Das SMV-System

H. Peter Gumm

Philipps-Universität Marburg

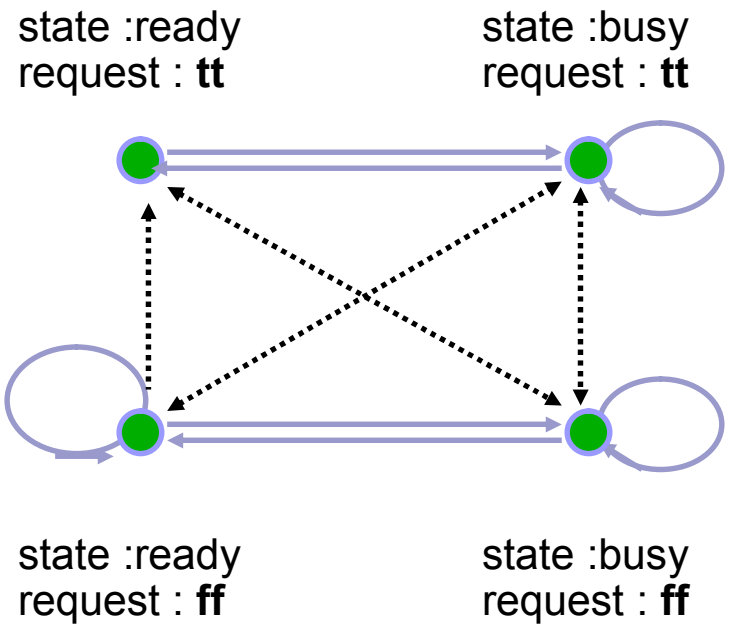
Sommersemester 2007

Ein Beispiel

Ein SMV-Programm beschreibt einen Transitionsgraphen durch Festlegung einer Menge von Anfangszuständen sowie einer Zustandsübergangsrelation.

In einem Spezifikationsteil können temporal-logische Ausdrücke angegeben werden, die von dem System überprüft werden sollen.

```
1
2  MODULE main
3  VAR
4   request : boolean ;
5   state   : { ready, busy };
6  ASSIGN
7    init(state) := ready ;
8    next(state) :=
9      case
10       state=ready & request : busy;
11   1   : { ready, busy};
12     esac;
13 LTLSPEC
14   G(request -> F state=busy)
15
```

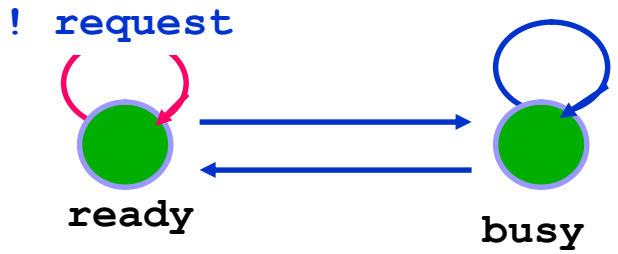


Input-Variablen

Externe Signale können als „independent variables“ **IVAR** erklärt werden. Semantisch macht dies keinen Unterschied. Begrifflich kann man jetzt den Transitions-Graphen als Automaten - mit (Kombinationen von) Signalwerten als Alphabet - auffassen.

```
1
2  MODULE main
3  IVAR
4      request : boolean ;
5  VAR
6    state : { ready, busy };
7  ASSIGN
8    init(state) := ready ;
9    next(state) :=
10     case
11       state=ready & request : busy;
12   1: { ready, busy};
13     esac;
14
15 LTLSPEC
16   G(request -> F state=busy)
```

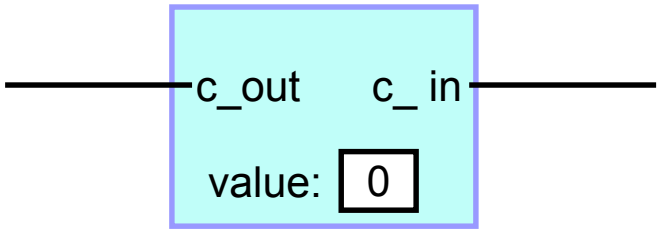
Die Variable **request** unterliegt keinerlei Restriktionen. Sie kann sich jederzeit beliebig ändern.



Die blauen Transitionen können bedingungslos ausgeführt werden, die rote nur falls `request = FALSE`.

Module und definierte Variablen

```
MODULE counter_cell(c_in)
VAR
  value : boolean;
ASSIGN
  init(value) := 0 ;
  next(value) := value+c_in mod 2
DEFINE
  c_out := value & c_in;
```



Mit parametrisierten Modulen können wiederverwertbare Systeme spezifiziert werden.

Definierte Variablen dienen als Abkürzungen. Sie werden wie Macros behandelt.

Module und definierte Variablen

```
1
2  MODULE main
3  VAR
4    bit0 : counter_cell(1);
5    bit1 : counter_cell(bit0.c_out);
6    bit2 : counter_cell(bit1.c_out);
7  LTLSPEC
8    G F bit2.c_out
9
```

```
10
11 MODULE counter_cell(c_in)
12 VAR
13   value : boolean;
14 ASSIGN
15   init(value) := 0 ;
16   next(value) := value+c_in mod 2;
17 DEFINE
18   c_out := value & c_in;
19
```

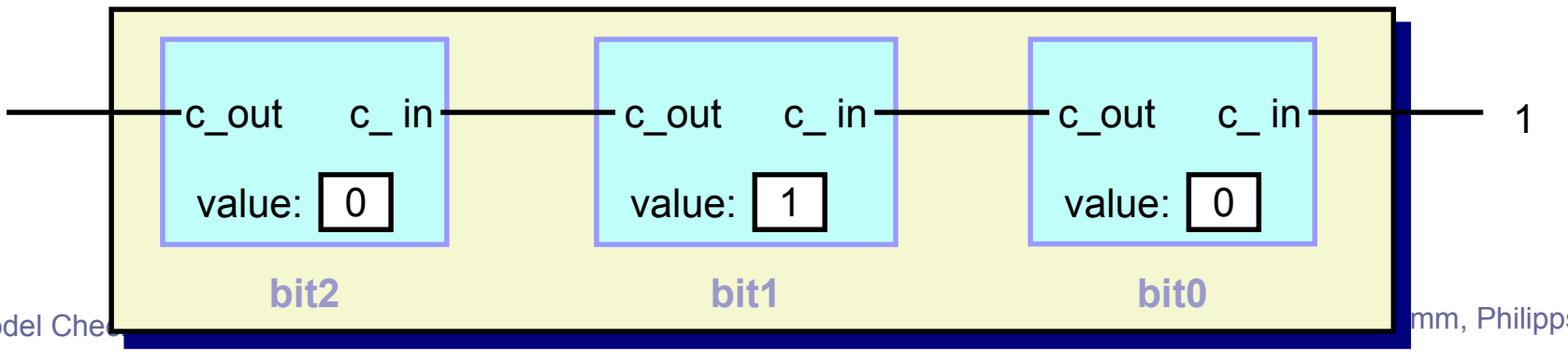
Hauptmodul muß "**main**" heißen
Kann weitere Module enthalten

Alle Zuweisungen
- im Hauptmodul und
- in den benutzten Modulen

werden synchron ausgeführt !

Mit parametrisierten Modulen können wiederverwertbare Systeme spezifiziert werden.

Definierte Variablen dienen als Abkürzungen / Macros



Prozesse

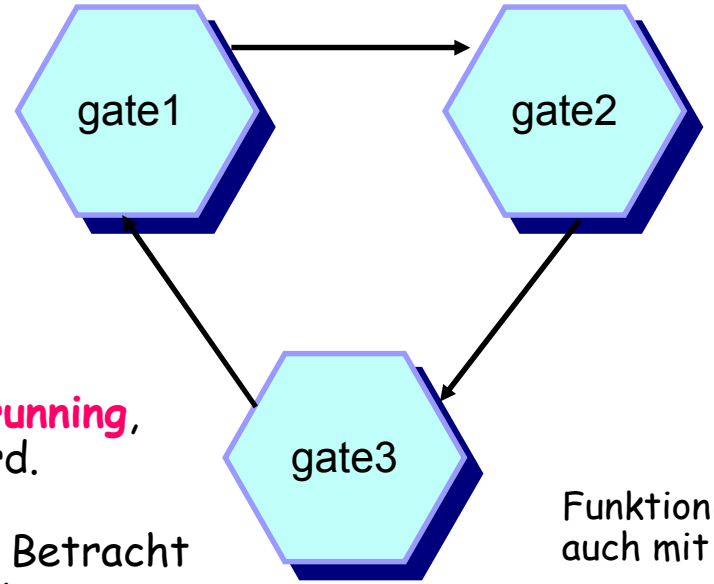
```
1
2  MODULE main
3  VAR
4    gate1 : process inverter(gate3.aus);
5    gate2 : process inverter(gate1.aus);
6    gate3 : process inverter(gate2.aus);
7
8   LTLSPEC
9      (G F gate1.aus) & (G F !gate1.aus);
10
```

```
11 MODULE inverter(ein)
12 VAR
13   aus : boolean;
14 ASSIGN
15   init(aus) := 0 ;
16   next(aus) := ! ein ;
17
18 FAIRNESS
19   running;
```

G F running

Prozesse arbeiten **asynchron**.
Es läuft immer nur ein Sohnprozeß eines gegebenen Moduls.

gate1.aus soll immer wieder **TRUE** und immer wieder **FALSE** sein.



