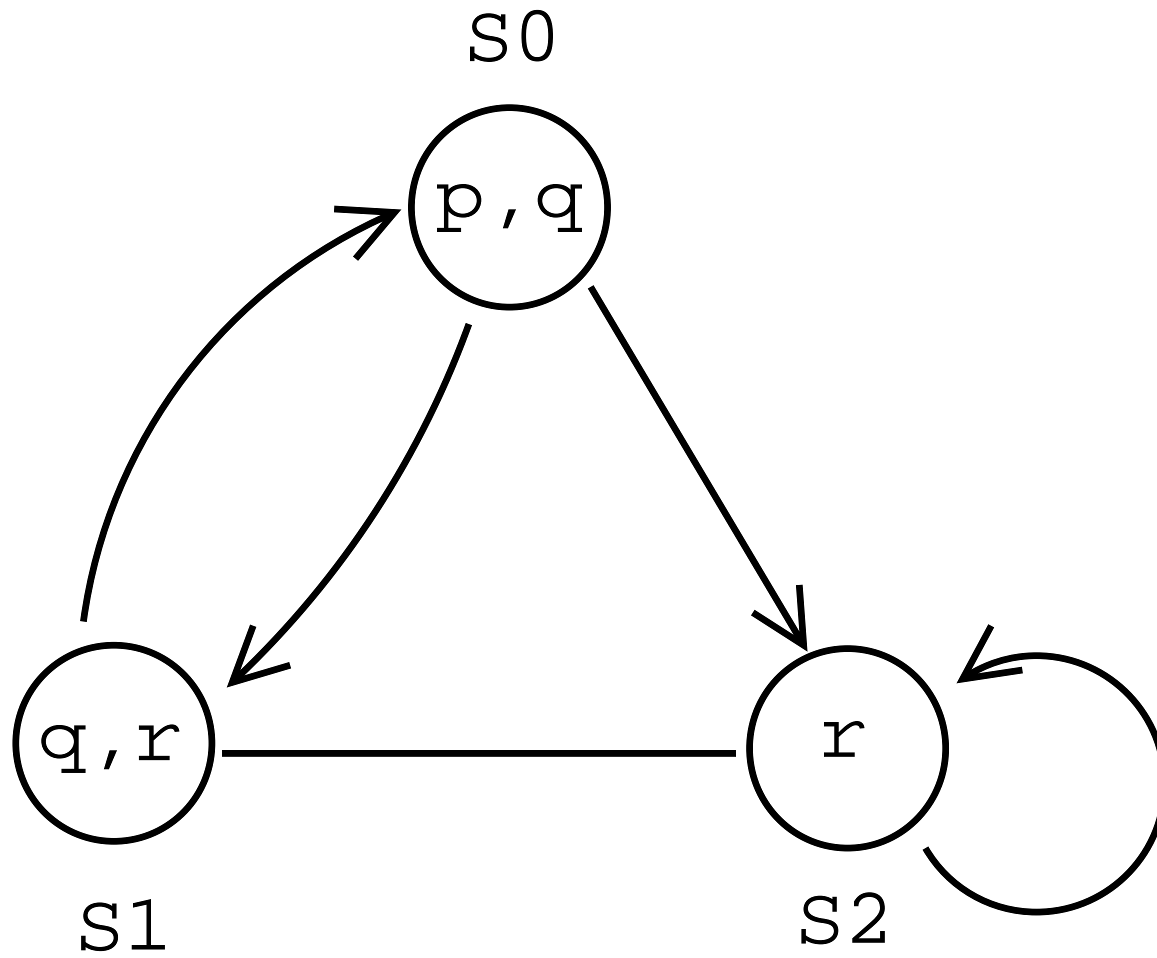
Beispiel



SMV - Überblick

Ziel: Überprüfung der Erfülltheit von CTL-Formeln in dem Anfangszustand s_0 eines Modells: $(M, s_0 \models \phi)$.

- Eingabesprache zur Darstellung von $M = (S, \rightarrow, L)$.
  - Zustandsraum, Zustände.
  - Übergangsrelation.
  - Markierung der Zustände.
  - Auszeichnung eines Initialzustandes s_0
- Eingabesprache zur Darstellung von Anforderungen in CTL.
- Prüfmechanismus.
  - Beweisverpflichtung.
  - Optimierungen "von Hand".
  - Analyse fehlgeschlagener Beweise.

