL: CTL*

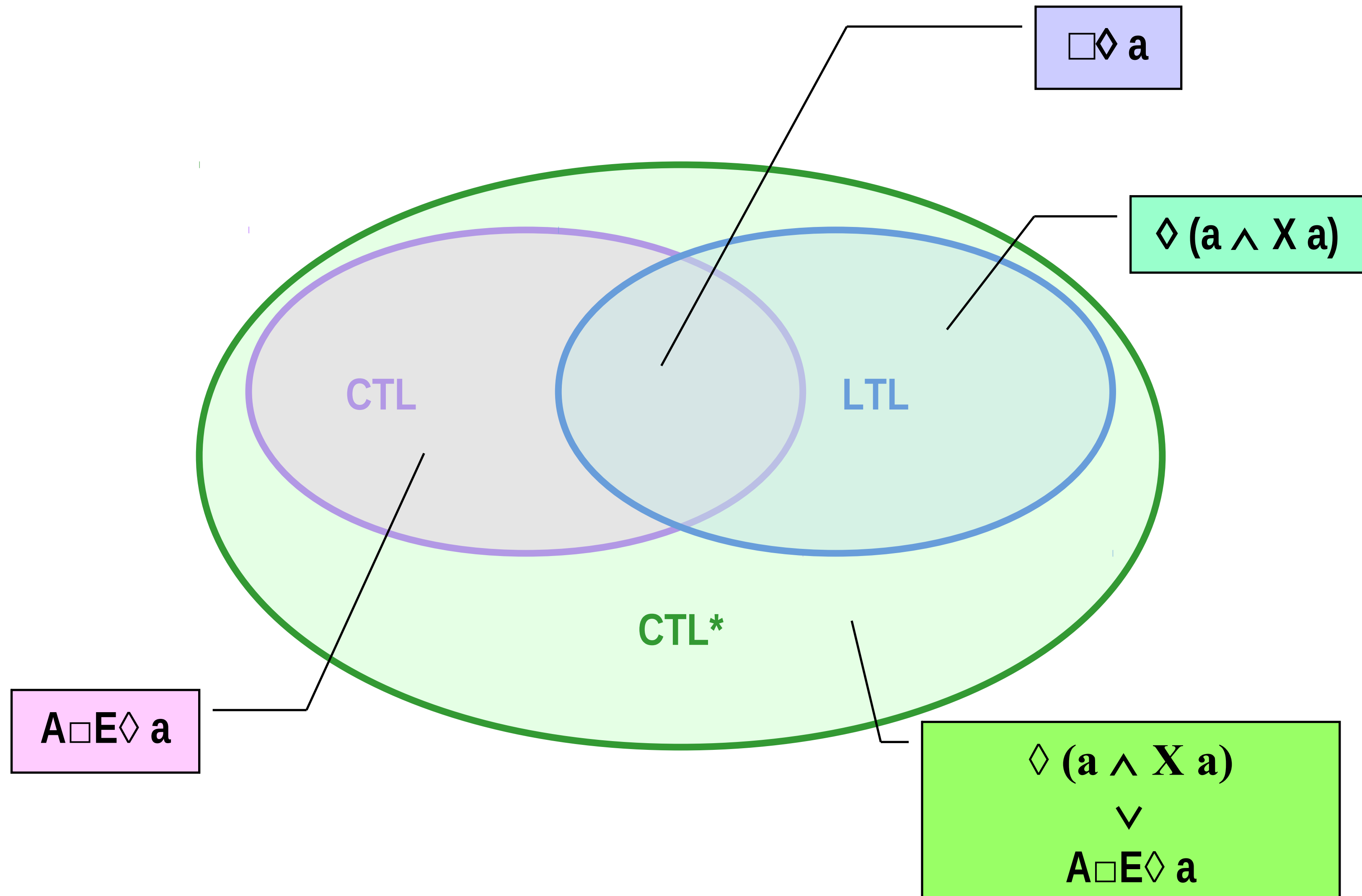


- CTL* kombiniert LTL und CTL
  - die temporalen Operatoren können in CTL* annähernd beliebig eingesetzt werden
  - Beispiele
    - » A (X φ ∨ X X φ)
      In allen Pfaden gilt φ im zweiten oder dritten Zustand
    - » E (□◊ φ)
      Es gibt einen Pfad, in welchem φ immer wieder gilt
  - φ ist Formel in CTL
    φ ist Formel in CTL*
  - φ ist Formel in LTL
    A φ 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, $true$, $false$, $\neg\, \varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Leftrightarrow \psi$, $\varphi \Rightarrow \psi$

    $A\, \Phi$, $E\, \Phi$
  - Pfadformeln

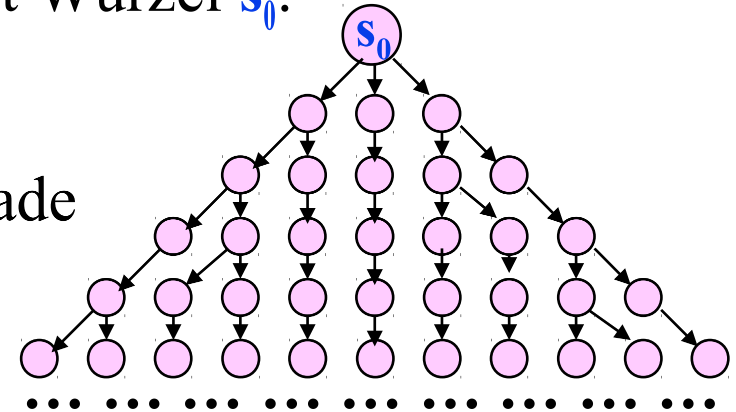
    φ

    $\neg\, \Phi$, $\Phi \wedge \Psi$, $\Phi \vee \Psi$, $\Phi \Leftrightarrow \Psi$, $\Phi \Rightarrow \Psi$

    $X\, \Phi$, $\Box\Phi$, $\Diamond\Phi$, $\Phi\, U\, \Psi$

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:

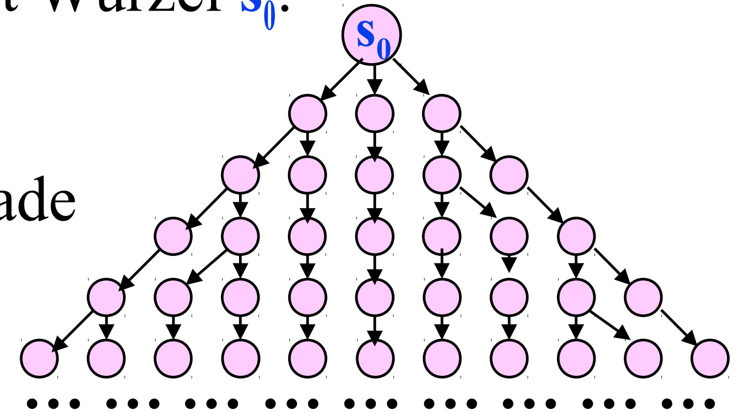
    $\neg \varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Leftrightarrow \psi$, $\varphi \Rightarrow \psi$ *wie üblich*
  - Pfadquantifizierte Formeln

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

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

F9: CTL* – Semantik: Pfadformeln

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

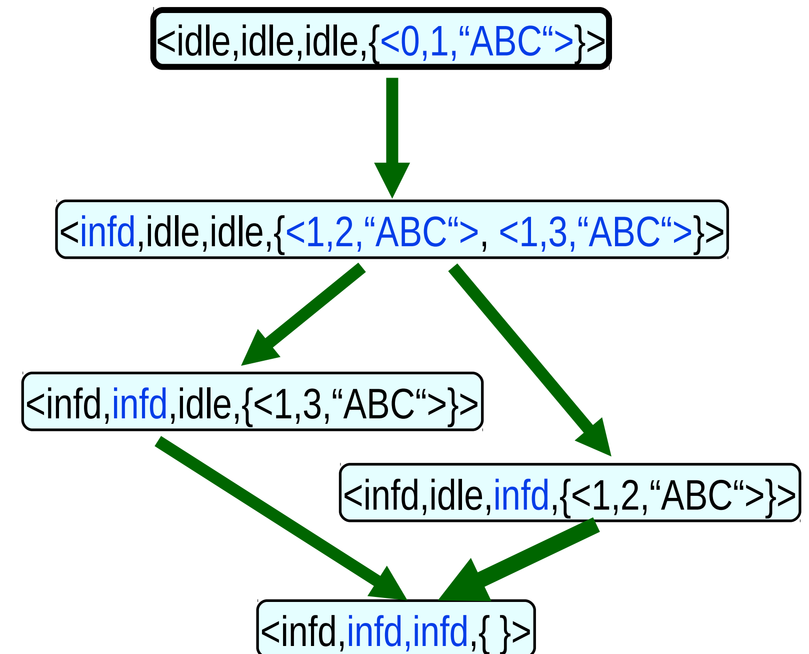


- $\Sigma(s_0)$ sei die Menge der in s_0 startenden Pfade der Form $\sigma = <s_0, s_1, s_2, s_3, \ldots >$.
- φ, ψ seien Zustandsformeln
  Φ, Ψ seien Pfadformeln
  a sei eine atomare Aussage
- Semantik („$\models$“ steht für „erfüllt“)
  - $\sigma \models \varphi \Leftrightarrow \rho, s_0 \models \varphi$
  - 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: $<s_i, s_{i+1}, s_{i+2}, s_{i+3}, \ldots > \models \varphi$

    $\sigma \models \Box \varphi \Leftrightarrow$
    $\forall$ i: $<s_i, s_{i+1}, s_{i+2}, s_{i+3}, \ldots > \models \varphi$

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

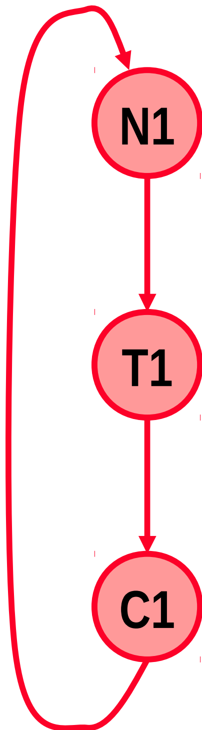
- Gegeben:
  Repräsentation eines (unendlichen) Baums ρ mit Wurzel s_0
- Gegeben: CTL-Formel φ
- Gesucht: Gilt $\rho, s_0 \models \varphi$?

