Kapitel II

Temporallogik und Model Checking

Inhalt Kapitel II

- Einführung
- Die Temporallogik CTL
  - Syntax und informelle Semantik
  - Semantik
  - Äquivalenzen
- CTL-Model Checking
  - Labelling Algorithmus
  - Optimierungen
- Das System SMV
- Fairness
- Das Alternating Bit Protokoll
- Symbolisches Model-Checking

Motivation

Unter *Model Checking* versteht man die automatische Überprüfung, ob ein Systemmodell eine Spezifikation erfüllt.

Die Modellierungen nebenläufiger Systeme aus Kapitel 1 waren bereits Beispiele dafür:

- Die Modellierungen mit SAT-Solvern sind Instanzen von *Bounded Model Checking* (da die Simulationszeit beschränkt ist).
- Die Modellierungen mit BDDs sind Instanzen von *Symbolic Model Checking* (da Zustandsmengen nicht explizit, sondern “symbolisch” repräsentiert wurden)

Arten von Eigenschaften

Typische Arten von spezifizierten Eigenschaften:

- **Safety**: System gerät in keinen “verbotenen” Zustand / alle erreichbaren Zustände sind “erlaubt” (hatten wir schon).
- **Liveness**: System verklemmt sich nicht; “Reset-Zustand” von überall ereichbar; jede “Anfrage” wird irgendwann “beantwortet”.
- **Fairness**: bestimmte “gute Eigenschaft” gilt für alle “fairen” Abläufe.

Diese Klassifikation erfasst die meisten Eigenschaften, bisweilen gibt es noch komplexere.

Temporallogik

Temporallogik erlaubt die kompakte Spezifikation von Eigenschaften von Systemabläufen.

Im Unterschied zur Aussagenlogik können auch Aussagen über den zeitlichen Ablauf gemacht werden.

Es gibt eine Reihe verschiedener Temporallogiken, z.B.

- CTL (Computation Tree Logic)
- LTL (Linear Time Logic)

In der Vorlesung wird CTL im Detail behandelt.

Temporallogik

Im Semaphorbeispiel haben wir überprüft, dass das System keinen unerwünschten Zustand erreichen kann.

In CTL kann das durch folgende Formel ausgedrückt werden.

$$\mathsf{AG}(\neg \mathit{undesired})$$

Diese Formel sagt aus, dass alle (A – all) Abläufe im Zustandsübergangssystem stets (G – generally) die Eigenschaft $\neg \mathit{undesired}$ erfüllen.

Temporallogik

Die Eigenschaft, dass stets wieder der Anfangszustand erreicht werden kann, kann in CTL wie folgt ausgedrückt werden:

$$\mathsf{AG}(\mathsf{EF}(\bigwedge_{p} q_{p\ sleep}))$$

Die Formel $\mathsf{EF}(\phi)$ besagt, dass ein Ablauf existiert (E – exists), auf dem irgendwann (F – finally) die Eigenschaft ϕ gilt.

Syntax von CTL

Die Menge der CTL-Formeln ist durch folgende Grammatik gegeben.

$$\begin{aligned}\phi, \psi ::= \; & p \mid \top \mid \bot \mid \neg\phi \mid \phi \oplus \psi \mid \mathsf{AX}\phi \mid \mathsf{EX}\phi \\ & \mid \mathsf{A}[\phi \mathsf{U} \psi] \mid \mathsf{E}[\phi \mathsf{U} \psi] \mid \mathsf{AG}\phi \mid \mathsf{AF}\phi \mid \mathsf{EG}\phi \mid \mathsf{EF}\phi\end{aligned}$$

Hier steht p für aussagenlogische Variablen und $\oplus$ steht für die zweistelligen Boole'schen Operatoren. Insbesondere ist also jede aussagenlogische Formel auch eine CTL-Formel.

Beispiel: $\mathsf{AG}(p \Rightarrow \mathsf{A}[p \mathsf{U} (\neg p \wedge \mathsf{A}[\neg p \mathsf{U} q])])$

Kein Beispiel: $\mathsf{A}[p]$ und $\phi \mathsf{U} \psi$ sind keine CTL-Formeln!

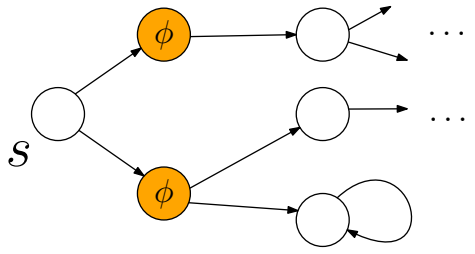
Informelle Semantik der CTL-Formeln

CTL Formeln werden relativ zu einem gegebenen Zustandsübergangssystem interpretiert.

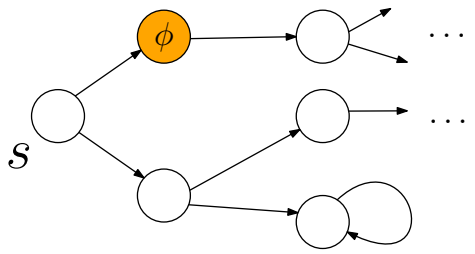
Eine CTL-Formel ϕ kann in jedem Zustand entweder gelten (= wahr sein) oder nicht.

In einem Zustand s gilt…

- …AXϕ, wenn ϕ in allen unmittelbaren Folgezuständen von s gilt.

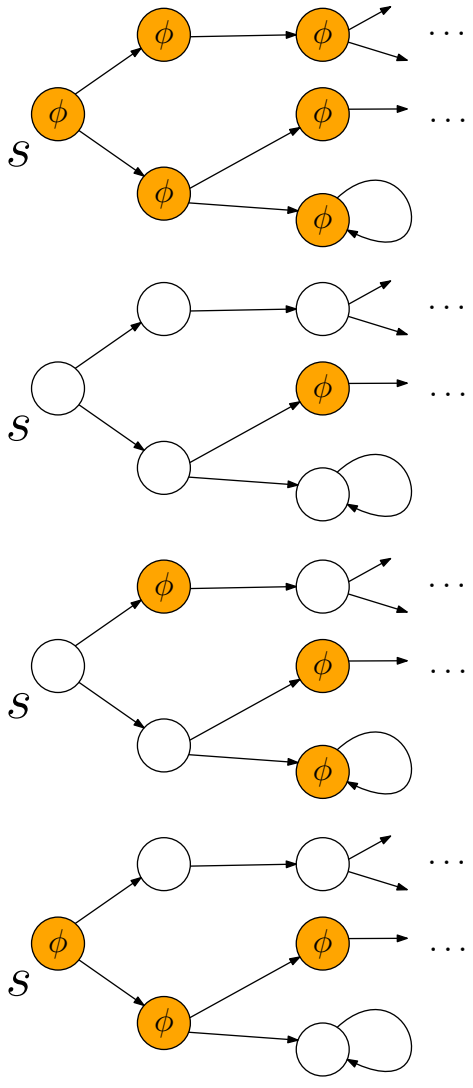


- …EXϕ, wenn ϕ in einem der unmittelbaren Folgezustände von s gilt.



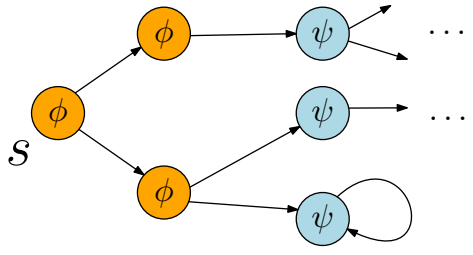
Informelle Semantik der CTL-Formeln, Forts.

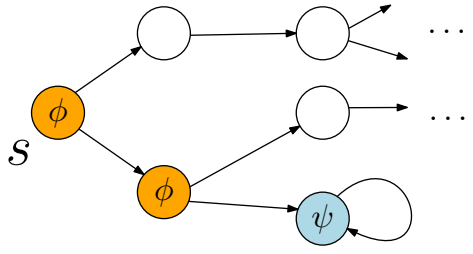
- …AGϕ, wenn ϕ auf allen von s aus erreichbaren Zuständen gilt.
- …EFϕ, wenn man von s aus einen Zustand erreichen kann, in dem ϕ gilt.
- …AF(ϕ), wenn auf allen von s ausgehenden Ausführungspfaden irgendwann ϕ gilt.
- …EG(ϕ), wenn von s aus die Ausführung so fortgesetzt werden kann, dass stets ϕ gilt.



Informelle Semantik der CTL-Formeln, Forts.

- …A$[\phi \mathrm{U} \psi]$, wenn auf allen von s ausgehenden Ausführungspfaden irgendwann ψ gilt und zumindest bis zum ersten Auftreten von ψ stets ϕ der Fall ist. (U = “until”).
- …E$[\phi \mathrm{U} \psi]$, wenn von s aus die Ausführung so fortgesetzt werden kann, dass irgendwann ψ gilt und bis dahin stets ϕ gilt.





Informelle Semantik der CTL-Formeln, Forts.

Beispiele:

- $\mathsf{AG}((\mathit{close\_door} \vee (\mathit{safe} \wedge \neg \mathit{open\_door})) \Rightarrow \mathsf{AX}\mathit{safe}) \wedge \mathsf{AG}(\mathit{heat} \Rightarrow \mathit{safe})$
- $\mathit{floor{=}2} \wedge \mathit{direction{=}up} \wedge \mathit{buttonpressed{=}5} \Rightarrow A[\mathit{direction{=}up}\ \mathsf{U}\ \mathit{floor{=}5}]$
- $\mathsf{AF}\mathit{fertig}$

Transitionssystem

Definition

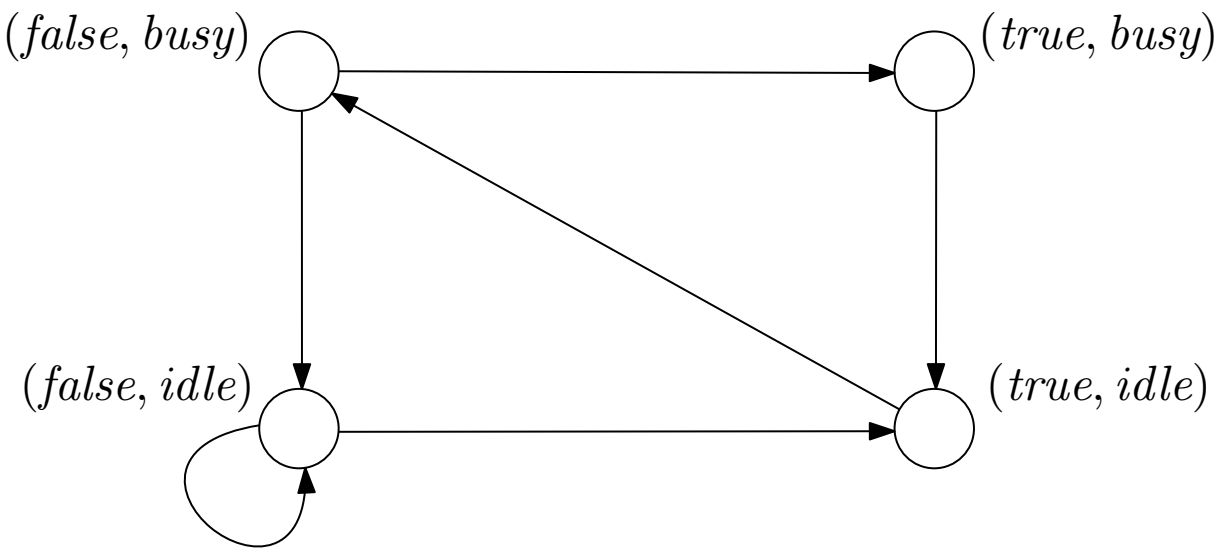
Ein *Transitionssystem* ist ein Paar $(S, \rightarrow)$, wobei

- S eine Menge von Zuständen ist, und
- $\rightarrow \subseteq S \times S$ eine binäre Relation auf S ist.
- Für jedes $s \in S$ existiert $s' \in S$ mit $s \rightarrow s'$.

- Die Menge S modelliert die Menge der globalen Zustände eines nebenläufigen Systems.
- Die Relation $\rightarrow$ heißt *Transitionsrelation*. Sie modelliert die möglichen Zustandsübergänge. Sie ergibt sich aus dem Programmtext, bzw. der Implementierung des Systems.
- Die dritte Bedingung hat technische Gründe. Liegt sie nicht bereits vor, so kann sie durch Hinzunahme eines Müllzustands s_d mit $s_d \rightarrow s_d$ künstlich hergestellt werden.

Beispiel

$$
\begin{aligned}
S &= \{(\mathit{request}, \mathit{status}) \mid \mathit{request} \in \{\mathit{true}, \mathit{false}\}, \mathit{status} \in \{\mathit{idle}, \mathit{busy}\}\} \\
\rightarrow &= \{((\mathit{false}, x), (\mathit{true}, x)) \mid x \in \{\mathit{idle}, \mathit{busy}\}\} \cup \\
&\quad \{((\mathit{true}, \mathit{idle}), (\mathit{false}, \mathit{busy}))\} \cup \\
&\quad \{((x, \mathit{busy}), (x, \mathit{idle})) \mid x \in \{\mathit{true}, \mathit{false}\}\} \cup \\
&\quad \{((\mathit{false}, \mathit{idle}), (\mathit{false}, \mathit{idle}))\}
\end{aligned}
$$



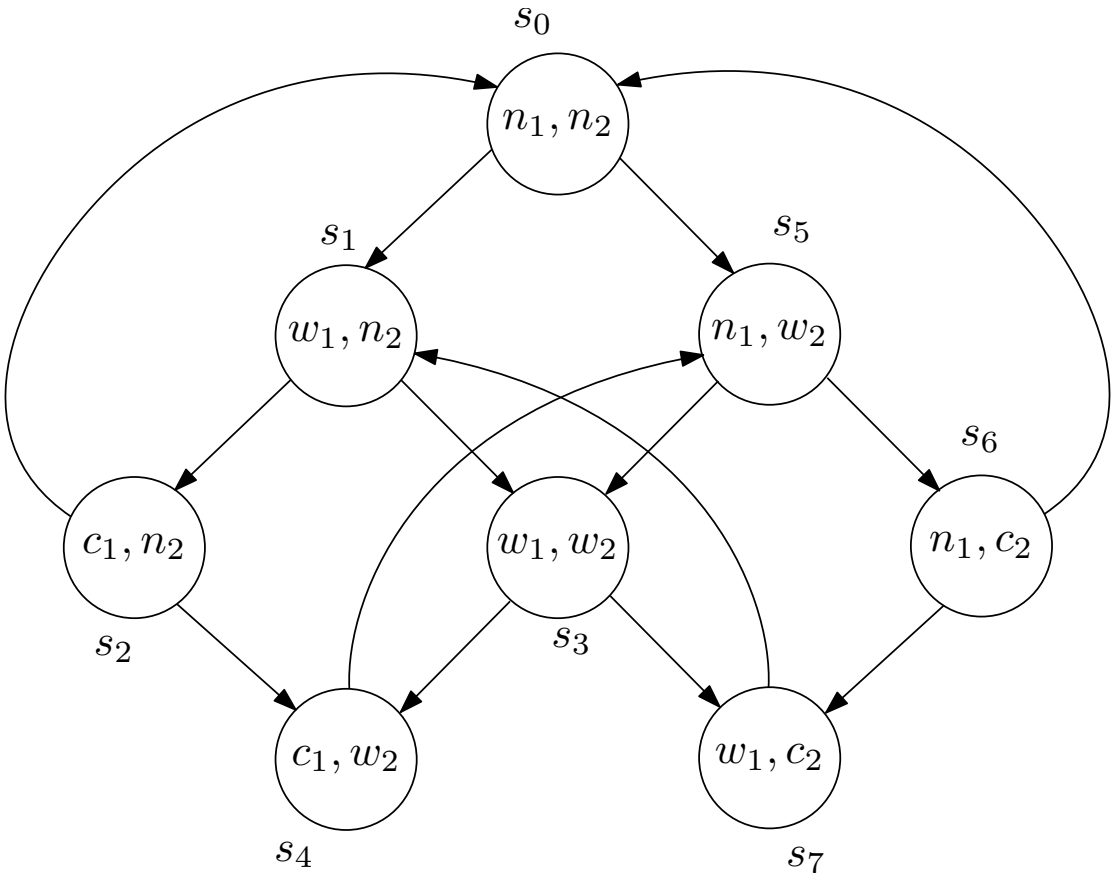
Weitere Beispiele

- Semaphor: $S = \{(\mathit{proc}_0, \mathit{proc}_1, \mathit{sem}) \mid \mathit{sem} \in \{\mathit{free}, \mathit{occ}\}, \forall i \in \{0,1\}.\mathit{proc}_i \in \{\mathit{sleep}, \mathit{wait}, \mathit{work}\}\}$
  Hier: $|S| = 18$
- Peterson:
  $S = \{(\mathit{flag}_0, \mathit{flag}_1, \mathit{turn}, \mathit{line}_0, \mathit{line}_1) \mid \forall i \in \{0,1\}.\mathit{flag}_i \in \{\mathit{true}, \mathit{false}\} \;\&\; \mathit{turn} \in \{0,1\} \;\&\; \mathit{line}_i \in \{0,1,2,3,4\}\}$
  Hier: $|S| = 2^2 \cdot 2 \cdot 5^2 = 200$
- Bauer, Hund, Katze, Maus: $S = \{(\mathit{pos}_B, \mathit{pos}_H, \mathit{pos}_K, \mathit{pos}_M) \mid \forall x \in \{B, H, K, M\}.\, \mathit{pos}_x \in \{\mathit{links}, \mathit{rechts}\}\}$
  Hier: $|S| = 2^4 = 16$.