Jeder Prozess besitzt eine implizite Boolesche Variable **running**, die genau dann wahr ist, wenn der Prozess ausgeführt wird.

Ein ***fairness-constraint*** bewirkt, daß nur solche Läufe in Betracht kommen, bei denen der *constraint* unendlich oft erfüllt ist.

Funktioniert das auch mit 3, 4, 5 ... Inverter-gates?

Nichtdeterminismus

```
1
2  MODULE main
3  VAR
4    gate1 : inverter(gate3.aus);
5    gate2 : inverter(gate1.aus);
6    gate3 : inverter(gate2.aus);
7  LTLSPEC
8    (G F gate1.aus)
9     & (G F !gate1.aus)
10
```

Diese Spezifikation wird nicht erfüllt. Warum ?

```
10
11 MODULE inverter(ein)
12 VAR
13   aus : boolean;
14 ASSIGN
15   init(aus) := 0 ;
16   next(aus) := (! ein ) union aus;
```

Asynchronizität kann auch ohne Prozessdeklarationen, bewirkt werden.

Dazu ändert man die deterministischen Zuweisungen zu nichtdeterministischen.

Auf der rechten Seite einer Zuweisung eine **Menge** von Termen, aus denen nicht-deterministisch einer ausgewählt wird.

Ein Wert v wird implizit als {v} aufgefasst, hier also

```
{ ! ein } union {out}
```

Teil der Syntax

program ::= *module* [*module*]*

module ::= **MODULE** *atom* [(*atom* { ,*atom* }*)]
decl [*decl*]*

decl ::= *vars* | *assign* | *define* | *fair* | *spec* | ...

vars ::= **IVAR** [*atom* : *type* ;]+
| **VAR** [*atom* : *type* ;]+

define ::= **DEFINE** [*atom* := *expr* ;]+

assign ::= **ASSIGN** [*dest* := *expr* ;]+

dest ::= **init(** *atom* **)** | **next(** *atom* **)**

fair ::= **FAIRNESS** *ctlexpr*

spec ::= **LTLSPEC** *ltlexpr* | **SPEC** *ctlexpr*

- Ein Programm besteht aus einer Folge von Modulen, einer davon muss **main** heißen.
- Module können formale Parameterlisten haben. Der Körper besteht aus Deklarationen.
- IVAR steht für (externe)Signale,
- VAR für interne Zustandsvariablen
- Macros können als Abkürzung für einen Ausdruck eingeführt werden.
- Hier wird die ***Dynamik*** festgelegt
- Anfangswerte durch init, Folgewerte durch next.
- Fairness constraint
- Die temporal-logische(TL) Spezifikation ist Teil des Programms.

Datentypen und Operationen

```
type      ::= boolean
            | enum
            | [ process ] modul_typ

enum      ::= { val { , val }* } | val .. val

modul_typ ::= atom [ ( expr { , expr }* ) ]
```

- Boolesche Werte werden durch 0, 1 oder FALSE,TRUE repräsentiert.
- Prozesse und Module sind Typen

- Mengen und Intervalle
- Module werde an Variablen gebunden

```
val       ::= atom | number

expr      ::= val | id
            | expr + expr | expr - expr| ...
            | expr = expr | expr < expr| ...
            | ! expr | expr & expr | ...
            | set_expr
            | case [ expr : expr ; ]+ esac
```

- Atome sind symbolische Konstanten.

- Arithmetische Operationen
- Atomare Aussagen
- Aussagen (Boolesche Kombinationen)
- Mengen-Ausdrücke

- bedingter Ausdruck

```
set_expr  ::= enum
            | id
            | expr in expr
            | expr union expr
```

- Angabe durch Aufzählung

- Test auf Mitgliedschaft
- Vereinigung

Temporale Formeln

ltl_expr ::= *expr*

| **X** *ltl_expr*

| **F** *ltl_expr*

| **G** *ltl_expr*

| *ltl_expr* **U** *ltl_expr*

Zustandsausdrücke -- am Anfang gilt ...

Folgezustand.. zum nächsten Zeitpunkt gilt

irgendwann gilt ...

immer gilt

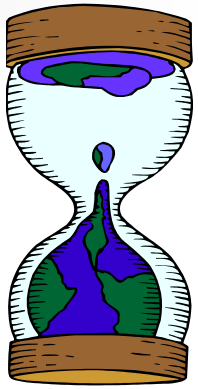
p U q heißt: p until q

Umgangssprachlich: „p bleibt wahr bis q eintritt"
bzw. „p wartet bis q"

Beispiele:

```
LTLSPEC -- Was von s gesendet wird, wird irgendwann von r empfangen

        G ( s.sent = k  ->  F r.received = k )

LTLSPEC -- will s eine Botschaft senden, versucht es das solange bis r diese empfängt

        G ( s.outMessage=k  -> ( s.outMessage=k U r.received=k) )
```

Vergangenheit



ltl _expr ::= ...

| **Y** *ltl_expr*

| **O** *ltl_expr*

| **H** *ltl_expr*

| *ltl_expr* **S** *ltl_expr*

Voriger Zustand .. zum vorigen Zeitpunkt galt..

once - irgendwann galt ...

historically - es galt immer

since - p *seit* q

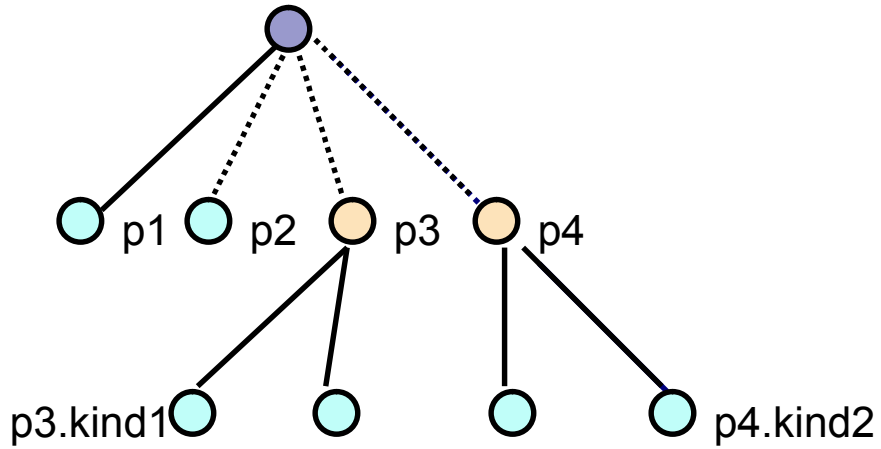
Beispiele:

```
LTLSPEC -- Was von r empfangen wurde irgendwann von s gesendet

    G ( r.received = k  ->  O s.sent = k )

LTLSPEC -- Solange r „undef" empfängt ist noch keine andere eingegangen:

    G ( r.received = undef  -> H r.received=undef )
```

Prozesshierarchie

```
MODULE Vater
VAR
        p1 :  Sohn(x) ;
        p2 :  process Sohn(y) ;
        p3 :  process Tochter(z) ;
        p4 :  process Tochter(u) ;

MODULE Sohn( .. )
..
MODULE Tochter( .. )
VAR
     kind1 : Sohn( .. )
     kind2 : Sohn( .. )
..
```

Eine Variable p, die an einen Modul M gebunden ist, heißt ***Instanz*** von M. Auf diese Weise entsteht eine Hierarchie von Instanzen. Die Wurzel dieser Hierarchie heißt "main".



