Kapitel II

Temporallogik und Model Checking

Inhalt Kapitel II

Motivation

Die Modellierungen nebenläufiger Systeme aus dem ersten Kapitel waren bereits Beispiele von Model Checking:

- Die Modellierungen mit SAT - Solver waren Instanzen von *Bounded Model Checking* (da die Simulationszeit beschränkt war).
- Die Modellierungen mit BDDs waren Instanzen von *Symbolic Model Checking* (da Zustandsmengen nicht explizit, sondern "symbolisch" repräsentiert wurden)

Allgemein bedeutet Model Checking die automatische Überprüfung von Eigenschaften nebenläufiger Systeme.

Arten von Eigenschaften

Grundsätzliche Arten von 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.

Sinn der Temporallogik

Temporallogik erlaubt die kompakte Notation von Eigenschaften von Systemabläufen.
Es gibt verschiedenen Temporallogiken, die einfachste ist CTL ("computation tree logic"). Die im Semaphorbeispiel bewiesene Eigenschaft lautet in CTL:

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

Der Operator AG besagt, dass auf allen (A) Pfaden (= Programmabläufen) stets (G) die Eigenschaft $\neg \mathit{undesired}$ gilt.

Eine Lebendigkeitseigenschaft

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

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

Der Operator $\mathsf{EF}(\phi)$ besagt, dass ein Fortsetzungspfad existiert (E), auf dem irgendwann ("*finally*",F) die Eigenschaft ϕ gilt.
Es ist $\mathsf{AG}(\phi) \Longleftrightarrow \neg\mathsf{AG}(\neg\phi)$.

Syntax von CTL

Gegeben sei eine Menge aussagenlogischer Variablen (z.B. die q_{pz} und die s_w).
Die Menge der CTL-Formeln über diesen Variablen ist durch folgende Grammatik gegeben.

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

Hier ist p eine aussagenlogische Variable und $\oplus$ irgendeiner der zweistelligen Boole'schen Operatoren. Insbesondere ist also jede aussagenlogische Formel auch eine CTL-Formel.

Beispiel: $\mathsf{AG}(p \Rightarrow \mathsf{A}[p \mathsf{U} (\neg p \wedge \mathsf{A}[\neg p \mathsf{U} q])])$
Gegenbeispiel: $\mathsf{A}[p]$ und $\phi \mathsf{U} \psi$ sind keine CTL-Formeln!

Informelle Semantik der CTL-Formeln

CTL Formeln werden relativ zu einem gegebenen nebenläufigen System interpretiert.
Es muss klar sein, in welchem Systemzustand welche aussagenlogische Variable gilt und welche nicht.
Eine CTL-Formel ϕ definiert dann eine Menge von Zuständen des Systems, bzw. kann man bei gegebenem Zustand sagen, ob die Formel ϕ in diesem Zustand gilt, oder nicht.
In einem Zustand s gilt...

- ...$\mathrm{AX}\phi$, wenn ϕ in allen unmittelbaren Folgezuständen von s gilt.
- ...$\mathrm{EX}\phi$, wenn ϕ in einem der unmittelbaren Folgezustände von s gilt.
- ...$\mathrm{AG}\phi$, wenn ϕ auf allen von s aus erreichbaren Zuständen gilt.
- ...$\mathrm{EF}\phi$, wenn man von s aus einen Zustand erreichen kann, in dem ϕ gilt.

Informelle Semantik der CTL-Formeln, Forts.

- . . . 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.
- . . . A[ϕUψ], 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[ϕUψ], wenn von s aus die Ausführung so fortgesetzt werden kann, dass irgendwann ψ gilt und bis dahin stets ϕ gilt.

Beispiele:

- AG((*close_door* $\vee$ *safe* $\wedge$ $\neg$*open_door*) $\Rightarrow$ AX*safe*) $\wedge$ AG(*heat* $\Rightarrow$ *safe*)
- *floor=2* $\wedge$ *direction=up* $\wedge$ *buttonpressed=5* $\Rightarrow$ *A*[*direction=up* U *floor=5*]
- AF*fertig*

Transitionssystem

Definition

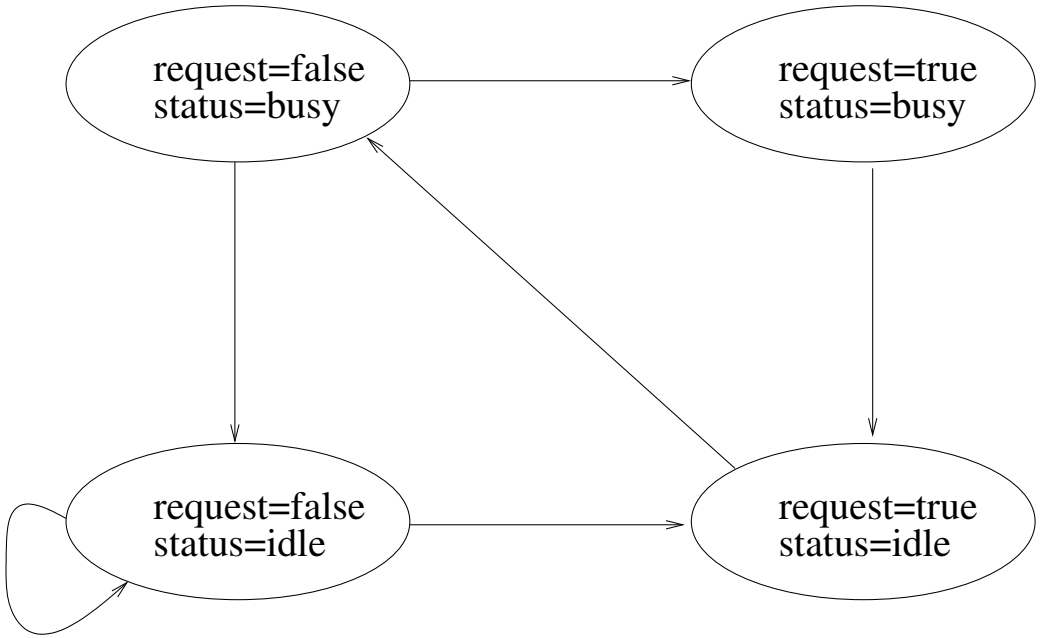
Ein Transitionssystem ist ein Paar $(S, \rightarrow)$, wobei

- S eine Menge von Zuständen ist
- $\rightarrow \subseteq S \times S$ 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 einer Deadlock-Zustandes s_d mit $s_d \rightarrow s_d$ künstlich hergestellt werden.