NB: Die Transitionsrelation $\rightarrow$ ist hier jeweils weggelassen.

Weitere Beispiele

Der von Zustand $s_0 = (\mathit{sleep}, \mathit{sleep}, \mathit{free})$ aus erreichbare Teil des Semaphor-Transitionssystems:



Formale Semantik von CTL

Die Semantik von CTL-Formeln wird bezüglich einer Interpretation festgelegt.

Definition

Eine *Interpretation* $\mathcal{I}$ besteht aus einem endlichen Transitionssystem $\mathit{Tr}(\mathcal{I}) = (S, \rightarrow)$ sowie einer Menge von Zuständen $\mathcal{I}(p) \subseteq S$ für jede aussagenlogische Variable p.

Sei $\mathcal{I}$ eine Interpretation $\mathcal{I}$ mit $\mathit{Tr}(\mathcal{I}) = (S, \rightarrow)$.

Die CTL-Semantik legt für jede Formel ϕ und jeden Zustand $s \in S$ fest, ob die Formel in diesem Zustand bezüglich der Interpretation $\mathcal{I}$ gilt.

Wir schreiben kurz $s \models_{\mathcal{I}} \phi$ für "ϕ gilt im Zustand s (bezüglich $\mathcal{I}$)" und definieren diesen Begriff auf den nächsten Folien.

Definition der Semantik

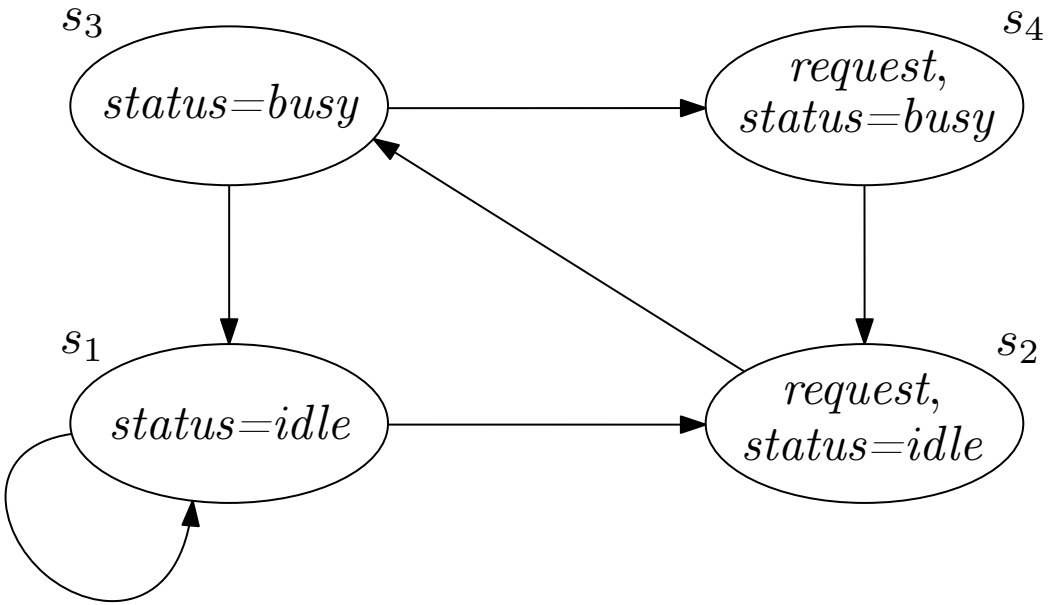
- $s \models_{\mathcal{I}} p$ genau dann wenn $s \in \mathcal{I}(p)$.
- $s \models_{\mathcal{I}} \neg\phi$ genau dann wenn $s \models_{\mathcal{I}} \phi$ nicht gilt (auch geschrieben als $s \not\models_{\mathcal{I}} \phi$).
- $s \models_{\mathcal{I}} \phi \wedge \psi$ genau dann wenn $s \models_{\mathcal{I}} \phi$ und $s \models_{\mathcal{I}} \psi$.
- die anderen Boole'schen Operatoren $\vee$, $\Rightarrow$, etc. sind analog.
- $s \models_{\mathcal{I}} \mathrm{EX}\phi$ genau dann wenn $s' \in S$ existiert mit $s \rightarrow s'$ und $s' \models_{\mathcal{I}} \phi$.
- $s \models_{\mathcal{I}} \mathrm{AX}\phi$ genau dann wenn für alle $s' \in S$ mit $s \rightarrow s'$ gilt: $s' \models_{\mathcal{I}} \phi$.

Definition der Semantik, Fortsetzung

- $s \models_{\mathcal{I}} \mathrm{AG}\phi$ gdw: Alle unendlichen Pfade der Form $s = s_0 \to s_1 \to s_2 \to s_3 \to \ldots$ haben die Eigenschaft, dass $s_i \models_{\mathcal{I}} \phi$ für alle $i \geqslant 0$ gilt.
- $s \models_{\mathcal{I}} \mathrm{EG}\phi$ gdw: Es gibt einen unendlichen Pfad der Form $s = s_0 \to s_1 \to s_2 \to s_3 \to \ldots$ mit der Eigenschaft, dass $s_i \models_{\mathcal{I}} \phi$ für alle $i \geqslant 0$ gilt.
- $s \models_{\mathcal{I}} \mathrm{EF}\phi$ gdw: Es gibt einen unendlichen Pfad der Form $s = s_0 \to s_1 \to s_2 \to s_3 \to \ldots$ mit der Eigenschaft, dass $s_i \models_{\mathcal{I}} \phi$ für ein $i \geqslant 0$ gilt.
- $s \models_{\mathcal{I}} \mathrm{AF}\phi$ gdw: Alle unendlichen Pfade der Form $s = s_0 \to s_1 \to s_2 \to s_3 \to \ldots$ haben die Eigenschaft, dass $s_i \models_{\mathcal{I}} \phi$ für ein $i \geqslant 0$ gilt.

Beispiel

Die Interpretation $\mathcal{I}$ mit dem Transitionssystem von Folie 120 sowie $\mathcal{I}(\mathit{request}) = \{s_2, s_4\}, \mathcal{I}(\mathit{status{=}idle}) = \{s_1, s_2\}$ und $\mathcal{I}(\mathit{status{=}busy}) = \{s_3, s_4\}$ wird folgendermaßen dargestellt.



Es gilt:

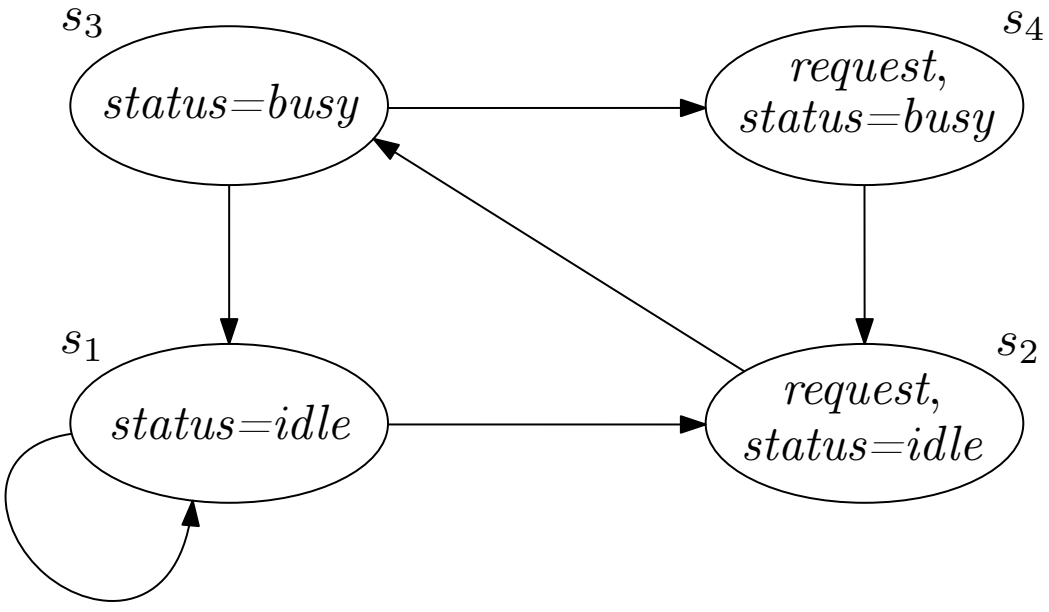
$s_1 \models_{\mathcal{I}} \mathsf{AF}\neg \mathit{request}$ $\quad$ $s_1 \models_{\mathcal{I}} \mathsf{AG}(\mathit{request} \Rightarrow \mathsf{EF}(\mathit{status{=}busy}))$

$s_1 \models_{\mathcal{I}} \mathsf{EG}\neg \mathit{request}$ $\quad$ $s_1 \models_{\mathcal{I}} \mathsf{AG}(\neg\mathsf{EG}(\mathit{status{=}busy}))$

Semantik der Until-Formeln

- $s \models_{\mathcal{I}} \mathsf{E}[\phi \mathsf{U} \psi]$ gdw: Es gibt einen Pfad $s = s_0 \to s_1 \to s_2 \to s_3 \to \cdots \to s_n$ mit der Eigenschaft, dass dass $s_n \models_{\mathcal{I}} \psi$ gilt sowie dass $s_i \models_{\mathcal{I}} \phi$ für alle $i < n$ gilt.
- $s \models_{\mathcal{I}} \mathsf{A}[\phi \mathsf{U} \psi]$ gdw: Alle unendlichen Pfade $s = s_0 \to s_1 \to s_2 \to s_3 \to \ldots$ haben die Eigenschaft, dass ein $n \geqslant 0$ existiert mit $s_n \models_{\mathcal{I}} \psi$ und $s_i \models_{\mathcal{I}} \phi$ für alle $i < n$.

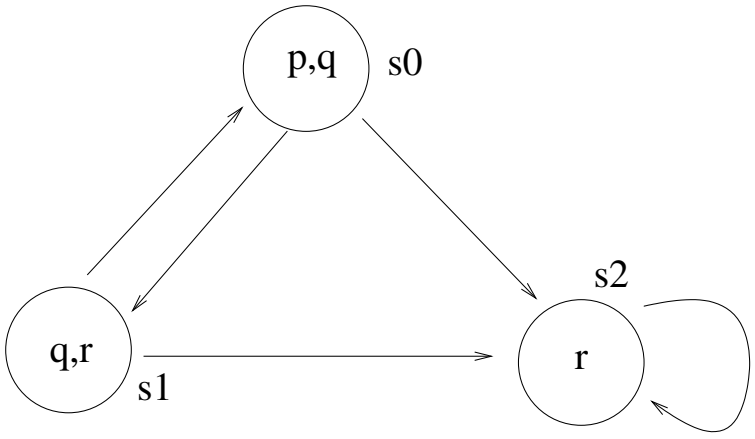
Beispiel



Es gilt:

$$s_1 \models_{\mathcal{I}} \mathsf{AG}(request \Rightarrow \mathsf{A}[request\ \mathsf{U}\ status{=}busy])$$

Beispiel



$s_0 \models_{\mathcal{I}} p \wedge q$

$s_0 \models_{\mathcal{I}} \top$

$s_0 \models_{\mathcal{I}} \neg \mathsf{AX}(q \wedge r)$

$s_1 \models_{\mathcal{I}} \mathsf{EG} r$

$s_0 \models_{\mathcal{I}} \mathsf{AF} r$

$s_0 \models_{\mathcal{I}} \mathsf{A}[p \mathsf{U} r]$

$s_0 \models_{\mathcal{I}} p \wedge \neg r$

$s_0 \models_{\mathcal{I}} \mathsf{EX}(q \wedge r)$

$s_0 \models_{\mathcal{I}} \neg \mathsf{EF}(p \wedge r)$

$s_2 \models_{\mathcal{I}} \mathsf{AG} r$

$s_0 \models_{\mathcal{I}} \mathsf{E}[(p \wedge q) \mathsf{U} r]$

Äquivalenzen

Äquivalenz von CTL-Formeln

Zwei CTL-Formeln ϕ und ψ sind *äquivalent*, geschrieben $\phi \iff \psi$, wenn für alle Interpretationen $\mathcal{I}$ und alle Zustände s gilt: $s \models_{\mathcal{I}} \phi$ gdw. $s \models_{\mathcal{I}} \psi$.

Sind $\phi \iff \psi$ aussagenlogisch äquivalente Formeln, so auch als CTL-Formeln. Z.B.: $\mathsf{AG}(p) \vee \mathsf{AG}(p) \iff \mathsf{AG}(p)$.

Wichtige Äquivalenzen:

$$\neg\mathsf{AG}(\phi) \iff \mathsf{EF}(\neg\phi) \qquad \neg\mathsf{AF}(\phi) \iff \mathsf{EG}(\neg\phi)$$

$$\neg\mathsf{EF}(\phi) \iff \mathsf{AG}(\neg\phi) \qquad \neg\mathsf{EG}(\phi) \iff \mathsf{AF}(\neg\phi)$$

$$\mathsf{AF}(\phi) \iff \mathsf{A}[\top\mathsf{U}\phi] \qquad \mathsf{EF}(\phi) \iff \mathsf{E}[\top\mathsf{U}\phi]$$

$$\mathsf{A}[\phi\mathsf{U}\psi] \iff \neg(\mathsf{E}[\neg\psi\mathsf{U}(\neg\phi \wedge \neg\psi)] \vee \mathsf{EG}(\neg\psi))$$

Äquivalenzen

Satz

Für jede CTL-Formel ϕ gibt es eine äquivalente Formel, in der neben Variablen nur die Operatoren $\neg$, $\wedge$, $\bot$, EX, AF, E$[-\mathsf{U}-]$ verwendet werden.