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

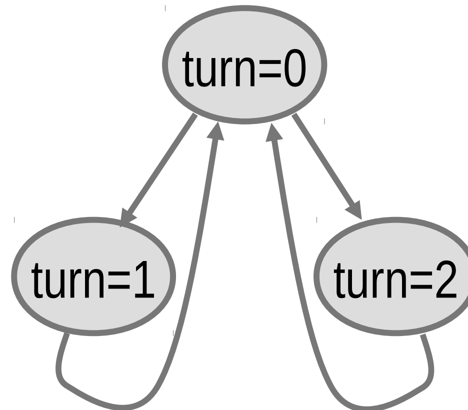
- Safety: **AG** φ
  - Erfüllen alle erreichbaren Zustände φ
- Liveness: **AG(**$\varphi \Rightarrow$ **AF** ψ**)**
  - Erfüllen alle Pfade die Bedingung $\varphi \rightsquigarrow \psi$

Model Checking: Beispiel „Gegenseitiger Ausschluss“

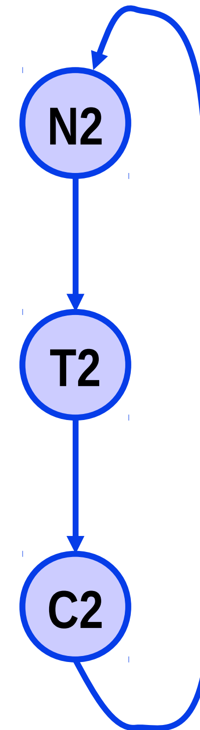
Prozess 1



Gegenseitiger Auschluss: Verwalter



Prozess 2

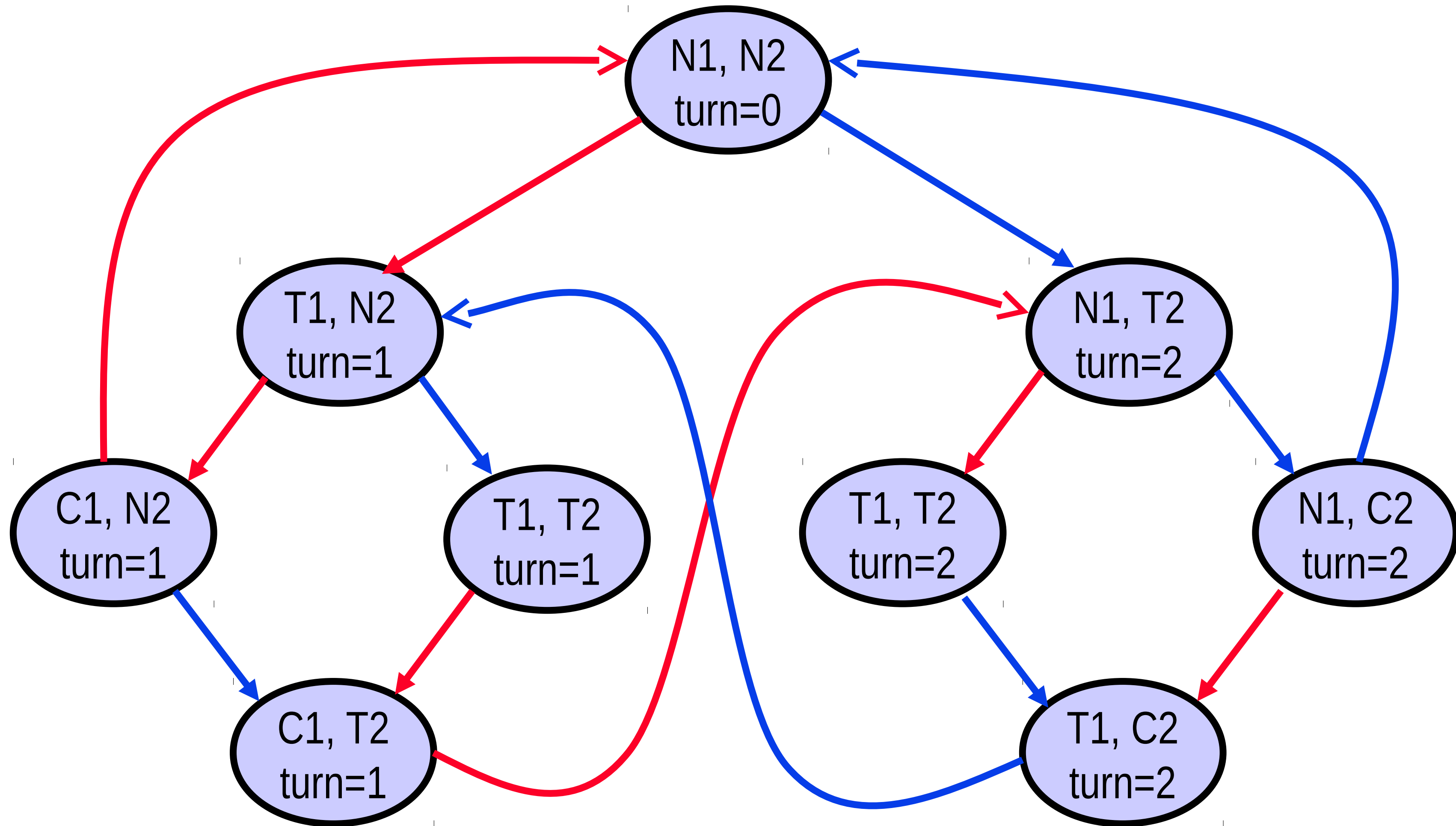


N: Nicht kritisch

T: Anfordernd

C: Kritisch

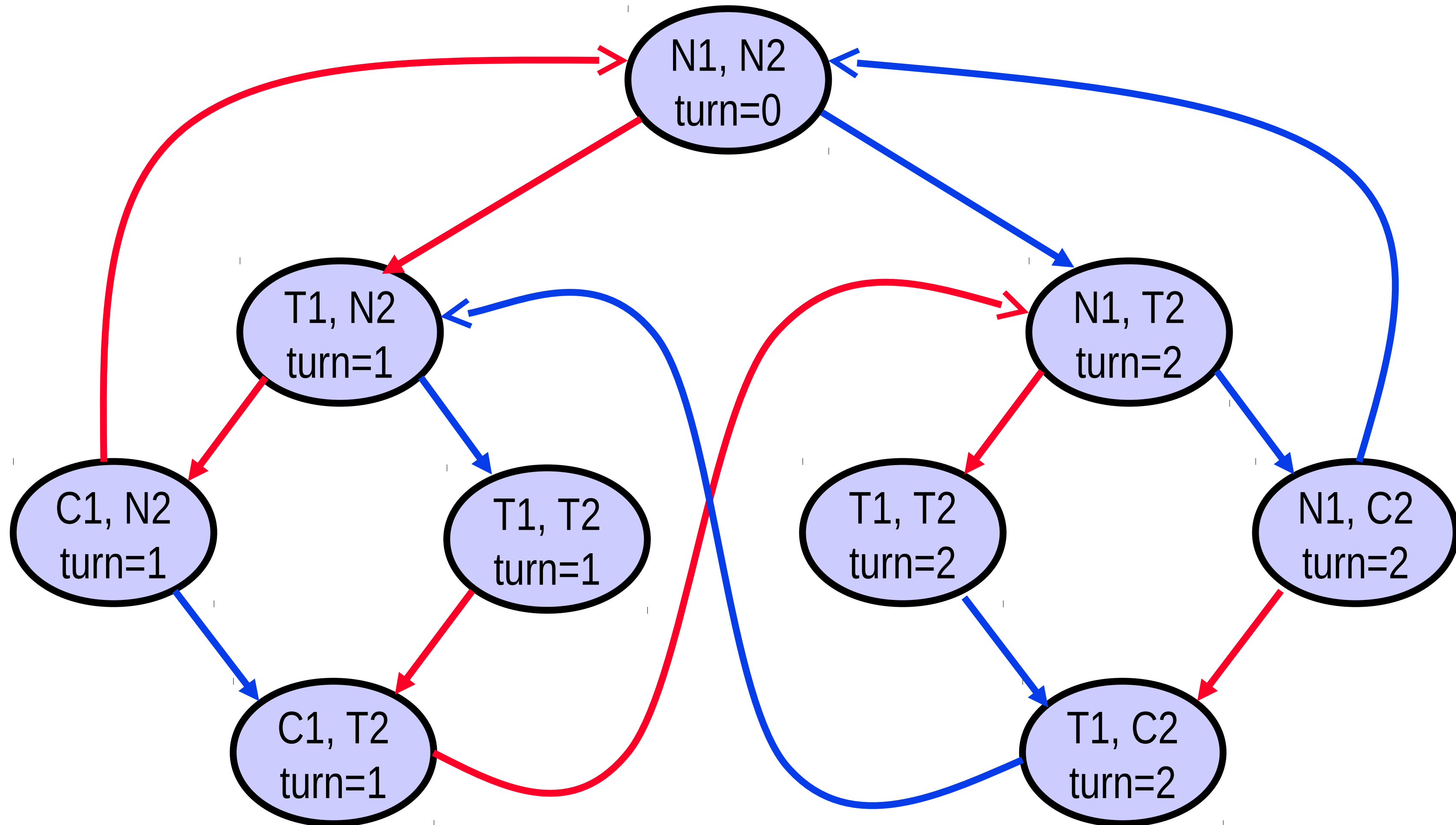
LTL Model Checking: Safety am Beispiel