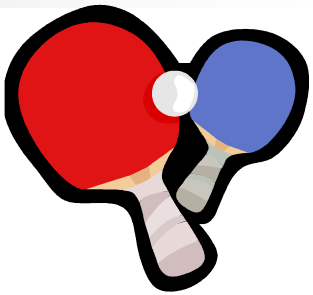
Ein **Prozess** ist eine durch das Schlüsselwort **process** gekennzeichnete Instanz. Jeder Prozess besitzt eine Variable "**running**".
Ein Assignment

next(x) := e

wird immer gleichzeitig für alle Prozesse **p** ausgeführt mit

p.running = 1

1. `main.running` ist immer wahr.
2. Für alle anderen Instanzen *P* mit Vater *V* gilt :
   $P.\texttt{running} \Rightarrow V.\texttt{running}$
3. Von den mit dem Schlüsselwort ***process*** spezifizierten Kind-Prozessen ist immer höchstens einer **running**.



Fairness

Ein ***Fairness constraint*** beschränkt die möglichen Pfade durch eine Bedingung, die unendlich oft wahr sein soll.

```
MODULE main
VAR
   a : foo
LTLSPEC
   tl-expr
...
MODULE foo
...
FAIRNESS
      running  & semaphore
```

Spezifikation braucht nur für solche Pfade gelten, die unendlich oft **`running & semaphore`** erfüllen.

Gegenbeispiele

Korrekte LTLSPECs werden ohne Begründung mit ***is true*** bestätigt.

Für nicht erfüllte Spezifikationen wird *Gegenbeispiel* geliefert.

Gegenbeispiel ist endliche oder unendliche Zustandsfolge, die in einem Anfangszustand beginnt.

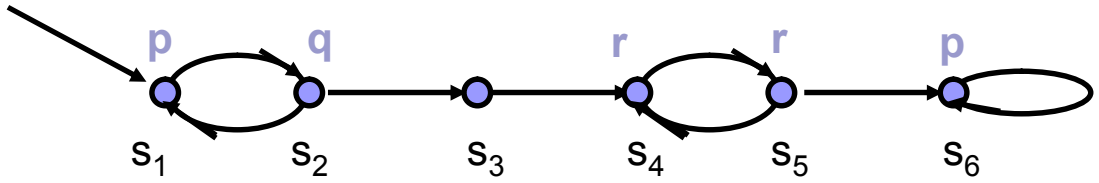
Spezifikation	Gegenbeispiel hat die Form :	
p	endliche Folge	mit Anfangszustand in dem !p gilt
X p	endliche Folge	deren zweiter Zustand !p erfüllt
F p	unendliche Folge	die zu jedem Zeitpunkt !p erfüllt
G p	endliche Folge	bei der an mindestens einer Stelle !p gilt
p **U** q	unendliche Folge oder endliche Folge	bei der immer !q gilt mit !p & !q vor dem ersten q.

In einem endlichen System kann man die unendliche Folge immer von der Form wählen:

$(s_0, s_1, \ldots, s_k,$ **loop** $s_{k+1}, \ldots, s_m)$

(SMV: *„ loop starts here “*).

Gegenbeispiele zu Spezifikationen




```
1
2  MODULE main
3  VAR
4  state : { s1, s2, s3, s4, s5, s6 };
5  ASSIGN
6    init(state) := s1 ;
7    next(state) :=
8      case
9        state=s1 : s2;
10       state=s2 : { s1, s3 };
11       state=s3 : s4;
12       state=s4 : s5;
13       state=s5 : { s4, s6 };
14       state=s6 : s6;
15     esac;
16
17 DEFINE
18   p := (state=s1|state=s6) ;
19   q := state=s2 ;
20   r := (state=s4|state=s5);
21
```

Gegenbeispiel zu :

`G ( p | q )`	: (s_1,s_2,s_3)
`r U q`	: (s_1,s_2)
`G ( r-> (r U q))`	: (s_1,s_2,s_3, loop: s_4,s_5)
`X X F q`	: (s_1,s_2,s_3, loop: s_4,s_5)
`G (p U !p )`	: (s_1,s_2,s_3,s_4,s_5, loop s_6)

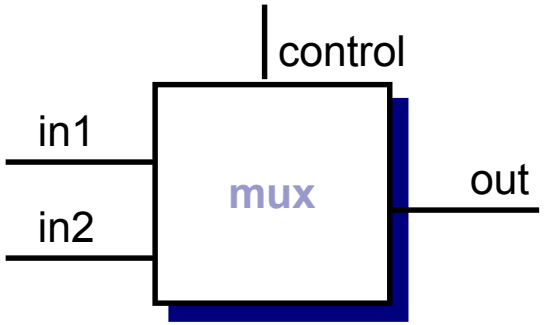
Probieren Sie dies in NuSMV aus !

Hardware Komponenten

```
MODULE delay(ein)
VAR
  out : 0 .. 255;
ASSIGN
  init(out) := 0;
  next(out) := ein;
```

$out_{t=k+1} = ein_{t=k}$

```
MODULE mux(in1,in2,control)
VAR
  out : 0 .. 255;
DEFINE
  out := case
           control  : in1;
           1        : in2;
         esac;
```

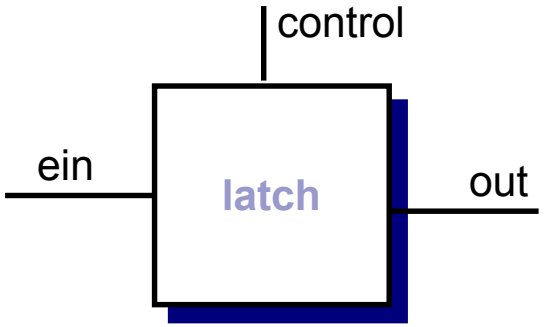


$out_{t=k}$ = **if** control **then** $in1_{t=k}$
else $in2_{t=k}$

Register

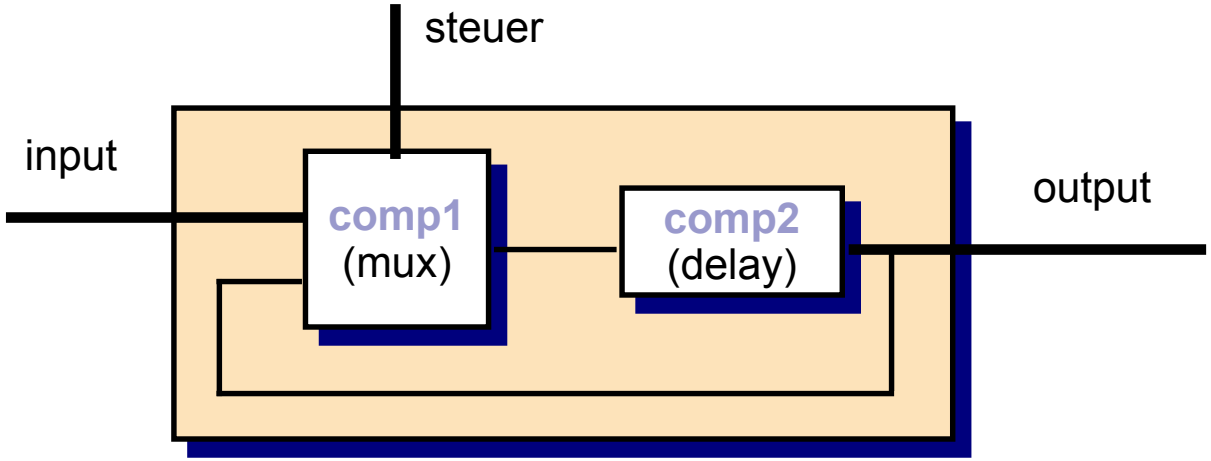
- Ein Register (engl. Latch) kann einen Wert speichern
  - Ist der Steuereingang 1 dann wird der am Eingang liegende Wert gespeichert. Der alte Wert wird überschrieben

```
MODULE latch (ein, control)
VAR
  out: 0 .. 255;

ASSIGN
 init(out) := 0;
 next(out) := case
         control  : ein;
          1       : out;
                esac;
```



$out_{t=k+1}$ = **if** control **then** $ein_{t=k}$
else $out_{t=k}$

Ein Register aus mux und delay




```
MODULE main
VAR
  input   : 0 .. 255;
  steuer  : boolean;
  comp1   : mux (input,comp2.out,steuer);
  comp2   : delay(comp1.out) ;

  k       : 0 .. 255;
ASSIGN
  next(k) := k;

DEFINE
  output  := comp2.out ;
```

Eigenschaften des Moduls :

LTLSPEC
G (steuer & input=k ⇒ X output=k)
G (! steuer & output=k ⇒ X output=k)

Macht *k* zu einem
- beliebigen, aber *festen* Byte-Wert, d.h. *k* ist beliebige Konstante.