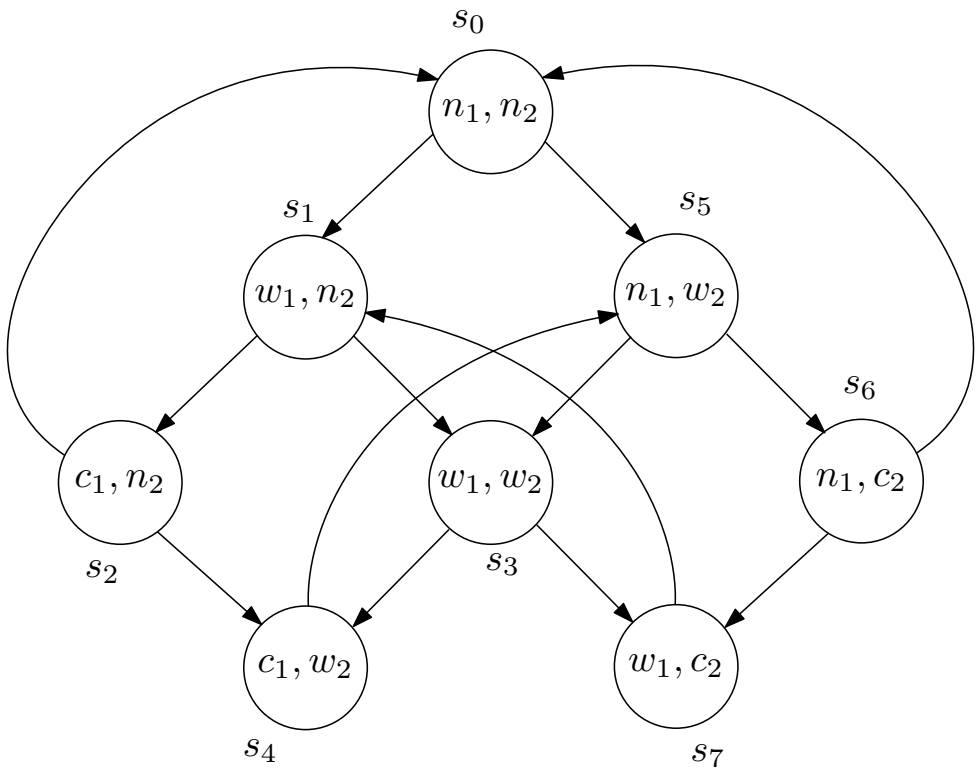
Beweis: Übung

Das Model Checking Problem für CTL

Gegeben:
Eine Interpretation $\mathcal{I}$ und eine CTL-Formel ϕ_0 und ein Zustand s_0.

Gefragt:
Gilt $s_0 \models_{\mathcal{I}} \phi_0$?

Mutual Exclusion Beispiel



- Safety: $\mathsf{AG}(\neg(c_1 \wedge c_2))$
- Liveness: $\mathsf{AG}(w_1 \Rightarrow \mathsf{AF}c_1)$
- Non-blocking: $\mathsf{AG}(n_1 \Rightarrow \mathsf{EX}w_1)$
- No strict sequencing: $\mathsf{EF}(c_1 \wedge \mathsf{E}[c_1 \mathsf{U}(\neg c_1 \wedge \mathsf{E}[\neg c_2 \mathsf{U} c_1])])$

Labelling Algorithmus

Der *Labelling Algorithmus* löst das Model Checking für CTL.

- Eingabe sind eine Interpretation $\mathcal{I}$ und eine Formel ϕ_0.
- Berechnet wird die Menge aller Zustände (im Transitionssystem von $\mathcal{I}$) in denen ϕ_0 gilt.

Aufgrund der Äquivalenzen können wir annehmen, dass die Formel ϕ_0 nur die Operatoren $\neg$, $\wedge$, $\bot$, EX, AF, E$[-\mathsf{U}-]$ verwendet.

Labelling Algorithmus

Grundidee: Berechne für jede Teilformel von ϕ von ϕ_0 die Menge aller Zustände, in denen ϕ gilt.

- Bildlich gesprochen beschriftet (labelt) man die Zustände in S mit denjenigen Teilformeln, die dort gelten.
- Der Algorithmus verfährt durch Rekursion über die Formel. Für Variablen ist die Aufgabe einfach.
  Bei einer zusammengesetzten Formel ϕ führt man den Algorithmus zunächst für die direkten Teilformeln aus. Aus dem Ergebnis kann man dann die Beschriftung für ϕ berechnen.

Labelling Algorithmus: Details (1)

Der Algorithmus macht eine Fallunterscheidung über die Eingabeformel ϕ_0.

- $\bot$: Markiere keinen Zustand mit $\bot$.
- p: Markiere Zustände mit aussagenlogischen Variablen, wie von der Interpretation vorgegeben.
- $\neg\phi$: Führe den Algorithmus rekursiv für ϕ aus. Markiere danach alle Zustände mit $\neg\phi$, die nicht mit ϕ beschriftet sind.
- $\phi \wedge \psi$: Führe den Algorithmus rekursiv für ϕ und ψ aus. Markiere danach alle Zustände mit $\phi \wedge \psi$, die sowohl mit ϕ als auch mit ψ beschriftet sind.
- EXϕ: Führe den Algorithmus rekursiv für ϕ aus. Markiere dann alle Zustände mit EXϕ, die einen unmittelbaren Nachfolger haben, der schon mit ϕ markiert ist.

Labelling Algorithmus: Details (2)

- AFϕ:
  1. Führe den Algorithmus rekursiv für ϕ aus.
  2. Markiere alle Zustände mit AFϕ, die schon mit ϕ markiert sind.
  3. Sind *alle* unmittelbaren Folgezustände eines Zustands s bereits mit AFϕ markiert, so markiere auch s mit AFϕ. Wiederhole Schritt 3 bis keine neuen Markierungen mehr hinzukommen.
- E$[\phi \mathsf{U} \psi]$:
  1. Führe den Algorithmus rekursiv für ϕ und ψ aus.
  2. Markiere alle Zustände mit E$[\phi \mathsf{U} \psi]$, die schon mit ψ markiert sind.
  3. Ist *ein* unmittelbarer Folgezustände eines Zustands s bereits mit E$[\phi \mathsf{U} \psi]$ markiert und ist s selbst mit ϕ markiert, so markiere s auch mit E$[\phi \mathsf{U} \psi]$. Wiederhole Schritt 3 bis keine neuen Markierungen mehr hinzukommen.

Komplexität

Eine direkte Implementierung des Algorithmus hat Laufzeit

$$O(f \cdot V \cdot (V + E))$$

wobei f die Größe der Ausgangsformel, V die Zahl der Zustände und E die Zahl der Transitionen ist.

Beispiel: AFϕ

1. rekursiver Aufruf: $O((f-1) \cdot V \cdot (V + E))$
2. Anfangsmarkierung: $O(V)$
3. einen Zustand, dessen Nachfolger alle schon markiert sind, finden und markieren: $O(V + E)$; maximal V Wiederholungen

Verbesserung

Labelling kann in Zeit $O(f \cdot (V + E))$ implementiert werden.

Dazu genügt es, alle Fälle so zu implementieren, dass zum rekursiven Aufruf jeweils nur Aufwand $O(V + E)$ hinzukommt.

- Die Fälle für p, $\neg$, $\wedge$, EX sind einfach.
- Der Fall für E$[\phi \mathsf{U} \psi]$ kann mit einer Rückwärts-Breitensuche implementiert werden.
  Ist ein Knoten mit E$[\phi \mathsf{U} \psi]$ markiert, so werden alle seine Vorgänger, die auch mit ϕ markiert sind, selbst mit $E[\phi \mathsf{U} \psi]$ markiert.
- Leider funktioniert Breitensuche für AF nicht, da ja alle Nachfolger und nicht nur einer markiert sein müssen.

Effiziente Behandlung von EGϕ

Anstatt ein effizienteres Verfahren für AF direkt anzugeben, ersetzen wir AF durch EG und geben ein Verfahren für EG an.

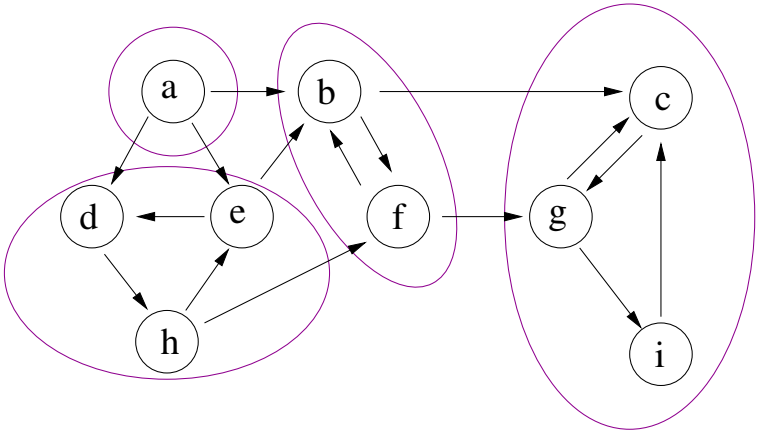
Beachte: $\mathsf{AF}\phi \iff \neg\mathsf{EG}(\neg\phi)$

Starke Zusammenhangskomponenten

Definition

Eine *starke Zusammenhangskomponente* (strongly connected component, SCC) eines gerichteten Graphs ist eine maximal große Menge U von Knoten mit folgender Eigenschaft: Für alle $s_1, s_2 \in U$ gilt, dass s_2 von s_1 aus erreichbar ist und umgekehrt.

Eine SCC ist *trivial*, wenn sie aus einem einzigen Knoten besteht, der keine Kante zu sich selbst hat.



Ein Graph kann in linearer Zeit in SCCs zerlegt werden (Tarjan's Alg.).

Effiziente Behandlung von EGϕ

Markierung der Zustände mit EGϕ.

- Führe den Algorithmus rekursiv für ϕ aus.
- Betrachte folgenden Teilgraphen G des Transitionsystems: Die Knoten sind alle mit ϕ markierten Zustände. Zwischen diesen Knoten hat G die gleichen Kanten wie das Transitonssystem.
- Berechne die SCCs von G.
- Markiere in G alle Knoten, von denen aus eine nichttriviale SCC erreichbar ist.
- Markiere im ursprünglichen Transitionssystem alle Zustände, die in G markiert sind.

Korrektheit der Markierung mit EGϕ

Angenommen der rekursive Aufruf für ϕ markiert genau die Zustände mit ϕ, in denen ϕ erfüllt ist.

Alle mit EGϕ markierten Zustände erfüllen die Formel EGϕ.

- Für jeden markierten Zustand gibt es in G einen Pfad in eine nichttrivale SCC von G.
- Ist s ein Knoten in einer nichttrivialen SCC, dann gibt es einen unendlichen Pfad $s \to s_0 \to s_1 \to \ldots$, der in der SCC bleibt.
- Es gibt also von jedem mit EGϕ markierten Knoten aus einen unendlichen Pfad in G.
- Da G nur mit ϕ markierte Knoten enthält, folgt daraus, das EGϕ erfüllt ist.

Vollständigkeit der Markierung mit EGϕ (1)

Alle Zustände, die EGϕ erfüllen, werden auch markiert.

Dazu ist zu zeigen, dass jeder Zustand s markiert wird, der $s \models_{\mathcal{I}} \text{EG}\phi$ erfüllt. Diese Eigenschaft gilt genau dann wenn es einen unendlichen Pfad $s = s_0 \to s_1 \to \ldots$ gibt, dessen Zustände alle mit ϕ markiert sind.

Lemma

In jedem unendlichen Pfad $s_0 \to s_1 \to s_2 \to \ldots$ in einem endlichen Graphen kommen ab einem bestimmten Index n nur noch Zustände vor, die auch unendlich oft im Pfad vorkommen.

Beweis: Für jeden Zustand s, der im Pfad nur endlich oft vorkommt, gibt es einen Index n_s des letzten Vorkommens. Sei n das Maximum aller dieser n_s (es gibt nur endlich viele, da S endlich ist). Ab Index n können nach Konstruktion nur Zustände vorkommen, die auch unendlich oft im Pfad vorkommen. □

Vollständigkeit der Markierung mit EGϕ (2)

Satz

Für jeden unendlichen Pfad $s_0 \to s_1 \to s_2 \to \ldots$ in einem endlichen Graphen gibt es eine nichttriviale SCC U und eine Zahl n, so dass $s_i \in U$ für alle $i \geqslant n$ gilt.

Beweis: Wähle n nach dem Lemma. Für beliebige $n \leqslant i < j$ können wir nun zeigen, dass s_i und s_j in derselben SCC liegen. Wegen $i < j$ gibt es einen Pfad von s_i nach s_j. Es gibt auch einen Pfad von s_j nach s_i. Es muss es ein $k > j$ geben mit $s_k = s_i$, denn andernfalls käme s_i nur endlich oft im Pfad vor. Somit ist s_i von s_j erreichbar und umgekehrt; sie liegen in einer SCC. Die SCC enthält eine Kante und ist daher nichttrivial. □

Nach dem Satz wird also jeder Zustand s markiert, für den es einen unendlichen Pfad $s = s_0 \to s_1 \to \ldots$ gibt, dessen Zustände alle mit ϕ markiert sind. Das sind alle Zustände, die EGϕ erfüllen.

State-Explosion-Problem

Die effizientere Version ist linear in der Größe des Transitionssystems, aber…

Die Größe des Transitionssystems ist exponentiell in der Anzahl seiner Komponenten: n Prozesse à k Zustände ergeben ein Transitionssystem mit k^n Zuständen.

Abhilfen:

- Ausnutzen von Symmetrie
- Abstraktion
- Symbolische Repräsentation von Zuständen (wie bei der Modellierung mit BDDs im ersten Kapitel)

Model Checking Tools

Model Checking wird zur automatischen Verifikation von Systemeigenschaften verwendet. Man verwendet dazu Model Checking Tools.

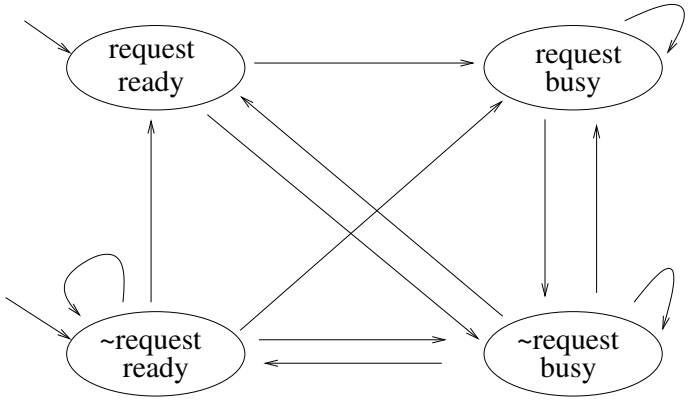
Im Folgenden betrachten wir das SMV System.