Zustandsraum und Markierungen in SMV

Der Zustandsraum ist das Kreuzprodukt der Wertebereiche sämtlicher Variablen.

- Variablen
  - Zugriff auf Zustandskomponenten (Projektionen).
  - Endliche Wertebereiche.
    - `boolean` mit Werten 0 (false), 1 (true).
    - Aufzählugstypen (z.B. {`gruen,gelb,rot`}).
    - Endliche Bereiche ganzer Zahlen (z.B. `2..5`).
  - Beispiel: `VAR signal:` {`gruen,gelb,rot`}`;`
  - Beispiel: `VAR bereit: boolean;`
  - Jede Variable kann mit boole'schen Variablen kodiert werden, z.B. `signal` durch ($\texttt{signal}_1$,$\texttt{signal}_2$) wobei
    - `signal=gruen` := $\texttt{signal}_1$=1
    - `signal=gelb` := $\texttt{signal}_1$=0 und $\texttt{signal}_2$=1
    - `signal=rot` := $\texttt{signal}_1$=0 und $\texttt{signal}_2$=0

    Anm.: In der Semantik von SMV ist i.W. dieses Verfahren zur Repräsentation von nicht-boole'schen Variablen angegeben.

- Definitionen.
  - Vergleichbar mit Makros: reiner Textersatz.
  - Bsp: `DEFINE unsicher := signal=gruen | signal=gelb;`

Markierungen entsprechen den (boole'schen) Variablen:

`bereit`$\in L(s)$ genau dann wenn in s `bereit`$=1$

Transitionssystem in SMV - ASSIGN

Beschreibung von Anfangszustand und Übergangsrelation im Stil einer Programmiersprache.

Beispiel:

```
ASSIGN
  init(signal) := rot;
  next(signal) :=
    case
      signal=rot   : gruen;
      signal=gruen : { gruen, gelb };
      1: rot;
    esac;
```

Anfangszustände sind die Zustände, in denen gilt: `signal=rot`, bzw.

- $\texttt{signal}_1 = 0$
- $\texttt{signal}_2 = 0$

Die Transitionsrelation ist die Menge $\{(s_1, s_2)\}$ mit

- Wenn `signal=rot` in s_1 dann `signal=gruen` in s_2,
- Wenn `signal=gruen` und nicht `signal=rot` in s_1 dann `signal=gruen` oder `signal=gelb` in s_2,
- Wenn nicht `signal=gruen` und nicht `signal=rot` in s_1 dann `signal=rot` in s_2.

Transitionssystem in SMV - ASSIGN (2)

Zu beachten:

- Nichtdeterminismus durch Angabe einer Menge von Werten. Alternative zu obigem case-Konstrukt:

```
case
  ...
  signal=gruen: signal union gelb;
  ...
esac;
```

- Vollständige Fallunterscheidung zwingend erforderlich (Totalität der Übergangsrelation).

- Randbedingungen verbieten zirkuläre Abhängigkeiten zwischen den Zuweisungen und garantieren somit die Implementierbarkeit.

Transitionssystem in SMV - INIT und TRANS

Beschreibung von Anfangszustand und Übergangsrelation im Stil einer Logik

Beispiel:

```
INIT
  signal = rot
TRANS
    (signal=rot & next(signal)=gruen)
   |(signal=gruen & next(signal)={gelb,gruen})
   |(signal=gelb & next(signal)=rot)
```

Anmerkung: Diese Übergangsrelation stimmt mit der vorangegangen definierten Übergangsrelation überein, da `signal` gleichzeitig nur eines der Werte `rot, gelb, gruen` annehmen kann.

Transitionssystem in SMV - INIT und TRANS (2)

Alternative:

```
TRANS
    (signal=rot & next(signal)=gruen)
   |(signal=gruen & next(signal)=gelb)
   |(signal=gruen & next(signal)=gruen)
   |(signal=gelb & next(signal)=rot)
```

Zu beachten:

- Partielle Übergangsrelationen möglich.
- Keine Überprüfung von Randbedingungen, die die Implementierbarkeit garantieren.

Anforderungen

Anforderungen werden in SMV direkt als CTL-Formeln ausgedrückt.

Syntax der Formeln ist im Wesentlichen die bereits behandelte Standard-Syntax von CTL. Besonderheiten:

`0,1` — False, True

`!` — Negation

`&` — Konjunktion

`|` — Disjunktion

`->` — Implikation

Neben Literalen können Ausdrücke mit Wahrheitswerten auftreten.

