Motivation und Einführung

H. Peter Gumm

Philipps-Universität Marburg

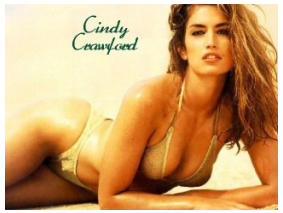
Sommersemester 2007

Model Checking hat nichts zu tun mit ...

- ... Schiffsmodellen

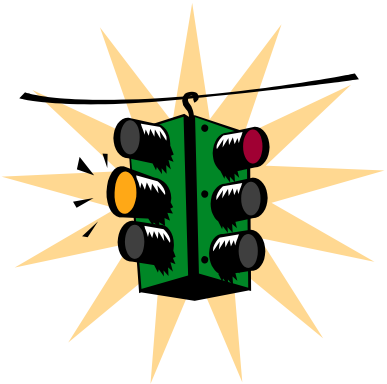
- ...Modelleisenbahnen

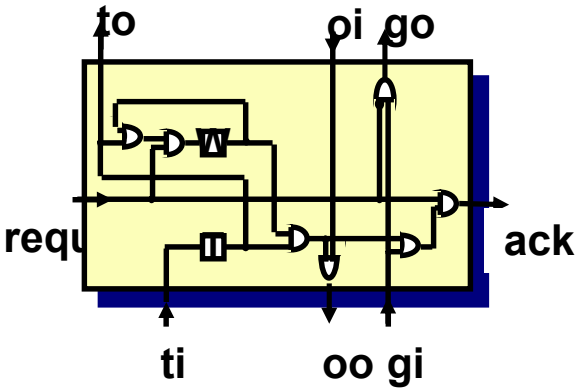
- ... solchen Modellen ...



Beim Model checking geht es um …

- Spezifikation und automatische Verifikation von
  - Hardware
  - Software
  - Protokollen
  - Steuerungen
- In dieser Vorlesung behandeln wir
  - Anwendungen von Model Checking
    - Viele Beispiele
  - Funktionsweise von Model Checkern
    - Insbesondere SMV
    - Effiziente Implementierungstechniken
  - Theoretische Hintergründe
    - Transitionssysteme
    - Temporale Logiken

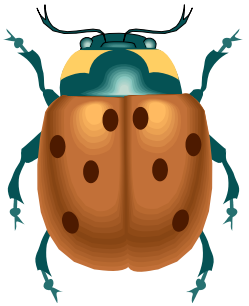




Software Qualitätssicherung

- Testen
  - Unit tests
  - Inspektionen
  - Reviews
  - …
- Es bleiben aber Fehler übrig
  - Sonderfälle, an die niemand gedacht hat
  - …

Durch Testen kann man die Anwesenheit, nicht die Abwesenheit von Fehlern feststellen