- Das Transitionssystem wird in einer speziellen Spezifikationssprache SMV definiert.
- Eigenschaften werden als CTL-Formeln ausgedrückt.
- Die Gültigkeit von Eigenschaften kann dann automatisch überprüft werden.
- Wir verwenden die aktuelle Implementierung NuSMV. Für Details siehe die Dokumentation.

Es gibt viele weitere Model Checking Tools mit verschiedenden Spezifikationssprachen und verschiedenen Temporallogiken.

Hello World in SMV

```
MODULE main
VAR
  request : boolean;
  status : {ready, busy};
ASSIGN
  init(status) := ready;
  next(status) := case
                    request : busy;
                    TRUE    : {ready,busy};
                  esac;
SPEC
  AG(request -> AF status=busy)
```



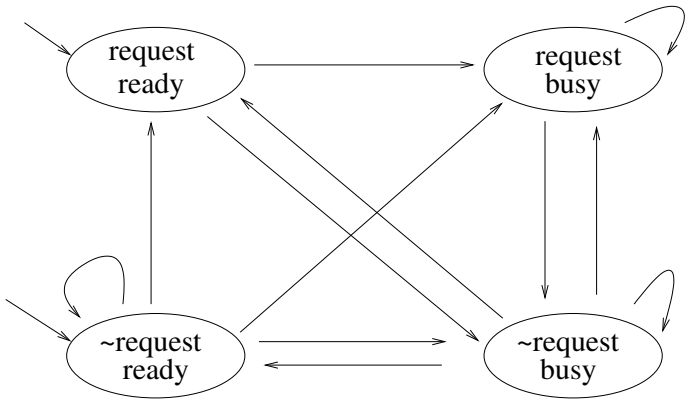
SMV — Explizite Notation

```
MODULE main
VAR
  request : boolean;
  status  : {ready, busy};

INIT
  status=ready

TRANS
      request & next(status)=busy
  | !request & next(status)=ready
  | !request & next(status)=busy

SPEC
  AG(request -> AF status=busy)
```



SMV — Explizite Notation

Das Modul definiert eine Interpretation und eine CTL-Formel.

- Die Zustände des Transitionssystems sind die Belegungen der mit VAR deklarierten Variablen. Hier:

$$S = \{(\texttt{request}, \texttt{status}) \mid \texttt{request} \in \{\mathit{true}, \mathit{false}\}, \texttt{status} \in \{\texttt{ready}, \texttt{busy}\}\}$$

- Es gibt eine Transition

$$(\texttt{request}, \texttt{status}) \rightarrow (\texttt{request}', \texttt{status}')$$

genau dann, wenn der Ausdruck in TRANS wahr ist.
Im Ausdruck schreibt man `next(request)` und `next(status)` für die Komponenten $\texttt{request}'$ und $\texttt{status}'$ des Nachfolgezustands.

SMV — Explizite Notation

- Die Menge der Anfangszustände sind alle Zustände, die den Ausdruck in `INIT` wahr machen.

Vergleiche die Kodierung von Zustandsübergangssystemen durch BDDs aus Kapitel 1.

In den aussagenlogischen Formeln kommen `boolean`-Variablen wie `request` sowie Gleichungen wie `status=ready` vor.

- Konzeptuell können diese alle wie aussagenlogische Variablen verstanden werden.
- Die Interpretation wird so gewählt, dass diese Variablen in den offensichtlichen Zuständen gelten.
  Im Zustand $(\mathit{true}, \texttt{ready})$ sind die aussagenlogischen Variablen `request` und `status=ready` wahr. Die Variable `status=busy` ist falsch.

SMV — Model Checking

Im SPEC-Teil der Datei steht die zu überprüfende Eigenschaft als CTL-Formel.

Aufruf von NuSMV (im CIP-Pool installiert):

```
NuSMV datei.smv
```

- Das in der Eingabedatei `datei.smv` enthaltene Modell wird syntax- und typüberprüft.
- Dann wird durch Model-Checking ermittelt, ob alle SPEC-Formeln in allen Startzuständen (`INIT`) erfüllt sind.
- Falls nicht, so wird ein Gegenbeispiel konstruiert.

SMV — Explizite Notation

Beispiele verschiedener Datentypen

```
VAR
  x : boolean;                    -- Boole'sche Werte
  y : {w1, w2, w3, w4};           -- endliche Menge
  z : 1..8;                       -- Zahlenbereiche
  u : array 0..10 of boolean;     -- Arrays
```

Mehrere Klauseln

Es sind mehrere `INIT`, `TRANS`, `SPEC` Klauseln erlaubt, diese verstehen sich jeweils als Konjunktion.

```
INIT u[0] & (u[z] -> y=w1)
INIT y=w2 -> x
```

entspricht

```
INIT u[0] & (u[z] -> y=w1) & (y=w2 -> x)
```

SMV — Implizite Notation

Die INIT und TRANS-Klauseln können durch ASSIGN-Klauseln ersetzt werden.

```
MODULE main
VAR request : boolean;
    status : {ready, busy};
ASSIGN
  init(status) := ready;
  next(status) := case
                    request : busy;
                    TRUE    : {ready, busy};
                  esac;
SPEC AG(request -> AF status = busy)
```

Es werden die möglichen Werte der Variablen in Anfangs- und Folgezuständen durch Wertzuweisung definiert.
Nichtdeterminismus (mehrere Möglichkeiten für Anfangs- und Folgezustände) wird durch Mengennotation erfasst.

SMV — Implizite Notation

```
next(status) := case
                   request : busy;
                   TRUE    : {ready, busy};
                esac;
```

Bedeutung:

- Wenn `request` wahr ist, dann muss `status` im Folgezustand gleich `busy` sein.
- Andernfalls hat `status` im Folgezustand einen Wert aus `{ready, busy}`.

Die Bedingungen in den `case`-Ausdrücken werden von oben nach unten der Reihe nach abgearbeitet, bis eine Bedingung zutrifft. Es müssen alle Möglichkeiten erfasst sein. Die Bedingung `TRUE` trifft natürlich immer zu.

SMV — Implizite Notation

Zuweisungen schränken die möglichen Werte der Variablen ein.
Gibt es keine Zuweisung, so sind beliebige Werte erlaubt.

Im Beispiel gibt es keine Zuweisung für `next(request)`.
In Transitionen darf der Wert von `request` im Nachfolger beliebig sein.

Semaphor mit zwei Prozessen in impliziter Notation

Wir verwenden eine zusätzliche Variable `selector`, die angibt, welcher Prozess einen Schritt macht.

```
MODULE main
VAR
  p1 : {sleep, wait, work};
  p2 : {sleep, wait, work};
  s  : {free, occ};
  selector : {1, 2};
ASSIGN
  init(p1) := sleep;
  init(p2) := sleep;
  init(s) := free;
```

(weiter auf nächster Folie)

Semaphore mit zwei Prozessen, Forts.

```
next(p1) := case selector=1 & p1=sleep          : wait;
                selector=1 & p1=wait & s=free   : work;
                selector=1 & p1=work            : sleep;
                TRUE                            : p1;
            esac;
next(p2) := case selector=2 & p2=sleep           : wait;
                selector=2 & p2=wait & s=free    : work;
                selector=2 & p2=work             : sleep;
                TRUE                             : p2;
            esac;
next(s) := case p1=wait & s=free & selector=1 : occ;
               p2=wait & s=free & selector=2 : occ;
               p1=work & selector=1          : free;
               p2=work & selector=2          : free;
               TRUE                          : s;
           esac;
```

Spezifikationen

- **Safety:** `AG(!(p1=work & p2=work))`. Gilt.
  Die beiden Prozesse sind niemals gleichzeitig im `work` Zustand.
- **Liveness:** `AG(p1=wait -> AF p1=work)`. Gilt nicht.
  Wenn Prozess `p1` irgendwann einmal im `wait` Zustand ist, dann wird er in jeder weiteren Ausführung irgendwann in den `work` Zustand kommen.
- **Weak Liveness:** `AG(p1=wait -> EF p1=work)`. Gilt.
  Wenn Prozess `p1` irgendwann einmal im `wait` Zustand ist, dann gibt es *eine* mögliche Ausführung, mit der er in den `work` Zustand kommt.
- **Non-blocking:** `AG(p1=sleep -> EX p1=wait)`. Gilt
  Wenn Prozess `p1` irgendwann einmal im `sleep` Zustand ist, dann gibt es einen möglichen Nachfolgezustand, in dem er im `wait` Zustand ist.

Spezifikationen

- **No strict sequencing:**

```
EF(p1=work & E[p1=work U
        (!(p1=work) & E[!(p2=work) U p1=work])])
```

  Gilt.
  Es kann sein, dass Prozess p1 den kritischen Bereich verlässt und ihn dann erneut betritt, ohne dass zwischendurch p2 im kritischen Bereich war.
  Die beiden Prozesse müssen also nicht strikt alternieren.

Abkürzende Schreibweise mit Modulen

```
MODULE prc(selector, s, pid)
VAR
  p : {sleep, wait, work};
ASSIGN
  init(p) := sleep;
  next(p) := case
               selector=pid & p=sleep          : wait;
               selector=pid & p=wait & s=free  : work;
               selector=pid & p=work           : sleep;
               TRUE                            : p;
             esac;
```

Das Modul `prc` hat drei Parameter (untypisiert, Einsetzung textuell).

Abkürzende Schreibweise mit Modulen, Forts.

```
MODULE main
VAR
  s  : {free, occ};
  selector : {1, 2};
  p1 : prc(selector, s, 1);
  p2 : prc(selector, s, 2);
ASSIGN
  init(s) := free;
  next(s) := case
               p1.p=wait & s=free & selector=1 : occ;
               p2.p=wait & s=free & selector=2 : occ;
               p1.p=work & selector=1          : free;
               p2.p=work & selector=2          : free;
               TRUE                            : s;
             esac;

SPEC AG(!(p1.p=work & p2.p=work))
```

Das Modulkonzept

Die Deklaration einer Variable mit Modultyp entspricht:

- Deklaration aller Variablen aus dem Modul.
  Mit Notation wie `p1.p` kann man auf diese Variablen zugreifen.
- Hinzufügen aller `ASSIGN`-Zuweisungen des Moduls zu den `ASSIGN`-Zuweisungen des umschließenden Moduls.

Das entspricht der *synchronen Komposition* der Module: zu jedem Zeitpunkt machen alle beteiligten Module gleichzeitig je einen Schritt.
In diesem Beispiel erzwingt die Variable `selector` explizit, dass tatsächlich nur ein Prozess einen echten Schritt macht, alle anderen einen Dummy-Schritt.

Die Zuweisungen in den Modulen dürfen sich nicht widersprechen.
Im Beispiel ist es nicht möglich in `prc` auch noch `next(s):=...` zu schreiben, da es in `main` eine schon eine solche Zuweisung gibt.

Prozesse in SMV

Bei der Modellierung nebenläufiger Systeme ist häufig die *asynchrone Komposition* von Prozessen gewünscht.

- Es werden k Prozesse parallel ausgeführt.
- Zu jedem Zeitpunkt ist genau ein Prozess aktiv und macht einen Transitionsschritt.

Asynchrone Komposition kann manuell wie eben gezeigt mit einer Variable `selector` implementiert werden.

Prozesse in SMV

In NuSMV kann asynchrone Komposition mit dem Schlüsselwort `process` definiert werden.

Deklariert man Modulinstanzen als `process`, so wird eine `selector`-Variable automatisch erzeugt.

- Es wird automatisch eine wird eine `selector`-Variable erzeugt, die sicherstellt, dass pro Zeiteinheit ist immer genau ein "Prozess" aktiv.
- In `process`-Modulen darf es auch Zuweisungen an gemeinsame (shared) Variablen (wie `s`) geben.
- Für jedes Modul (einschließlich `main`) wird eine `running` Variable vom Typ `boolean` angelegt. Sie ist `TRUE`, wenn der Prozess gerade einen Schritt macht.

Semaphore mit Prozessen

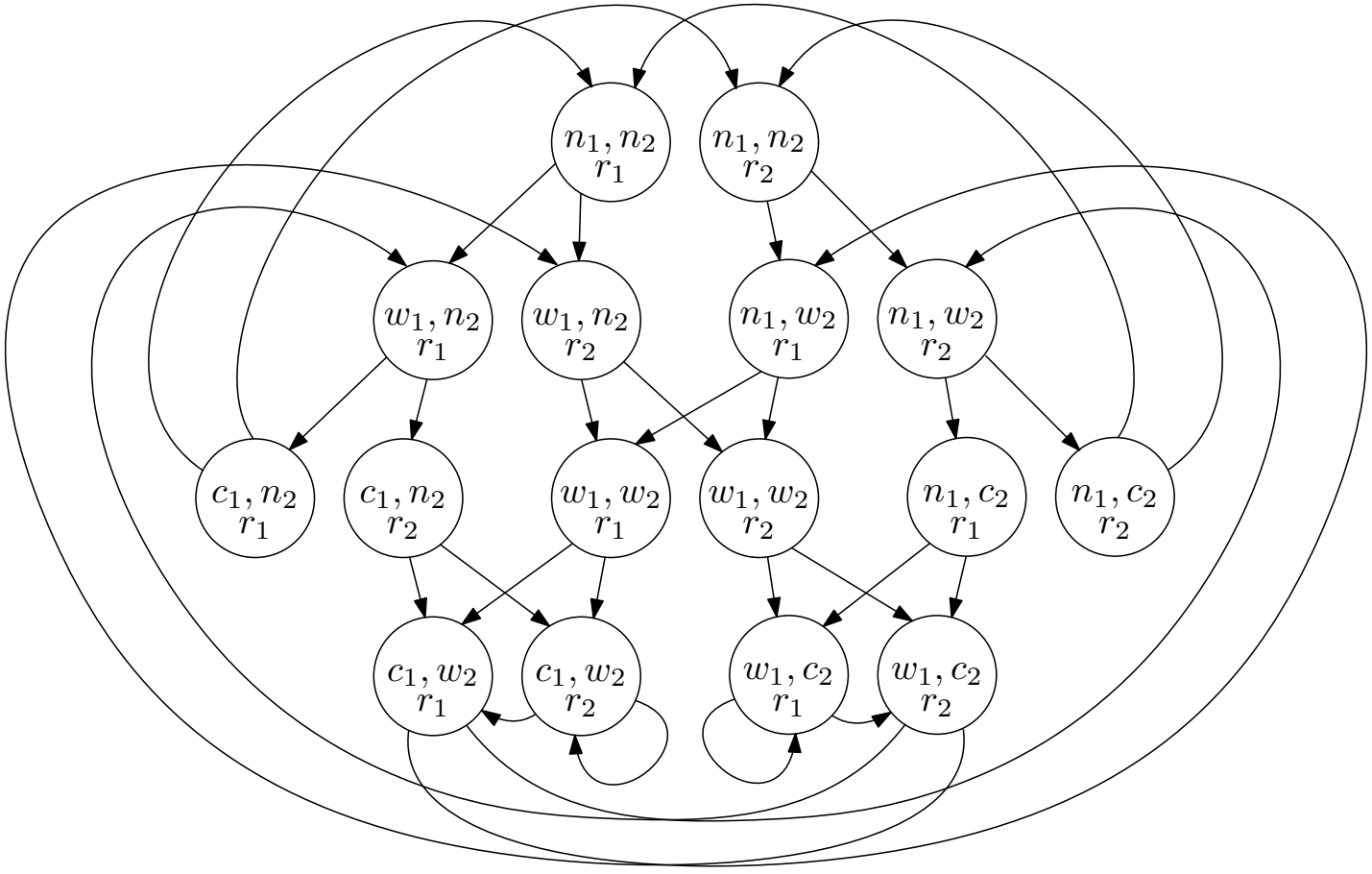
```
MODULE prc(s)
VAR
  p : {sleep, wait, work};
ASSIGN
  init(p) := sleep;
  next(p) := case
               p=sleep           : wait;
               p=wait & s=free   : work;
               p=work            : sleep;
               TRUE              : p;
             esac;
  next(s) := case
               p=wait & s=free   : occ;
               p=work            : free;
               TRUE              : s;
            esac;
```

Semaphore mit Prozessen, Forts.

```
MODULE main
VAR
  s  : {free, occ};
  p1 : process prc(s);
  p2 : process prc(s);
ASSIGN
  init(s) := free;
SPEC AG(!(p1.p=work & p2.p=work))
SPEC AG(p1=sleep -> EX p1=wait)
SPEC AG(p1=wait -> AF p1=work)
SPEC EF(p1=work &
        E[p1=work U (!(p1=work) & E[!(p2=work) U p1=work])])
```

Beispiel: Semaphore mit Prozessen

Transitionssystem: (n — sleep, w — wait, c — work, r — running):



Weitere SMV Konstrukte

- Mit `DEFINE` kann man Definitionen (Abkürzungen) einführen.
- Mit `INVAR` kann man Zustandsinvarianten festlegen. Es sind dann nur solche Transitionen erlaubt, die diese Invariante sicherstellen. Man kann sich Invarianten als zusätzliche Einschränkung der `next`-Werte in einer `TRANS` Deklaration vorstellen.
- Mit `JUSTICE` (synonym `FAIRNESS`) kann man eine Fairness-Bedingung festlegen. Die Semantik der CTL-Operatoren wird dann dahingehend verändert, dass nur solche unendliche Pfade betrachtet werden, entlang derer die Fairness-Bedingung immer wieder (“unendlich oft”) wahr ist.