Anforderungen: nach dem Schlüsselwort `SPEC`

Zur Spezifikation von erwünschten Eigenschaften, z.B. Sicherheitseigenschaften eines Systems:

```
SPEC
  AG !(nord_sued_signal=gruen & ost_west_signal=gruen)
```

Sie können auch zur Durchführung von Plausibilitätstests dienen, z.B. zur Überprüfung der Totalität der Übergangsrelation:

```
SPEC
  AG EX 1
```

Prüfung von Anforderungen

Gegeben: Transitionssystem M, Formel ι zur Charakterisierung des Initialzustandes, Formel ϕ zur Spezifikation der auf Erfülltheit zu prüfenden Anforderung.

Gesucht: Gilt $M, s \models \phi$ für alle Initialzustände s (d.h. für alle Zustände s, in denen ι gilt).

Viele Model-Checker (SMV?) prüfen $M, s \models \iota \Rightarrow \phi$ für alle Zustände s (alternative Schreibweise: $M \models \iota \Rightarrow \phi$).

Wenn die geprüfte Anforderung nicht erfüllt ist, wird ein Gegenbeispiel in Form eines Ausführungspfades generiert, von dem die Anforderung verletzt wird.
Diese Gegenbeispiele sind insbesondere dann nützlich, wenn CTL-Formeln mit universellem Pfadquantor (**A**) auf Erfülltheit geprüft werden.

Effizienzsteigerung

Hauptproblem: riesige Zustandsräume.
Ansatzmöglichkeit: Verwendung von Definitionen anstatt Variablendeklarationen sofern möglich: Definitionen vergrössern nicht den Zustandsraum.

SMV bietet zudem verschiedene "Schalter", die geeignet sein können, um die Effizienz der Modell-Prüfung zu steigern.

-f berechnet vorab die erreichbaren Zustände - nützlich insbesondere in Fällen, in denen der Gesamtzustandsraum wesentlich grösser ist, als die tatsächlich erreichbaren Zustände.

-reorder ordnet die Variablen neu an, wenn die Anzahl der erzeugten BDD-Knoten (BDD = Binary Decision Diagram) eine vorgegebene Grenze (-**reordersize**) überschreitet.

- Die Effizienz des Prüfmechanismus hängt von der Grösse der BDD-Repräsentation ab und die hängt wiederum von der Reihenfolge ab, in der die Variablen beim Aufbau der BDD-Struktur betrachtet werden.

- Faustregel für “gute” Variablenordnungen: Variablen, die stark voneinander Abhängen, sollten in der Ordnung nah beieinander stehen.
- Man kann versuchen gute Variablenordnungen für kleine Systeminstanzen zu finden und sie für grosse Systeminstanzen zu nutzen: SMV gibt auf Anweisung die mit -**reorder** erzeugte Variablenordnung aus.

Module in SMV (I)

Module sind “syntaktischer Zucker” zur Spezifikation von Teilsystemen. Ihre Übergangsrelationen können miteinander synchronisiert sein oder sie werden mit einer Interleaving-Semantik verknüpft.

Beispiel - synchronisierte Teilsysteme:

```
MODULE Ampel
  VAR signal: ...

MODULE main
  VAR nord_sued: Ampel;
      ost_west: Ampel;
  ...
```

Das Gesamtsystem ist definiert durch ein als `main` ausgezeichnet Modul. Der Zustandsraum eines Moduls ergibt sich als kartesisches Produkt aus den Zustandsräumen und Wertebereichen der in ihm enthaltenen Module und Variablen.

Module in SMV (II)

Beispiel - asynchrone Kombination von Teilsystemen:

```
MODULE Ampel
  ...

MODULE main
  VAR nord_sued: process Ampel;
      ost_west: process Ampel;
```

SMV bietet darüber hinaus

- Parametrisierte Module
- Anforderungen in Modulen

SMV: ein einfaches Beispiel

```
MODULE main

   VAR
      request : boolean;
      status  : {ready,busy};

   ASSIGN
      init(status) := ready;
      next(status) :=
         case
           request : busy;
           1 : {ready,busy};
         esac;

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

SMV: einfaches Hardware-Beispiel

3-bit-Zähler, zusammengesetzt aus 3 1-bit-Zählern

```
MODULE main

 VAR   bit0 : counter_cell(1);
       bit1 : counter_cell(bit0.carry_out);
       bit2 : counter_cell(bit1.carry_out);