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

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

LTL Model Checking: Liveness am Beispiel

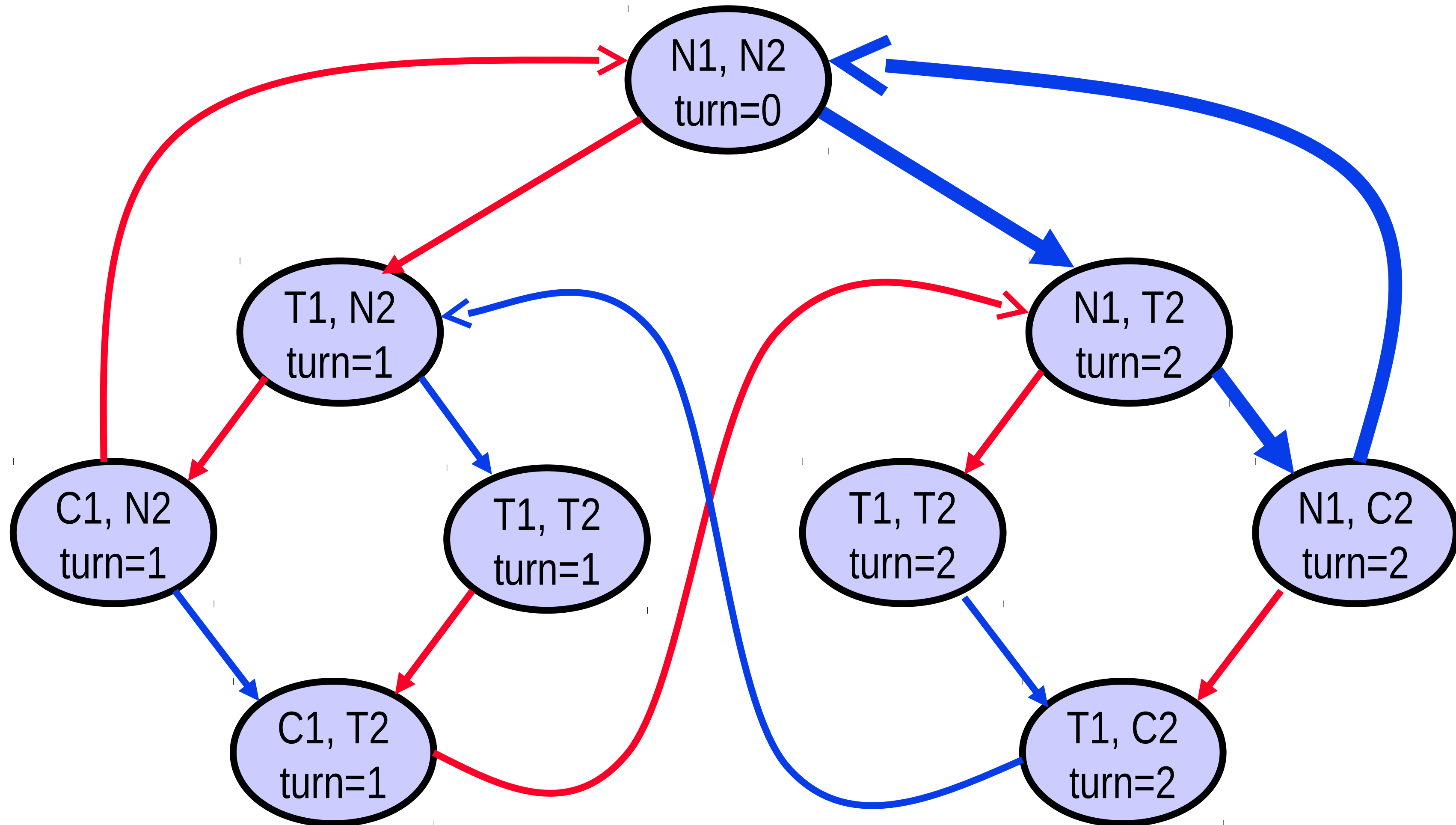


Gilt: T1 ~> C1

„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

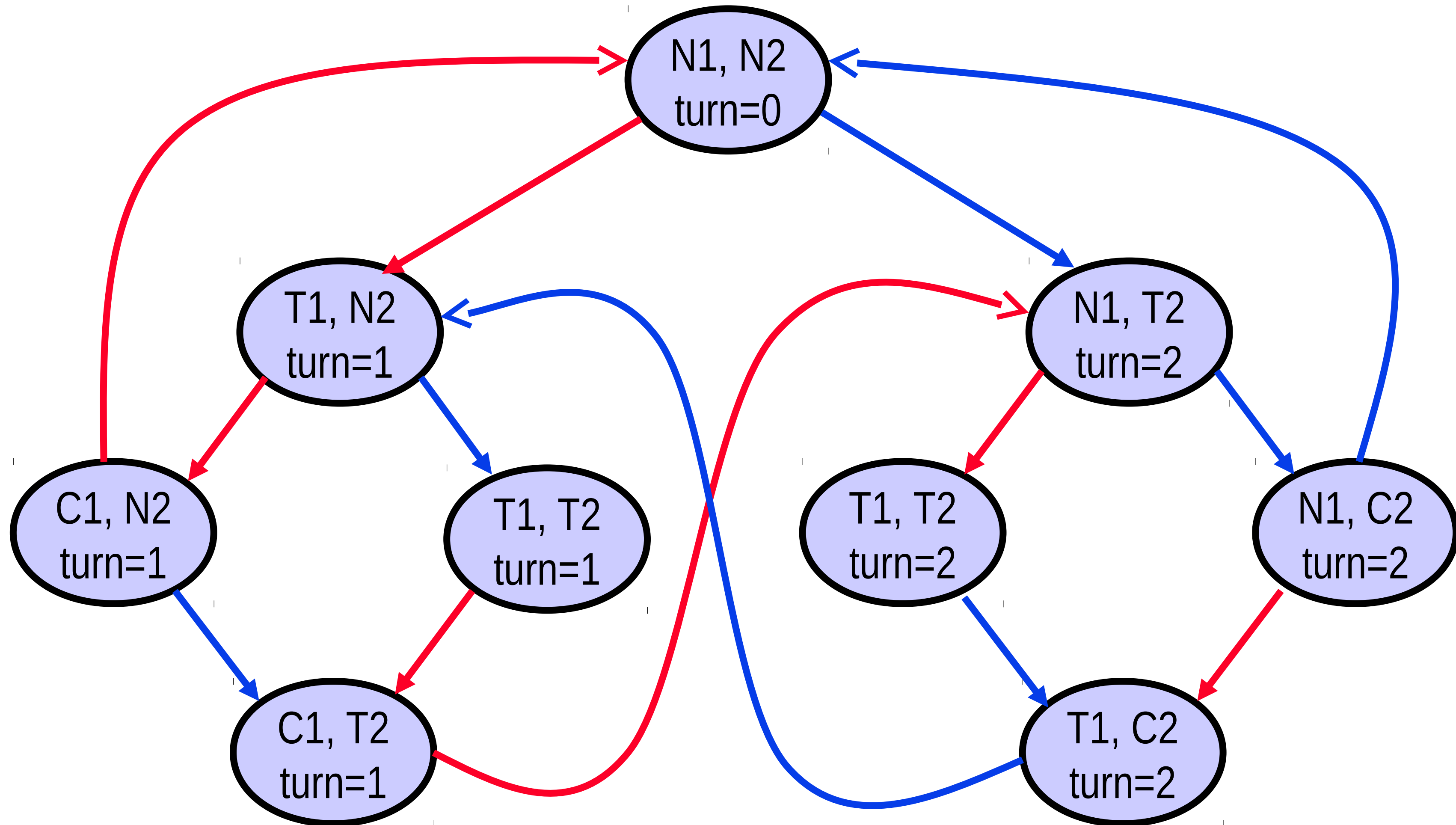


Gilt: □◊ C1

„immer wieder C1"

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

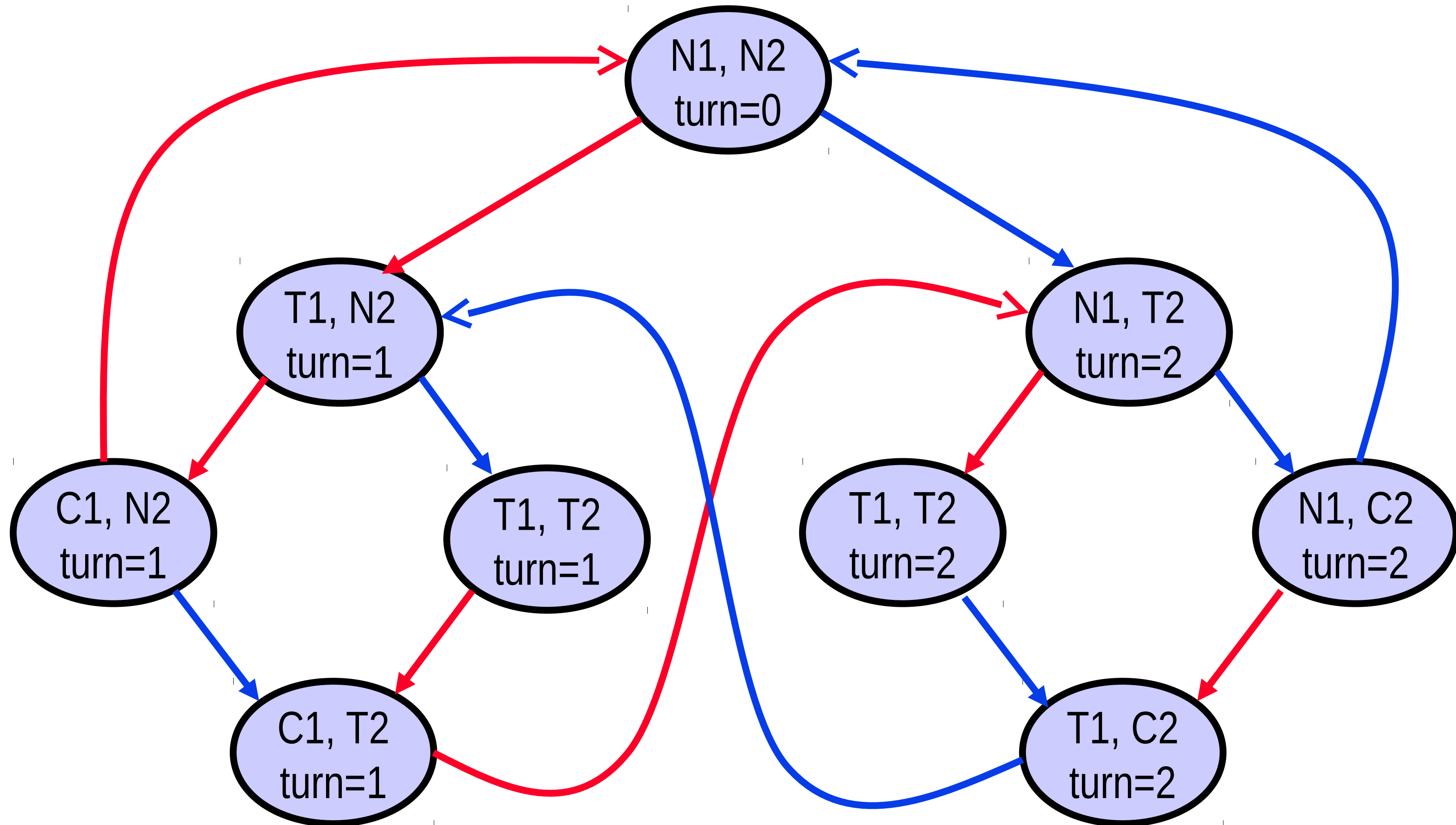
CTL Model Checking: Safety am Beispiel



Gilt: AG¬(C1∧C2) ?

Ja: Alle erreichbaren Zustände erfüllen „¬(C1∧C2)“

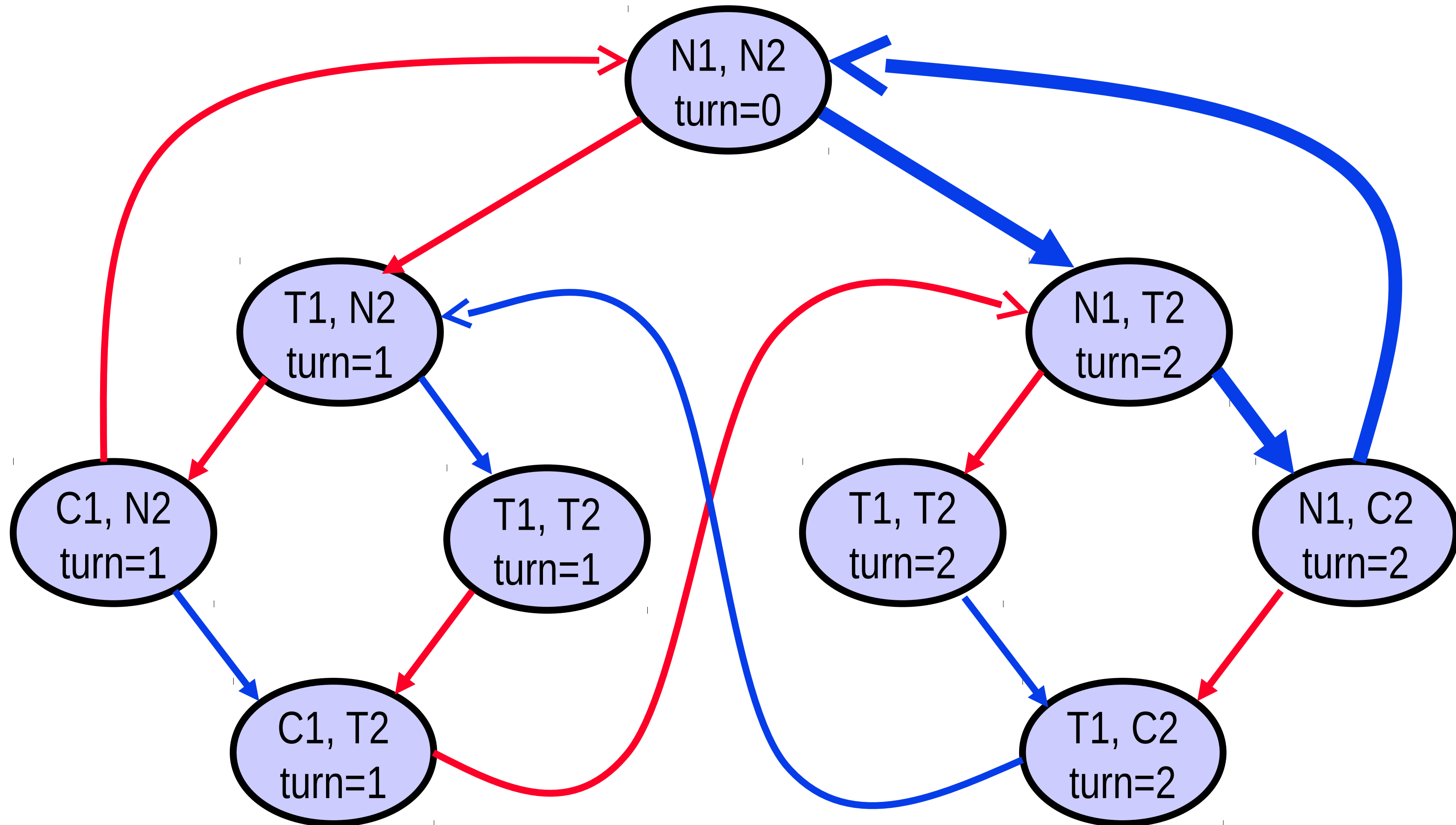
CTL Model Checking: Liveness am Beispiel