Invarianten: Bauer, Hund, Katze, Maus

```
MODULE bauer
  VAR pos : {links, rechts};
  ASSIGN
    init(pos) := links;
    next(pos) := case pos=links  : rechts;
                      pos=rechts : links;
                 esac;

MODULE passagier(mitnehmen, bauer)
  VAR pos : {links, rechts};
  ASSIGN
    init(pos) := links;
    next(pos):=
      case
        mitnehmen & pos=bauer.pos : next(bauer.pos);
        TRUE                      : pos;
      esac;
```

Invarianten: Bauer, Hund, Katze, Maus

```
MODULE main
  VAR bauer   : bauer();
      auswahl : {h, k, m, keiner};
      hund    : passagier(auswahl=h, bauer);
      katze   : passagier(auswahl=k, bauer);
      maus    : passagier(auswahl=m, bauer);

  INVAR auswahl=h -> bauer.pos=hund.pos
  INVAR auswahl=k -> bauer.pos=katze.pos
  INVAR auswahl=m -> bauer.pos=maus.pos
  INVAR katze.pos=hund.pos -> bauer.pos=katze.pos
  INVAR katze.pos=maus.pos -> bauer.pos=katze.pos

  SPEC !EF(bauer=rechts & hund.pos=rechts &
           katze.pos=rechts & maus.pos=rechts)
```

Man erhält die gesuchte Lösung als “Gegenbeispiel”.

Beispiel für Fairness

Im Semaphoren-Beispiel ist die Liveness-Eigentschaft

```
SPEC AG(p1.p=wait -> AF p1.p=work)
```

trivialerweise falsch, da es möglich ist, dass der Prozess `p1` überhaupt nie ausgeführt wird, sondern immer nur `p2`.

Gilt Liveness für Ausführungen, in denen beide Prozesse fair behandelt werden und immer wieder dran kommen?

- Mit der Klausel "`JUSTICE running`" in `prc` kann man sich auf Ausführungen beschränken, in denen `running` immer wieder gilt, also in denen der Prozess immer wieder drankommt.
- Bei Nicht-Verwendung von `process` nimmt man stattdessen die Klausel "`JUSTICE selector=pid`" hinzu.

Auch mit dieser Fairnessbedingung gilt Liveness nicht; das System ist mangelhaft.

CTL mit Fairness

Fairness kann mit den vorhandenen CTL Operatoren nicht ausgedrückt werden.

CTL mit Fairness hat die gleichen Formeln wie CTL, aber die Semantik beachtet Fairnessbedingungen.

Gegeben sei ein Transitionssystem $(S, \rightarrow)$.

Definition

Eine *Fairness-Bedingung* (auch Justice-Bedingung) ist eine Teilmenge $f \subseteq S$ von Zuständen.

Definition

Eine *Interpretation für CTL mit Fairness* ist eine Interpretation wie für CTL sowie mit einer Liste von Fairnessbedingungen $F = (f_1, \ldots, f_n)$.

CTL mit Fairness

Wir definieren die Semantik von CTL mit Fairness wie für CTL, beschränken aber alle Quantoren auf Pfade, die *fair* bezüglich der Liste von Fairnessbedingungen in der Interpretation $\mathcal{I}$ sind.

Definition

Ein unendlicher Pfad $s_0 \to s_1 \to \ldots$ ist *fair bezüglich einer Fairness-Bedingung* f, wenn die Menge $\{i \in \mathbb{N} \mid s_i \in f\}$ unendlich ist ("es kommen immer wieder Zustände aus f vor").

Definition

Ein unendlicher Pfad ist *fair bezüglich einer Liste* $(f_1, \ldots, f_n)$ *von Fairness-Bedingungen*, wenn er bzgl. jeder einzelnen fair ist.

Semantik von CTL mit Fairness

- $s \models_{\mathcal{I}} p$ genau dann wenn $s \in \mathcal{I}(p)$.
- $s \models_{\mathcal{I}} \neg\phi$ genau dann wenn $s \models_{\mathcal{I}} \phi$ nicht gilt (auch geschrieben als $s \not\models_{\mathcal{I}} \phi$).
- $s \models_{\mathcal{I}} \phi \wedge \psi$ genau dann wenn $s \models_{\mathcal{I}} \phi$ und $s \models_{\mathcal{I}} \psi$.
- die anderen Boole'schen Operatoren $\vee$, $\Rightarrow$, etc. sind analog.
- $s \models_{\mathcal{I}} \mathrm{EX}\phi$ genau dann wenn $s' \in S$ existiert mit $s \rightarrow s'$ und $s' \models_{\mathcal{I}} \phi$.
- $s \models_{\mathcal{I}} \mathrm{AX}\phi$ genau dann wenn für alle $s' \in S$ mit $s \rightarrow s'$ gilt: $s' \models_{\mathcal{I}} \phi$.

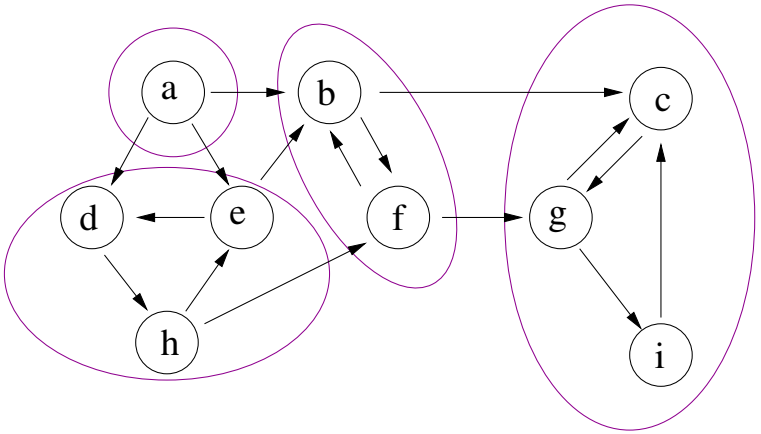
Semantik von CTL mit Fairness, Fortsetzung

- $s \models_{\mathcal{I}} \mathrm{AG}\phi$ gdw: Alle *fairen* unendlichen Pfade $s = s_0 \to s_1 \to s_2 \to s_3 \to \ldots$ haben die Eigenschaft, dass $s_i \models_{\mathcal{I}} \phi$ für alle $i \geqslant 0$ gilt.
- $s \models_{\mathcal{I}} \mathrm{EG}\phi$ gdw: Es gibt einen *fairen* unendlichen Pfad $s = s_0 \to s_1 \to s_2 \to s_3 \to \ldots$ mit der Eigenschaft, dass $s_i \models_{\mathcal{I}} \phi$ für alle $i \geqslant 0$ gilt.
- $s \models_{\mathcal{I}} \mathrm{EF}\phi$ gdw: Es gibt einen *fairen* unendlichen Pfad $s = s_0 \to s_1 \to s_2 \to s_3 \to \ldots$ mit der Eigenschaft, dass $s_i \models_{\mathcal{I}} \phi$ für ein $i \geqslant 0$ gilt.
- $s \models_{\mathcal{I}} \mathrm{AF}\phi$ gdw: Alle *fairen* unendlichen Pfade $s = s_0 \to s_1 \to s_2 \to s_3 \to \ldots$ haben die Eigenschaft, dass $s_i \models_{\mathcal{I}} \phi$ für ein $i \geqslant 0$ gilt.

Model-Checking mit Fairness

- Eine nichttriviale SCC ist *fair*, wenn sie einen fairen Pfad enthält. Das ist genau dann der Fall, wenn sie für jede Fairnessbedingung f_i einen Zustand $s_i \in f_i$ enthält.
- Von einem Zustand aus gibt es einen fairen Pfad, genau dann, wenn von ihm aus eine faire (echte) SCC erreicht werden kann.

Sei $f_1 = \{a, d, f\}$ und $f_2 = \{d, b\}$.



Die Komponenten $\{d, e, h\}$ und $\{b, f\}$ sind fair.
Von a, b, d, e, f, h aus gibt es faire Pfade, von c, g, i aus nicht.

Model-Checking mit Fairness

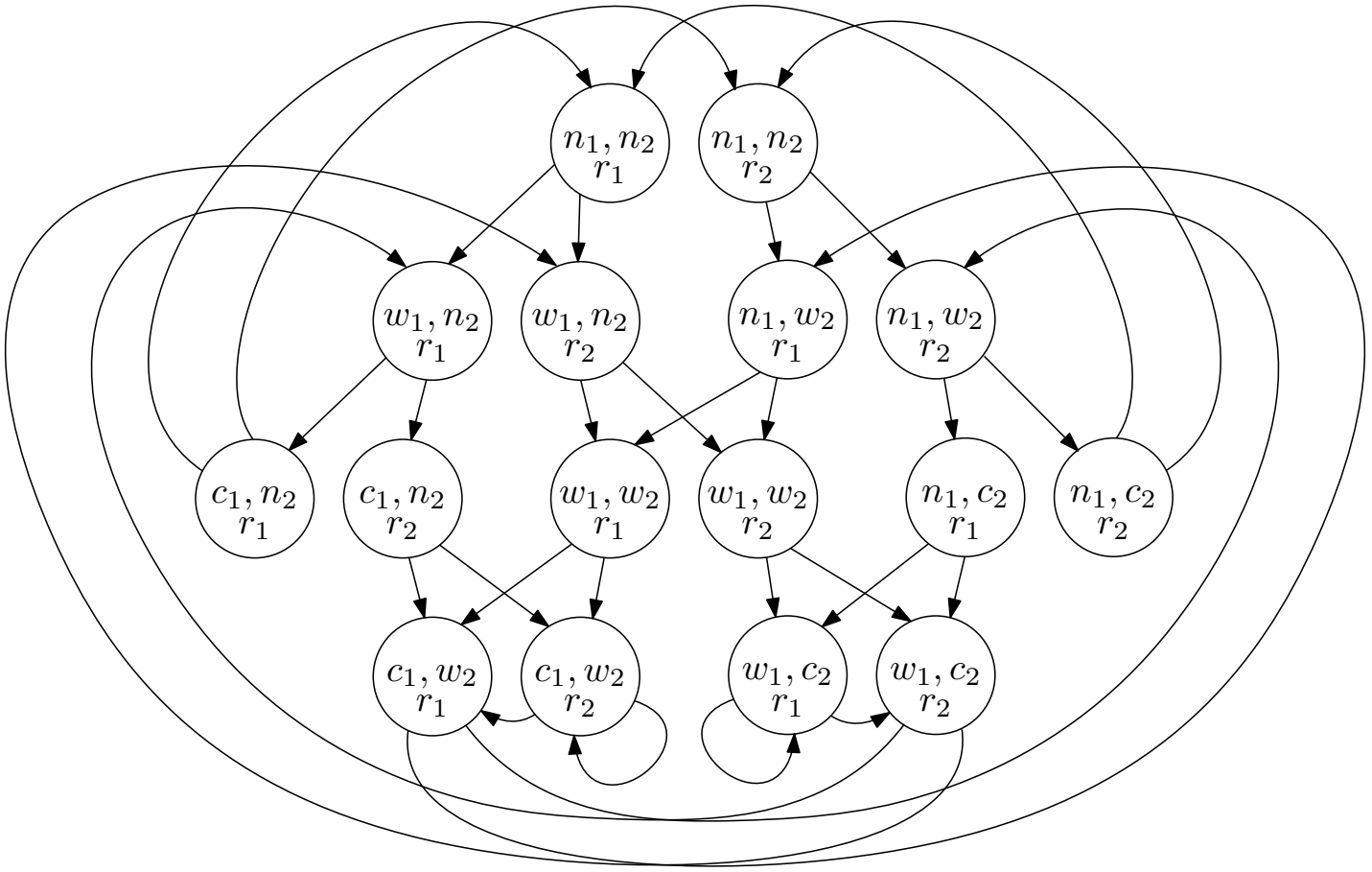
- Um also die Zustände zu finden, in denen EGϕ gilt, betrachtet man den Teilgraphen aller Zustände, die ϕ erfüllen, und sucht nach erreichbaren fairen SCCs.
- In einem Zustand s gilt EXϕ, wenn er einen Folgezustand besitzt, in dem ϕ gilt und von dem außerdem ein fairer Pfad ausgeht. Letzteres stellt man durch Prüfen von EG$\top$ fest.
- E$[\phi$U$\psi]$ wird analog behandelt.

Strong Fairness

Es gibt auch allgemeinere Fairness-Bedingungen der Form: Falls g unendlich oft, dann auch f unendlich oft. In SMV können diese mit `COMPASSION` angegeben werden. Model-Checking solcher *strong fairness* oder "*compassion*" Bedingungen ist in ähnlicher Weise möglich, aber etwas komplizierter.