 SPEC  AG AF bit2.carry_out


MODULE counter_cell(carry_in)
 VAR   value : boolean;

 ASSIGN
   init(value) := 0;
   next(value) := value + carry_in mod 2;

 DEFINE
  carry_out := value & carry_in;
```

Beispiel: *Mutual Exclusion*

Mutual Exclusion – “gegenseitiger Ausschluß

Wichtiges Prinzip bei nebenläufigen Prozessen: es soll verhindert werden, daß meherer Prozesse gleichzeitig Zugriff auf eine kritische Ressource haben (z.B. Schreiben einer Datenbank-Tabelle)

Realisierung durch “kritische Regionen” (*critical section*) in den jeweiligen Programmen, und ein Protokoll, unter welchen Bedingungen die kritische Region “betreten” werden darf.

Abläufe von zwei Prozessen verschränkt (*interleaving*): wir nehmen an, daß jeder Prozeß separat und unabhängig, aber nicht gleichzeitig mit einem anderen Prozeß, einen Zustandsübergang ausführt.

Mutual Exclusion: Eigenschaften

Geforderte Eigenschaften:

1. *Sicherheit*: zu jedem Zeitpunkt darf sich nur ein Prozeß in der kritischen Region befinden.

2. *Lebendigkeit*: Wenn ein Prozeß die kritische Region anfordert will, dann erhält er schließlich auch den Zugang.

3. *Kein Blockieren*: Ein Prozeß kann jederzeit die kritische Region anfordern.