Beispiel

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

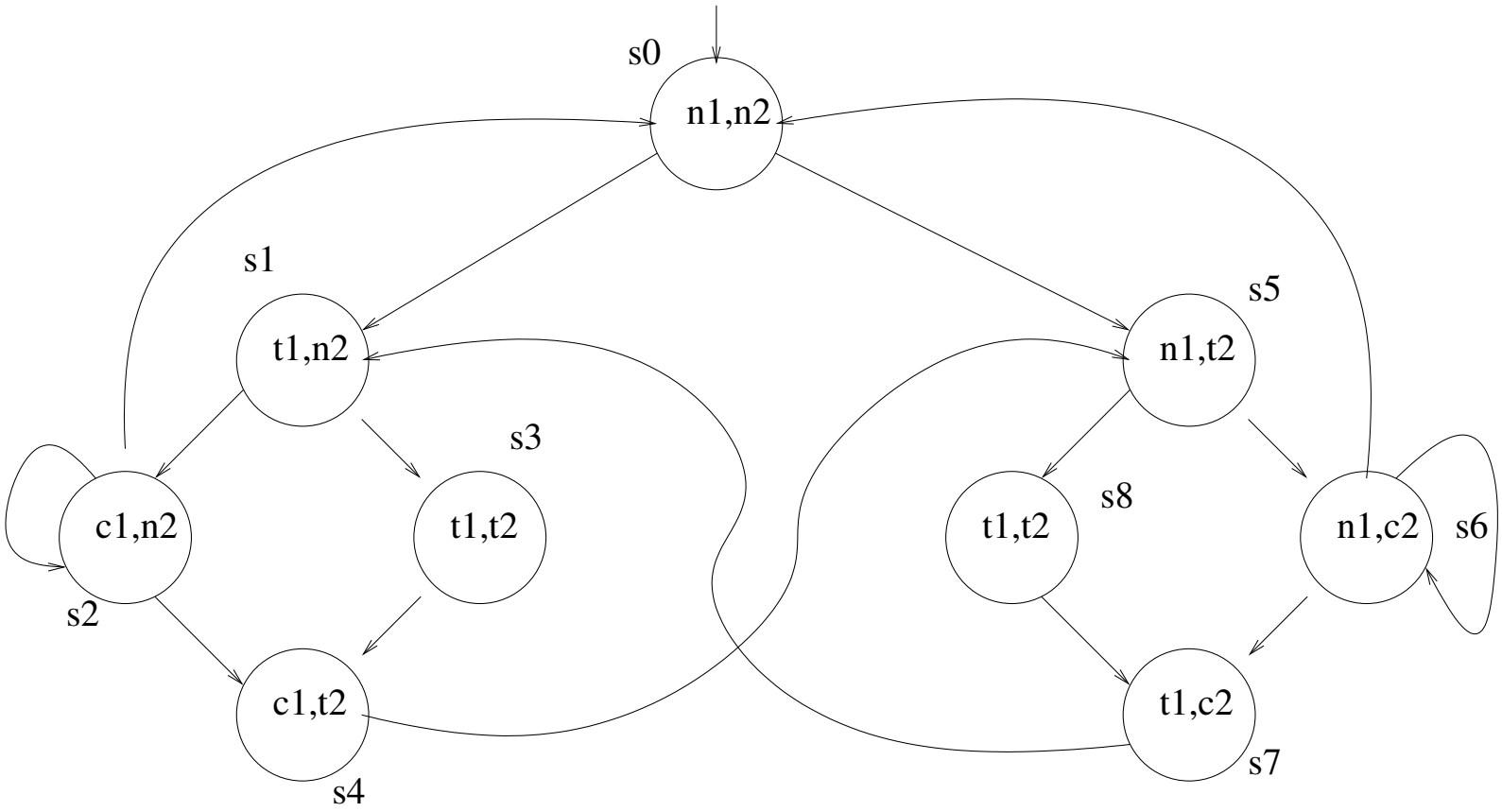


Weitere Beispiele

- Semaphor: $S = \{(sem, proc_0, proc_1) \mid sem \in \{free, occ\}, \forall i \in \{0,1\}.proc_i \in \{sleep, wait, work\}\}$; hier: $|S| = 8$
- Peterson: $S = \{(flag, turn, line) \mid \forall i \in \{0,1\}.flag_i \in \{true, false\}\ \&turn \in \{0,1\}\ \&\ line_i \in \{0,1,2,3,4\}\}$; hier: $|S| = 2^2 \cdot 2 \cdot 5^2 = 200$
- Kohlkopf: $S = \{pos \mid \forall x \in \{wolf, ziege, kohlkopf, bauer\}.pos_x \in \{links, rechts\}\}$; Hier: $|S| = 2^4 = 16$.

NB: *flag*, *pos*, *line* sind Tupel von Variablen oder Arrays.
NB: Die Transitionsrelation $\rightarrow$ ist hier jeweils weggelassen.

Weiteres Beispiel: Mutual Exclusion



Formale Semantik von CTL

Interpretation

Sei eine Menge von aussagenlogischen Variablen gegeben.

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

Man schreibt auch $s \models_{\mathcal{I}} p$ (lies "s erfüllt p (in $\mathcal{I}$)") für $s \in \mathcal{I}(p)$.

Bei gegebener Interpretation $\mathcal{I}$ mit $Tr(\mathcal{I}) = (S, \rightarrow)$ definiert jede CTL-Formel ϕ auch eine Menge von Zuständen $\mathcal{I}(\phi)$.

Auch hier schreibt man häufig $s \models_{\mathcal{I}} \phi$ für $s \in \mathcal{I}(\phi)$.

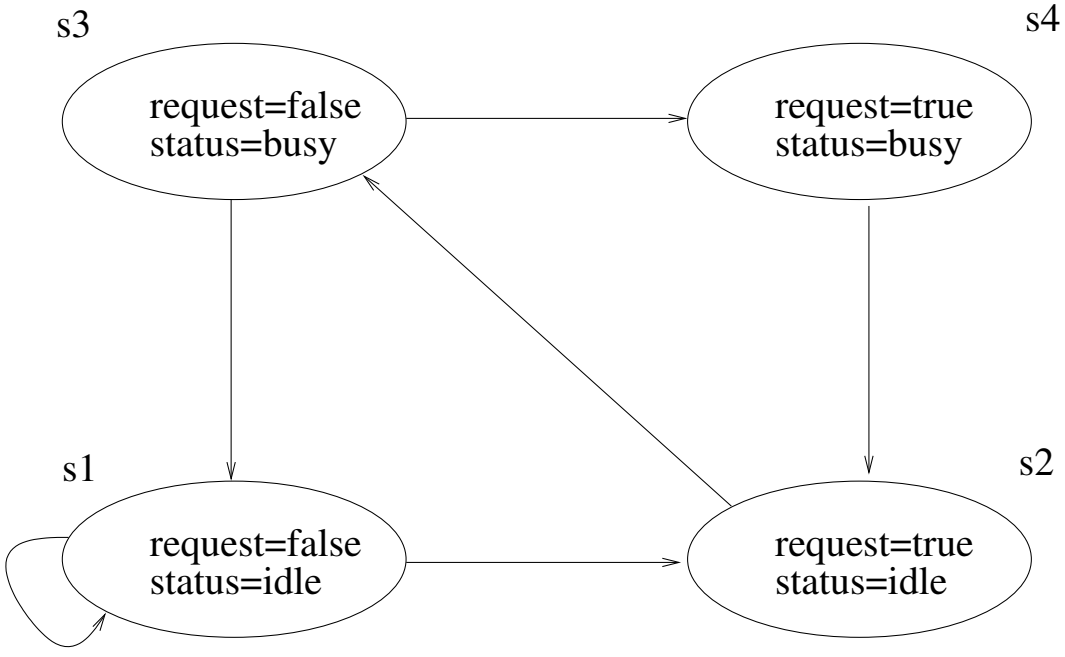
Definition der Semantik

- $s \models_{\mathcal{I}} p$ ist bereits durch $\mathcal{I}$ festgelegt.
- $s \models_{\mathcal{I}} \neg\phi$ genau dann, wenn $s \not\models_{\mathcal{I}} \phi$
- $s \models_{\mathcal{I}} \phi \wedge \psi$ genau dann, wenn $s \models_{\mathcal{I}} \phi$ und $s \models_{\mathcal{I}} \psi$.
- die anderen Boole'schen Operatoren $\vee, \Rightarrow$, etc. sind analog.
- $s \models_{\mathcal{I}} \mathsf{EX}\phi$ genau dann, wenn $s' \in S$ existiert mit $s \to s'$ und $s' \models_{\mathcal{I}} \phi$.
- $s \models_{\mathcal{I}} \mathsf{AX}\phi$ genau dann, wenn für alle $s' \in S$ mit $s \to s'$ gilt: $s' \models_{\mathcal{I}} \phi$.

Definition der Semantik, Forts.

- $s \models_{\mathcal{I}} \mathrm{AG}\phi$ genau dann, wenn für alle $s' \in S$ mit $s \rightarrow^* s'$ gilt: $s' \models_{\mathcal{I}} \phi$. NB: $\rightarrow^*$ ist die reflexive, transitive Hülle von $\rightarrow$.
- $s \models_{\mathcal{I}} \mathrm{EF}\phi$ genau dann, wenn $s' \in S$ existiert mit $s \rightarrow^* s'$ und $s' \models_{\mathcal{I}} \phi$.
- $s \models_{\mathcal{I}} \mathrm{EG}\phi$ genau dann, wenn ein unendlicher Pfad $s = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ existiert, derart dass $s_i \models_{\mathcal{I}} \phi$ für alle i.
- $s \models_{\mathcal{I}} \mathrm{AF}\phi$ genau dann, wenn für alle unendlichen Pfade $s = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ jeweils ein i existiert, sodass $s_i \models_{\mathcal{I}} \phi$.

Beispiel



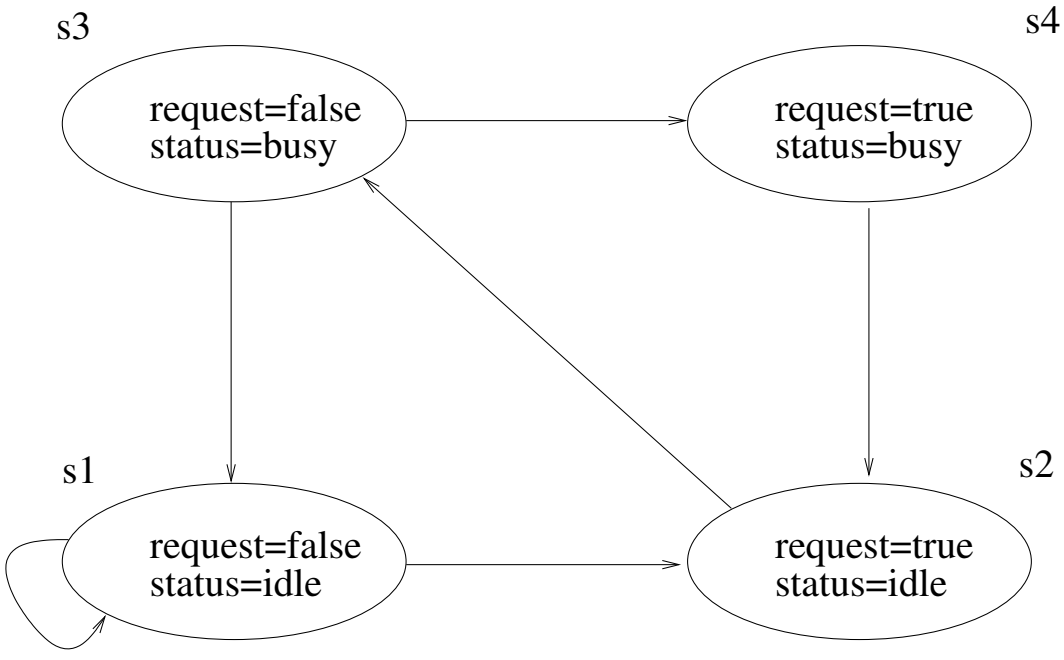
Die aussagenlogischen Variablen: *request* und *status=idle* und *status=busy* werden wie im Diagramm interpretiert, also z.B. $s_1 \not\models_{\mathcal{I}}$ *request* und $s_1 \models_{\mathcal{I}}$ *status=idle*. Es gilt:

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

Semantik der Until-Formeln

- $s \models_{\mathcal{I}} \mathrm{E}[\phi \mathrm{U} \psi]$ genau dann, wenn ein Pfad $s = s_0 \to s_1 \to s_2 \to s_3 \to \cdots \to s_n$ existiert, derart dass $s_n \models_{\mathcal{I}} \psi$ und für $i < n$ gilt $s_i \models_{\mathcal{I}} \phi$.
- $s \models_{\mathcal{I}} \mathrm{A}[\phi \mathrm{U} \psi]$ genau dann, wenn für alle unendlichen Pfade $s = s_0 \to s_1 \to s_2 \to s_3 \to \ldots$ jeweils ein n existiert, sodass $s_n \models_{\mathcal{I}} \psi$ und für $i < n$ gilt $s_i \models_{\mathcal{I}} \phi$.