Gilt: AG(T1⇒ 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

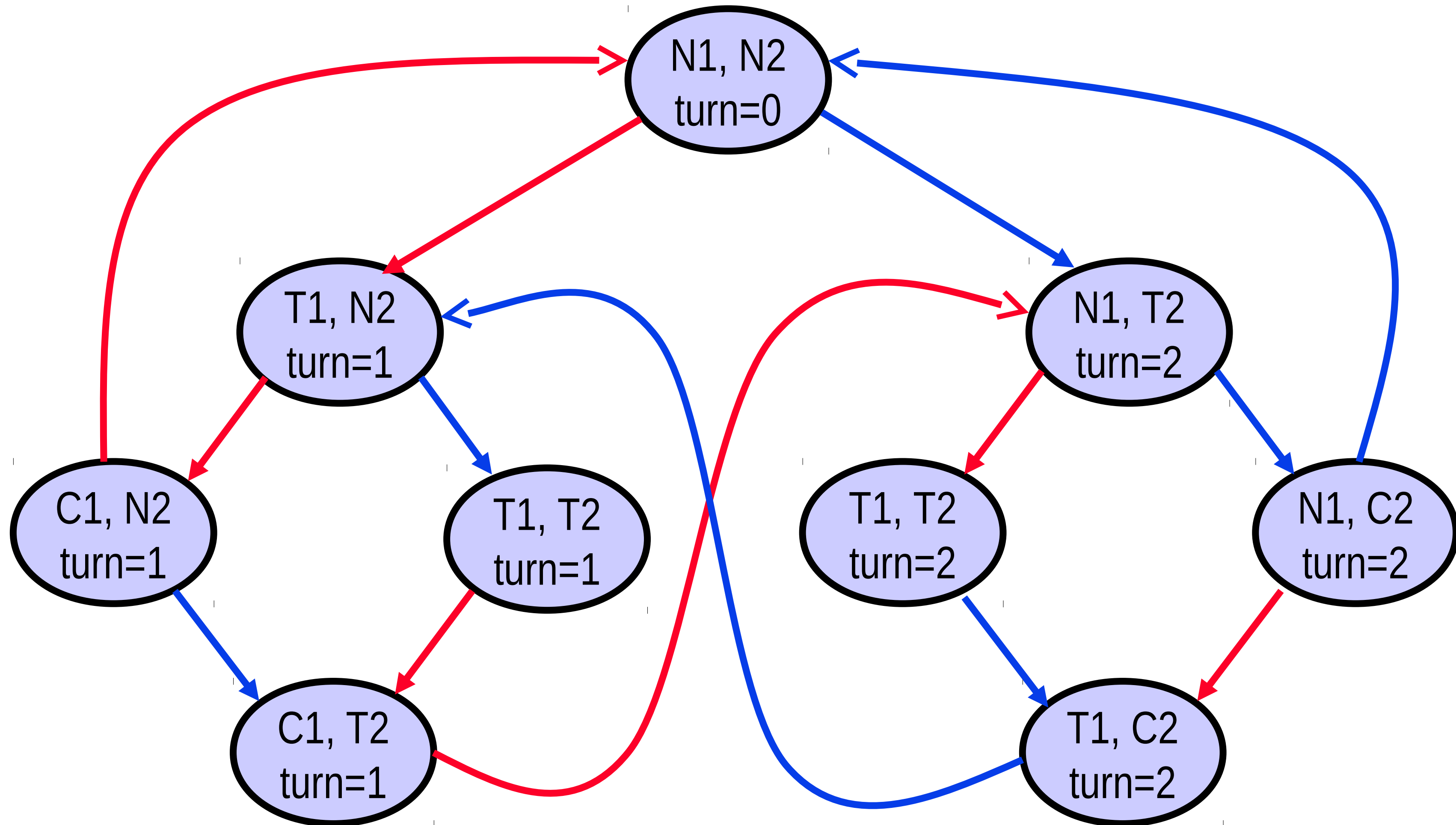


Gilt: AGAF C1

„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



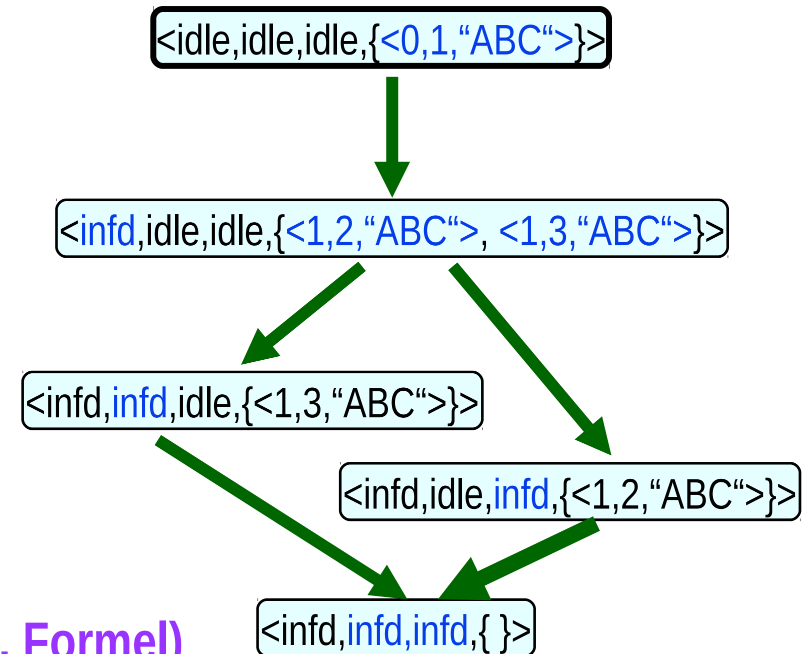
Gilt: AG(N1 ⇒ EF T1)

„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



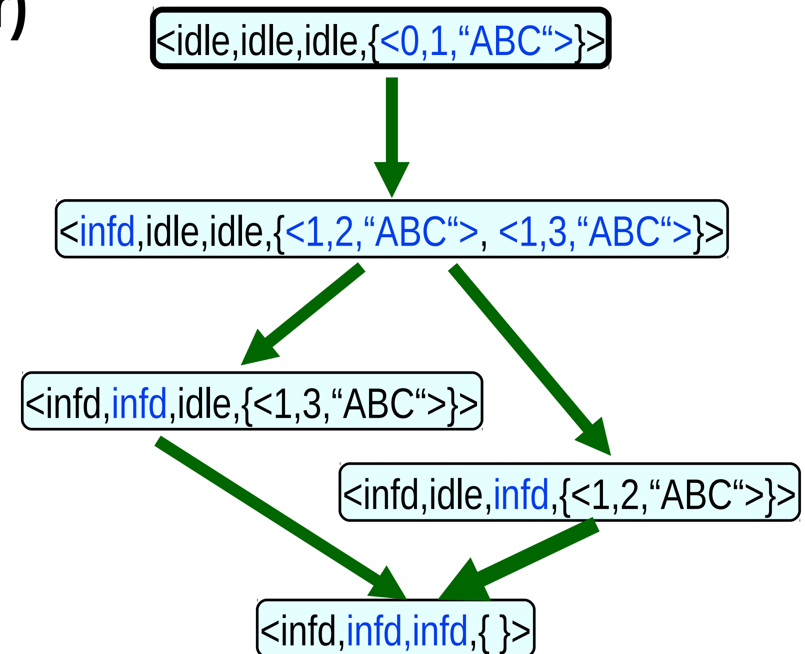
- Weitere Stufe:
  On-the-Fly-Model-Checking
  Erreichbarkeitsgraph wird
  nicht vollständig gespeichert.

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. |φ|	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 System zustände.
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

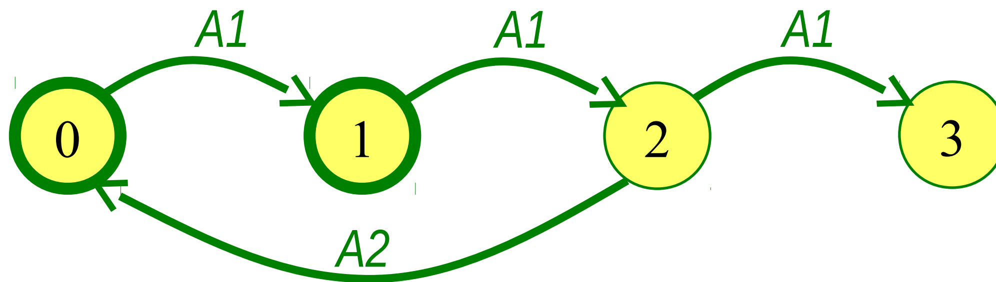
- Livenessbeweise, Regeln
  - Schon da
  - Ein WF-Schritt
  - Ein SF-Schritt
  - Transitivität
  - Lattice-Regel

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.



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 ;

Beispiel
„Dem Zustand 3 geht immer Zustand 2 voraus"

I3 $V=3 \Rightarrow H=2$

F11: Safetybeweise – Induktion über Ablauf

- gegeben: **STS**, φ
  (**STS** per Variablen, Init und Aktionen α_1, .., α_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→n+1
    $\varphi \wedge (\alpha_1 \vee \ldots \vee \alpha_n) \Rightarrow \varphi'$
    - » 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 $\alpha_1, .., \alpha_n$ definiert)
- gesucht: Beweis für „φ ist Zustandsinvariante (es soll gelten $\Box\varphi$)"
- Verfahren: Induktion über Ablauf des Systems
  - n=0
    Init $\Rightarrow \varphi$
  - n→n+1
    $\varphi \wedge (\alpha_1 \vee \ldots \vee \alpha_n) \Rightarrow \varphi'$