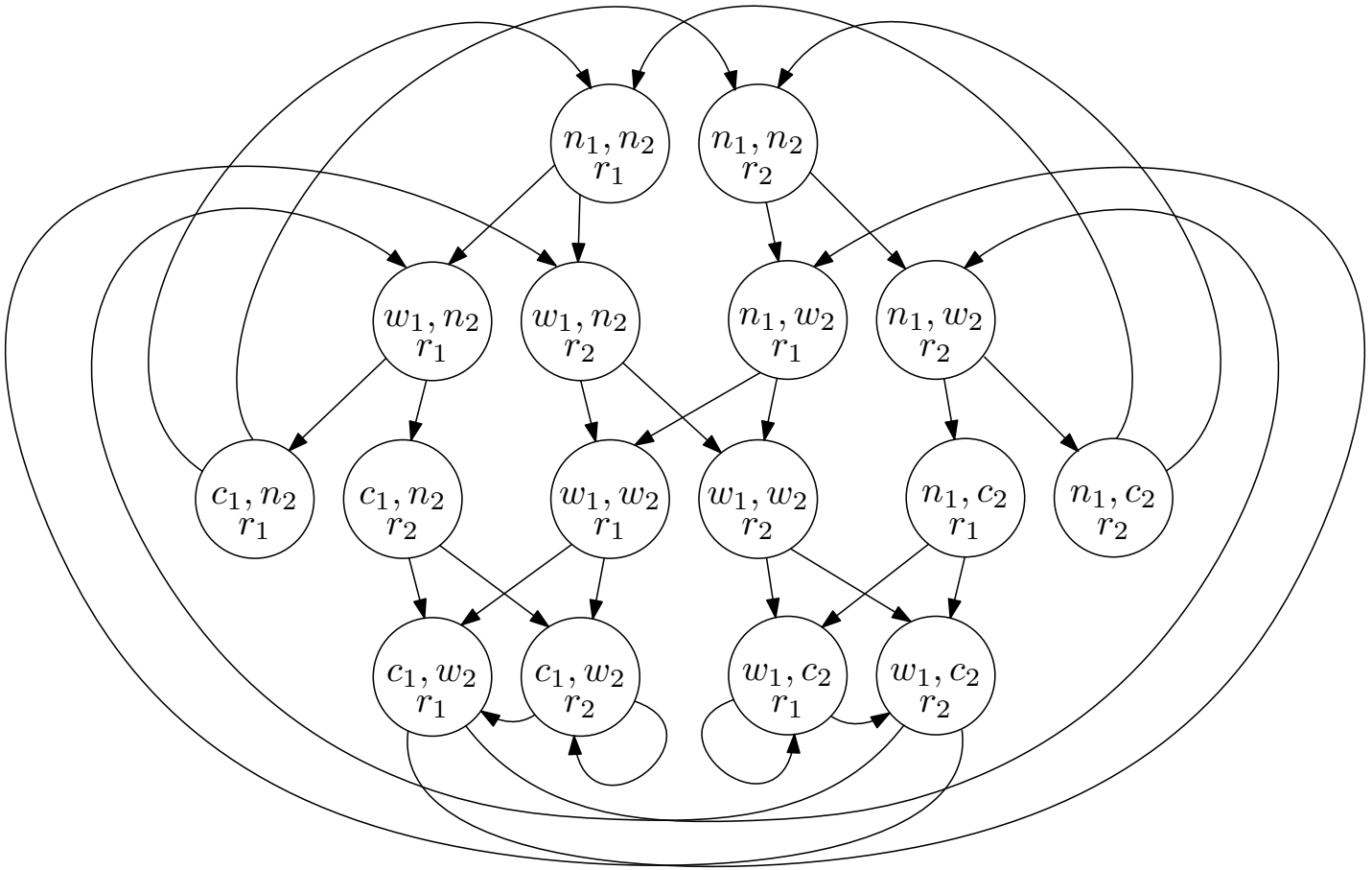
Beispiel: Semaphore mit Fairness

Überprüfe die Liveness-Eigenschaft $w_1 \Rightarrow \mathsf{AF} c_1$ (äquivalent zu $w_1 \Rightarrow \neg\mathsf{EG}\neg c_1$) für das Semaphorenbeispiel von Folie 166.
Transitionssystem: (n — sleep, w — wait, c — work, r — running):



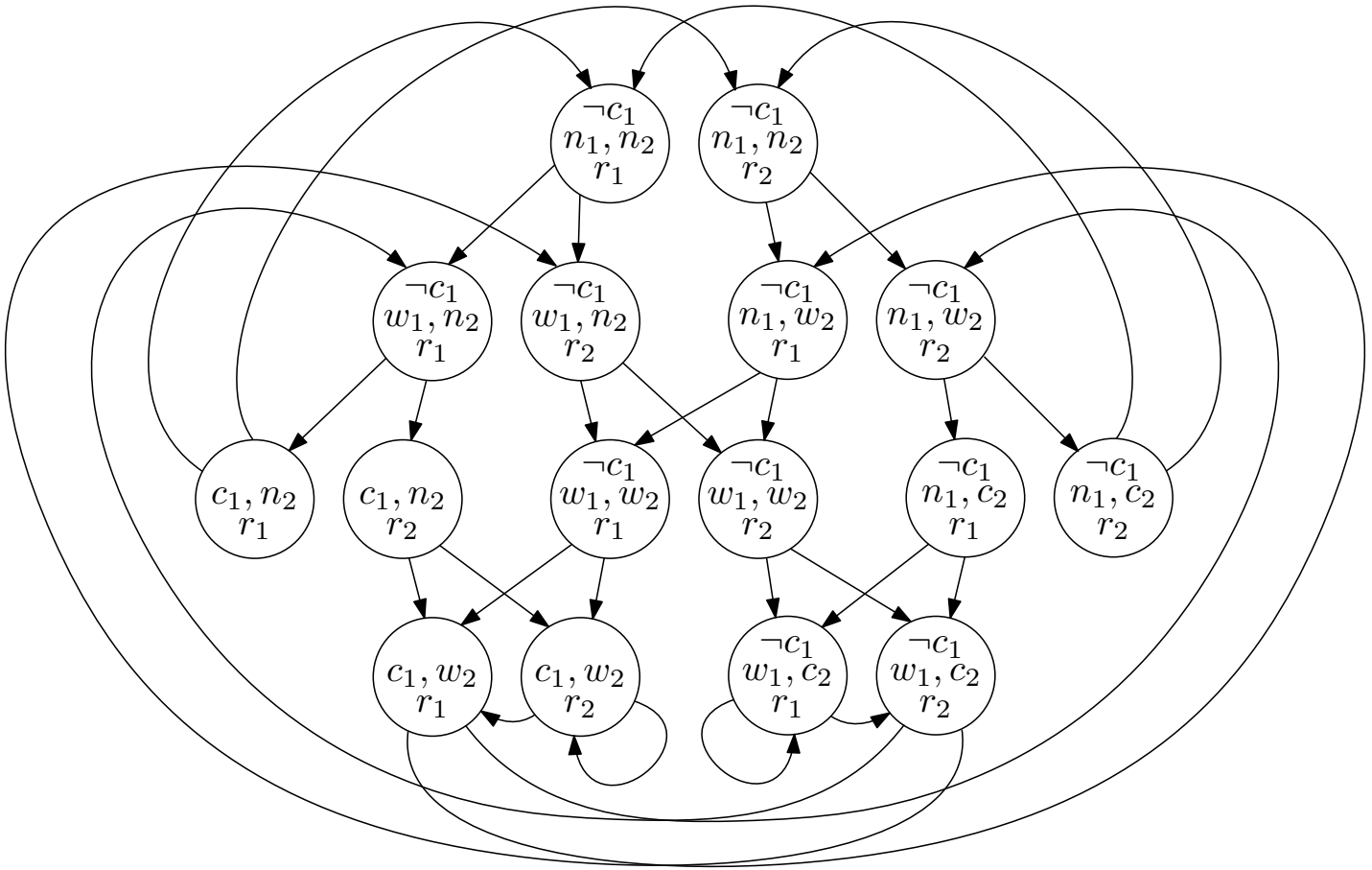
Beispiel: Semaphore mit Fairness

Formel: $\phi = (w_1 \Rightarrow \neg \mathsf{EG} \neg c_1)$
Fairnessbedingungen: $\{r_1\}, \{r_2\}$



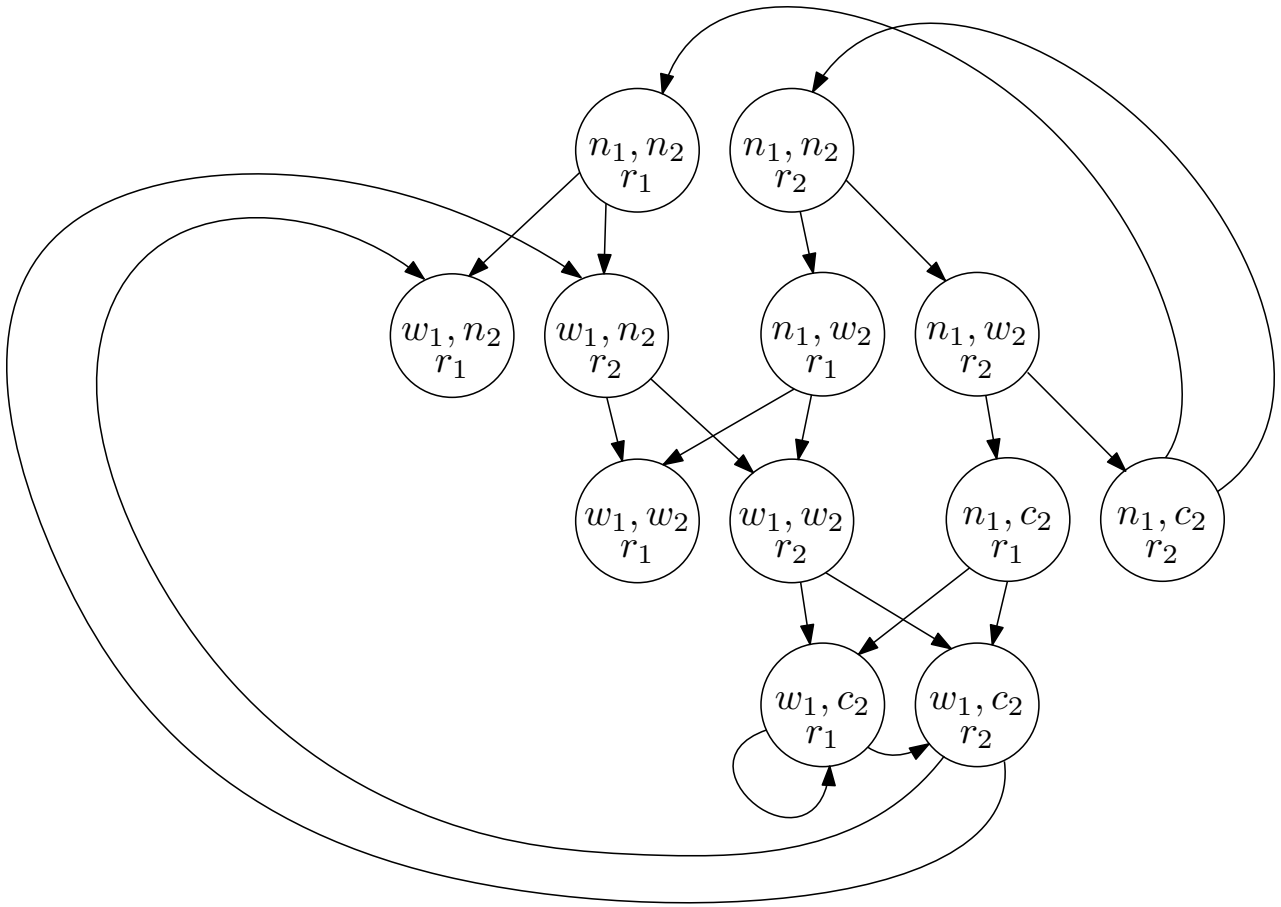
Beispiel: Semaphore mit Fairness

Formel: $\phi = (w_1 \Rightarrow \neg \mathsf{EG} \neg c_1)$
Fairnessbedingungen: $\{r_1\}, \{r_2\}$



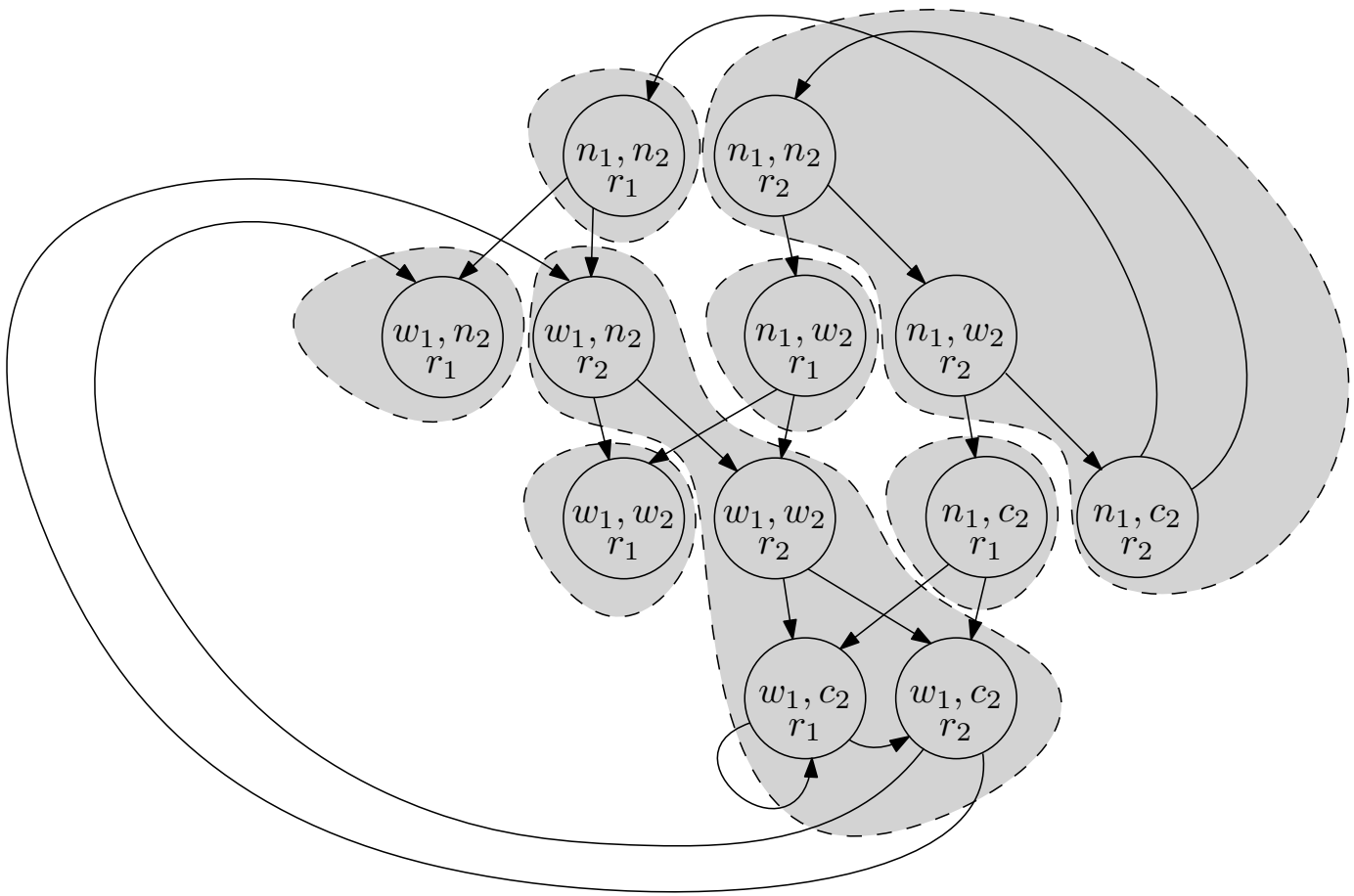
Beispiel: Semaphore mit Fairness

Formel: $\phi = (w_1 \Rightarrow \neg \mathsf{EG} \neg c_1)$
Fairnessbedingungen: $\{r_1\}, \{r_2\}$