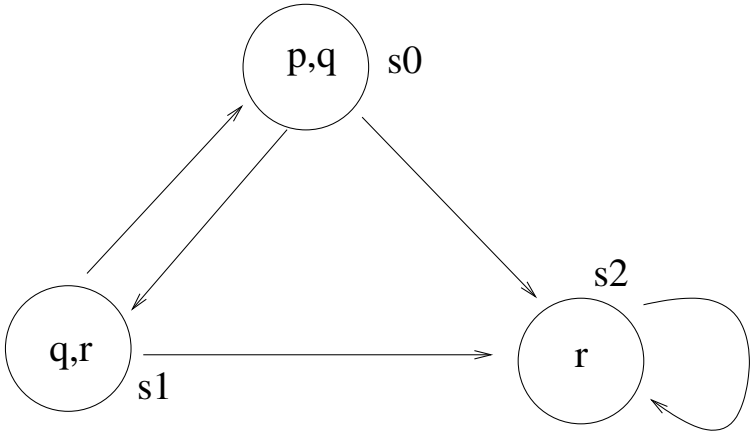
Beispiel



Es gilt:

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

Beispiel



$s_0 \models_{\mathcal{I}} p \wedge q$
$s_0 \models_{\mathcal{I}} \top$
$s_0 \models_{\mathcal{I}} \neg \mathsf{AX}(q \wedge r)$
$s_1 \models_{\mathcal{I}} \mathsf{EG} r$
$s_0 \models_{\mathcal{I}} \mathsf{AF} r$
$s_0 \models_{\mathcal{I}} \mathsf{A}[p \mathsf{U} r]$

$s_0 \models_{\mathcal{I}} p \wedge \neg r$
$s_0 \models_{\mathcal{I}} \mathsf{EX}(q \wedge r)$
$s_0 \models_{\mathcal{I}} \neg \mathsf{EF}(p \wedge r)$
$s_2 \models_{\mathcal{I}} \mathsf{AG} r$
$s_0 \models_{\mathcal{I}} \mathsf{E}[(p \wedge q)\ U r]$

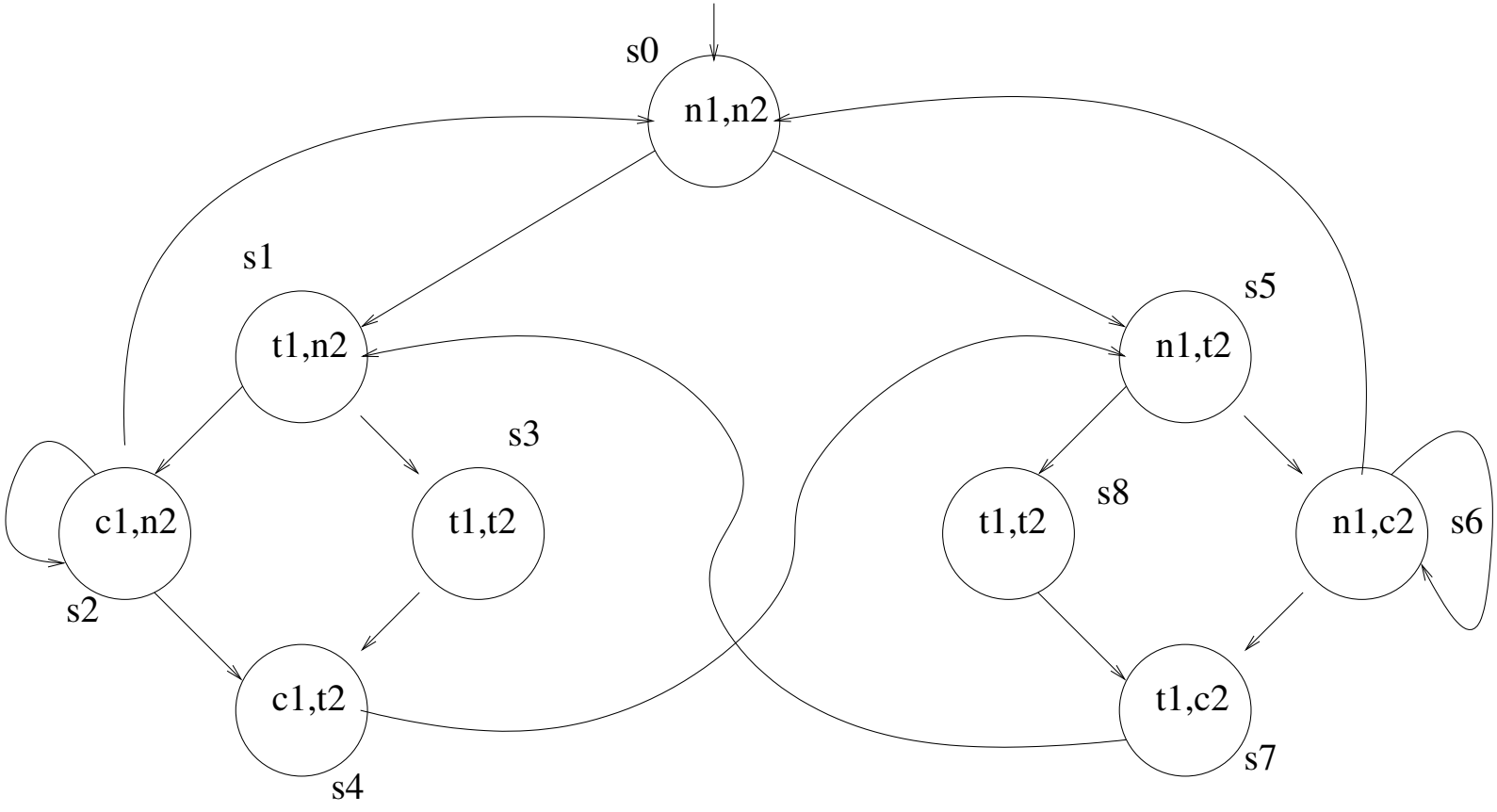
Äquivalenz

Äquivalenz von CTL-Formeln

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

- Sind $\phi \iff \psi$ aussagenlogisch äquivalente Formeln, so auch als CTL-Formeln. Z.B.: $\mathsf{AG}(p) \vee \mathsf{AG}(p) \iff \mathsf{AG}(p)$.
- $\mathsf{AG}(\phi) \iff \neg\mathsf{EF}(\neg\phi)$
- $\mathsf{AF}(\phi) \iff \neg\mathsf{EG}(\neg\phi)$
- $\mathsf{EF}(\phi) \iff \neg\mathsf{AG}(\neg\phi)$
- $\mathsf{EG}(\phi) \iff \neg\mathsf{AF}(\neg\phi)$
- $\mathsf{AG}(\phi \wedge \psi) \iff \mathsf{AG}(\phi) \wedge \mathsf{AG}(\psi)$
- $\mathsf{AF}(\phi) \iff \mathsf{A}[\top \mathsf{U} \phi]$
- $\mathsf{EF}(\phi) \iff \mathsf{E}[\top \mathsf{U} \phi]$
- $\mathsf{A}[\phi \mathsf{U} \psi] \iff \neg(\mathsf{E}[\neg\psi \mathsf{U}(\neg\phi \wedge \neg\psi)] \vee \mathsf{EG}(\neg\psi))$

Mutual Exclusion Beispiel



- Safety: $\mathsf{AG}(\neg(c_1 \wedge c_2))$
- Liveness: $\mathsf{AG}(t_1 \Rightarrow \mathsf{AF}c_1)$
- Non-blocking: $\mathsf{AG}(n_1 \Rightarrow \mathsf{EX}t_1)$
- No strict sequencing: $\mathsf{EF}(c_1 \wedge \mathsf{E}[c_1 \mathsf{U}(\neg c_1 \wedge \mathsf{E}[\neg c_2 \mathsf{U} c_1])])$

Problemstellung

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

Gefragt: Gilt $s \models_{\mathcal{I}} \phi$.

Labelling Algorithmus: Grundidee

Gegeben $\mathcal{I}$ mit $Tr(\mathcal{I}) = (S, \rightarrow)$ und Formel ϕ_0.

Man berechnet $\mathcal{I}(\phi)$ für alle Teilformeln ϕ von ϕ_0.
Bildlich gesprochen beschriftet (labelt) man die Zustände S mit denjenigen Teilformeln, die dort gelten.

Hat man dies für die unmittelbaren Teilformeln einer Formel ϕ bereits getan, so kann man es für die Formel ϕ selbst auch tun, indem man die Definition der Semantik benutzt.

Aufgrund der Äquivalenzen können wir annehmen, dass unsere Formeln nur die Operatoren $\neg, \wedge, \bot, \mathrm{AF}, \mathrm{E}[-\mathrm{U}-]$ verwenden.

Labelling Algorithmus: Details

- $\bot$: Markiere keinen Zustand mit $\bot$
- p: Markiere Zustände mit aussagenlogischen Variablen wie von der Interpretation vorgegeben.
- $\neg\phi$: Ist s nicht mit ϕ markiert, so markiere s mit $\neg\phi$.
- $\phi \wedge \psi$: Ist s mit ϕ und mit ψ markiert, so markiere s mit $\phi \wedge \psi$.
- EXϕ: Ist einer der Folgezustände von s mit ϕ markiert, so markiere s mit EXϕ.