Direkter Vergleich

```
MODULE latch(ein,control)
VAR
  aus : 0 .. 25;
ASSIGN
  next(aus) := case
      control : ein;
      1       : aus;
  esac;
```

```
MODULE main
VAR
  input  : 0 .. 255;
  steuer : boolean;
  comp1  : mux (input,comp2.out,steuer);
  comp2  : delay(comp1.out) ;
  comp3  : latch(input,steuer);
```

```
LTLSPEC
  G ( comp2.aus = comp3.aus)
```

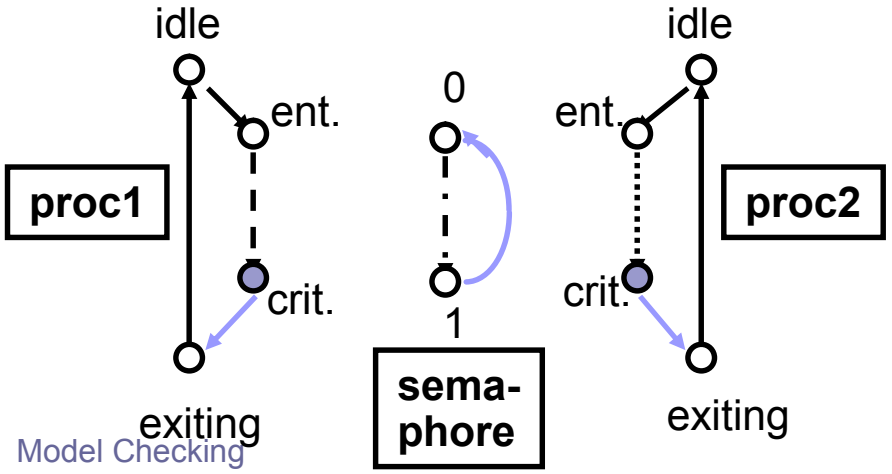
Semaphor

Mit Semaphor garantieren wir, dass zwei Prozesse nie gleichzeitig in ihrem kritischen Bereich sind.

```
MODULE main
VAR
  semaphore : boolean;
  proc1 : process user(semaphore);
  proc2 : process user(semaphore);
ASSIGN
  init(semaphore) := 0;

LTLSPEC
  G !  (      proc1.state = critical
         &    proc2.state = critical )

LTLSPEC
  G (  proc1.state = entering
       -> F proc1.state = critical )
```

Sicherheit

Lebendigkeit



Lebendigkeit ist noch nicht gegeben ! Warum ?
Modifizieren Sie das Modell, so dass
Lebendigkeit für beide Prozesse erfüllt ist.

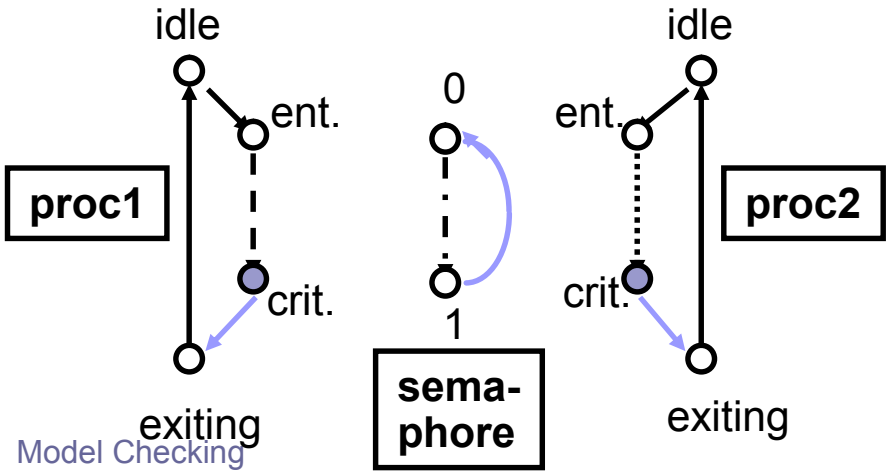
Semaphor

Mit Semaphor garantieren wir, dass zwei Prozesse nie gleichzeitig in ihrem kritischen Bereich sind.

```
MODULE main
VAR
  semaphor : boolean;
  proc1 : process user(semaphor);
  proc2 : process user(semaphor);
ASSIGN
  init(semaphor) := 0;

LTLSPEC
  G !  (   proc1.state = critical
         & proc2.state = critical)

LTLSPEC
  G (  proc1.state = entering
       -> F proc1.state = critical)
```

```
MODULE user(sema)
VAR
  state : {idle,entering,critical,exiting};
ASSIGN
  init(state) := idle;

next(state) := case
   state = idle      : { idle,entering};
   state = entering & !sema : critical;
   state = critical  : {critical,exiting};
   state = exiting   : idle;
      1              : state;
esac;

next(sema) := case
      state = entering : TRUE;
      state = exiting  : FALSE;
      1                : sema;
esac;
FAIRNESS  running
```

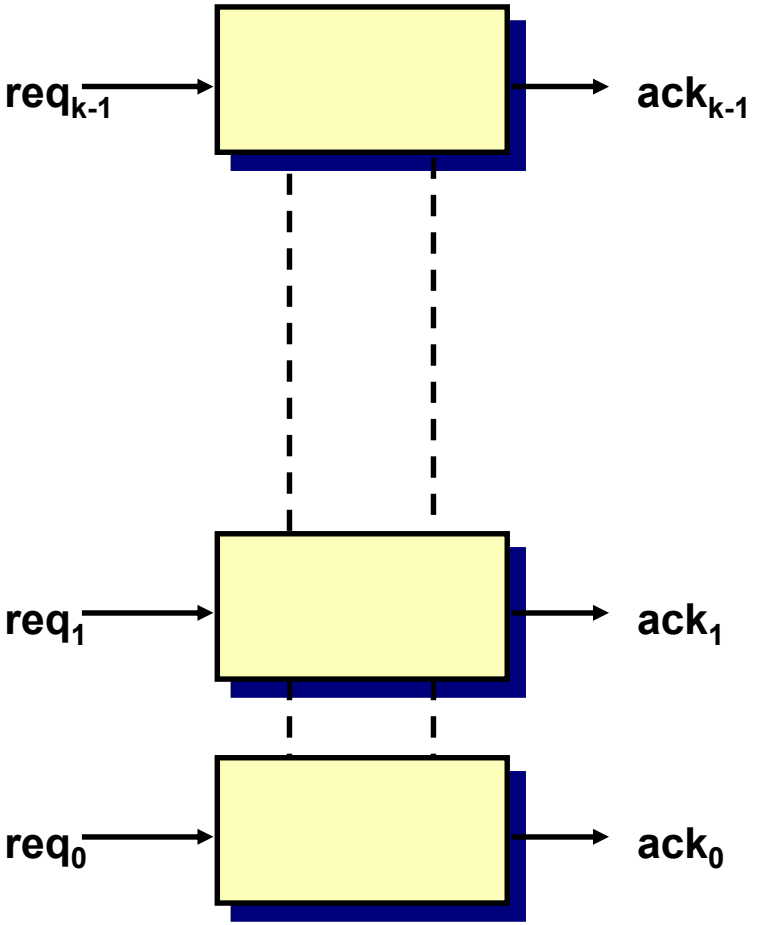
call by reference



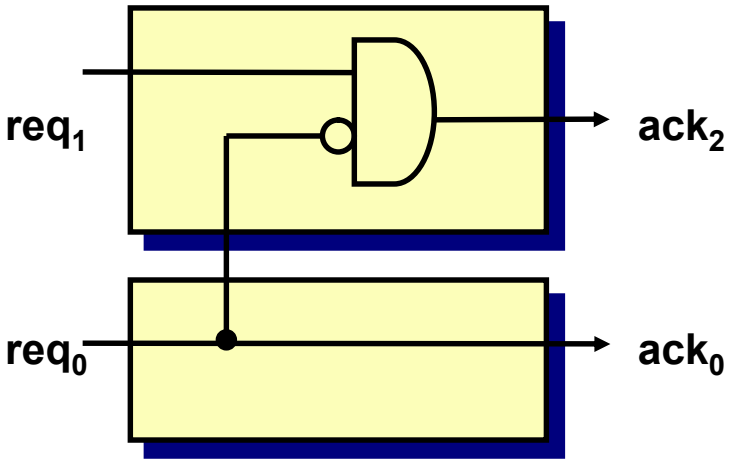
Lebendigkeit ist noch nicht gegeben ! Warum ?
Modifizieren Sie das Modell, so dass
Lebendigkeit für beide Prozesse erfüllt ist.