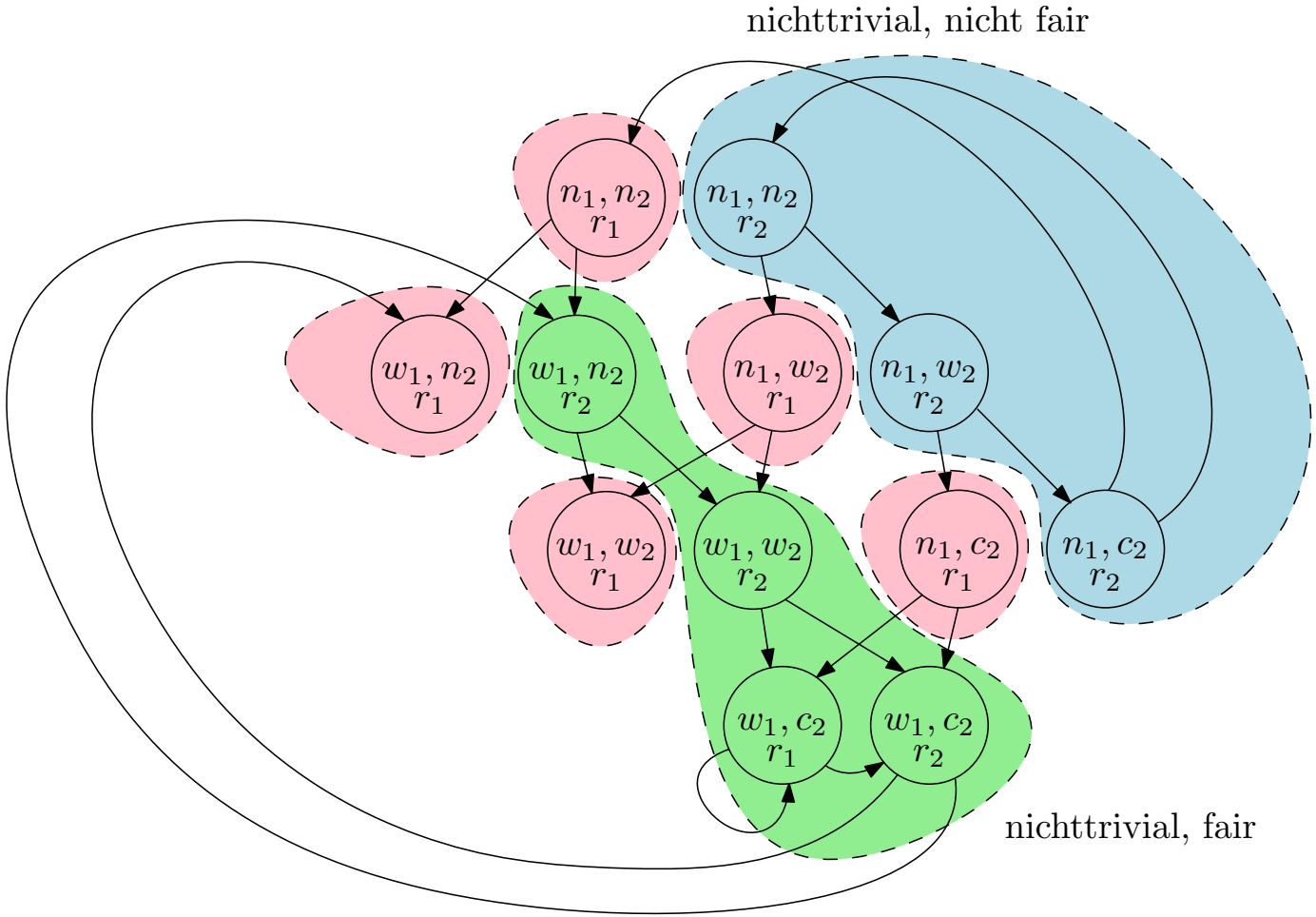
Beispiel: Semaphore mit Fairness

Formel: $\phi = (w_1 \Rightarrow \neg \mathsf{EG} \neg c_1)$
Fairnessbedingungen: $\{r_1\}, \{r_2\}$



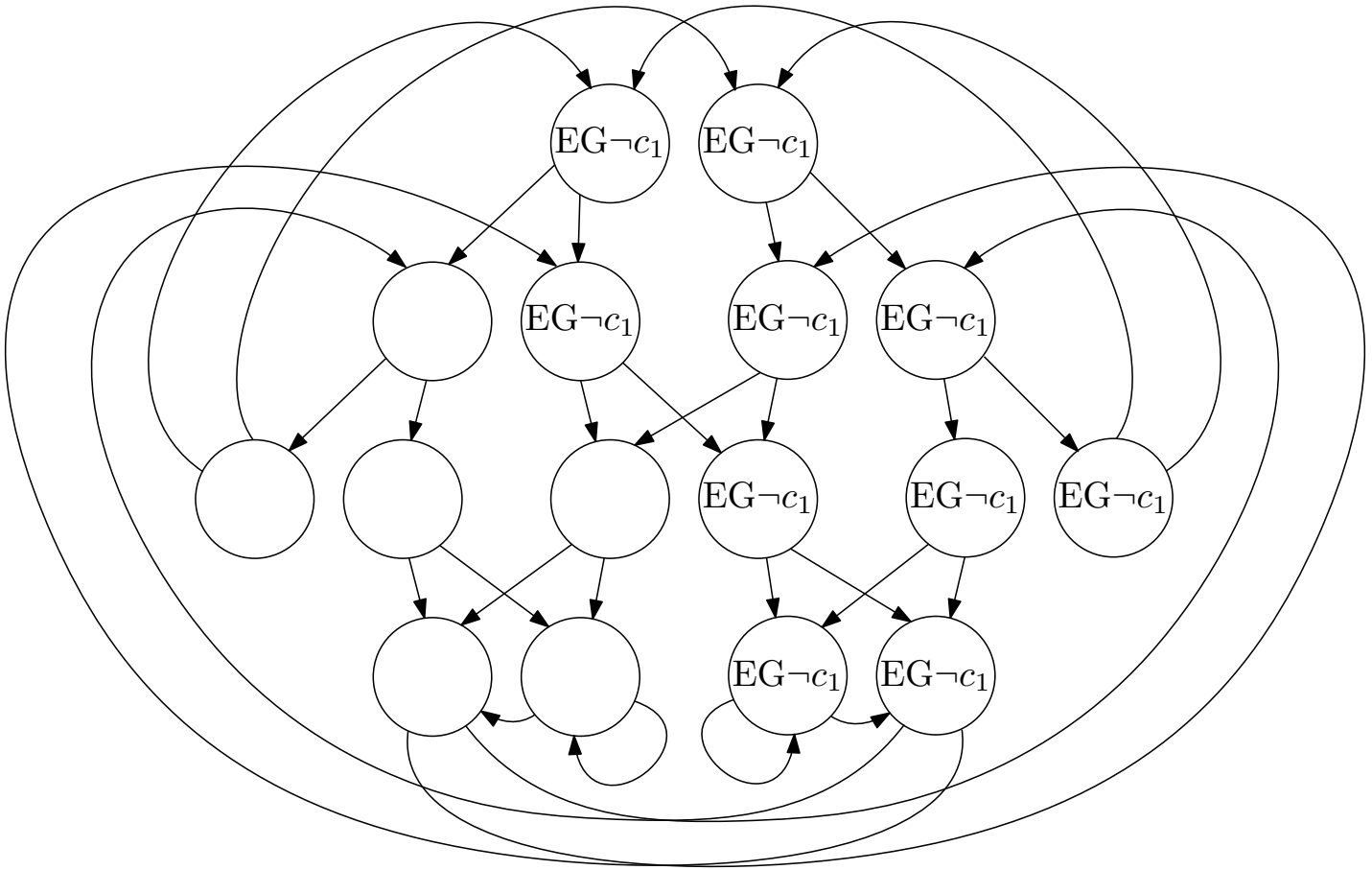
Beispiel: Semaphore mit Fairness

Formel: $\phi = (w_1 \Rightarrow \neg \mathsf{EG} \neg c_1)$
Fairnessbedingungen: $\{r_1\}, \{r_2\}$



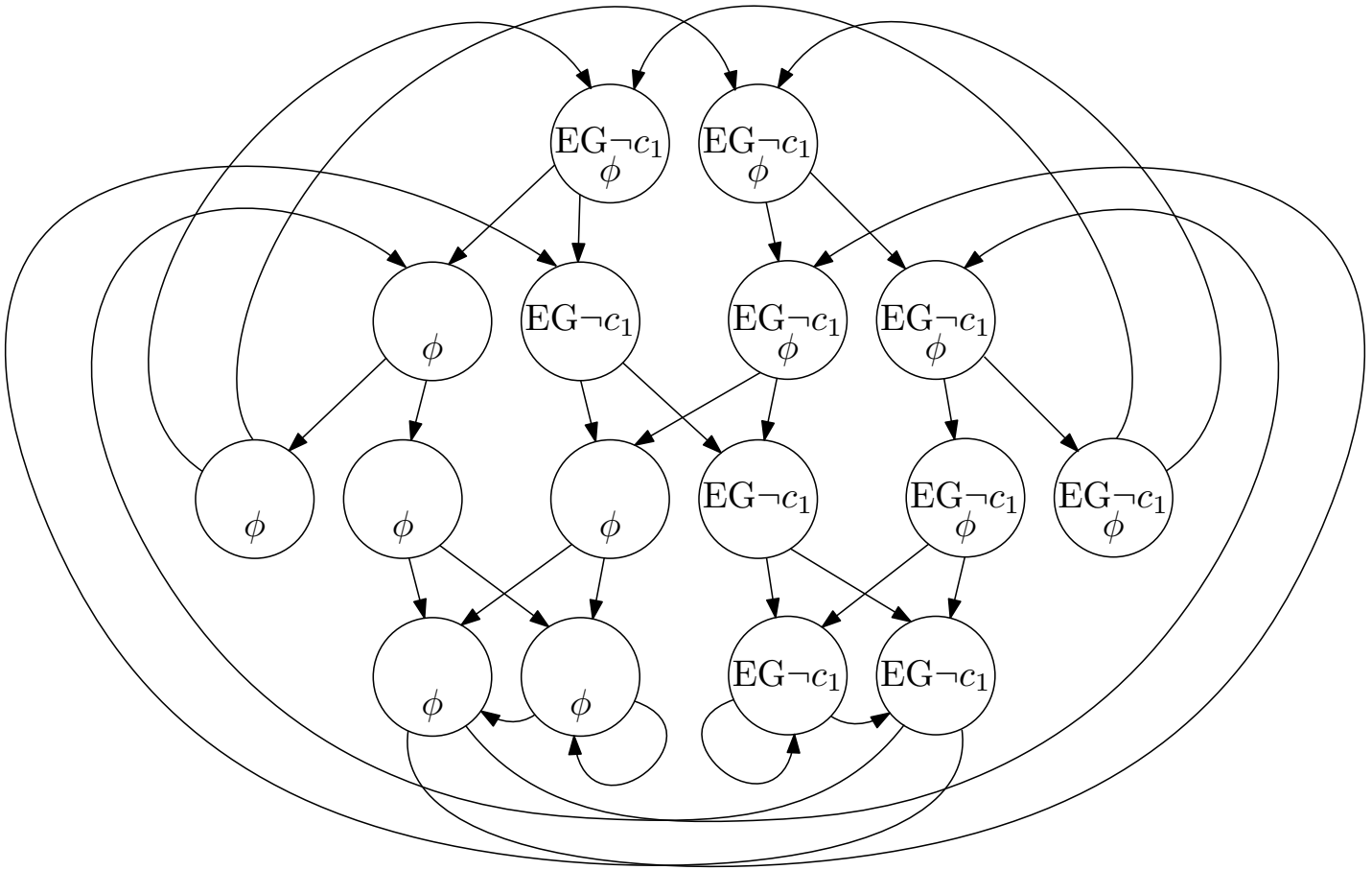
Beispiel: Semaphore mit Fairness

Formel: $\phi = (w_1 \Rightarrow \neg \mathsf{EG} \neg c_1)$
Fairnessbedingungen: $\{r_1\}, \{r_2\}$



Beispiel: Semaphore mit Fairness

Formel: $\phi = (w_1 \Rightarrow \neg \mathsf{EG} \neg c_1)$
Fairnessbedingungen: $\{r_1\}, \{r_2\}$



Beispiel: Semaphore mit Fairness

Das Semaphorenbeispiel von Folie 166 erfüllt also Liveness nicht.

Um Liveness zu garantieren kann man die Implementierung zum Beispiel so ändern, dass ein Prozess nicht mehrfach hintereinander den kritischen Bereich betritt, wenn ein anderer Prozess wartet.

Beispiel: Peterson Algorithmus

Diese Idee ist im z.B. im Peterson-Algorithmus implementiert (siehe Folie 63).

```
MODULE main
VAR
  turn : {0, 1};
  flag : array 0..1 of boolean;
  p0: process proc(0, turn, flag);
  p1: process proc(1, turn, flag);
ASSIGN
  init(turn) := 0;
  init(flag[0]) := FALSE;
  init(flag[1]) := FALSE;
SPEC
  AG(!(p0.critical & p1.critical)) -- safety
SPEC
  AG(p0.waiting -> AF(p0.waiting)) -- liveness
```

Beispiel: Peterson Algorithmus

Pseudocode:

```
0: flag[i] := 1; goto 1
1: turn := 1-i; goto 2
2: if !flag[1-i] then
     goto 4
   else
     goto 3
3: if turn = i then
     goto 4
   else
     goto 2
4: /* kritischer Bereich */
   flag[i] := 0; goto 0
```

SMV-Modul:

```
MODULE proc(i, turn, flag)
VAR line : {0, 1, 2, 3, 4};
ASSIGN
  init(line) := 0;
  next(line) := case
      line=2 & !flag[1-i] : 4;
      line=3 & turn=1-i   : 2;
      line=4              : 0;
      TRUE                : line + 1;
    esac;
  next(turn) := case
      line=1: 1-i;
      TRUE  : turn;
    esac;
  next(flag[i]) := case
      line=0 : TRUE;
      line=4 : FALSE;
      TRUE   : flag[i];
    esac;
DEFINE
  waiting := (line>=1) & (line<=3);
  critical := (line=4);
JUSTICE running
```

Beispiel: Alternating Bit Protokoll

Als Anwendungsbeispiel verifizieren wir einige Eigenschaften des *Alternating Bit Protokoll*.

Problemstellung:

- Ein Sender und ein Empfänger sind über einen bidirektionalen Kanal verbunden.
- Der Kanal kann Nachrichten entweder unverfälscht übermitteln, oder unlesbar machen. Er kann die Nachricht aber nicht unerkannt verfälschen (Sicherstellung dieser Annahme durch redundante Codierung).
- Ist die Nachricht am Kanalende überhaupt lesbar, so stimmt sie mit der Nachricht am Eingang überein.
- Der Kanal kann die Reihenfolge von Nachrichten nicht ändern.
- Der Sender möchte eine Folge von Nachrichten mit Sicherheit zum Empfänger übermitteln.

(typisch für die Transportschicht in Netzwerkprotokollen)

Alternating Bit Protokoll

Idee:

- Der Sender sendet die Nachricht ggf. mehrmals.
- Der Empfänger benutzt die Rückrichtung des Kanals für Empfangsbestätigungen, ebenfalls ggf. mehrmals, da diese auch unlesbar werden können.
- Um wiederholt gesendete Nachrichten erkennen zu können, werden die Nachrichten mit einer Nummer versehen.
- Im Alternating Bit Protokoll genügt eine 1-Bit-Nummer, da sich die Reihenfolge der Nachrichten nicht ändern kann.