Natürlich muss man erst alle Teilformeln einer Formel verarbeitet haben.
Man darf also nicht gleich am Anfang sagen: "s nicht mit ϕ markiert, also schnell mit $\neg\phi$ markieren."

Labelling Algorithmus: Details, Forts.

- AFϕ: Ist ein Zustand mit ϕ markiert, so markiere ihn mit AFϕ. Sind alle unmittelbaren Folgezustände eines Zustandes s auf diese Weise bereits mit AFϕ markiert, so markiere auch s mit AFϕ. Fahre so fort, bis "nichts mehr geht".
- E[ϕUψ]: Ist ein Zustand mit ψ markiert, so markiere ihn mit E[ϕUψ]. Ist einer der Folgezustände eines Zustandes s auf diese Weise bereits mit E[ϕUψ] markiert und ist s bereits mit ϕ markiert, so markiere s auch mit E[ϕUψ]. Fahre so fort, bis "nichts mehr geht".

Die Komplexität dieses Algorithmus ist $O(f \cdot V \cdot (V + E))$, wobei f die Größe der Ausgangsformel, V die Zahl der Zustände und E die Zahl der Transitionen ist.

Effizientere Version

Wäre AF nicht vorhanden, so liefe das gegebene Verfahren unter Verwendung von Rückwärts-Breitensuche in $O(f \cdot (V + E))$: Ist ein Knoten mit $E[\phi \mathrm{U} \psi]$ markiert, so werden alle seine Vorgänger, die auch mit ϕ markiert sind, selbst mit $E[\phi \mathrm{U} \psi]$ markiert.

Leider funktioniert das für AF nicht, da ja alle Nachfolger und nicht nur einer markiert sein müssen.

Man kann aber anstelle von AF den Operator EG verwenden (wegen der Äquivalenz $\mathrm{AF}\phi \Longleftrightarrow \neg \mathrm{EG}(\neg \phi)$ und für EG folgendes direkte und effizientere Verfahren einsetzen:

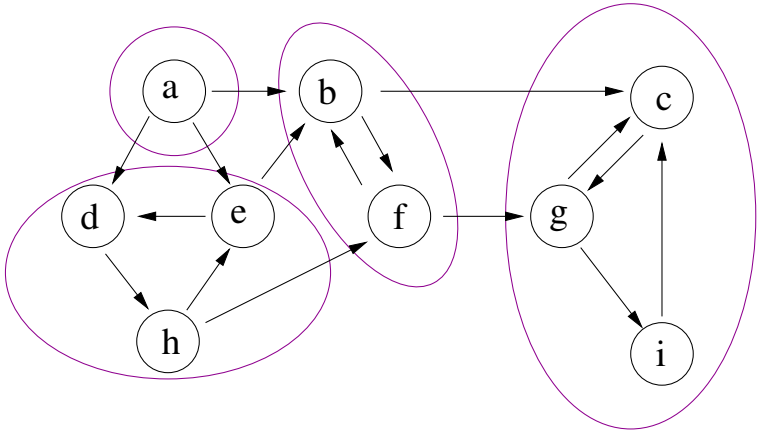
Effiziente Behandlung von EGϕ

Um die mit EGϕ zu markierenden Zustände zu finden, nachdem bereits mit ϕ markiert wurde:

- Streiche (temporär) alle Zustände, die nicht mit ϕ markiert sind.
- Zerlege den so erhaltenen Graphen in starke Zusammenhangskomponenten.
- Markiere alle Zustände mit EGϕ von denen aus (im neuen Graphen) eine echte starke Zusammenhangskomponente erreicht werden kann.

Wdh.: Starke Zusammenhangskomponenten

- Eine starke Zusammenhangskomponente (SCC) ist eine maximal große Teilmenge U von Zuständen, so dass für je zwei Knoten $s_1, s_2 \in U$ gilt: s_2 ist von s_1 aus erreichbar und umgekehrt.
- Eine starke Zusammenhangskomponente ist echt, wenn sie nicht nur aus einem Zustand besteht oder aus einen Zustand s mit $s \to s$ besteht, also einen unendlichen Pfad enthält.
- Mit Tarjan's Algorithmus (siehe Cormen) kann ein Graph in linearer Zeit in SCCs zerlegt werden.



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:

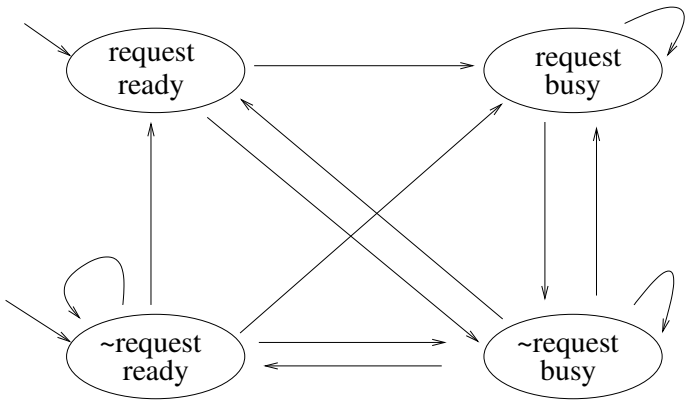
- Ausnutzen von Symmetrie
- Abstraktion
- Symbolische Repräsentation von Zuständen (wie bei der Modellierung mit BDDs im ersten Kapitel)

Das System SMV

- In SMV kann durch programmiersprachliche Notation ein Transitionssystem definiert werden.
- Die Gültigkeit von CTL-Formeln kann dann automatisch überprüft werden.
- Wir verwenden die aktuelle Implementierung NuSMV. Für Details siehe Doku.

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)
```



Dasselbe mit expliziter 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)
```

Erklärung der expliziten 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)
```

- Das Modul definiert eine Interpretation und eine CTL-Formel.
- Zustände sind alle Belegungen der mit `VAR` deklarierten Variablen.
- Aussagenlogische Variablen sind die `boolean`-Variablen, sowie Gleichungen zwischen Variablen und / oder Konstanten. Sie werden in der offensichtlichen Weise interpretiert. Z.B. sind im Zustand $s := [\texttt{request} \mapsto \mathit{true}, \texttt{status} \mapsto \texttt{ready}]$ die aussagenlogischen Variablen `request` und `status=ready` wahr.

Erklärung der expliziten Notation, Forts.

```
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)
```

- Will man keine so strukturierten Zustände, sondern z.B. fünf abstrakte Zustände $s_1, \ldots, s_5$, so deklariert man eine einzige Variable

  ```
  VAR
     zustand : {s1, s2, s3, s4, s5}
  ```

  In der Praxis kommt das selten vor.
- Der Boole'sche Ausdruck in der `INIT`-Klausel definiert die Startzustände.
- Der Boole'sche Ausdruck in der `TRANS`-Klausel definiert die Transitionsrelation. Alle Zustandspaare, die ihn wahrmachen, sind Transitionen.
  Mit der Syntax `next` bezieht man sich auf den (die Komponenten des) Folgezustand(s).

Erklärung der expliziten Notation, Forts.

```
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)
```

- Es sind mehrere `INIT` und `TRANS` Klauseln erlaubt, diese verstehen sich als Konjunktion.
- In der `SPEC`-Klausel steht eine CTL-Formel in der üblichen Syntax. Mehrere `SPEC`-Klauseln verstehen sich als Konjunktion.
- Mit `NuSMV` *datei*`.smv` ruft man NuSMV auf. Das in der Eingabedatei enthaltene Modell wird syntax- und typüberprüft und sodann wird durch Model-Checking ermittelt, ob alle `SPEC`-Formeln in allen Startzuständen (`INIT`) erfüllt sind. Falls nicht, so wird ein Gegenbeispiel konstruiert.

Erklärung der impliziten Notation

```
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)
```

Die `INIT` und `TRANS`-Klauseln können durch `ASSIGN`-Klauseln ersetzt werden.
In diesen wird der Anfangs- und Folgezustand durch Wertzuweisung definiert. Nichtdeterminismus (mehrere Anfangs- und Folgezustände) wird durch Mengennotation erfasst.

Semaphor mit zwei Prozessen in impliziter Notation

Wir verwenden eine zusätzliche Variable `running`, die angibt, welcher Prozess gerade eine Transition gemacht hat.

```
MODULE main
VAR
  p0 : {sleep, wait, work};
  p1 : {sleep, wait, work};
  s  : {free, occ};
  running : {0, 1};
ASSIGN init(p0):=sleep; init(p1):=sleep; init(s):=free;
next(p0) := case
              next(running)=0 & p0=sleep        : wait;
              next(running)=0 & p0=wait&s=free  : work;
              next(running)=0 & p0=work         : sleep;
              TRUE                              : p0;
            esac;
```

Semaphor mit zwei Prozessen, Forts.

```
next(p1) := case
              next(running)=1 & p1=sleep        : wait;
              next(running)=1 & p1=wait&s=free  : work;
              next(running)=1 & p1=work         : sleep;
              TRUE                          : p1;
            esac;
next(s) := case
              p0=wait & s=free & next(running)=0 : occ;
              p1=wait & s=free & next(running)=1 : occ;
              p0=work & next(running)=0          : free;
              p1=work & next(running)=1          : free;
              TRUE                           : s;
   esac;

SPEC AG(!(p0=work & p1=work))
```

NB: Die Bedingungen in den case-Ausdrücken, müssen alle Möglichkeiten erfassen. Sie werden der Reihe nach abgearbeitet, bis eine zutrifft. Die Bedingung TRUE trifft natürlich immer zu.

Weitere Spezifikationen

- Safety: `AG(!(p0=work & p1=work))`. Gilt.
- Liveness: `AG(p0=wait -> AF p0=work)`. Gilt nicht.
- Liveness: `AG(p0=wait -> EF p0=work)`. Gilt.
- Non-blocking: `AG(p0=sleep -> EX p0=wait)`. Gilt
- No strict sequencing:

```
EF(p0=work & E[p0=work U
        (!(p0=work) & E[!(p1=work) U p0=work])])
```

  Gilt.

Version mit Modulen

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

Das Modul `prc` hat drei Parameter (untypisiert, “by reference”).

Version mit Modulen, Forts.