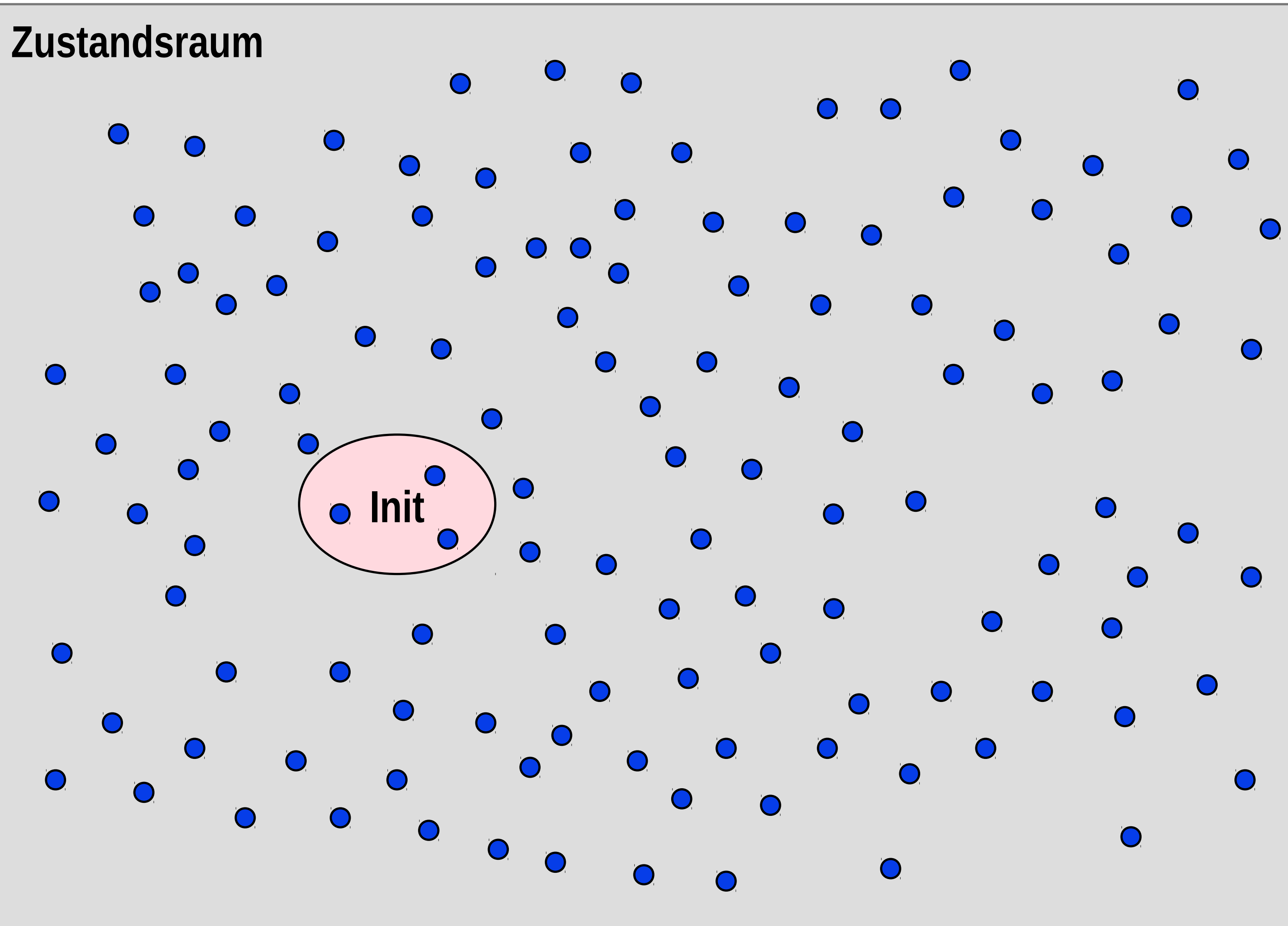
Induktionsfolgerung

Induktionsvoraussetzung

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

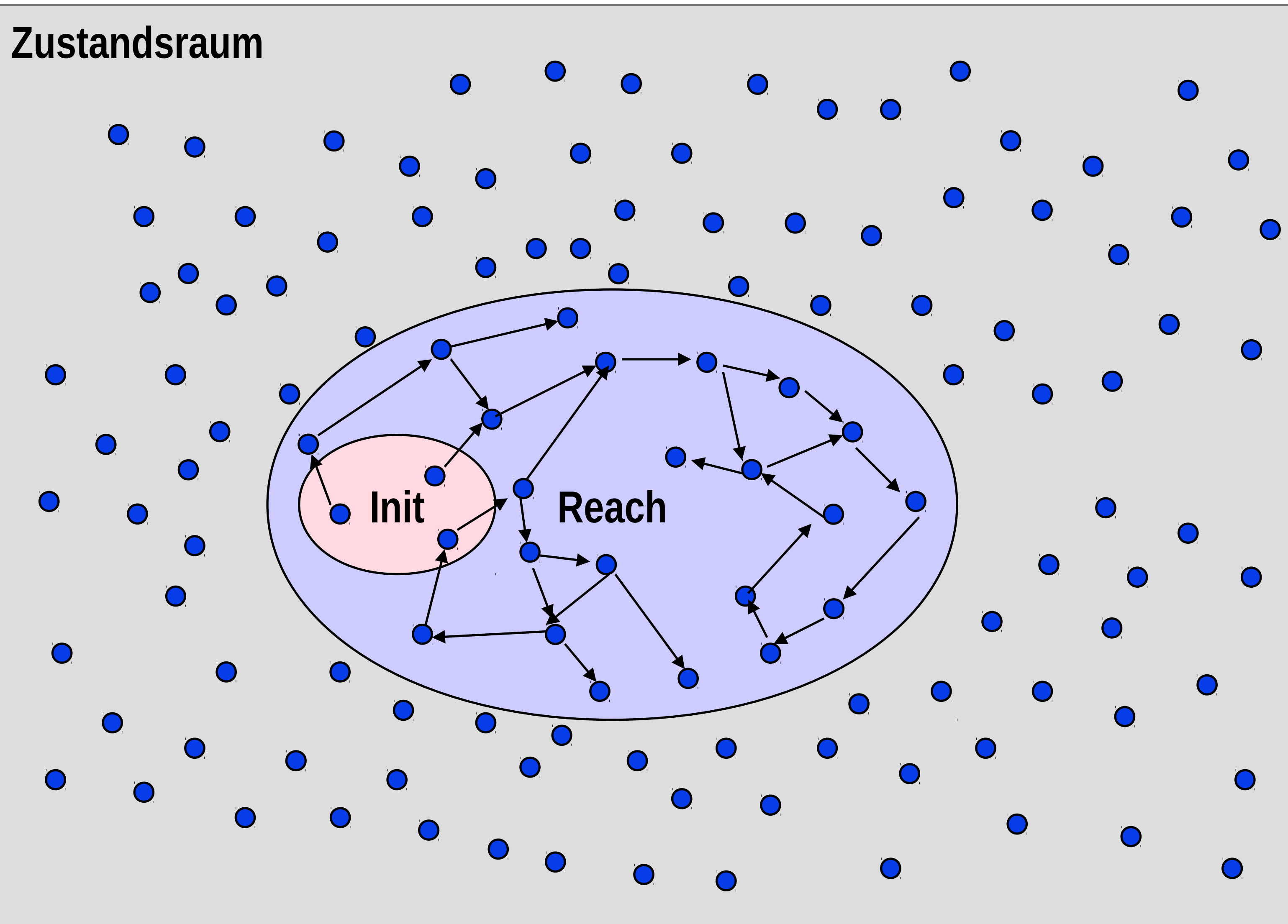
F11: Safetybeweise – Induktion über Ablauf



Zustandsraum enthält die kombinatorisch möglichen Zustände

davon sind die Initialzustände unbedingt direkt erreichbar

F11: Safetybeweise – Induktion über Ablauf



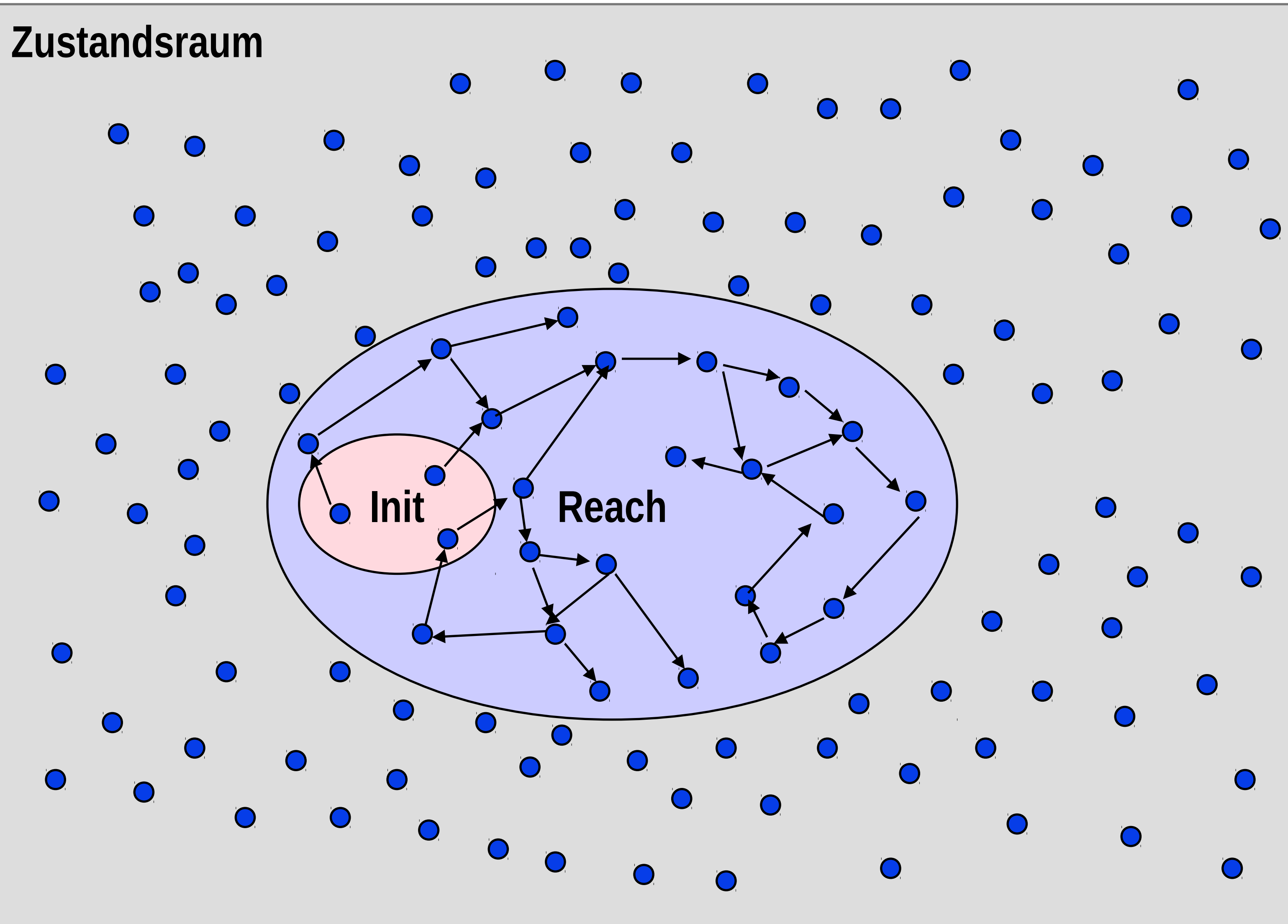
Zustandsraum enthält die kombinatorisch möglichen Zustände

davon sind die Initialzustände unbedingt direkt erreichbar

davon aus die per Transitionen erreichbaren etc.

➔ **Reach**: erreichbare Zustände

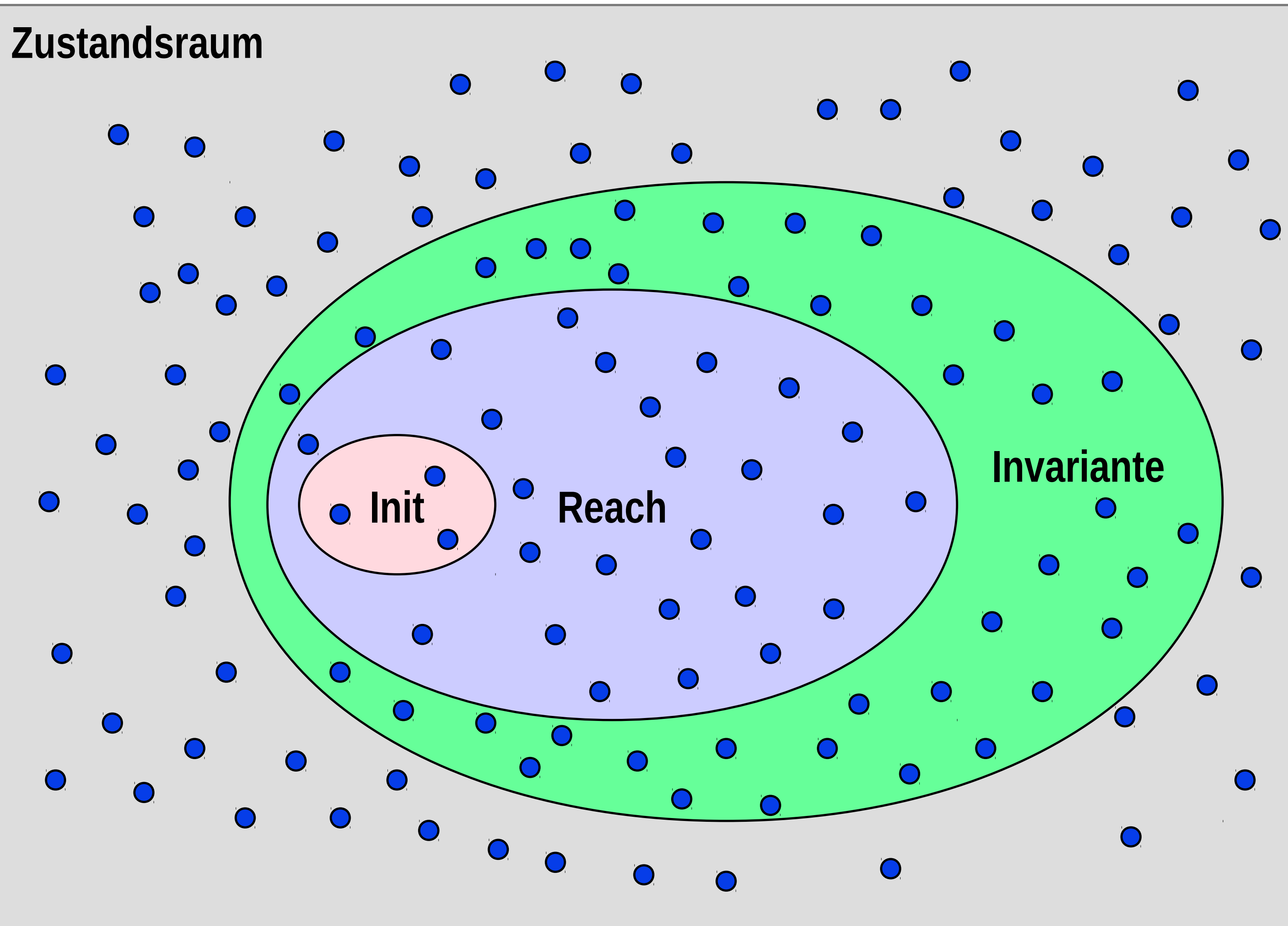
F11: Safetybeweise – Induktion über Ablauf



Beachte:
Während eines beliebigen Systemablaufs gilt immer $s \in$ Reach.
Die Bedingung $s \in$ Reach ist **Invariante**.