Buscontroller - *Arbiter*

- Mehrere Geräte G_1, ..., G_k verlangen Zugang zu einem Bus
  - Signal req_i zeigt an, dass Gerät G_i Zugang wünscht
  - Signal ack_i signalisiert, dass G_i Zugang erhält
- Sicherheit und Lebendigkeit:
  - Keine zwei Geräte haben gleichzeitig Zugang
  - Jedes Gerät, das Zugang wünscht, erhält ihn auch irgendwann



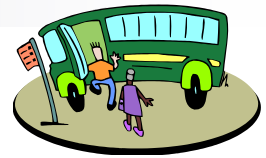
Ein (unfairer) Arbiter für zwei Geräte




```
MODULE main

IVAR
  req0 : boolean;
  req1 : boolean;

DEFINE
  ack0 := req0;
  ack1 := req1 & ! req0;

LTLSPEC  -- Sicherheit
  G !(ack0 & ack1)

LTLSPEC  -- Gerät0 funktioniert
  G (req0 ->  F ack0)

LTLSPEC -- Nicht erfüllt !
  G (req1 ->  F ack1)
```

Erster Versuch:
Sicher, aber nicht fair für G_1

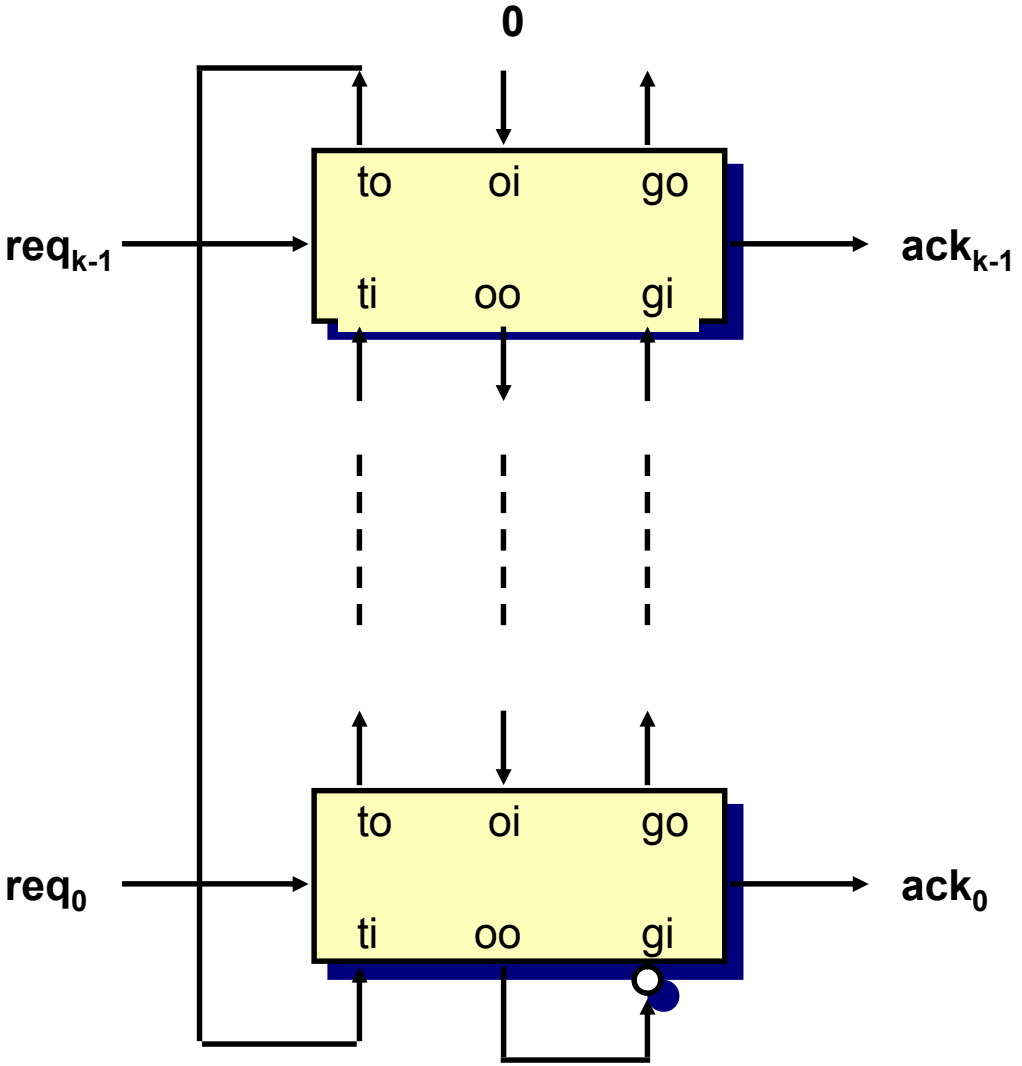
Modifizieren Sie diesen Arbiter, so dass er für beide Geräte fair wird.



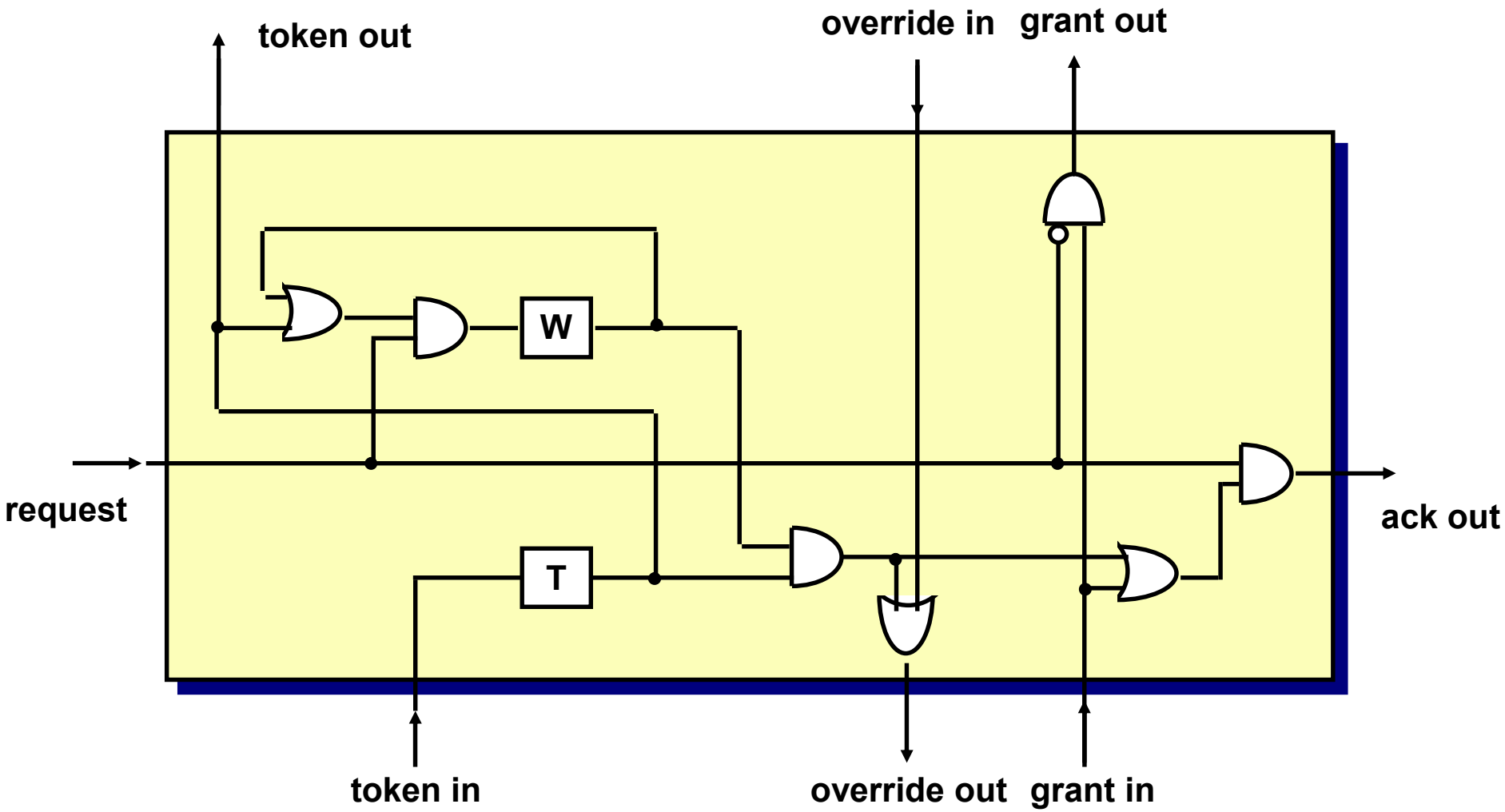
Ein flexibles Buszugangsprotokoll

Normalerweise erhält das Gerät mit niedrigstem Index den Zugang.