```
MODULE main
VAR
  s  : {free, occ};
  running : {0, 1};
  p0 : prc(running,s,0);
  p1 : prc(running,s,1);
ASSIGN
  init(s) := free;
  next(s) := case
               p0.p=wait & s=free & next(running)=0 : occ;
               p1.p=wait & s=free & next(running)=1 : occ;
      p0.p=work & next(running)=0           : free;
      p1.p=work & next(running)=1           : free;
               TRUE                           : s;
   esac;
SPEC
  AG(!(p0.p=work & p1.p=work))
```

Das Modulkonzept

- Deklariert man eine Variable eines Moduls, so entspricht das der Deklaration aller Variablen in dem Modul. Mit Dot-Notation kann man auf diese zugreifen.
- Die Transitionsrelationen werden konjungiert (ver-undet). Das entspricht der *synchronen Komposition*: zu jedem Zeitpunkt machen alle beteiligten Module gleichzeitig je einen Schritt. Wegen der `running`-Variable macht natürlich nur einer der Prozesse einen echten Schritt, der andere macht einen Dummy-Schritt, bei dem sich der Zustand nicht ändert.
- Module können (a priori untypisierte) Parameter enthalten. Zur Typüberprüfung werden die Moduldefinitionen einfach in ihren Verwendungskontext eingesetzt.
- Die `ASSIGN` Definitionen in verschiedenen Instanzen dürfen sich nicht widersprechen. Im Beispiel ist es nicht möglich die Variable `s` in `prc` zu verändern. Die statische Prüfung von SMV erkennt nicht, dass jeweils nur ein Modul aktiv ist.

Prozesse in SMV

- Deklariert man Modulinstanzen mit `process`, so werden automatisch entsprechende `running` Variablen erzeugt;
- pro Zeiteinheit ist immer nur höchstens ein "Prozess" aktiv.
- In solchen Modulen können dann auch gemeinsame (shared) Variablen (wie `s`) verändert werden.
- Die `running` Variablen sind vom Typ `boolean`: es gibt eine für jedes Modul (einschl. `main`). Die `running`-Variablen der "Prozesse" dürfen nicht gleichzeitig gesetzt (`TRUE`) sein und stets ist mindestens eine gesetzt.

Semaphor 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;
```

Semaphor mit Prozessen, Forts.

```
MODULE main
VAR
  s  : {free, occ};
  p0 : process prc(s);
  p1 : process prc(s);
ASSIGN
  init(s) := free;
SPEC
  AG(!(p0.p=work & p1.p=work))
```

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.

Beispiel für Fairness

Nimmt man in `prc` die Klausel `JUSTICE running` hinzu, so muss jeder Prozess immer wieder drankommen und die Spezifikation

```
SPEC
AG(p0.p=wait -> AF p0.p=work)
```

wird wahr!
Bei Nicht-Verwendung von `process` nimmt man stattdessen die Klausel `JUSTICE running=pid` hinzu.

NB: Fairness kann nicht mit den vorhandenen CTL Operatoren ausgedrückt werden. Man muss den Model-Checking Algorithmus geeignet modifizieren.

Wolf, Ziege, Kohlkopf

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

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

Wolf, Ziege, Kohlkopf, Forts.

```
MODULE main
  VAR
     wolf : process passagier(bauer);
     ziege : process passagier(bauer);
     kohlkopf : process passagier(bauer);
     bauer : process bauer();

  INVAR (ziege.pos=wolf.pos -> bauer.pos=ziege.pos)&
        (ziege.pos=kohlkopf.pos -> bauer.pos=ziege.pos)

  SPEC
     !EF(
         bauer.pos = rechts & ziege.pos=rechts &
         wolf.pos=rechts & kohlkopf.pos=rechts)
```

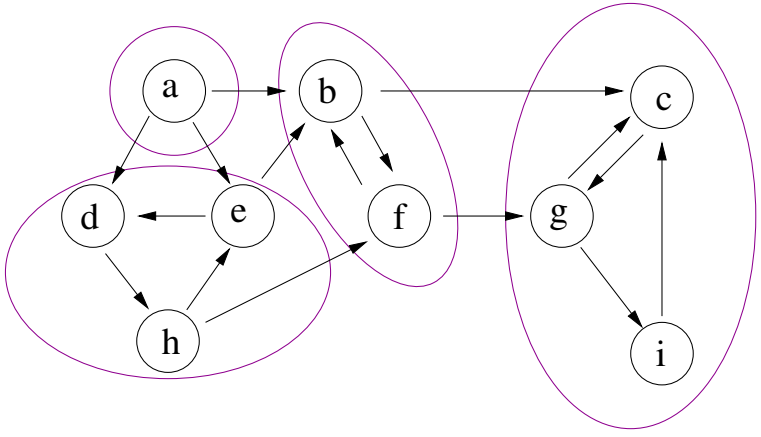
Man erhält die gesuchte Lösung als “Gegenbeispiel”.

CTL mit Fairness

- Gegeben sei ein Transitionssystem $(S, \rightarrow)$.
- Eine Fairness-Bedingung (auch Justice-Bedingung) ist eine Teilmenge $f \subseteq S$ von Zuständen.
- Ein unendlicher Pfad in $(S, \rightarrow)$ ist *fair* bezüglich f, wenn er immer wieder (also unendlich oft) Zustände aus f besucht.
- Ein Pfad ist fair bezüglich einer Liste $(f_1, \ldots, f_n)$ von Fairness-Bedingungen, wenn er bzgl. jeder einzelnen fair ist.
- Sind solche Fairnessbedingungen gegeben, so schränkt man die Semantik von CTL auf solche fairen Pfade ein: $\mathrm{EG}\phi$ gilt dann in s, wenn es einen von s ausgehenden fairen Pfad gibt, entlang dem stets ϕ gilt. $\mathrm{E}[\phi \mathrm{U} \psi]$ gilt dann, wenn es einen fairen Pfad von s aus gibt, auf irgendwann ψ gilt und bis dahin ϕ.

Model-Checking mit Fairness

- Eine (echte) 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 $f_1 = \{a, d, f\}, f_2 = \{d, b\}$. 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.

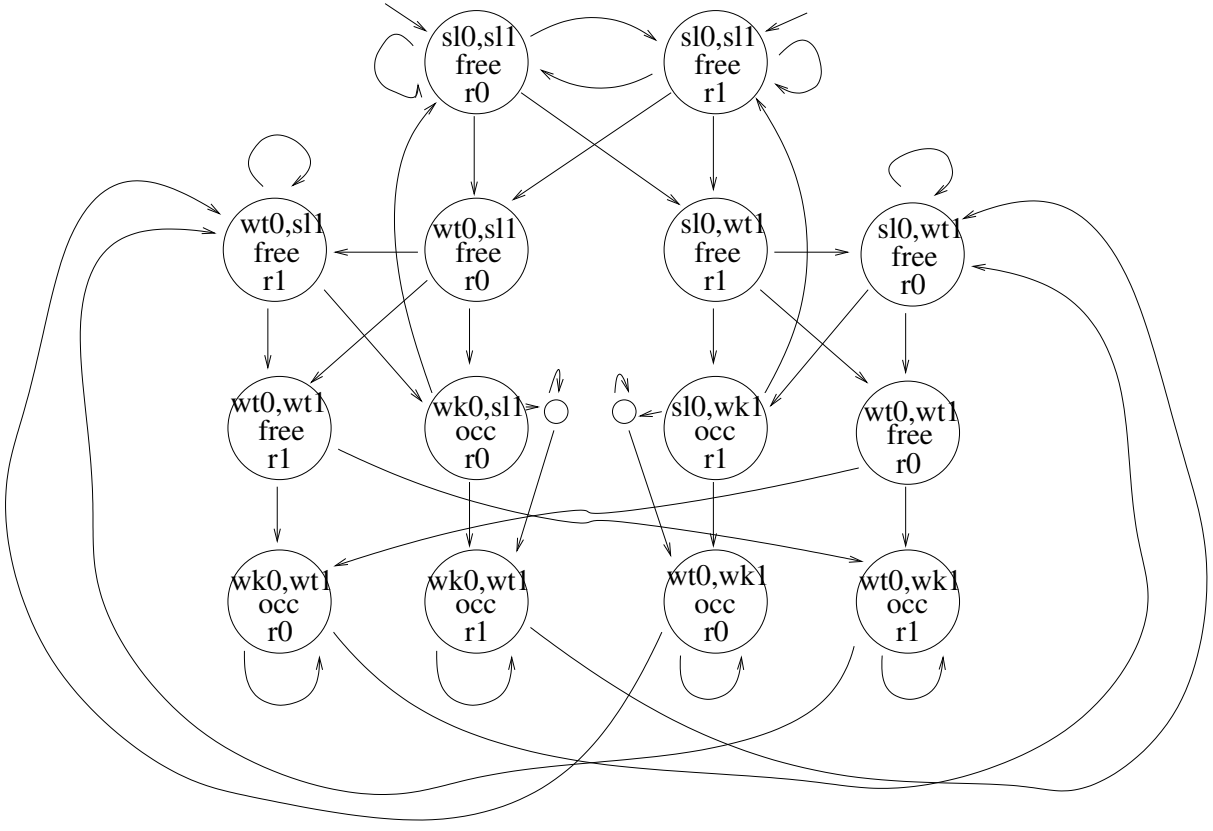
Model-Checking mit Fairness

- Um also die Zustände zu finden, in denen EGϕ gilt, schränkt man zunächst auf alle Zustände, die ϕ erfüllen ein 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[ϕUψ] 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.