Beachte:
Es gibt keine Transitionen, die aus Reach hinausführen!
Die Bedingung $s \in$ Reach ist **induktiv**.

F11: Safetybeweise – Induktion über Ablauf

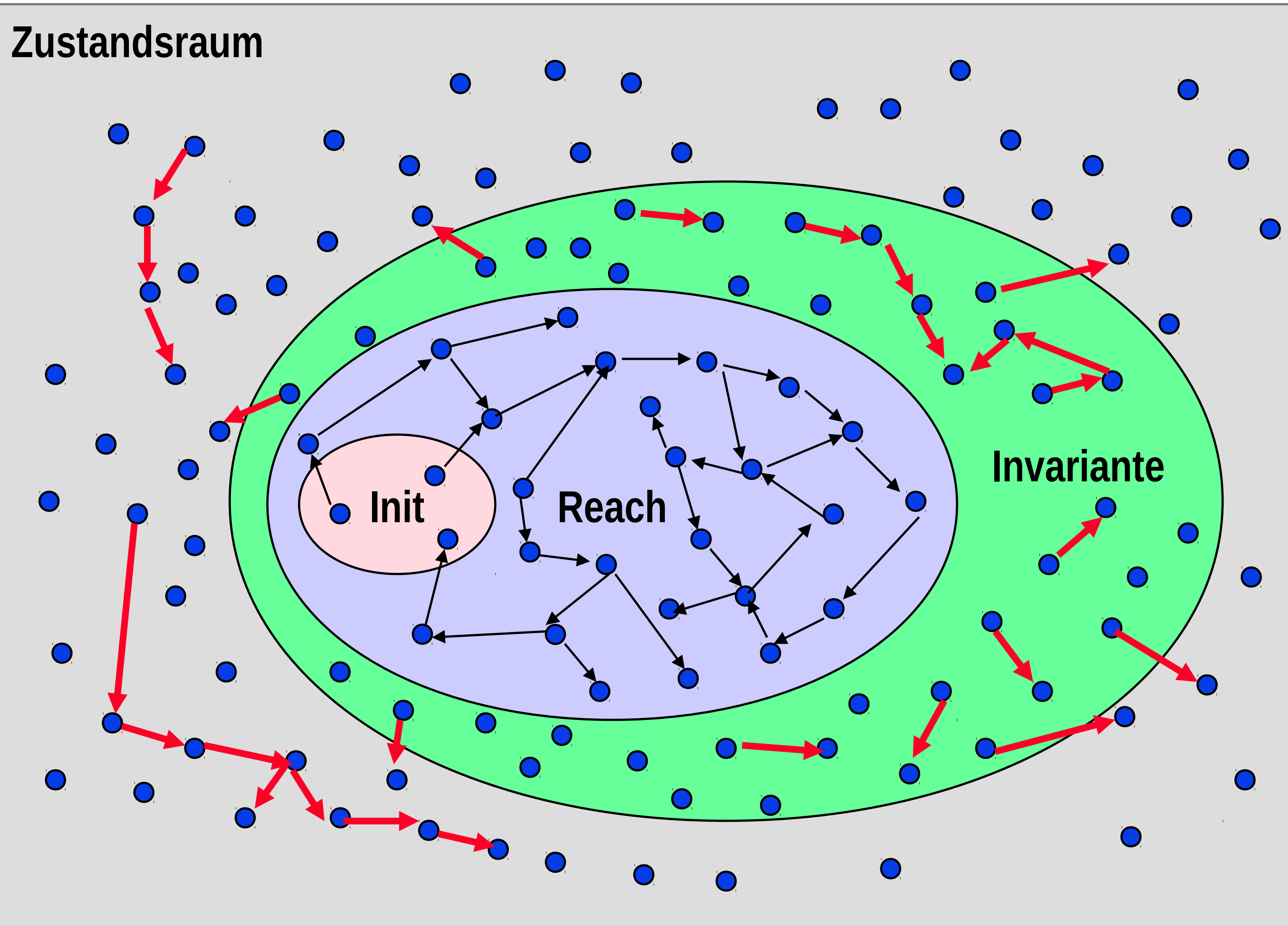


Jede Zustandsmenge, die Reach ganz umfasst, ist eine **Invariante**.

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

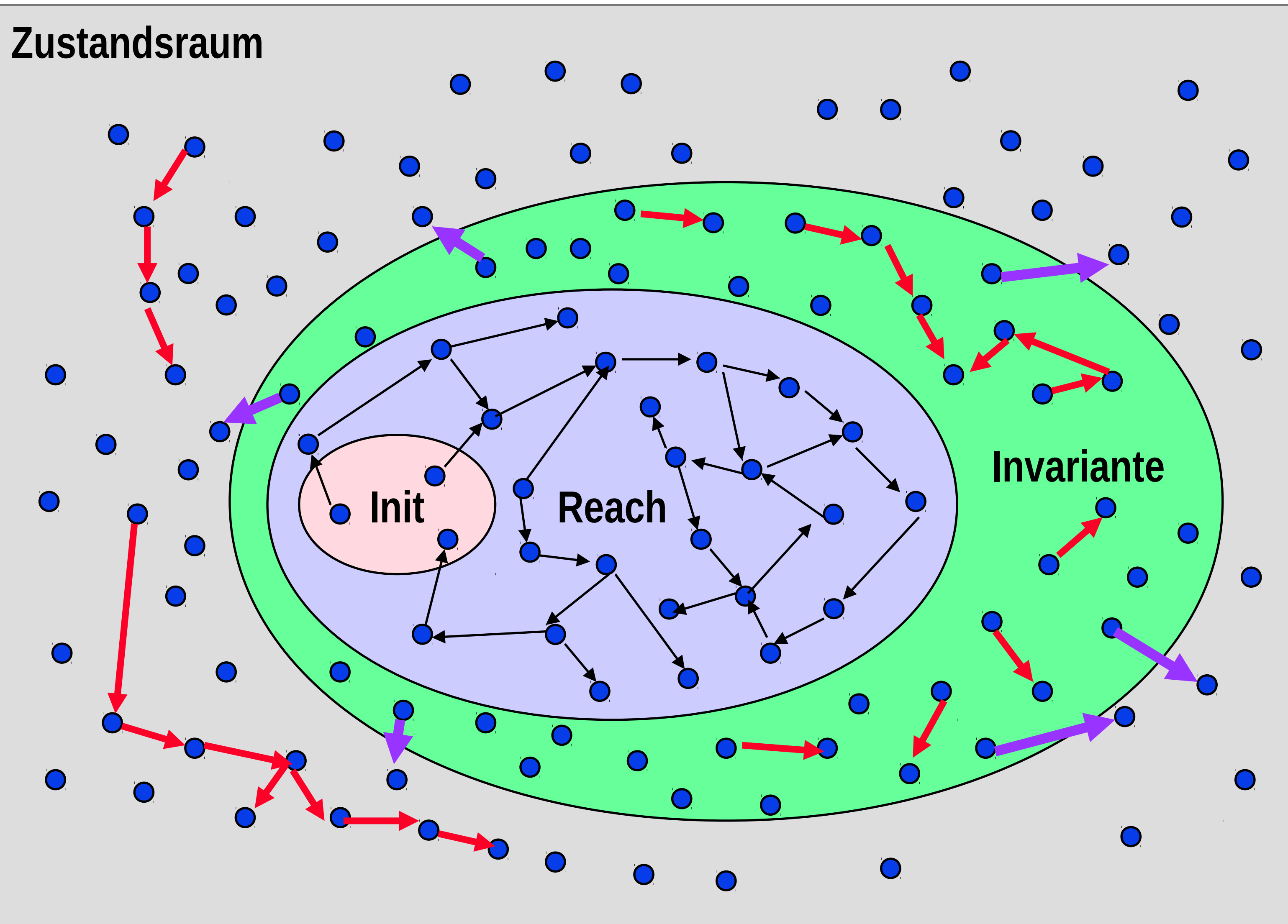
Reach ist die **schärfste Invariante** des STS.

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

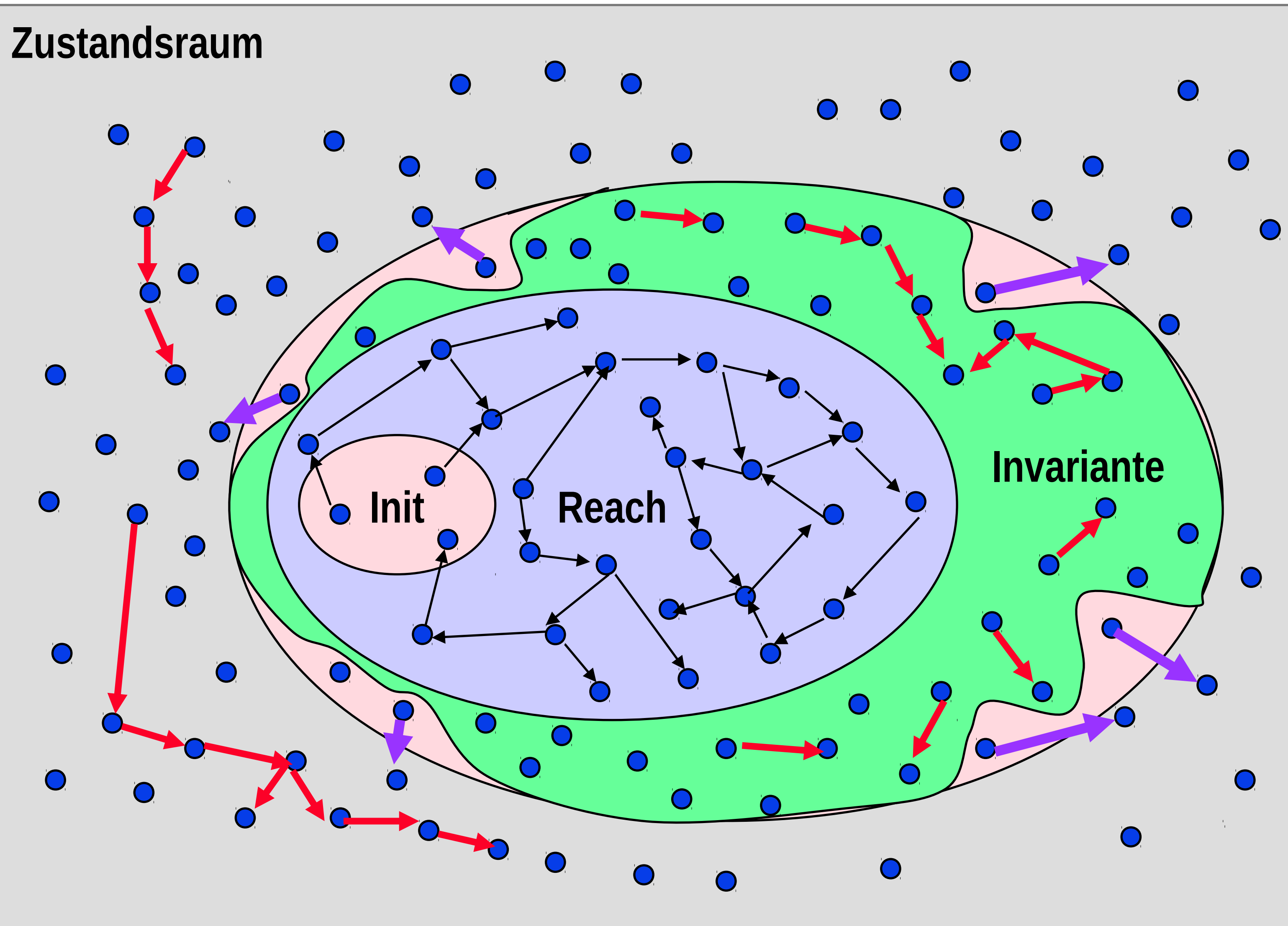


Transitionen sind nicht nur über erreichbaren Zuständen definiert.
Es gibt Transitionen, die nie gefeuert werden!