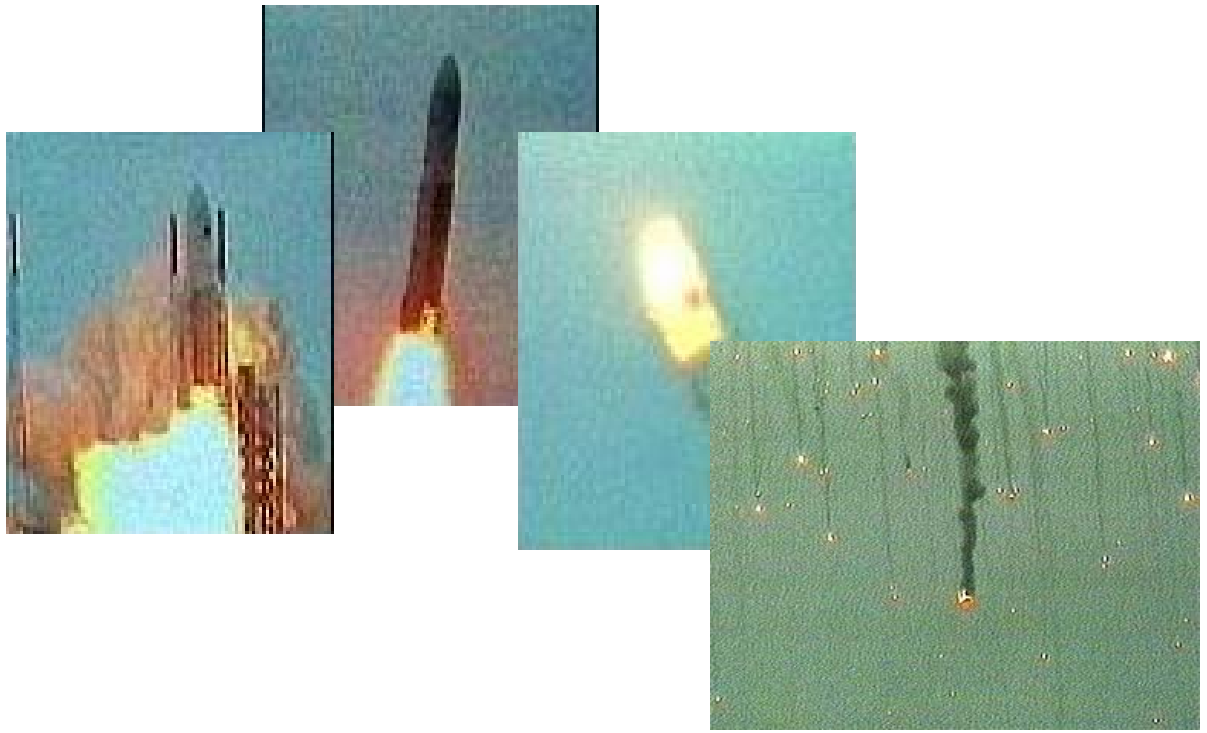


Neulich in der Karibik …

- wandelte ein Rechner irgendwo in Südamerika eine 64-Bit Integer in eine 16-Bit Integer um
- es war keine *exception* programmiert, um den Fehler zu behandeln.
- der Rechner stürzt ab.
- … aber zum Glück gibt es einen backup rechner
- der Backup Rechner führt den gleichen Code aus
- der gleiche Fehler tritt auf

- noch ein Absturz … davon gibt es Bilder:

nicht nur ein Rechner stürzt ab



ein Feuerwerk für 4 Milliarden DM
wegen eines dummen Softwarefehlers ….

Nichtdeterministische Systeme

Verteilte Systeme sind *nichtdeterministisch*.
Unklar, in welcher Reihenfolge

- Signale
- Messages
- Interrupts

ankommen.

Testen führt nicht zum Ziel. Es bleiben Fehler übrig, man sagt dann:

- *„Verkettung unglücklicher Umstände“*, **oder**
- **Murphy's Law** *„what can go wrong will go wrong“*

Ein Produzent und zwei Konsumenten

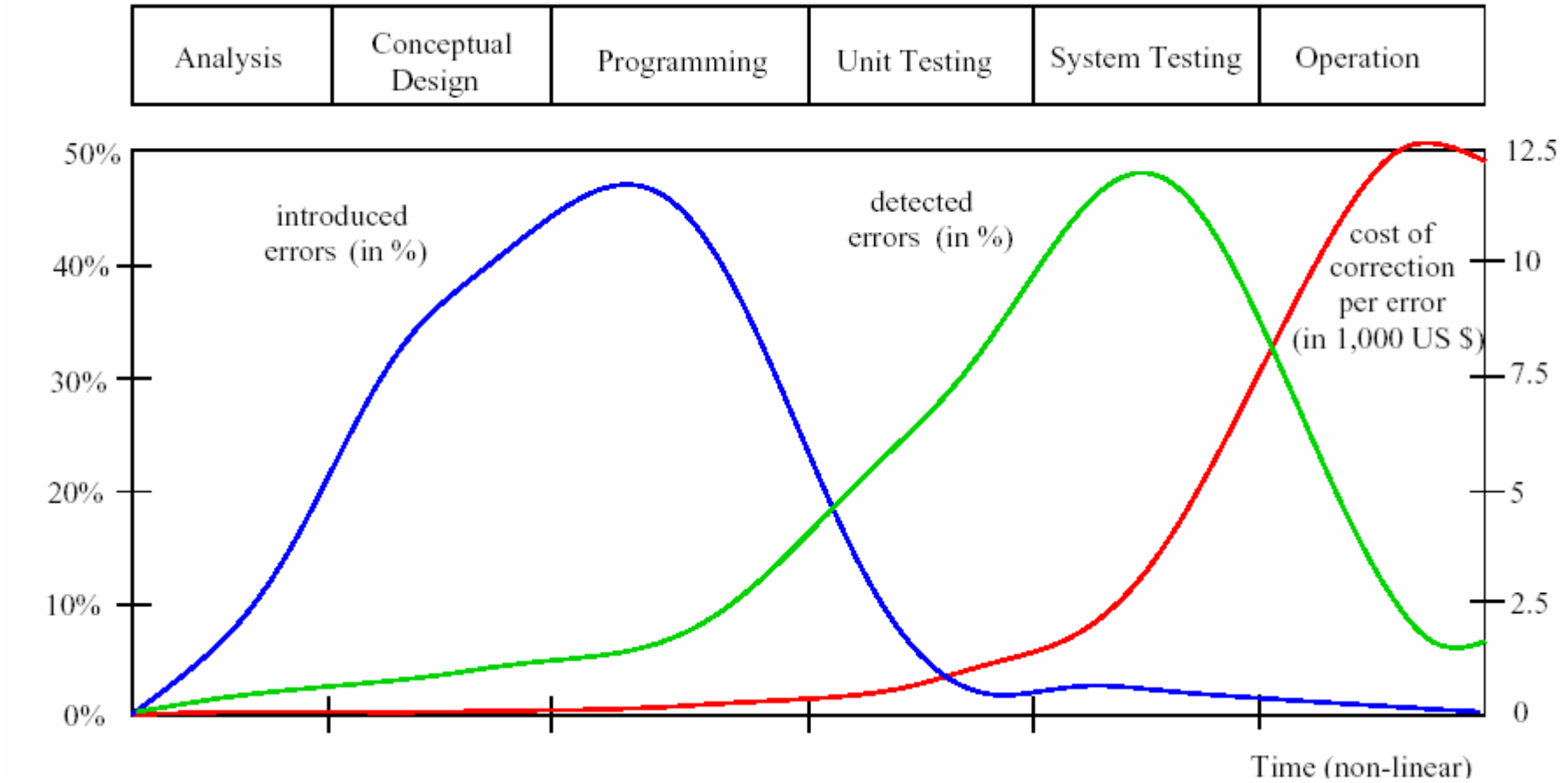
```
public class Banking {

  int konto = 0;

  class Produzent extends Thread{
    public void run(){
        while(true){
            if (konto < 100){
               System.out.println(konto);
               konto += 10;
   }}}}

  class Konsument extends Thread{
    public void run(){
        while(true){
            if (konto >= 10) {
              System.out.println(konto);
              konto -= 10;
    assert konto >= 0 : "Konto in den Miesen";
    }}}}
```

```
  public void test(){
    Produzent otto = new Produzent();
    Konsument eva = new Konsument();
    Konsument anna = new Konsument();

    otto.start();
    eva.start();
    anna.start();
    }

}// Ende der Klasse Banking
```