Wenn sehr viele Requests anfallen, wird der Zugang reihum gewährt. Dazu kreist ein Token. Gelangt es zu einer Zelle, die bereits einen Umlauf lang gewartet hat, so erhält diese sofort Zugang.

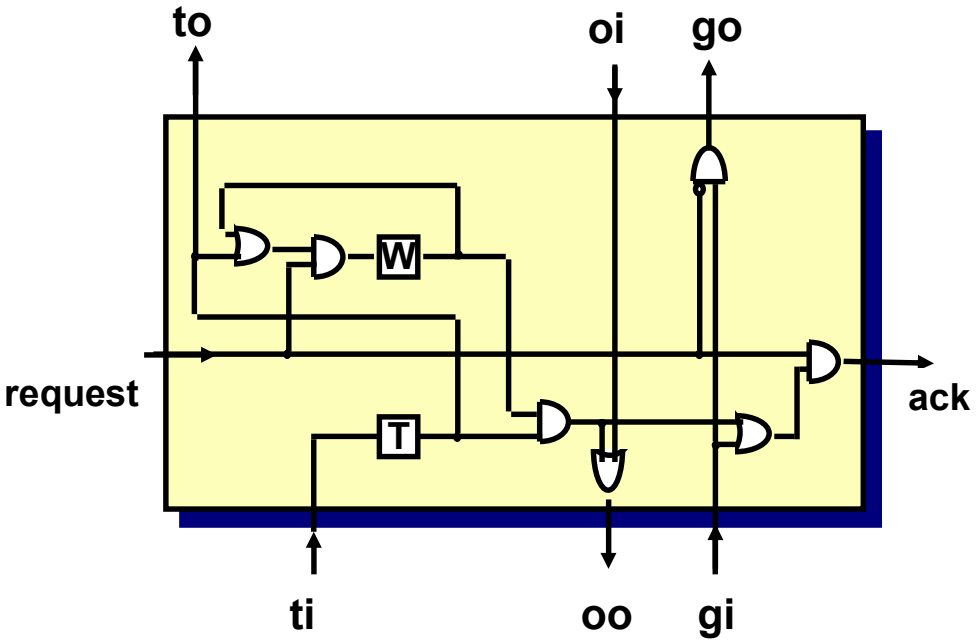


Eine einzelne Arbiterzelle



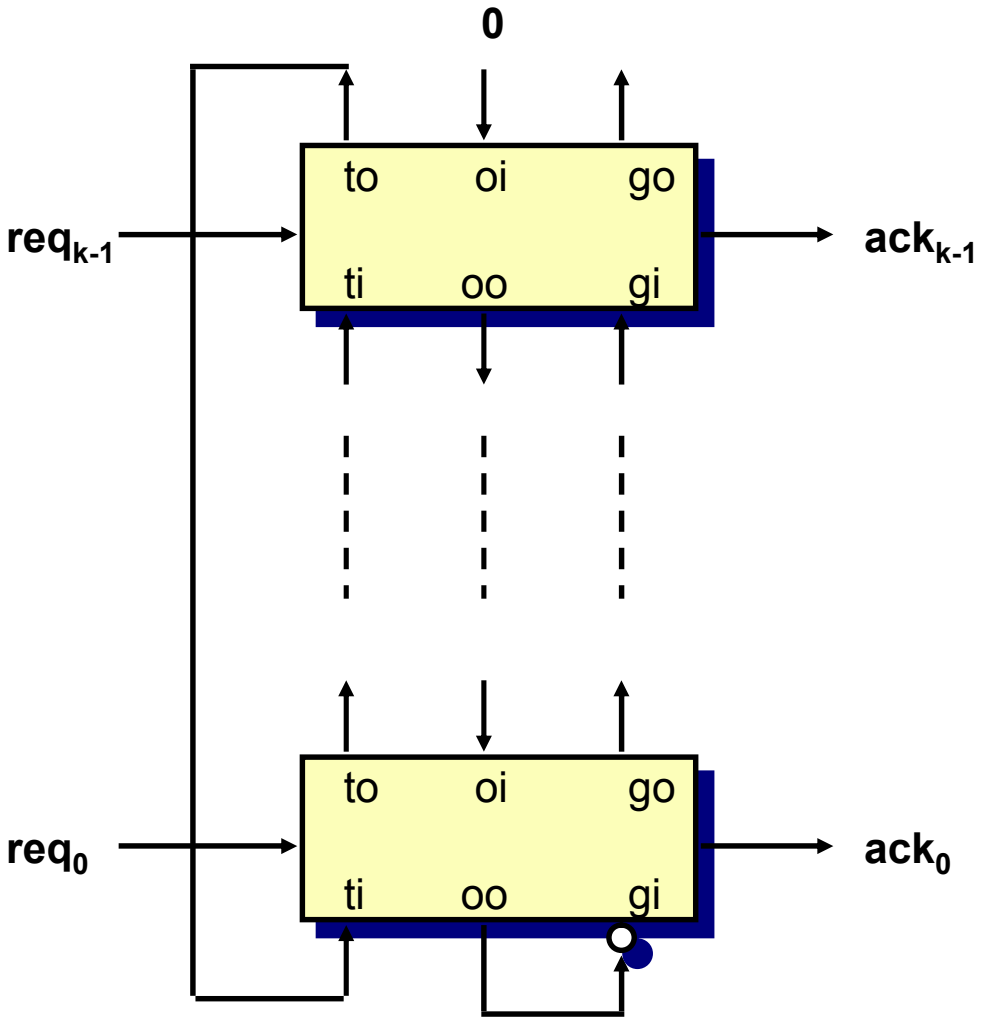
Die Arbiterzelle in SMV

```
1
2  MODULE element(init-token,ti,oi,gi)
3  VAR
4    Waiting : boolean;
5    Token   : boolean;
6    Request : boolean;
7
8  ASSIGN
9    init(Token)       := init-token;
10   next(Token)    := ti;
11   init(Waiting)     := 0;
12   next(Waiting) := Request & (Waiting | Token);
13
14 DEFINE
15   to    := Token;
16   oo   := oi | (Waiting & Token);
17   go   := !Request & gi;
18   ack := Request & (gi | Waiting & Token );
19
20 LTLSPEC  -- Kein ack ohne  request
21   G (ack -> Request)
22
23 LTLSPEC  -- Jeder Request fuehrt mal zum Ack.
24   G F ( Request -> ack )
25
```



Ein Arbiter für k Geräte

```
25
26  MODULE main
27  VAR
28    e5: element(0,e4.to,    0,e4.go);
29    e4: element(1,e3.to,e5.oo,e3.go);
30    e3: element(0,e2.to,e4.oo,e2.go);
31    e2: element(0,e1.to,e3.oo,e1.go);
32    e1: element(0,e0.to,e2.oo,e0.go);
33    e0: element(0,e5.to,e1.oo,!e0.oo);
34
35  DEFINE
36    sum :=  e0.ack + e1.ack + e2.ack
37    + e3.ack + e4.ack + e5.ack;
38
39  LTLSPEC  -- Es wird immer hoechstens
40           -- ein request durchgelassen
41     G sum <= 1
```

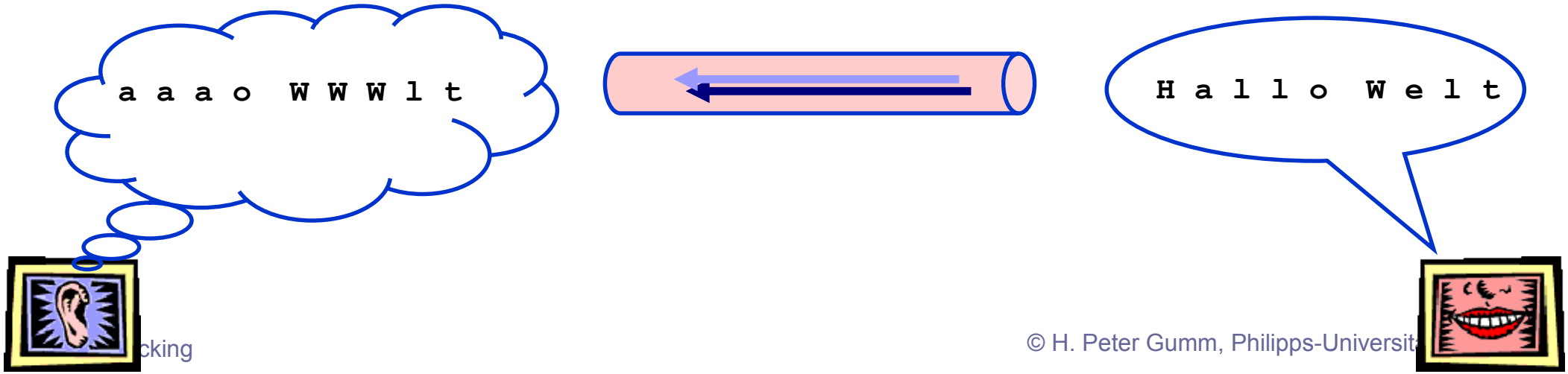


Sichere Datenübertragung

- Sender und Empfänger kommunizieren über unsichere Kanäle
  - Kleine Fehler können mit Codierungstheorie automatisch korrigiert werden
  - große Fehler nicht
  - Verluste ganzer Datenpakete erst recht nicht
- Wie kann man trotzdem sichere Datenübertragung gewährleisten ?
  - Quittungskanal einführen
  - ein Nachrichtenblock muss mehrmals gesendet werden
  - so oft, bis er quittiert wird
- Quittungskanal ist genauso unsicher
  - Quittungen können verloren gehen
  - Quittungen müssen auch wiederholt werden