4. *Flexibilität*: Prozesse müssen nicht ihre kritische Region in fester Reihenfolge betreten (Ausschluß eines rigiden Plans, wie z.B. Zyklus).

Mutual Exclusion: Modellierung

Jeder Prozeß durchläuft einen Zyklus

$$n \to t \to c \to n \to \ldots,$$

wobei

n – “ nicht kritisch”

t – “ kritische Region angefordert” (*trying*)

c – “ in kritischer Region”

Menge der Zustände für System von 2 Prozessen: Zustände markiert mit Elementen aus einer Teilmenge des kartesischen Produkts $\{n_1, t_1, c_1\} \times \{n_2, t_2, c_2\}$

Initialer Zustand: mit Markierung n_1, n_2

Zustandsübergänge: *einer* der Prozesse macht einen Übergang entsprechend dem angegebenen Zyklus; z.B.

$$(n_1, t_2) \to (n_1, c_2)$$

Mutual Exclusion: formal

Formalisierung der geforderten Eigenschaften:

Sicherheit: $\phi_1 := \textbf{AG}\ \neg(c_1 \wedge c_2)$

Lebendigkeit: $\phi_2 := \textbf{AG}\ (t_1 \Rightarrow \textbf{AF}\ c_1)$

Kein Blockieren: $\phi_3 := \textbf{AG}\ (n_1 \Rightarrow \textbf{EX}\ t_1)$

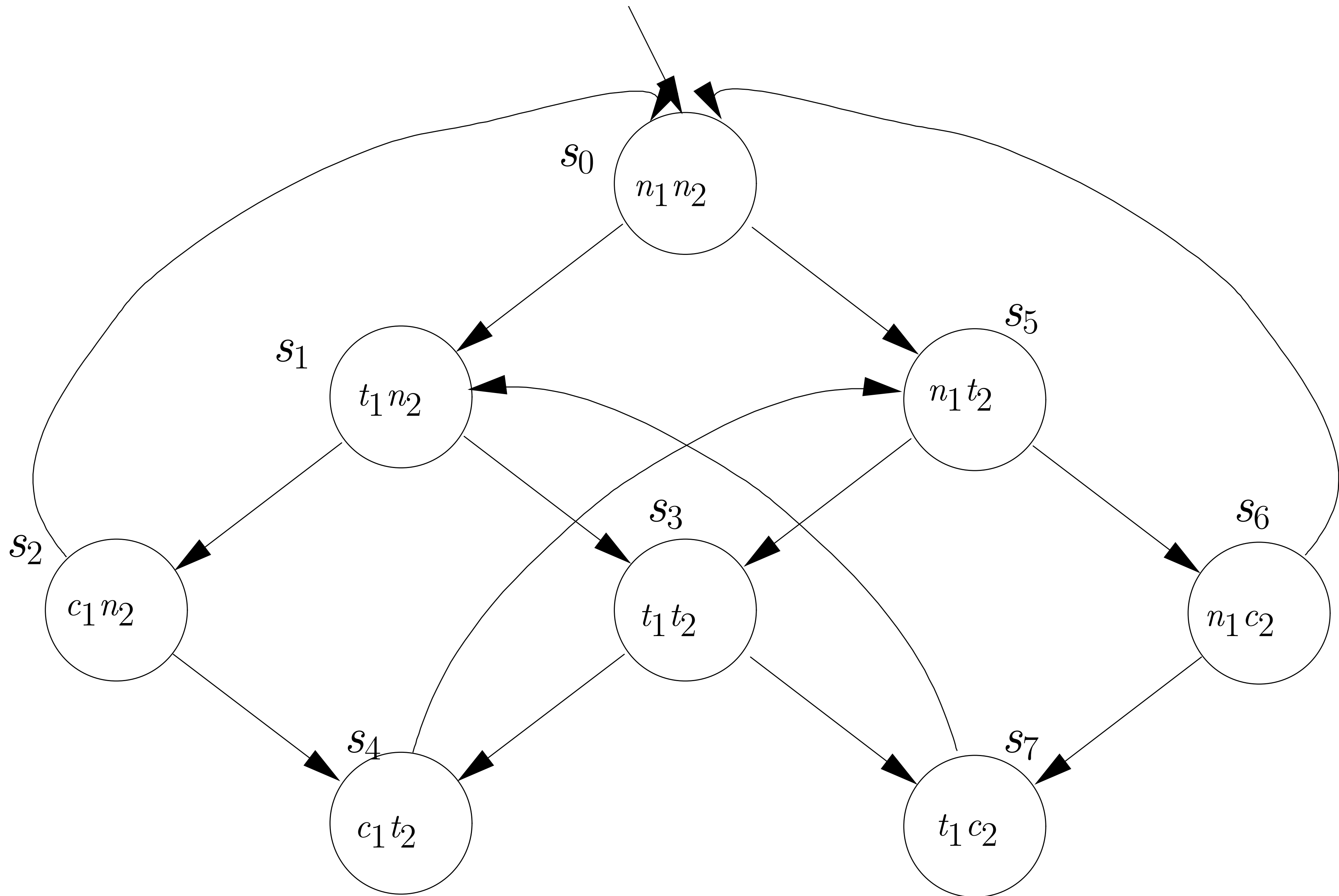
Flexibilität: $\phi_4 := \textbf{EF}\ (c_1 \wedge \textbf{E}\ [c_1\ \textbf{U}\ (\neg c_1 \wedge \textbf{E}\ [\neg c_2\ \textbf{U}\ c_1])])$

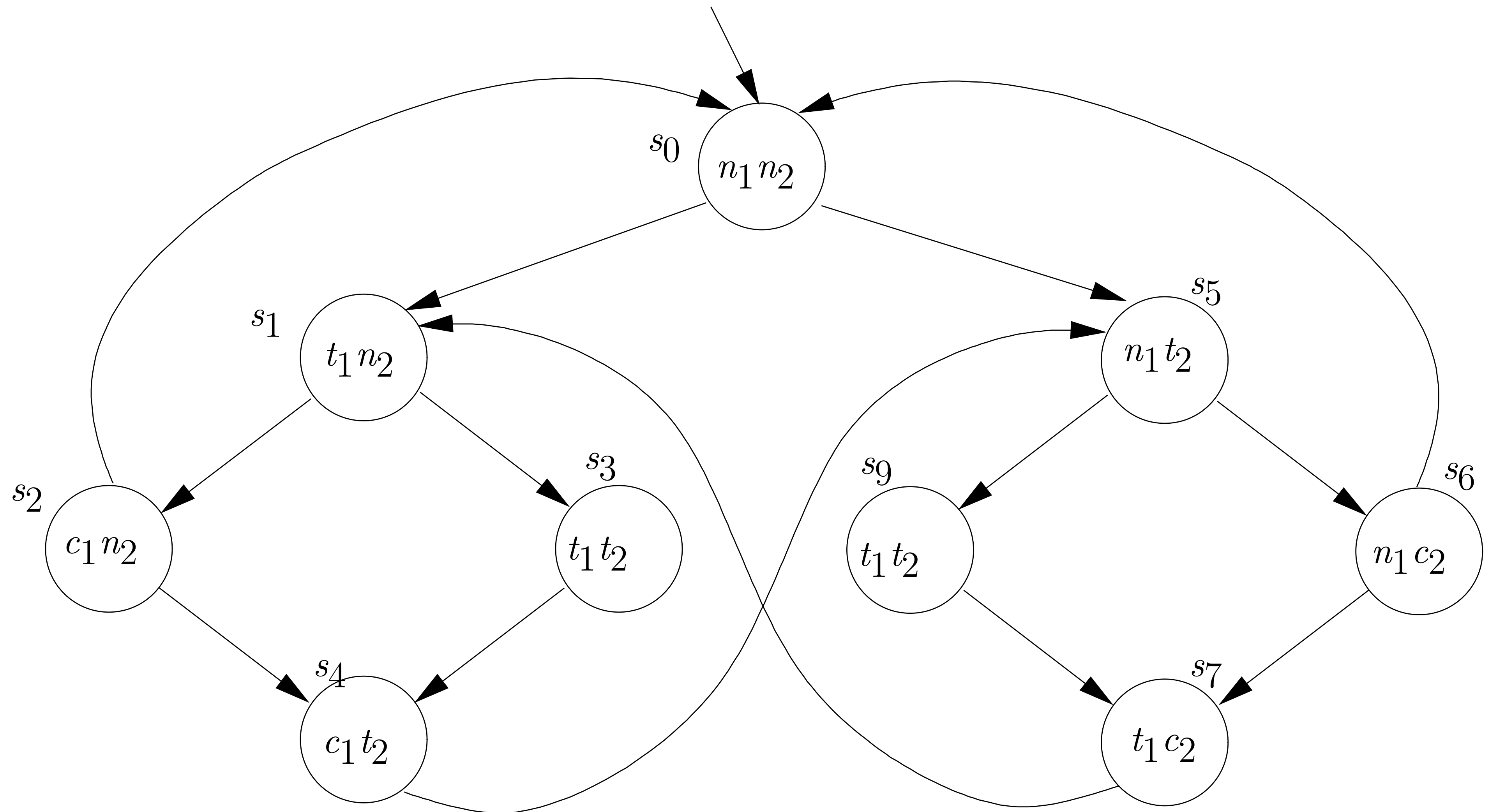
SMV-Code für Mutual Exclusion