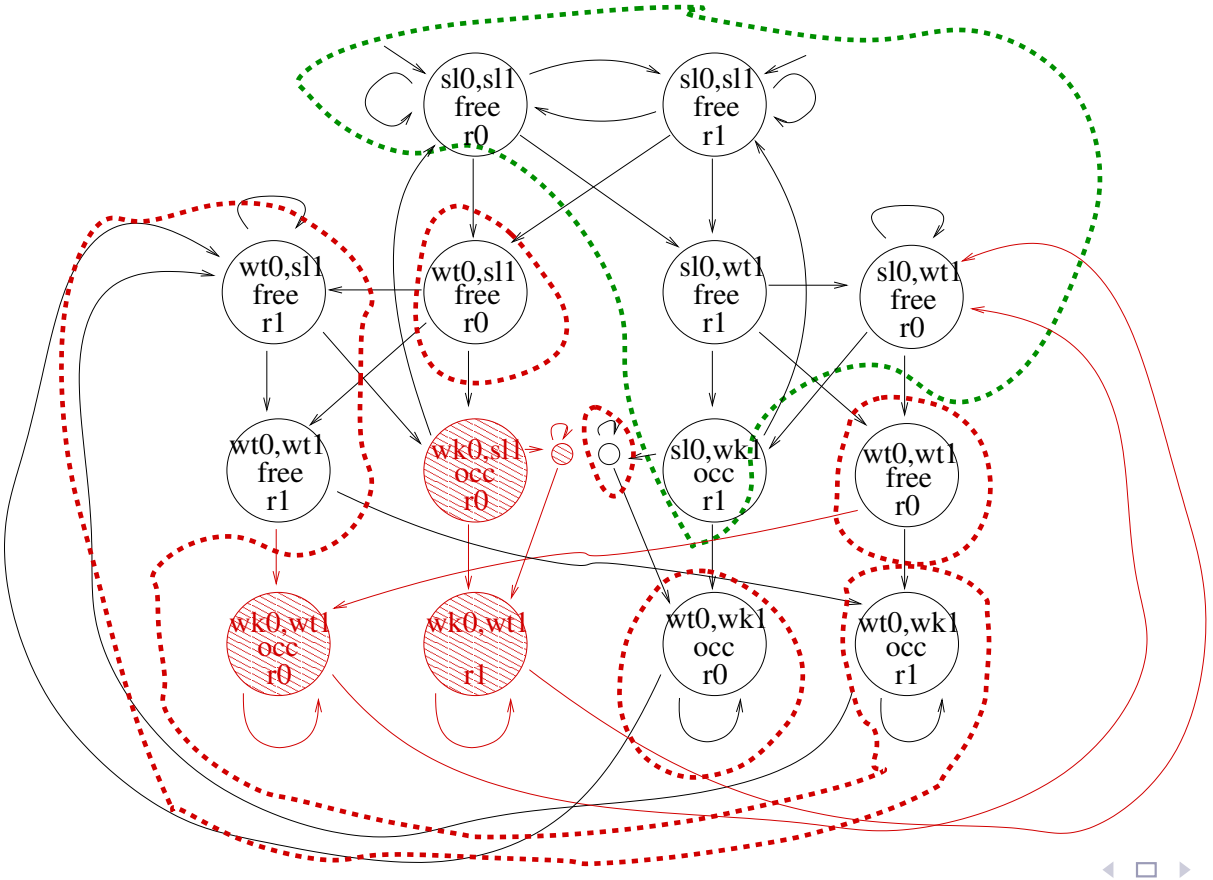
Beispiel: Semaphor



- Wir erlauben die zusätzliche Transition `sleep` $\rightarrow$ `sleep`.
- Nur die erreichbaren Zuständen sind dargestellt.
- Zwei Zustände wurden aus Platzgründen verkleinert.

Fairness

Wir betrachten die Fairness-Bedingungen `p0.running` und `p1.running`, also r0 und r1.
Wir suchen die Zustände, in denen AF(wk0), also ¬EG(¬wk0) gilt.
Dazu suchen wir faire SCC im auf ¬wk0 eingeschränkten Graphen.



Die SCCs sind rot gestrichelt, die einzige faire SCC ist grün gestrichelt.

Wir schließen: wt0⇒¬EG(¬wk0) gilt in allen erreichbaren Zuständen.

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 Sender möchte eine Nachricht mit Sicherheit übermitteln.
- Er sendet dazu die Nachricht ggf. mehrmals.
- Die Rückrichtung des Kanals kann für Empfangsbestätigungen verwendet werden, welche allerdings auch unlesbar werden können.

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.
- ...

Animation einer etwas allgemeineren Version (Nachrichten werden in beide Richtungen verschickt): `http://www4.cs.uni-dortmund.de/RVS/MA/hk/OrdnerVertAlgo/AltBit.html`

Implementierung in SMV

Synchrones System: alle Komponenten führen ihre Schritte gleichzeitig aus.

```
MODULE main
 VAR
   sen : sender();
   rec : receiver();
   out_c : two_bit_channel(sen.msg,sen.bit,rec.in0,rec.in1);
   ret_c : one_bit_channel(rec.out,sen.in0);
 SPEC AG(AF sen.state=sent)
 SPEC AG(sen.state=sent -> AX(sen.msg=c_0 ->
          A[rec.result=c_X U rec.result=c_0]))
 SPEC AG(sen.state=sent -> AX(sen.msg=c_1 ->
          A[rec.result=c_X U rec.result=c_1]))
```

Implementierung der fehlenden Module während der Vorlesung.

Symbolisches Model-Checking

- Der Labelling-Algorithmus manipuliert Zustände explizit.
- Alternativ können wir Zustandsmengen als Boole'sche Funktionen, also z.B. durch BDDs repräsentieren. Das nennt man symbolische Repräsentation.
- Model-Checking mit symbolischen Repräsentationen von Zustandsmengen heißt also symbolisches Model-Checking.
- Man geht davon aus, dass, ähnlich wie in SMV der gesamte Zustandsraum durch die Belegungen Boole'scher Variablen gegeben ist. (Nicht-Boole'sche Variablen kann man sich (unär oder binär) durch Boole'sche Variablen codiert vorstellen.)
- Durch Induktion über den Formelaufbau bestimmt man dann für jede Formel ϕ ein BDD $\mathcal{B}(\phi)$, welches die Menge der Zustände beschreibt, für die ϕ gilt.

Übersetzung von CTL-Formeln in BDDs: Variablen und Boole'sche Op.

Das BDD $\mathcal{B}\phi$ versteht sich bezüglich der Boole'schen Variablen, die den Zustandsraum beschreiben.

- Ist p eine solche Variable, so ist $\mathcal{B}(p) = p$.
- Boole'sche Operationen werden durch die analogen Operationen auf BDDs behandelt, also z.B. $\mathcal{B}(\phi \wedge \psi) = \mathcal{B}(\phi) \wedge \mathcal{B}(\psi)$.
- Für den Operator EX benötigen wir ein BDD $\mathit{next}(s, s')$ mit gestrichenen und ungestrichenen Variablen, welches die erlaubten Zustandsübergänge beschreibt. Vgl. Kapitel 1, Folie 75.

Übersetzung von CTL-Formeln in BDDs: (EXϕ)

- $\mathcal{B}(\mathrm{EX}(\phi)) = \mathit{exists'}(\mathit{next} \wedge \mathit{subst}^{-1}(\mathcal{B}(\phi)))$, wobei *exists'* die existentielle Quantifizierung der *gestrichenen* Variablen und subst^{-1} die Ersetzung der ungestrichenen durch die gestrichenen Variablen bezeichnet. Also intuitiv:

$$\mathcal{B}(\mathrm{EX}(\phi))(s) = \exists s'.\mathit{next}(s, s') \wedge \mathcal{B}(\phi)(s')$$

- Wie üblich: $\mathcal{B}(\mathrm{AX}(\phi)) = \neg\mathcal{B}(\mathrm{EX}(\neg\phi))$.

Übersetzung von CTL-Formeln in BDDs: (E[ϕUψ])

Für durch BDD B gegebene Zustandmenge sei EX(B) das BDD (gemäß letzter Folie), welches der Zustandsmenge $\{s \mid \exists s'.s \rightarrow s' \ \ \& \ \ B(s)\}$ entspricht.

- Setze $B_0 = \mathcal{B}(\psi)$ und falls B_n schon definiert ist, setze

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

- Es gilt: $B_n \Rightarrow B_{n+1}$ für alle n.
- Sei also n_0 der kleinste Wert, sodass $B_{n_0+1} = B_{n_0}$. Es gilt: $\mathcal{B}(\mathrm{E}[\phi \mathrm{U} \psi]) = B_{n_0}$.
- Zum Beweis zeigt man durch Induktion, dass B_n gerade all die Zustände s erfasst, für die ein Pfad $s = s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_k$ mit $k \leq n$ existiert mit $s_k \models \psi$ und $s_i \models \phi$ für alle $i < k$. Die Behauptung, dass $B_{n_0} = \mathcal{B}(\mathrm{E}[\phi \mathrm{U} \psi])$ ergibt sich dann aus der Tatsache, dass nur endlich viele Zustände vorhanden sind.