Es kann Transitionen geben, die aus einer Invarianten herausführen!

Die Invariante ist dann nicht induktiv.

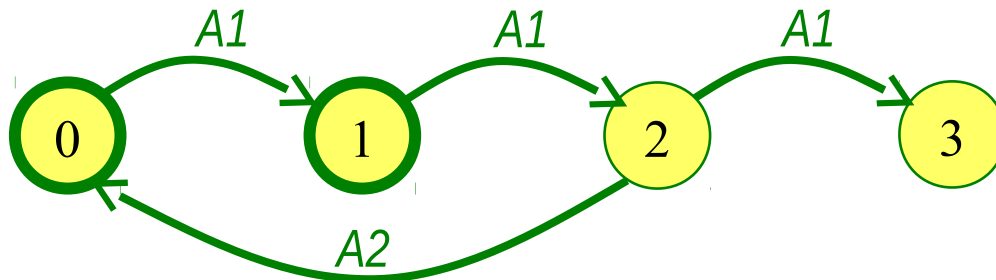
F11: Safetybeweise – Induktion über Ablauf



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



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

Beispiel
„Dem Zustand 3 geht immer Zustand 2 voraus“
I3 $V=3 \Rightarrow H=2$

☒ n=0: *init* $\Rightarrow$ I3

$$((V=0 \lor V=1) \land H=V) \Rightarrow (V=3 \Rightarrow H=2)$$

- $n \rightarrow n+1$: I3 $\land$ *A1* $\Rightarrow$ I3'

$$(V=3 \Rightarrow H=2) \land (V<3 \land V'=V+1 \land H'=V) \Rightarrow (V'=3 \Rightarrow H'=2)$$

- $n \rightarrow n+1$: I3 $\land$ *A2* $\Rightarrow$ I3'

$$(V=3 \Rightarrow H=2) \land (V=2 \land V'=0 \land H'=V) \Rightarrow (V'=3 \Rightarrow H'=2)$$

F11: Safetybeweise – Beispiel

1. n=0: *init*⇒I3
   $((V=0 \lor V=1) \land H=V) \Rightarrow (V=3 \Rightarrow H=2)$
2. n→n+1: I3 ∧ *A1* ⇒ I3'
   $(V=3 \Rightarrow H=2) \land (V<3 \land V'=V+1 \land H'=V) \Rightarrow (V'=3 \Rightarrow H'=2)$
3. n→n+1: I3 ∧ *A2* ⇒ I3'
   $(V=3 \Rightarrow H=2) \land (V=2 \land V'=0 \land H'=V) \Rightarrow (V'=3 \Rightarrow H'=2)$

Die Invariante I3 ist induktiv.

Die Schritte 2 und 3 sind erfolgreich.

I3 umfasst alle Zustände, bei welchen V≠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={} ⇒ ∀ 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 < from, to, msg > ! *Transportsystem*

Init

cs = < idle, idle, …, idle> ∧
nts = {<0, *Initiator*,*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}

F11: Safetybeweise – Beispiel „Broadcast“

Bei Terminierung wurden alle Stationen informiert.

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

n=0

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

$n \rightarrow n+1$: Forward

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

$\wedge$

$Forward) \Rightarrow$

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

$n \rightarrow n+1$: Skip

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

$\wedge$

$Skip) \Rightarrow$

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

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

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

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

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

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

F11: Safetybeweise – Beispiel „Broadcast“

Bei Terminierung wurden alle Stationen informiert:

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

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

$\wedge$

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

$\Rightarrow$

$\exists$ m: <k, l, m> $\in$ nts

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> }
\ {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 \ {m}

F11: Safetybeweise – Beispiel „Broadcast“

Invar Is:

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

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

$cs[l] \neq infd \Rightarrow \exists\ m: \langle k, l, m\rangle \in nts$

n=0

Init ⇒ **Is**

! nts ist nicht leer, alle Stationen sind idle

n → n+1: Forward

Is ∧ Forward ⇒ **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 → n+1: Skip

Is ∧ Skip ⇒ **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

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

Aktionen

Forward

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

Skip

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

F11: Safetybeweise – Beispiel „Broadcast“

Es werden höchstens 2e Nachrichten ausgetauscht.

AnzSend ≤ 2e

(Hilfsvariable AnzSend geeignet eingeführt)

n=0

Init ⇒ AnzSend ≤ 2e

n→n+1: Forward

((AnzSend ≤ 2e) ∧ **Forward**) ⇒

(AnzSend‘ ≤ 2e)

n→n+1: Skip

((AnzSend ≤ 2e) ∧ **Skip**) ⇒

(AnzSend‘ ≤ 2e)

Bei Forward lässt sich die Implikation nicht beweisen:

Invariante verschärfen!

- *Station i wird frisch informiert*
- *Es kommen #Nachbar(i)-1 Nachrichten dazu*

→ *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> ∧

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

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}
∧ AnzSend‘=AnzSend + #Nachbar(i)-1

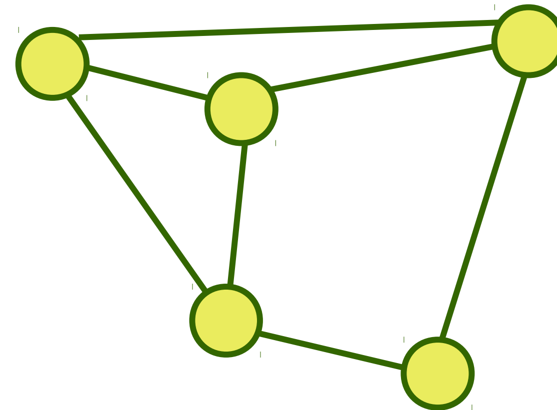
Skip

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

F11: Safetybeweise – Beispiel „Broadcast“

Graphentheorie
Kante verbindet je 2 Knoten:

Σi: #Nachbar(i) = 2e



Verschärfungsidee
Wir achten auf die informierten Stationen.

AnzSend ≤ Σi,cs[i]=infd: #Nachbar(i)

Verschärfte Invariante wäre

AnzSend ≤ 2e $\wedge$ AnzSend ≤ Σi,cs[i]=infd: #Nachbar(i)

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

Σi,cs[i]=infd: #Nachbar(i) **≤ 2e**

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

AnzSend ≤ Σi,cs[i]=infd: #Nachbar(i)

F11: Safetybeweise – Beispiel „Broadcast“

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

AnzSend ≤ Σi,cs[i]=infd: #Nachbar(i)

n=0

Init⇒ AnzSend ≤ Σi,cs[i]=infd: #Nachbar(i)

n→n+1: Forward

((AnzSend ≤ Σi,cs[i]=infd: #Nachbar(i))
∧ **Forward**)
⇒
(AnzSend‘ ≤ Σi,cs[i]‘=infd: #Nachbar(i))

n→n+1: Skip

((AnzSend ≤ Σi,cs[i]=infd: #Nachbar(i))
∧ **Skip**)
⇒
(AnzSend‘ ≤ Σi,cs[i]‘=infd: #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> ∧
nts = {<0, *Initiator*,*Text*>} ∧ AnzSend=0

Aktionen

Forward

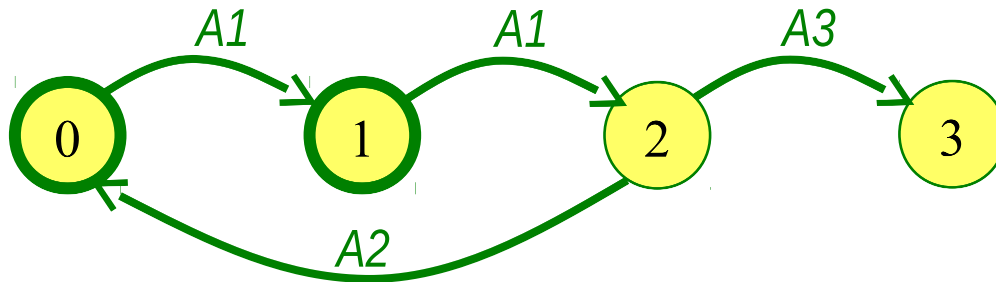
∃ i, m: cs[i]=idle ∧ m∈ nts ∧ m.to=i ∧
cs[i]‘=infd ∧ ∀ j ≠ i: cs[j]‘=cs[j] ∧
nts‘=
nts ∪ { <i, k, m.msg>: istNachbar(k,i) ∧
k ≠ m.from> } \ {m}
∧ AnzSend‘=AnzSend + #Nachbar(i)-1

Skip

∃ i, m: cs[i]=infd ∧ m∈ nts ∧ m.to=i
∧ AnzSend‘=AnzSend
∧ ∀ j : cs[j]‘=cs[j] ∧ nts‘=nts \ {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 ~> Q** notiert.
- Mögliche Abläufe des Systems sind in der Safetyeigenschaft des Systems enthalten.
- Beweis der Liveness vor diesem Hintergrund mithilfe von Regeln



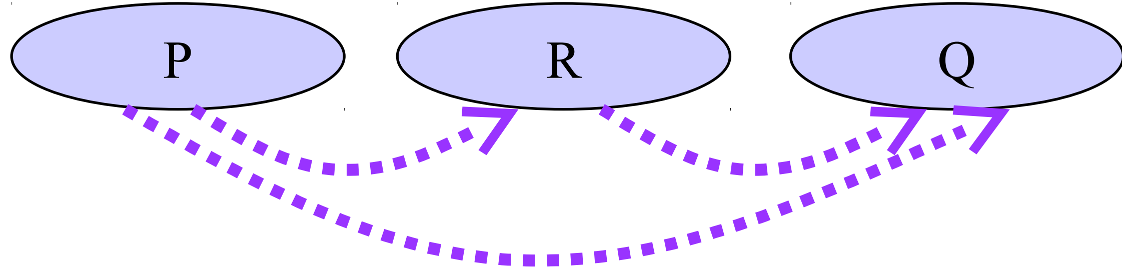
var   *V : (0, 1, 2, 3) ;*
init   *(V=0 ∨ V=1) ;*
act   *A1: V<2 ∧ V'=V+1;*
A2: V=2 ∧ V'= 0;
A3: V=2 ∧ V'=3 ;
WF(A1), SF(A3)