```
MODULE main
 VAR
   pr1 : process prc(pr2.st, turn, 0);
   pr2 : process prc(pr1.st, turn, 1);
   turn : boolean;

 ASSIGN
   init(turn) := 0;

--safety
 SPEC AG!((pr1.st = c) & (pr2.st = c))

--liveness
 SPEC AG((pr1.st = t) -> AF (pr1.st = c))
 SPEC AG((pr2.st = t) -> AF (pr2.st = c))

--no strict sequencing
 SPEC EF(pr1.st = c & E[pr1.st = c U (!pr1.st = c &
                 E[! pr2.st = c U pr1.st = c ])])
```

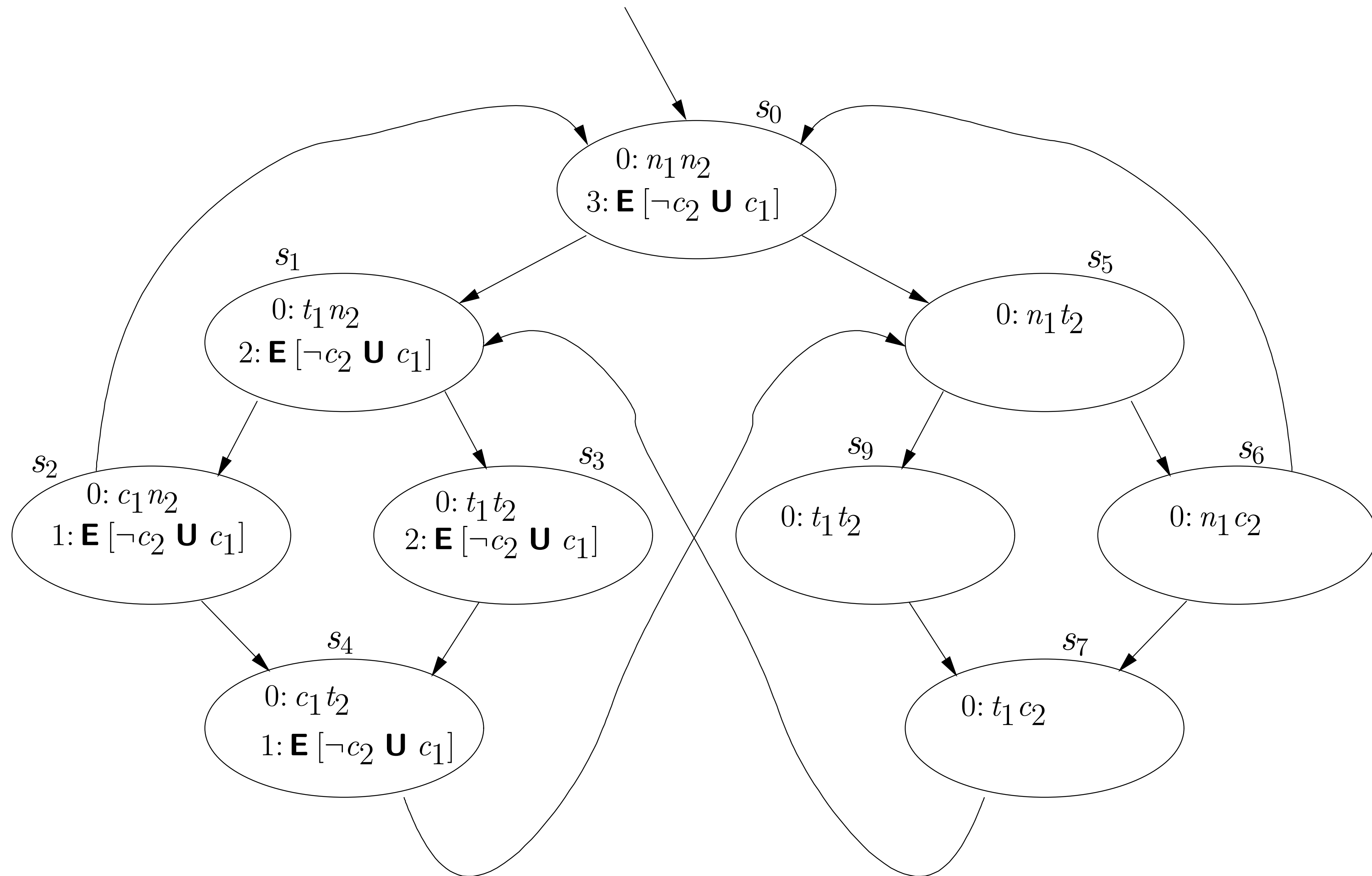
SMV-Code für Mutual Exclusion (2)

```
MODULE prc(other-st, turn, myturn)
 VAR
   st : {n, t, c};

 ASSIGN
   init(st) := n;
   next(st) :=
     case  (st = n)                    :{t, n);
           (st = t) & (other-st = n) : c;
           (st = t) & (other-st = t) & (turn = myturn) : c;
           (st = c)                    : {c, n};
           1                           : st;
     esac;
   next(turn) :=
     case   turn = myturn & st = c  : !turn;
            1                       : turn;
     esac;

FAIRNESS running      FAIRNESS !(st = c)
```


s_0
$n_1 n_2$
s_1
$t_1 n_2$
s_5
$n_1 t_2$
s_2
$c_1 n_2$
s_3
$t_1 t_2$
s_6
$n_1 c_2$
s_4
$c_1 t_2$
s_7
$t_1 c_2$


s_0
$n_1 n_2$
s_1
$t_1 n_2$
s_5
$n_1 t_2$
s_2
$c_1 n_2$
s_3
$t_1 t_2$
s_9
$t_1 t_2$
s_6
$n_1 c_2$
s_4
$c_1 t_2$
s_7
$t_1 c_2$