Übersetzung von CTL-Formeln in BDDs: (EG(ϕ))

- Setze $B_0 = \mathcal{B}(\phi)$ und falls B_n schon definiert ist, setze

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

- Es gilt: $B_{n+1} \Rightarrow B_n$ für alle n.
- Sei also n_0 der kleinste Wert, sodass $B_{n_0+1} = B_n$. Es gilt: $\mathcal{B}(\mathsf{EG}) = B_{n_0}$.
- Zum Beweis zeigt man durch Induktion, dass B_n gerade all die Zustände s erfasst, für die ein Pfad $s = s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_k$ mit $k \geq n$ existiert mit $s_i \models \phi$ für alle $i < k$. Die Behauptung, dass $B_{n_0} = \mathcal{B}(\mathsf{EG}(\phi))$ ergibt sich dann aus der Tatsache, dass nur endlich viele Zustände vorhanden sind.

Übersetzung von CTL-Formeln in BDDs: Fairness

Es seien zwei Fairness-Bedingungen f_1 und f_2 gegeben. Um $\mathcal{B}(\mathsf{EG}(\phi))$ unter diesen Fairness-Bedingungen zu definieren, geht man wie folgt vor:

- Setze $B_0 = \mathcal{B}(\phi)$ und falls B_n schon definiert ist, setze

$$B_{n+1} = \mathcal{B}(\mathsf{E}[\phi \;\; \mathsf{U} \;\; \phi \wedge f_1 \wedge \mathsf{E}[\phi \;\; \mathsf{U} \;\; \phi \wedge f_2 \wedge \mathsf{EX}(B_n)]])$$

- Es gilt: $B_{n+1} \Rightarrow B_n$ für alle n.
- Sei also n_0 der kleinste Wert, sodass $B_{n_0+1} = B_n$. Es gilt: $\mathcal{B}(\mathsf{EG}(\phi)) = B_{n_0}$ (unter den Fairness-Annahmen f_1, f_2).
- Zum Beweis zeigt man durch Induktion, dass B_n gerade all die Zustände s erfasst, für die ein Pfad $s = s_0 \to s_1 \to \cdots \to s_k$ mit $k \geq n$ existiert mit $s_i \models \phi$ für alle $i \leq k$ und so, dass auf dem Pfad mindestens n-Mal erst f_1 und anschließend f_2 erfüllt sind.

Die Behauptung, dass $B_{n_0} = \mathcal{B}(\mathsf{EG}(\phi))$ ergibt sich dann aus der Tatsache, dass nur endlich viele Zustände vorhanden sind.

Bounded Model-Checking

- Man kann die Iterationen, die in der Definition der BDDs für E[−U−] und EG eingestzt wurden auch willkürlich bei einem bestimmten n_0, z.B. $n_0 = 19$ abbrechen.
- Dadurch erfasst man zwar nicht die wirkliche CTL-Semantik, aber, bei "vernünftiger" Wahl der Schranke n_0 doch eine gute Näherung.
- Mithilfe eines SAT-Solvers kann dann die Gültigkeit einer solchen Formel überprüft werden. (Gemeint ist die Formel, die dem BDD entspricht.)
- Wie schon die Beispiele aus Kapitel 1 gezeigt haben, geht das i.a. effizienter als mit BDDs.
- Man spricht dann von *Bounded Model Checking*.
- Wir haben im Kapitel 1 den Spezialfall einer Formel $\neg\mathrm{EF}(\phi)$ mit $\phi =$ undesired betrachtet.

Die Temporallogik LTL

Neben CTL gibt es eine Reihe anderer Temporallogiken: CTL*, LTL, PDL, μ-Kalkül, ... Wir betrachten hier nur kurz LTL (linear temporal logic), die populärste und älteste von allen.

- LTL-Formeln definieren Eigenschaften unendlicher Pfade.
- Per Definition "erfüllt" eine Interpretation (i.S.v. CTL) eine LTL-Formel wenn die Formel *für alle* von Startzuständen ausgehenden, unendlichen Pfade gilt.
- Grammatik der LTL-Formeln:

$$\phi ::= p \mid \top \mid \neg\phi \mid \phi \wedge \psi \mid \mathsf{X}\phi \mid \mathsf{F}\phi \mid \mathsf{G}\phi \mid \phi \mathsf{U} \psi$$

- Intuitive Bedeutung: Ein Pfad $s_0 \to s_1 \to \ldots$ erfüllt $p \wedge \mathsf{XXX}(q \mathsf{U} r)$, wenn $s_0 \models p$ und ein Zeitpunkt $i \geq 3$ existiert, sodass $s_i \models r$ und für alle $3 \leq j < i$ gilt $s_j \models q$.

Semantik von LTL

- Die Bedeutung der LTL-Formeln versteht sich relativ zu einem Pfad π und einem Zeitpunkt i.
- p gilt zu einem Zeitpunkt i, wenn der i-te Zustand von π in p ist.
- Zu einem Zeitpunkt i gilt $\mathsf{X}\phi$, wenn ϕ zum Zeitpunkt $i+1$ gilt.
- Zu einem Zeitpunkt i gilt $\phi\mathsf{U}\psi$, wenn es einen Zeitpunkt $j \geq i$ gibt, zu dem ψ gilt und zu allen Zeitpunkten k mit $i \leq k < j$ die Formel ϕ gilt.
- $\mathsf{F}\phi \iff \top\mathsf{U}\phi$ und $\mathsf{G}\phi \implies \neg\mathsf{F}(\neg\phi)$

Bedeutung von LTL

- In SMV können auch LTL Spezifikationen mit `LTLSPEC` eingebracht werden. Solche verstehen sich wie gesagt für alle Pfade von Startzuständen aus.
- In LTL können Fairness-Aussagen direkt formuliert werden, z.B.:

$$(\mathsf{GF}\texttt{p0.running} \wedge \mathsf{GF}\texttt{p1.running}) \Rightarrow \mathsf{G}(\texttt{p0.st=wait} \Rightarrow \mathsf{F}(\texttt{p0.st=work}))$$

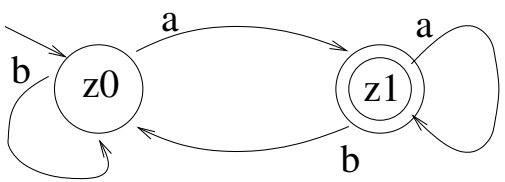
- Model-Checking für LTL ist komplizierter als für CTL und exponentiell in der Formelgröße (aber auch linear in der Zustandszahl).
- LTL ist komfortabler als CTL, aber nicht strikt ausdrucksfähiger: $\mathsf{AG}(\mathsf{EF}p)$ lässt sich nicht in LTL ausdrücken.

Definition

Ein Büchi Automat ist durch dieselben Daten wie ein nichtdet. endl. Automat gegeben.

Büchi-Automat

Ein Büchi-Automat ist ein Quintupel $\mathcal{B} = (\Sigma, Z, I, E, \delta)$, wobei Σ (Alphabet), Z (Zustände) endliche Mengen sind und I (Anfangszustände), E (Endzustände), Teilmengen von Z sind und $\delta \subseteq Z \times \Sigma \times Z$ die Menge der Zustandsübergänge ist.



Hier ist $\Sigma = \{a, b\}, Z = \{z_0, z_1\}, I = \{z_0\}, E = \{z_1\}, \delta = \{(z_0, a, z_1), (z_0, b, z_0), (z_1, a, z_1), (z_1, b, z_0)\}$.
Im Gegensatz zu "normalen" nichtdeterministischen endlichen Automaten verarbeitet ein Büchi Automat *unendliche* Wörter.

Akzeptierte Sprache

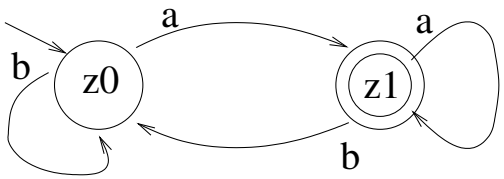
Lauf eines Büchi Automaten

Sei $\mathcal{B} = (\Sigma, Z, E, I, \delta)$ ein Büchi Automat und sei $w = a_0, a_1, a_2, \ldots$ ein unendliches Wort über Σ. Ein Lauf von $\mathcal{B}$ ist eine unendliche Folge von Zuständen $z_0, z_1, z_2, \ldots$, so dass $z_0 \in I$ und $(z_i, a_i, z_{i+1}) \in \delta$ für $i \geq 0$.
Der Lauf ist *akzeptierend*, wenn $z_i \in E$ für unendlich viele i ("immer wieder").

Erkannte Sprache

Die von einem Büchi Automaten $\mathcal{B}$ erkannte Sprache $L(\mathcal{B})$ besteht aus allen unendlichen Wörtern, zu denen ein akzeptierender Lauf existiert.

Beispiel



Ein akzeptierender Lauf auf $w = aabaabaabaabaab\ldots$:

$$z_0 \xrightarrow{a} (z_1) \xrightarrow{a} (z_1) \xrightarrow{b} z_0 \xrightarrow{a} (z_1) \xrightarrow{a} (z_1) \xrightarrow{b} z_0 \longrightarrow \ldots$$

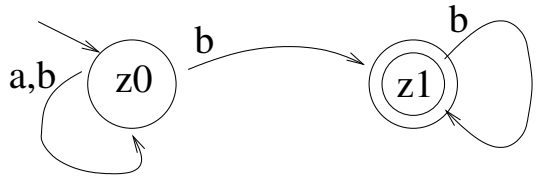
Ein nicht akzeptierender Lauf auf $w = aabbbbbbbb\ldots$:

$$z_0 \xrightarrow{a} (z_1) \xrightarrow{a} (z_1) \xrightarrow{b} z_0 \xrightarrow{b} z_0 \xrightarrow{b} z_0 \xrightarrow{b} z_0 \longrightarrow \ldots$$

Die erkannte Sprache umfasst alle Wörter mit unendlichen vielen a's. Man schreibt: $L = (b^*a)^\omega$.