Das Alternating Bit Protokoll ist ein Spezialfall eines Schiebefensterprotokolls (Sliding Window Protocol), siehe Vorlesung Rechnernetze.

Das Protokoll

- Der Sender paart die zu sendende Botschaft mit dem Kontrollbit 0 und sendet sie immer wieder.
- Sobald der Empfänger eine unverfälschte Botschaft mit Kontrollbit 0 erhält, so hat er die Botschaft korrekt empfangen und sendet das empfangene Kontrollbit (also 0) zum Sender zurück. Anderenfalls sendet er 1 zurück.
- Empfängt der Empfänger weitere Botschaften mit Kontrollbit 0, so ignoriert er sie und sendet weiterhin das Kontrollbit 0 zurück.
- Empfängt der Sender schließlich das Kontrollbit 0, so kann er davon ausgehen, dass die Botschaft korrekt empfangen wurde und sendet die nächste Botschaft, diesmal gepaart mit Kontrollbit 1, immer wieder an den Empfänger.

Das Protokoll, Forts.

- Sobald der Empfänger eine unverfälschte Botschaft mit Kontrollbit 1 erhält, so hat er die Botschaft korrekt empfangen und sendet das empfangene Kontrollbit (also 1) zum Sender zurück.
- Bis dahin sendet er das alte Kontrollbit, also 0, denn er kann ja nicht sicher sein, dass die Empfangsbestätigung schon durchgedrungen ist.
- Alte Kontrollbits werden vom Sender ignoriert.
- …

Einfache Implementierung

Das Senden und Empfangen von Nachrichten wird durch das Setzen von Programmvariablen erreicht.

Sender:

```
bit_out : {0, 1}
msg_out : {Nothing, a, b, ..., z}
ack_in  : {Nothing, 0, 1}
```

Empfänger:

```
bit_in : {Nothing, 0, 1}
msg_in : {Nothing, a, b, ..., z}
ack_out  : {0, 1}
```

Es gibt jeweils einen parallelen Prozess, der ständig die Werte der `out`-Variablen über den Kanal sendet und die empfangenen Nachrichten in die `in`-Variablen schreibt.

Pseudocode Nachrichtenkanal

```
while (true) {

   // Nachricht und Kontrollbit werden als Einheit behandelt
   if (error() || sender.msg_out == Nothing) {
     receiver.msg_in = Nothing;
     receiver.bit_in = Nothing;
   } else {
     receiver.msg_in = sender.msg_out;
     receiver.bit_in = sender.bit_out;
   }

   if (error()) {
     sender.ack_in = Nothing;
   } else {
     sender.ack_in = receiver.ack_out;
   }
}
```

Pseudocode Sender

```
bit_out = 0;
msg_out = Nothing;

while (true) {
   if (msg_out == Nothing) {
      // Setze zu sendende Nachricht
      msg_out = getNextMessage();
   } else if (ack_in == bit_out) {
      // Nachricht angekommen und bestätigt
      bit_out = 1 - bit_out;
      msg_out = Nothing;
   }
}
```

Pseudocode Empfänger

```
ack_out = 1;
received = Nothing;
expected = 0;

while (true) {
   if (bit_in == expected) {
      received = msg_in;
      // hier etwas mit der emfangenen Nachricht machen
      received = Nothing;
      ack_out = expected;
      expected = 1 - expected;
   }
}
```

Synchrone Komposition

Wir nehmen an, dass die Prozesse *synchron* ausgeführt werden:

- In jedem Taktschritt machen alle drei Prozesse gleichzeitig einen Schritt.
- Am Anfang führen alle drei die Initialisierungsanweisungen aus und gehen bis zum Kopf der `while`-Schleife.
- Danach für jeder Prozess in jedem Schritt den Körper der `while`-Schleife einmal komplett aus.
- Nach jedem Schritt ist jeder Prozess also wieder am Kopf der `while`-Schleife.

Andere Implementierungen (z.B. asynchron oder mit kleineren Schritten) sind ebenso möglich.

Beispielablauf

t	Sender	Empfänger
0	bit_out = 0 msg_out = Nothing ack_in = Nothing	bit_in = Nothing expected = 0 msg_in = Nothing received = Nothing ack_out = 1
1	bit_out = 0 msg_out = a ack_in = Nothing	bit_in = Nothing expected = 0 msg_in = Nothing received = Nothing ack_out = 1
2	bit_out = 0 msg_out = a ack_in = Nothing	bit_in = 0 expected = 0 msg_in = a received = Nothing ack_out = 1
3	bit_out = 0 msg_out = a ack_in = Nothing	bit_in = 0 expected = 1 msg_in = a received = a ack_out = 0
4	bit_out = 0 msg_out = a ack_in = 0	bit_in = 0 expected = 1 msg_in = a received = Nothing ack_out = 0

Beispielablauf, Forts.

5	bit_out = 1 msg_out = Nothing ack_in = 0	bit_in = 0 msg_in = a ack_out = 0 expected = 1 received = Nothing
6	bit_out = 1 msg_out = b ack_in = 0	bit_in = Nothing msg_in = Nothing ack_out = 0 expected = 1 received = Nothing
7	bit_out = 1 msg_out = b ack_in = Nothing	bit_in = Nothing msg_in = Nothing ack_out = 0 expected = 1 received = Nothing
8	bit_out = 1 msg_out = b ack_in = 0	bit_in = 1 msg_in = b ack_out = 0 expected = 1 received = Nothing
9	bit_out = 1 msg_out = b ack_in = 0	bit_in = 1 msg_in = b ack_out = 1 expected = 0 received = b

Im Schritt von 6 auf 7 ist der Kanal beidseitig fehlerhaft.

Modellierung in SMV

```
MODULE main
VAR
  sender   : Sender();
  receiver : Receiver();
  channel  : Channel(sender, receiver);

SPEC AG(AF sender.msg_out=Nothing)
SPEC AG(sender.msg_out=Nothing ->
        AX(sender.msg_out=a ->
          A[receiver.received=Nothing U receiver.received=a]))
SPEC AG(sender.msg_out=Nothing ->
        AX(sender.msg_out=b ->
          A[receiver.received=Nothing U receiver.received=b]))
...
```

Implementierung der fehlenden Module während der Vorlesung.

Symbolisches Model-Checking

Die wichtigste Limitierung des Model-Checking ist das State-Explosion-Problem: Die Anzahl der möglichen Zustände eines Systems wächst exponentiell mit mit der Anzahl seiner Variablen.

Man möchte den Labelling-Algorithmus zum CTL-Model-Checking auf sehr große Transitionssystemen ausführen.

Im *symbolischen Model-Checking* verwendet man BDDs für die kompakte Repräsentation von Zustandsmengen. Das nennt man symbolische Repräsentation.

NuSMV arbeitet intern mit BDDs.

Symbolisches Model-Checking

Im symbolischen Model-Checking wird der Labelling-Algorithmus mit BDDs implementiert.

Gegeben sei eine Interpretation $\mathcal{I}$ mit Transitionssystem $(S, \rightarrow)$. Für jeden Zustand in S wird eine eindeutige Binärkodierung mit n Bits festgelegt, wobei n hinreichend groß gewählt wird.

Aus der Systembeschreibung werden BDDs berechnet:

- *sanity*, *initial* und *next* (wie auf Folie 96).
- Die Interpretation definiert für jede aussagenlogische Variable p eine Menge $\mathcal{I}(p)$ der Zustände, in denen die Variable wahr ist.
  Für jede dieser Mengen $\mathcal{I}(p)$ wird ein BDD $\mathcal{B}(p)$ erzeugt.

Symbolisches Model-Checking

Der Labelling-Algorithmus wird nun mit BDDs implementiert.

Für jede CTL-Formel ϕ wird ein BDD $\mathcal{B}(\phi)$ berechnet, welches die Menge der Zustände beschreibt, in denen ϕ wahr ist.

Der Algorithmus funktioniert durch Rekursion über den Formelaufbau. Bei Eingabe ϕ wird das BDD $\mathcal{B}(\phi)$ zurückgegeben.

Berechnung der Vorgänger

Im Algorithmus müssen wir die Menge der Vorgänger einer Menge von Zuständen berechnen.

Ist B eine Zustandsmenge $B \subseteq S$, so schreiben wir $\mathsf{EX}(B)$ für die Menge der Vorgänger von B:

$$\mathsf{EX}(B) := \{s \in S \mid \exists s' \in B.\, s \to s'\}$$

Ist b ein BDD für B, so ist

$$\mathsf{EX}(b) := \mathit{sanity} \wedge (\exists x'_0. \ldots \exists x'_{n-1}.\, b' \wedge \mathit{next})$$

ein BDD für $\mathsf{EX}(B)$.
Dabei ist b' das BDD, das aus b durch Umbenennung von x_i in x'_i für $i = 0, \ldots, n-1$ entsteht.

Berechnung der Vorgänger: Korrektheit

Angenommen eine Belegung von $x_0, \ldots, x_{n-1}$ macht

$$\mathit{sanity} \wedge (\exists x'_0. \ldots \exists x'_{n-1}. b' \wedge \mathit{next})$$

wahr. Dann:

- $x_0, \ldots, x_{n-1}$ macht *sanity* wahr, also müssen die $x_0, \ldots, x_{n-1}$ einen Zustand $s \in S$ kodieren.
- $x_0, \ldots, x_{n-1}$ macht $(\exists x'_0. \ldots \exists x'_{n-1}. b' \wedge \mathit{next})$ wahr. Also muss es eine Belegung der Variablen $x'_0, \ldots, x'_{n-1}$ geben, so dass alle Variablen zusammen dann $b' \wedge \mathit{next}$ wahr machen.
- Dass b' wahr ist, impliziert, dass die Werte von $x'_0, \ldots, x'_{n-1}$ einen Zustand in B kodieren, nennen wir ihn s'.
- Dass *next* wahr ist, liefert dann zusätzlich, dass $s \to s'$ gilt.

Das heißt, die Belegung von $x_0, \ldots, x_{n-1}$ kodiert dann einen Zustand, der einen Nachfolger in B hat.

Berechnung der Vorgänger: Korrektheit

Angenommen ein Zustand $s \in S$ hat einen Nachfolger $s' \in B$.
Dann:

- Belege die Variablen $x_0, \ldots, x_{n-1}$ so, dass sie s kodieren. Für diese Belegung muss *sanity* wahr sein.
- Die Belegung von $x_0, \ldots, x_{n-1}$ macht auch $(\exists x'_0 . \ldots \exists x'_{n-1} . b' \wedge \mathit{next})$ wahr
- Als Zeugen können wir die Belegung der Variablen $x'_0, \ldots, x'_{n-1}$ verwenden, die s' kodiert.
  - Für diese Belegung muss *b'* wahr sein, da s' in B liegt.
  - Für die Belegung aller Variablen muss auch *next* wahr sein, da s' ein Nachfolger von s ist.

Unter der Annahme macht die Belegung von $x_0, \ldots, x_{n-1}$, die s kodiert, dann auch $\mathit{sanity} \wedge (\exists x'_0 . \ldots \exists x'_{n-1} . b' \wedge \mathit{next})$ wahr.

Labelling-Algorithmus mit BDDs

Für eine gegebene Formel ϕ ist das BDD $\mathcal{B}(\phi)$ zu berechnen.

Fallunterscheidung über ϕ:

- $\mathcal{B}(p)$ wurde bereits konstruiert.
- $\mathcal{B}(\neg\phi) := \mathit{sanity} \wedge \neg\mathcal{B}(\phi)$
  (berechne BDD $\mathcal{B}(\phi)$, negiere das BDD und berechne die Konjunktion mit *sanity*)
- $\mathcal{B}(\phi \wedge \psi) := \mathcal{B}(\phi) \wedge \mathcal{B}(\psi)$
  (berechne BDDs $\mathcal{B}(\phi)$ und $\mathcal{B}(\psi)$ und dann ihre $\wedge$-Verknüpfung)
  Andere Boolesche Operationen werden ebenfalls mit den analogen Operationen auf BDDs behandelt.
- $\mathcal{B}(\mathsf{EX}\phi) := \mathsf{EX}(\mathcal{B}(\phi))$.
  (berechne $\mathcal{B}(\phi)$ und dann ein BDD für die Menge der Vorgänger)

Labelling-Algorithmus mit BDDs: E$[\phi \mathsf{U} \psi]$

Berechne zunächst BDDs $\mathcal{B}(\phi)$ und $\mathcal{B}(\psi)$.

- Definiere:

$$B_0 := \mathcal{B}(\psi)$$
$$B_{n+1} := \mathcal{B}(\psi) \vee (\mathcal{B}(\phi) \wedge \mathsf{EX}(B_n))$$

- Das BDD B_n repräsentiert die Menge der Zustände s, für die ein Pfad $s = s_0 \to s_1 \to \cdots \to s_k$ mit $k \leqslant n$ existiert, so dass $s_k \models \psi$ und $s_i \models \phi$ für alle $i < k$.
- Berechne nun $B_1, B_2, \ldots, B_{n_0}$, wobei n_0 die kleinste Zahl mit $B_{n_0} = B_{n_0+1}$ ist.
- Setze $\mathcal{B}(\mathsf{E}[\phi \mathsf{U} \psi]) := B_{n_0}$.

Labelling-Algorithmus mit BDDs: AFϕ

Berechne zunächst das BDD $\mathcal{B}(\phi)$.

- Definiere:

$$B_0 := \mathcal{B}(\phi)$$
$$B_{n+1} := \mathcal{B}(\phi) \vee \mathsf{AX}(B_n)$$

  wobei $\mathsf{AX}(B_n) := \mathit{sanity} \wedge \neg\mathsf{EX}(\mathit{sanity} \wedge \neg B_n)$.
- Das BDD B_n repräsentiert die Menge der Zustände s, so dass für alle Pfade der Form $s = s_0 \to s_1 \to \cdots \to s_n$ ein $k \leqslant n$ existiert mit $s_k \models \phi$.
- Berechne nun $B_1, B_2, \ldots, B_{n_0}$, wobei n_0 die kleinste Zahl mit $B_{n_0} = B_{n_0+1}$ ist.
- Setze $\mathcal{B}(\mathsf{AF}\phi) := B_{n_0}$.

Labelling-Algorithmus mit BDDs: EGϕ

Berechne zunächst das BDD $\mathcal{B}(\phi)$.

- Definiere:

$$E_0 := \mathcal{B}(\phi)$$
$$E_{n+1} := \mathcal{B}(\phi) \wedge \mathsf{EX}(E_n)$$

- Berechne nun $E_1, E_2, \ldots, E_{n_0}$, wobei n_0 die kleinste Zahl mit $E_{n_0} = E_{n_0+1}$ ist.
- Setze $\mathcal{B}(\mathsf{EG}\phi) := E_{n_0}$.
- Man zeigt durch Induktion über n: Das BDD *sanity* $\wedge\ \neg E_n$ ist gleich dem BDD B_n in der Berechnung von $\mathsf{AF}(\neg\phi)$.
  Korrektheit folgt dann wegen $\mathsf{EG}\phi = \neg\mathsf{AF}(\neg\phi)$.

Zusammenfassung Kapitel II

- Transitionssysteme als Modell nebenläufiger Systeme
- Die Temporallogik CTL
- Der Labelling-Algorithmus für CTL-Model-Checking
- CTL mit Fairness
- Das System SMV
- Alternating Bit Protokoll
- Symbolisches Model-Checking für CTL