Markierung der Zustände, die die Formel **E** $[\neg c_2$ **U** $c_1])$ erfüllen.



SMV-Beispiel: *Alternating Bit Protocol*

Protokoll für die Übertragung von Nachrichten von einem Sender zu einem Empfänger, die über einen Nachrichten-Kanal und einen Bestätigungskanal verbunden sind.

- 4 Komponenten: Sender, Empfänger, Nachrichten-Kanal und Bestätigungskanal (*acknowledgement channel*)
- Kanäle sind unzuverlässig, d.h. können Nachrichten verlieren oder verdoppeln.
- Sender sendet Nachricht zusammen mit Kontroll-Bit.
- Empfänger sendet bei Empfang einer Nachricht das Kontroll-Bit als Bestätigung, falls es dem erwarteten Bit entspricht, andernfalls das alte Bestätigungsbits ("*ack bit*").
- Sender wechselt Kontrollbit für nächste Nachricht, falls er das richtige Bit als Bestätigung erhalten hat;
- andernfalls wird alte Nachricht (mit altem Kontrollbit) solange wiederholt, bis er das richtige Bestätigungsbits empfangen hat.

SMV-Beispiel: *Alternating Bit Protocol* (2)

- Durch alternierendes Kontrollbit können Sender und Empfänger Verlust oder Verdopplung einer Nachricht entdecken.

Bermerkung: Um Korruption von Nachrichten zu entdecken, müssen andere Methoden angewendet werden (vgl. Paritätsbit).

Alternating Bit Protocol: **SMV-Code**