Beispiel

$\mathcal{B} = (\Sigma, Z, E, I, \delta)$ wobei $\Sigma = \{a, b\}, Z = \{z_0, z_1\}, I = \{z_0\}, E = \{z_1\}, \delta = \{(z_0, a, z_0), (z_0, b, z_0), (z_0, b, z_1), ((z_1, b, z_0)\}$.



Ein akzeptierender Lauf auf $w = aababbbbbb\ldots$:

$$z_0 \xrightarrow{a} z_0 \xrightarrow{a} z_0 \xrightarrow{b} z_0 \xrightarrow{a} z_0 \xrightarrow{b} \textcircled{z_1} \xrightarrow{b} \textcircled{z_1} \xrightarrow{b} \textcircled{z_1} \longrightarrow \ldots$$

Ein nicht akzeptierender Lauf auf demselben Wort:

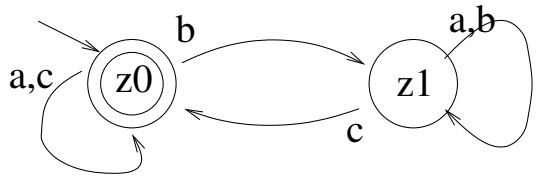
$$z_0 \xrightarrow{a} z_0 \xrightarrow{a} z_0 \xrightarrow{b} z_0 \xrightarrow{a} z_0 \xrightarrow{b} z_0 \xrightarrow{b} z_0 \xrightarrow{b} z_0 \xrightarrow{b} z_0 \xrightarrow{b} \ldots$$

Ein nicht akzeptierender Lauf auf $w = abababababab\ldots$

$$z_0 \xrightarrow{a} z_0 \xrightarrow{b} z_0 \xrightarrow{a} z_0 \xrightarrow{b} z_0 \xrightarrow{a} z_0 \xrightarrow{b} z_0 \xrightarrow{a} z_0 \xrightarrow{b} \ldots$$

Die erkannte Sprache umfasst alle Wörter mit nur endlich vielen a's. Man schreibt: $L = (a + b)^* b^\omega$.

Beispiel



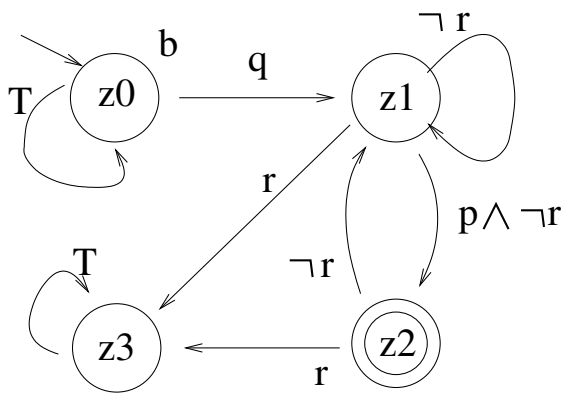
- Hier umfasst die erkannte Sprache alle Wörter, bei denen auf jedes b ("Tür auf") irgendwann ein c ("Tür zu") folgt.
- Man schreibt:
  $L = (a+c)^\omega + (a+b+c)^* c (a+c)^\omega + (a+b+c)^* (b(a+b+c)^* c)^\omega$.
- So ein regulärer Ausdruck mit "hoch omega" (wird hier nicht formal definiert) heißt *omega regulärer Ausdruck*.
- Man kann diese Sprache auch durch die LTL-Formel $\mathsf{G}(b \Rightarrow \mathsf{F}c)$ beschreiben.
- Die Sprache "nur endlich viele a's" entspricht dann $\mathsf{FG}b$. Die Sprache "unendlich viele a's" entspricht dann $\mathsf{GF}a$.

LTL und Automaten

Satz (ohne Beweis)

Zu jeder LTL-Formel ϕ kann ein Büchi Automat angegeben werden, dessen Alphabet die Belegungen der Boole'schen Variablen sind und der gerade die Pfade (aufgefasst als unendliche Wörter) erkennt, die die LTL-Formel erfüllen.

Beispiel: $\phi = \neg(\mathsf{GF}p \Rightarrow \mathsf{G}(q \Rightarrow \mathsf{F}r))$.



Eine mit einer Boole'schen Formel beschriftete Kante steht für alle Belegungen, die die Formel erfüllen.

Model-Checking mit Büchi Automaten

Gegeben sei eine Interpretation $\mathcal{I}$, also ein Transitionssystem $(S, \rightarrow)$ und für jeden Zustand $s \in S$ die Menge $\mathcal{I}(s)$ der Boole'schen Variablen, die dort gelten. Außerdem eine Menge $S_I \subseteq S$ von Anfangszuständen.

Gegeben sei weiter ein Büchi Automat $\mathcal{B} = (\Sigma, Z, I, E, \delta)$ dessen Alphabet die Belegungen der Boole'schen Variablen sind.

Man konstruiert ein Produkt-Transitionssystem $\mathcal{I} \times \mathcal{B}$, dessen Zustände Paare (s, z) mit $s \in S$ und $z \in Z$ sind. Eine Transition $(s, z) \rightarrow (s', z')$ gibt es, wenn $s \rightarrow s'$ und $(z, \mathcal{I}(s), z') \in \delta$. (Wir fassen $\mathcal{I}(s)$ als Belegung auf.)

Model-Checking mit Büchi Automaten, Forts.

Satz

Die folgenden drei Aussagen sind äquivalent:

- Im Transitionssystem $(S, \to)$ gibt es einen von S_I ausgehenden Pfad, dessen Beschriftungen ein von $\mathcal{B}$ akzeptiertes Wort ergeben,
- Im Produkttransitionssystem $\mathcal{I} \times \mathcal{B}$ gibt es einen von $S_I \times I$ ausgehenden unendlichen Pfad, der immer wieder Zustände aus $S \times E$, also von der Form (s, z) mit $z \in E$ besucht.
- Im Produkttransitionssystem $\mathcal{I} \times \mathcal{B}$ gibt es ein von $S_I \times I$ ausgehendes "Lasso", also einen endlichen Pfad

$$(s_0, z_0) \to (s_1, z_1) \to \ldots$$
$$\ldots \to (s_n, z_n) \to (s_{n+1}, z_{n+1}) \to \ldots \to (s_{n+m}, z_{n+m})$$

wobei $s_{n+m} = s_n$ und $z_{n+m} = z_n$ und $z_n \in E$.

Model-Checking mit Büchi-Automaten, Schluss

Um festzustellen, ob eine LTL-Formel ϕ in einem Transitionssystem gilt, kann man einen Büchi Automaten für $\neg\phi$ konstruieren und im entsprechenden Produkt-Transitionssystem nach einem "Lasso" suchen.

Man muss dafür das Produkt-Transitionssystem nicht vollständig im Speicher halten, sondern kann es während der Suche nach Bedarf aufbauen. ("on the fly" Model Checking).

Gerne wird der Büchi-Automat direkt, also ohne Umweg über LTL, angegeben. Dieser ist dann so zu wählen, dass er die "verbotenen" Pfade charakterisiert. Die operationelle Sichtweise der Büchi-Automaten ist manchmal attraktiver als die deklarative Natur der temporallogischen Formeln.

Der Model Checker SPIN

Der Model-Checker SPIN arbeitet nach diesem Muster. Er hat ein Front-End, welches aus einer Modellbeschreibung und einer Spezifikation einen Graphen in Form eines abstrakten Datentyps konstruiert:

- Ein abstrakter Typ `node` von Knoten, dessen Elemente in 64bit abgespeichert werden können;
- Eine Liste von Startknoten;
- Eine Funktion `next: node->node list`, die zu einem Knoten seine unmittelbaren Nachfolger liefert;
- Eine Funktion `final : node -> bool`;

so, dass die Spezifikation erfüllt ist, wenn es kein Lasso von Knoten gibt, in dessen Schleife die Bedingung `final` mindestens einmal wahr ist.

Das Back-End sucht nach solch einem Lasso mit geschickter Kombination von Tiefen- und Breitensuche. Man kann auch eigene Back-Ends einpflegen.

Ausblick auf andere Anwendungen der Büchi Automaten

- Entscheidbarkeit der monadischen Logik zweiter Stufe (Satz von Büchi).
  - Man konstruiert einen Büchi Automaten, der Modelle einer gegebenen Formel akzeptiert. Kann der Automat überhaupt ein Wort akzeptieren, so hat die Formel ein Modell (ist erfüllbar).
  - Hierzu ist es notwendig, Büchi Automaten zu komplementieren. Nicht so einfach, da i.a. nichtdeterministisch und keine Determinisierung möglich ist.
- Terminationsanalyse (feststellen, ob ein Programm terminiert)
  - Man lässt einen Büchi Automaten auf allen Ausführungspfaden eines funktionalen Programms mitlaufen, welcher prüft, ob mindestens eine Variable immer wieder dekrementiert wird.
  - Nichtdeterminismus erlaubt es, diese Variable zu “erraten”.
  - Akzeptiert der Büchi Automat alle möglichen Ausführungspfade (ist also der komplementierte Automat leer), so terminiert das gegebene Programm.
  - Natürlich funktioniert das nur für bestimmte Programme (Halteproblem!), ist aber trotzdem nützlich.

Zusammenfassung Kapitel 2

- 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 und Bounded Modelchecking für CTL
- Die Temporallogik LTL
- Büchi Automaten