Lossy Channel

- Ein ***verlustbehafteter Kanal*** (engl.: *lossy channel*) kann Datenpakete
  - verlieren
  - duplizieren
  - korrekt weiterleiten
- Die Reihenfolge der Datenpakete bleibt dabei erhalten
- Der Kanal heißt ***fair***, falls ein Datenpaket nicht unendlich oft verloren gehen kann.
  - D.h. der Kanal ist nicht völlig kaputt

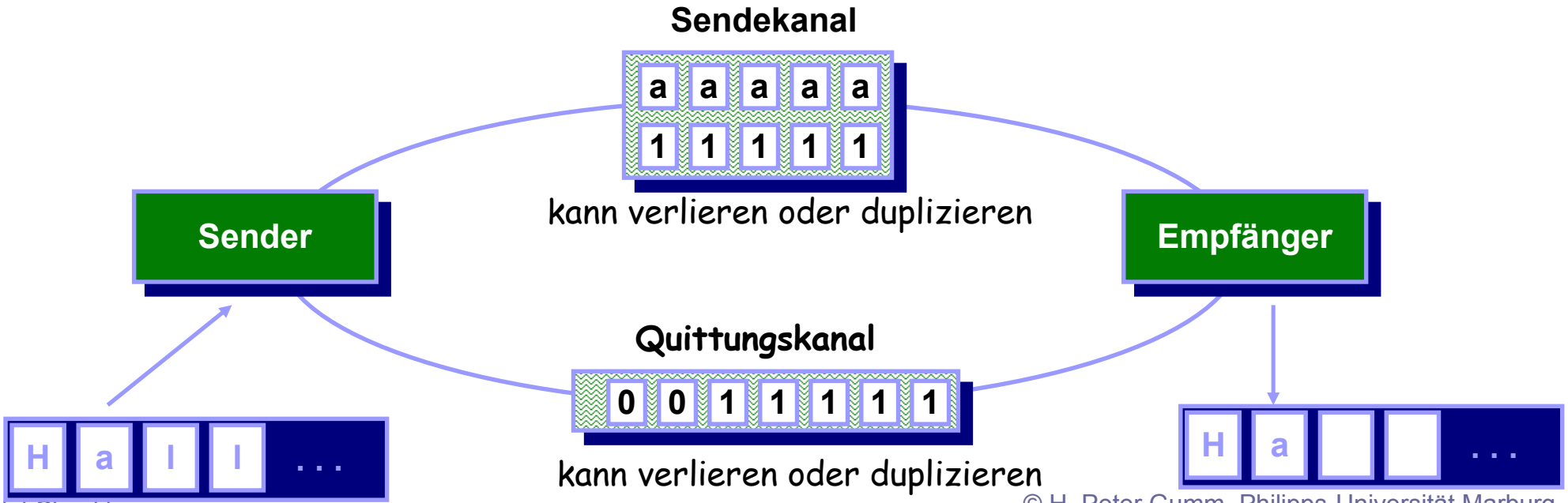


Alternierendes-Bit-Protokoll

- Sichere Datenübertragung über faire lossy channels:
  - Hefte Quittung an Nachricht
  - Sende Nachricht immer wieder
  - bis die Quittung zurückkommt
- Quittungskanal:
  - sende Quittung immer wieder
  - bis nächste Nachricht kommt
- Wie stelle ich fest
  - ob Nachricht neu ist, oder Wiederholung einer Nachricht mit verloren gegangener Quittung
  - ob Quittung neu ist, oder Wiederholung einer alten Quittung
- Reicht als Quittung ein einzelnes Bit ?
  - Ein veröffentlichter Beweis sagt: unmöglich
- Dennoch: Es funktioniert:
  - Bei jeder neuen Nachricht wechselt (alterniert) das Bit
  - Beweis mittels SMV

ABP

- Jedes Nachrichtenpaket mit einem Bit sichern (p_i+b)
  - aufeinanderfolgende Pakete mit verschiedenen Bits. Bit-Änderung signalisiert Empfänger, dass Paket p_i neu ist
  - Empfänger sendet immer wieder b in den Quittungskanal
  - Sender wiederholt das Senden von (p_i+b) solange, bis b quittiert wurde, dann sendet er das nächste Paket p_{i+1} mit Bit $\neg$b

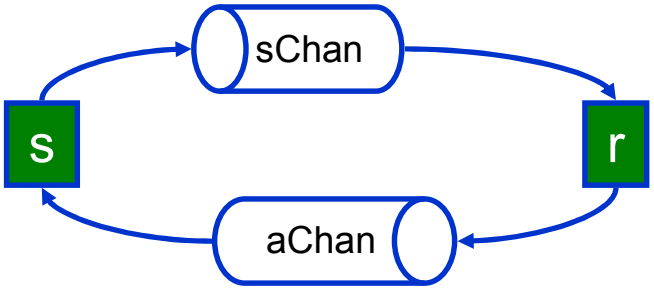


ABP in SMV

ABP.smv*

```
MODULE main
VAR
        s       : process sender(aChan.outBit);
        r       : process receiver(sChan.sent,sChan.outBit);
        sChan   : process sendeKanal(s.sentMessage,s.checkBit);
        aChan   : process ackKanal(r.ack);
```

For Help, press F1 | Ln 88, Col. 13, C0 | DOS | Mod: 3/23/2003 9:33:28PM | File Size: 565



Komponenten arbeiten asynchron
→ als Prozesse modelliert

ackKanal(inBit)

Wenn er funktioniert,
sendet er das eingehende Bit zum Ausgang

Wenn nicht, wiederholt er das letzte Bit.

Immer mal wieder kommt er
an die Reihe und funktioniert

ABP.smv*

```
MODULE ackKanal(inBit)
VAR
  working : boolean;
  outBit  : boolean;
ASSIGN
  next(outBit) := case
     working : inBit;
     1       : outBit;
  esac;
FAIRNESS
  running & working
```

For Help, press F1 | Ln 88, Col. 13, C0

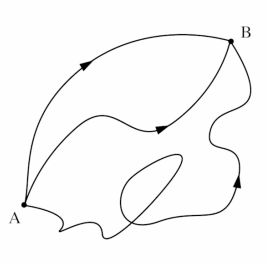
Verifikation

```
MODULE main
VAR
        s       : process sender(aChan.outBit);
        r       : process receiver(sChan.sent,sChan.outBit);
        sChan   : process sendeKanal(s.sentMessage,s.checkBit);
        aChan   : process ackKanal(r.ack);

------ Beliebige aber feste Nachricht:
        k       : {wie, hallo, junge, alter, gehts, bla, blabla};
ASSIGN
       next(k) := k;

LTLSPEC  -- jede(!!!) Message, die versandt wird, wird auch empfangen
   G (s.sent=k -> F r.received=k)
```

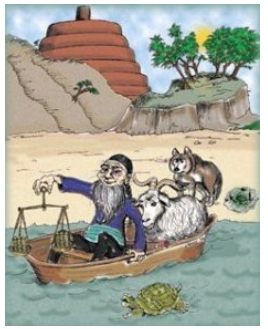
For Help, press F1 | Ln 96, Col. 1, C0 | DOS | Mod: 3/23/2003 9:33:28PM | Bytes Sel: 40

- Wir wollen zeigen, dass *jede beliebige* Nachricht erfogreich durchkommt
- Wir möchten gerne spezifizieren:
  - $\forall$ k $\in$ {wie, hallo, junge, alter, gehts, bla, blabla} G s.sent= k $\rightarrow$ F r.received=k
  - Aber: Quantoren nicht möglich in SMV/LTL
- Trick: Führe eine Variable k ein
  - k beliebig
  - aber fest