```
MODULE sender(ack)
VAR
    st       : {sending, sent};
    message1 : boolean;
    message2 : boolean;
ASSIGN
  init(st) := sending;
  next(st) := case  ack = message2 & !(st = sent) : sent;
                    1                             : sending;
              esac;
  next(message1) :=  case  st = sent : {0, 1};
                           1         : message1;
                     esac;
  next(message2) :=  case  st = sent : !message2;
                           1         : message2;
                     esac;
  FAIRNESS running
  SPEC AG AF st = sent
```

Alternating Bit Protocol: SMV-Code (2)

```
MODULE receiver(message1, message2)
VAR
    st       : {receiving, received};
    ack      : boolean;
    expected : boolean;
ASSIGN
  init(st) := receiving;
  next(st) := case message2 = expected & !(st = received): received;
                   1                                      : receiving;
              esac;
  next(ack):= case st = received : message2;
                   1             : ack;
              esac;
  next(expected) :=  case  st = received : !expected;
                           1             : expected;
                     esac;
  FAIRNESS running
  SPEC AG AF st = received
```

Alternating Bit Protocol: SMV-Code (3)

```
MODULE one-bit-chan(input)
  VAR
    output : boolean;
  ASSIGN
    next(output) := {input, output};
  FAIRNESS running
  FAIRNESS (input = 0 -> AF output = 0)
      & (input = 1 -> AF output = 1)
```

Alternating Bit Protocol: **SMV-Code (4)**

```
MODULE two-bit-chan(input1, input2)
  VAR
    output1 : boolean;
    output2 : boolean;
  ASSIGN
    next(output2) := {input2, output2};
    next(output1) :=
      case
        input2 = next(output2) : input1;
        1                      : {input1, output1};
      esac;
  FAIRNESS running
  FAIRNESS (input1 = 0 -> AF output1 = 0)
      & (input1 = 1 -> AF output1 = 1)
      & (input2 = 0 -> AF output2 = 0)
      & (input2 = 1 -> AF output2 = 1)
```

Alternating Bit Protocol: **SMV-Code (5)**

```
MODULE main
VAR
   S : process sender(ack_chan.output);
   R : process receiver(msg_chan.output1, msg_chan.output2);
   msg_chan : process two-bit-chan(S.message1, S.message2);
   ack_chan : process one-bit-chan(R.ack);

ASSIGN
  init(S.message2)         := 0;
  init(R.expected)         := 0;
  init(R.ack)              := 1;
  init(msg_chan.output2)   := 1;
  init(ack_chan.output)    := 1;

SPEC AG( S.st = sent & S.message1 = 1  -> msg_chan.output1 = 1 )
```

Fairness

Bedeutung von Fairness für ABP: Nachrichtenkanal darf nicht unendlich viele Nachrichten verlieren – wenn eine Nachricht genügend oft gesendet wird, dann kommt sie schließlich auch an.

Die Forderung nach Fairness eines Systems (bzw. eines Systemteils) schränkt die Menge der zulässigen Ausführungen ein
$\leadsto$ Verfeinerung der Spezifikation

Ausgedrückt als eine Fairness-Bedingung:

`FAIRNESS` ϕ

soll bedeuten, daß ϕ entlang eines jeden Ausführungspfads unendlich oft erfüllt ist

In SMV spezielle Form der Bedingung:

```
FAIRNESS running
```

soll Einschränkung auf solche Pfade bedeuten, entlang denen der Modul, in dem die Bedingung auftritt, unendlich oft zur Ausführung kommt.

Fairness (2)

Nur solche Pfade sind zugelassen, die die Bedingun(en) erfüllen – die "Pfad-Quantoren" **A** und **E** können aufgefaßt werden als auf solche Pfade beschränkt.

Modellprüfung für Fairness-Bedingungen (dieser einfachen Art) ist fest eingebaut in SMV.