… lange getestet, ging auch immer gut, doch eines Tages, durch unglückliche Verkettung von Umständen

„Konto in den Miesen"

Fehler in Software und Hardware



aus J.P.Katooen

Software Qualitätssicherung

Am besten: Verifikation

aber ...

- Programmverifikation geht nicht vollautomatisch
- Rechnerunterstützung notwendig
- Verifikation ist mühsam und daher teuer

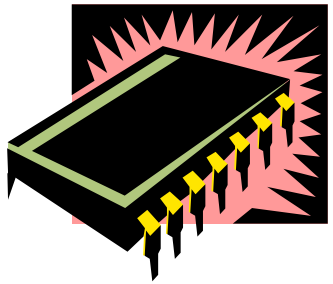
Daher: Allgemeine Programmverifikation nur anwendbar bei überschaubaren und gut verstandenen Softwaresystemen oder punktuell bei kritischen Routinen.

Endliche Systeme

Viele Systeme der Praxis haben nur *endlich viele* Zustände

- Boolesche Schaltungen
- kombinatorische Schaltungen
  (mit speichernden Gliedern)
- Programme mit endlichen Zustandsmengen
- Protokolle

Kann man solche Systeme vollständig testen ?



Die Bank - modelliert in NuSMV

```
MODULE producer(konto)
VAR
  pc : 0 .. 1;

ASSIGN
  init(pc) := 0;

  next(pc) := case
    pc =0 & konto < 100 : 1;
    pc =1               : 0;
    TRUE                : pc;
  esac;

  next(konto) := case
    pc = 1     : (konto+10) mod 256;
    TRUE       : konto;
  esac;
```

```
MODULE consumer(konto)
VAR
  pc : 0 .. 1;

ASSIGN
  init(pc) := 0;

  next(pc) := case
    pc=0 & konto >= 10: 1;
    pc =1             : 0;
    TRUE              : pc;
  esac;

  next(konto):= case
    pc=1 : (konto + 246) mod 256;
    TRUE : konto;
  esac;
```

Entspräche in PseudoAssembler:

```
0   : cmp konto 100
      jge 0

1   : add konto 10
      jmp 0
```

```
0   : cmp konto 10
      jb 0

1   : sub konto 10
      jmp 0
```