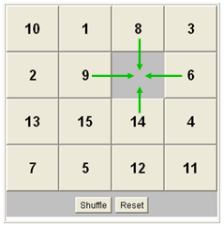
Planen mit SMV

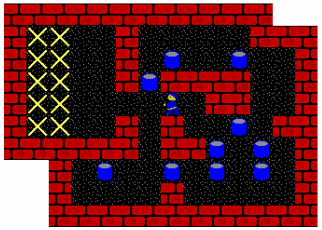


- Planungsprobleme oft gegeben durch
  - Menge von Zuständen
    - Untermenge von erlaubten Zuständen
  - Menge von Transitionen
  - Startzustände und Zielzustände
  - Gesucht: Weg von einem Start- zu einem Zielzustand



- Beispiele:
  - Wolf, Ziege, Kohlkopf-Problem
    - Wolf frisst Ziege, Ziege frisst Kohlkopf , außer der Fährmann ist dabei
    - Fährmann soll die drei im Boot übersetzen,
    - es passt nur ein Passagier ins Boot
  - Spiele
    - TicTacToe
    - Hanoi
  - Puzzles
    - Schiebepuzzle, **Sokoban** ( )
  - Sudoku
    - *Abkürzung für* (*s ji wa dokushin ni kagiru*)





- Lösung mit NuSMV
  - codiere Zustände
  - codiere Transitionen
  - Behaupte mittels LTL-Ausdruck: Es gibt keine Lösung
  - Lasse NuSMV ein Gegenbeispiel produzieren.

Wolf, Ziege, Kohlkopf in SMV

```
 3 MODULE main
 4 -- Modellierung der Zustände - wer ist auf welcher Seite
 5 VAR
 6   kohl  : boolean;   ziege  : boolean;
 7   wolf  : boolean;   boot   : boolean;
 8
 9 DEFINE  -- aus optischen Gründen:
10   links  := FALSE;  rechts := TRUE;
11
12 -- anfangs sind alle auf der linken Seite
13 ASSIGN
14   init(kohl)  := links;  init(ziege)  := links;
15   init(wolf)  := links;  init(boot) := links;
16
17 -- Sichere Zustände und Zielzustand
18 DEFINE
19   sicher := (ziege = wolf | ziege = kohl) -> ziege = boc
20   ziel   := kohl = rechts & ziege = rechts & wolf = rech
21
22 -- Mögliche Transaktionen als Prozesse modelliert
23 -- >> nur ein Prozess wird jeweils ausgeführt
24 VAR
25   wolf_trans : process transport(boot,wolf);
26   ziege_trans: process transport(boot,ziege);
27   kohl_trans : process transport(boot,kohl);
28   leerfahrt  : process leer(boot);
29 -- Behauptung: Es gibt keinen Weg zum Ziel,
30 -- der nur sichere Zustände enthält
31 -- Natürlich hoffen wir auf ein Gegenbeispiel !!
32 ------------------------------
33 LTLSPEC
34   G sicher -> G !ziel
```

```
48 -- Das Boot transportiert einen Passagier,
49 MODULE transport(boot,passagier)
50 ASSIGN
51   next(boot) := case
52     boot = passagier : ! boot;
53     1                : boot;
54   esac;
55
56   next(passagier) := case
57     boot = passagier : ! passagier;
58     1                : passagier;
59   esac;
60
61 -- Das Boot macht eine Leerfahrt
62 MODULE leer(boot)
63 ASSIGN
64   next(boot) := ! boot;
65
66 ----------------------------------------
67 -- Bounded Model Checking liefert ein
68 -- Gegenbeispiel in 7 Schritten
```

Lösung des „ferryman‘s problem“

- **Minimale** Lösung mit „bounded model checking“ gefunden:
- Aufruf
  `NuSMV.exe -bmc wzk.smv`

```
Output Window
-- no counterexample found with bound 0
-- no counterexample found with bound 1
-- no counterexample found with bound 2
-- no counterexample found with bound 3
-- no counterexample found with bound 4
-- no counterexample found with bound 5
-- no counterexample found with bound 6
-- no counterexample found with bound 7
-- specification ( G sicher ->  G !ziel)    is false
-- as demonstrated by the following execution sequence
```

```
 4 Trace Type: Counterexample
 5 -> State: 1.1 <-
 6   kohl = 0, ziege = 0, wolf = 0, boot = 0,
 7   rechts = 1, links = 0, ziel = 0, sicher = 1
 8   ...
 9   _process_selector_ = ziege_trans
10 -> State: 1.2 <-
11   ziege = 1, boot = 1
12   ...
13   _process_selector_ = leerfahrt
14 -> State: 1.3 <-
15   boot = 0
16   ...
17   _process_selector_ = wolf_trans
18 -> State: 1.4 <-
19   wolf = 1, boot = 1
20   ...
21   _process_selector_ = ziege_trans
22 -> State: 1.5 <-
23   ziege = 0, boot = 0
24   ...
25   _process_selector_ = kohl_trans
26 -> State: 1.6 <-
27   kohl = 1, boot = 1
28   ...
29   _process_selector_ = leerfahrt
30 -> State: 1.7 <-
31   boot = 0
32   ...
33   _process_selector_ = ziege_trans
34 -> State: 1.8 <-
35   ziege = 1, boot = 1. ziel = 1
```

Hausaufgaben

- 1. Verbessern Sie den einfachen (unfairen) Arbiter für *zwei* Geräte
  - Der neue sollte sicher sein, und fair zu beiden Geräten,
  - aber nicht so aufwendig wie das allgemeine Buszugangsprotokoll.
- 2. Verändern Sie die Modellierung der Arbiterzelle, so dass Sie verifizieren können:
  - Spätestens nach einem Umlauf des Tokens folgt auf req_i ein ack_i.
- 3. Im direkten Vergleich von Latch und selbstgebautem Register
  - begründen Sie, warum LTLSPEC G (comp3.out = output) nicht erfüllt ist.
  - Verbessern Sie die Modellierung, (d.h. die SMV-Implementierung) von latch, delay und main, so dass die obige temporale Formel wahr wird.
- 4. Verbessern Sie die Semaphoren-Modellierung des wechselseitigen Ausschlusses zweier Prozesse, so dass sie nicht nur sicher, sondern auch insofern fair wird, dass beide Prozesse abwechselnd ihren kritischen Bereich betreten dürfen.
  - Hinweis: Verwenden Sie eine Variable turn, die besagt, welcher Prozess „dran" ist.
  - Jeder user prozess hat neue Parameter : user(semaphore,pId,turn). pId ist die Prozess_Id (0 oder 1), turn die neue Variable.
  - Spezifizieren und beweisen Sie als LTLSPEC die Sicherheits-und Lebendigkeitseigenschaften.

Hausaufgabe

1. Ergänzen Sie dazu das folgende SMV-Fragment des ABP. Sie können weitere Variablen einführen, das ist aber nicht notwendig.

2. Spezifizieren Sie die folgenden Eigenschaften in LTL und prüfen Sie, dass sie in Ihrer Modellierung gelten.

- unendlich viele Nachrichten werden erfolgreich quittiert
- jede gesendete Nachricht wird auch empfangen
- eine Nachricht wird so lange gesendet, bis sie empfangen wird
- Nur gesendete Nachrichten werden empfangen
- wenn eine message nie gesendet wird, dann wird sie auch nie empfangen
- ab einem Zeitpunkt wird "undef" nicht mehr empfangen
- nachdem die erste Nachricht != undef empfangen wurde, wird später nie mehr undef empfangen.

```
MODULE main
VAR
  s       : process sender(aChan.outBit);
  r       : process receiver(sChan.sent,sChan.outBit);
  sChan   : process sendeKanal(s.sentMessage,s.checkBit);
  aChan   : process ackKanal(r.ack);

------ Beliebige aber feste Nachricht:
        k       : {wie, hallo, junge, alter, gehts};
ASSIGN
       next(k) := k;

MODULE sender(inBit)
VAR
  nextMessage    : {hallo, alter, junge, wie, gehts};
  checkBit       : boolean;
  sentMessage    : {hallo, alter, junge, wie, gehts};

-- ...

MODULE receiver(inMessage,inBit)
VAR
  expected : boolean;
  ack      : boolean;
-- ...
```

```
MODULE ackKanal(inBit)
VAR
  working : boolean;
  outBit  : boolean;
ASSIGN
  next(outBit) := case
     working : inBit;
     1       : outBit;
  esac;
FAIRNESS
  running & working
MODULE sendeKanal(inMessage,inBit)
VAR
   working : boolean;
   sent    : {undef, hallo, alter, junge,
wie, gehts};
   outBit  : boolean;
-- ...
```