Beispiel
„Das System terminiert in 3"
L   *init ~> V=3*

F11: Livenessbeweise – Regeln

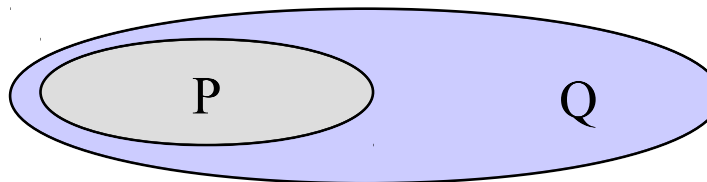
1. Transitivität von ~>

P ~> R
R ~> Q
P ~> Q



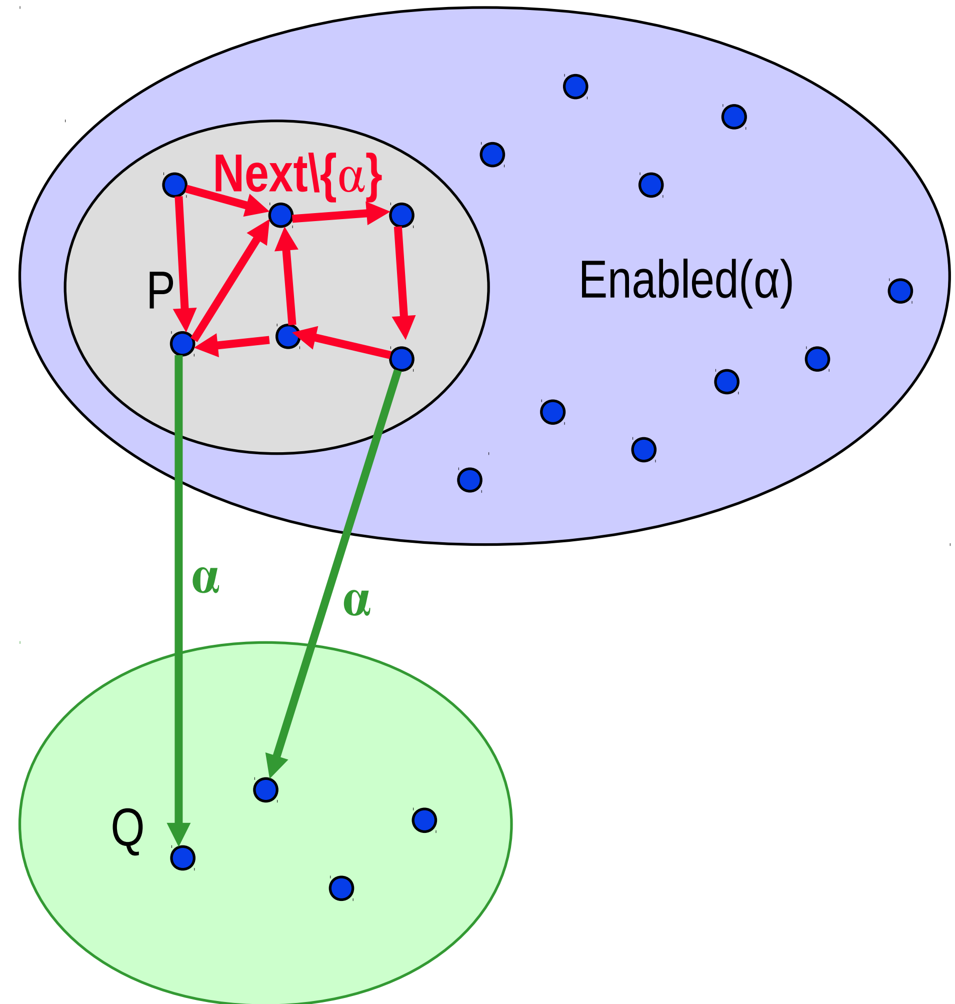
2. Trivialfall „Schon da“

P ⇒ Q
P ~> Q



3.1 Ein Weak Fair Schritt

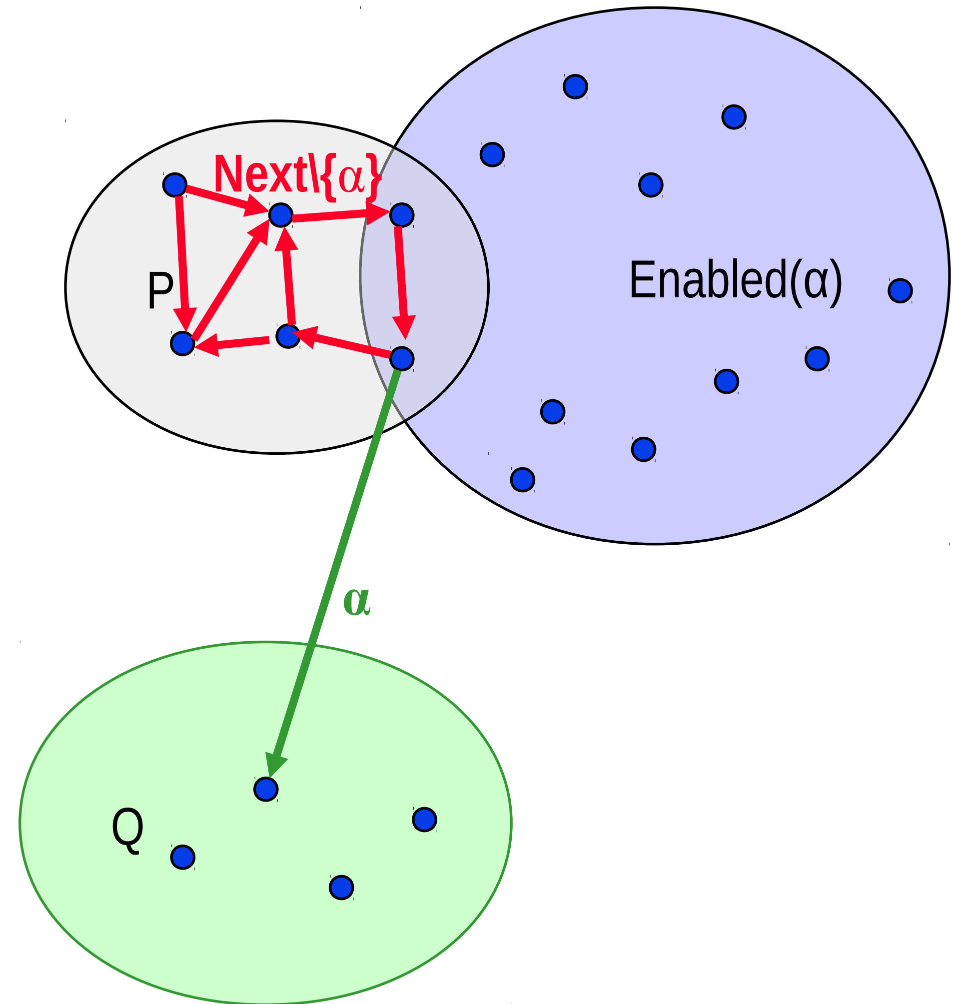
(1) WF(α)
(2) P ⇒ Enabled(<α>)
(3) P ∧ Next ∧ ¬<α> ⇒ P'
(4) <u>P ∧ <α> ⇒ Q'</u>
(c) P ~> Q



F11: Livenessbeweise – Regeln

3.2 Ein Strong Fair Schritt

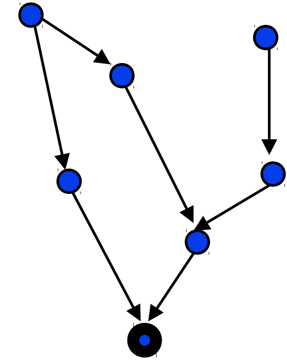
(1) SF(α)
(2) P ~> Enabled(<α>)
(3) P ∧ Next ∧ ¬<α> ⇒ P'
(4) P ∧ <α> ⇒ Q'
(c) P ~> Q



F11: Livenessbeweise – Regeln

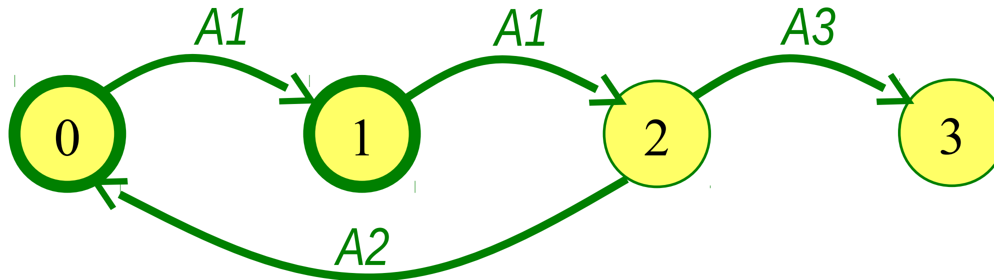
4. Lattice

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 ::= < M,∠>, der durch eine Wertemenge M und eine Halbordnung ∠ gegeben ist.
V muss ein Minimum m haben.
Für jeden Wert aus M gibt es nur endlich viele entsprechend ∠ 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 ∠ kleiner ist:
   expsf = x ~> expsf ∠ x.

F11: Livenessbeweise – Beispiel zu Ein-Schritt-Regeln



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), SF(A3)

Beispiel
„Das System terminiert in 3"
L *init ~> V=3*

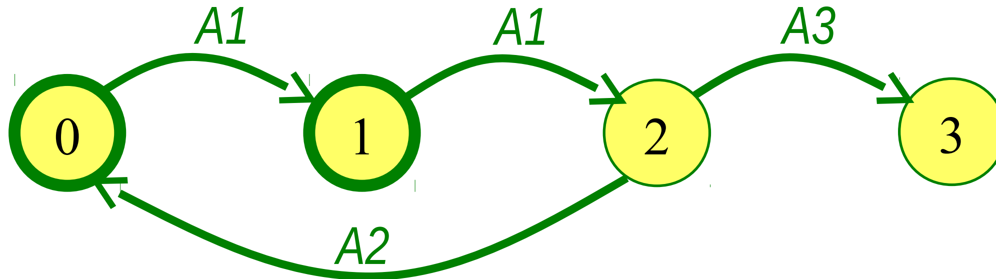
3.1 Ein Weak Fair Schritt

(1) WF(α)
(2) P ⇒ Enabled(<α>)
(3) P ∧ Next ∧ ¬<α> ⇒ P'
(4) P ∧ <α> ⇒ Q'
(c) P ~> Q

1WF: V=0 ~> V=1
1WF: V=1 ~> V=2

Trans: V=0 ~> V=2

F11: Livenessbeweise – Beispiel zu Ein-Schritt-Regeln



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), SF(A3)

Beispiel
„Das System terminiert in 3"
L init ~> V=3

1WF: V=0 ~> V=1
1WF: V=1 ~> V=2

Trans: V=0 ~> V=2

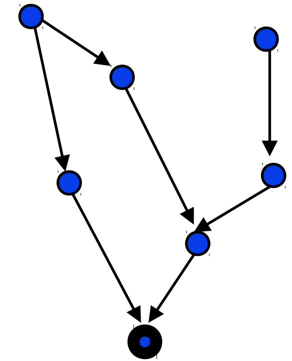
1SF: V≤2 ~> V=3

Trans: V=0 ~> V=3

3.2 Ein Strong Fair Schritt

(1) SF(α)
(2) P ~> Enabled(<α>)
(3) P ∧ Next ∧ ¬<α> ⇒ P'
(4) P ∧ <α> ⇒ Q'
(c) P ~> Q

F11: Livenessbeweise – Beispiel zu Lattice



- ☒ Formalisierung der Eigenschaft als eine Bedingung über den Systemzuständen, welche nach endlich vielen Schritten gelten muss
  - Def.: Term ≡ cs[Initiator]=infd ∧ 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.: lat ≡ <Anzahl Stationen im Zustand idle, Anzahl Nachrichten im nts>
    Lattice: Nat x Nat mit Ordnung **<**
    <x,y> **<** <x‘,y‘> ⇔ x **<** x‘ v (x = x‘ ∧ y **<** y‘)
  - **lat=<x,y> ~> lat < <x,y>** !
  - **lat=<0,0> ⇒ Term** !

F11: Livenessbeweise – Beispiel zu Lattice

Term ≡ cs[Initiator]=infd ∧ nts={}

Def.: lat ≡ <Anzahl Stationen im Zustand idle,
Anzahl Nachrichten im nts>

Lattice: Nat x Nat mit Ordnung **<**

<x,y> **<** <x',y'>

⇔

x **<** x' v (x = x' ∧ y **<** y')

1. lat=<x,y> ~> lat < <x,y>

! Forward *informiert neue Station*

! Skip *nimmt Nachr. aus nts*

! *Wegen* ***WF(Forward), WF(Skip)*** *kann System nicht unendlich lange stottern*

2. lat=<0,0> ⇒ 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 < from, to, msg > ! *Transportsystem*

Init

cs = < idle, idle, ..., idle> ∧
nts = {<0, *Initiator*,*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}

WF(Forward), WF(Skip)