Überprüfung in NuSMV

```
MODULE main
  VAR
    konto : 0 .. 255;
    otto : process  producer(konto);
    eva  : process  consumer(konto);
    anna : process  consumer(konto);

ASSIGN
  init(konto) := 0;

LTLSPEC
  G konto <= 100
```

```
-- specification  G konto <= 100     is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  konto = 0
  otto.pc = 0
  eva.pc = 0
  anna.pc = 0
-> Input: 1.2 <-
  _process_selector_ = otto
  running = 0
  anna.running = 0
  eva.running = 0
  otto.running = 1
-> State: 1.2 <-
  otto.pc = 1
```

```
er Tool 1        Ln 2, Col. 23, C0        DOS
```

```
 1 Trace Type: Counterexample
 2 -> State: 1.1 <-
 3   konto = 0
 4   otto.pc = 0
 5   eva.pc = 0
 6   anna.pc = 0
 7   ... etc ...
 8   _process_selector_ = otto
 9   ...
10 -> State: 1.2 <-
11   otto.pc = 1
12   _process_selector_ = otto
13   ...
14 -> State: 1.3 <-
15   konto = 10
16   _process_selector_ = anna
17   ...
18 -> State: 1.4 <-
19   anna.pc = 1
20   _process_selector_ = eva
21   ...
22 -> State: 1.5 <-
23   eva.pc = 1
24   _process_selector_ = anna
25   ...
26 -> State: 1.6 <-
27   konto = 0
28   _process_selector_ = eva
29   ...
30 -> State: 1.7 <-
31   konto = 246
```

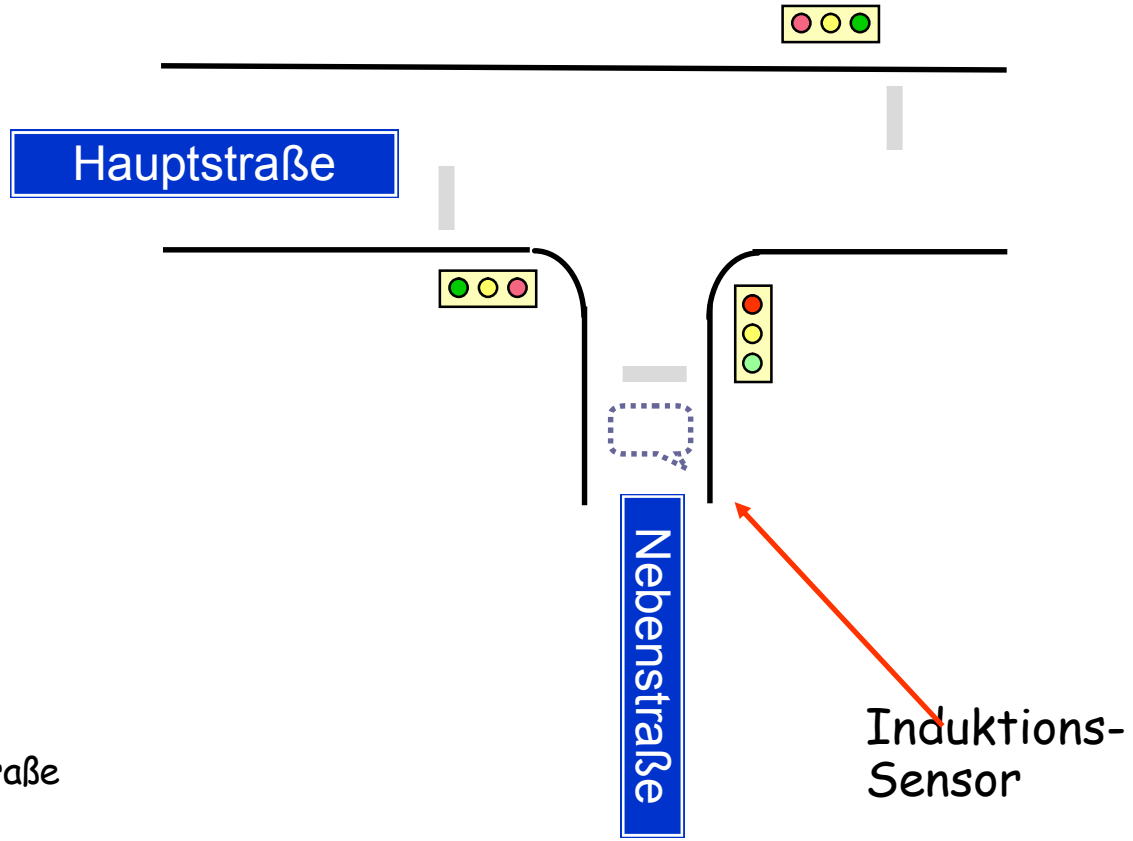
Wir nehmen drei Prozesse an.

Wir behaupten, dass das Konto immer <= 100 sei

NuSMV liefert minimales Gegenbeispiel

Eine Ampelsteuerung

- **Nichtdeterminismus**
  - Man weiß nicht, wann und von wo ein Auto kommt
- **Gewünschte Eigenschaften**
  - **Sicherheit**
    - Die Ampeln von Haupt- und Nebenstraße dürfen nie gleichzeitig grün sein
  - **Lebendigkeit**
    - Wenn ein Auto in der Nebenstraße wartet, wird irgendwann seine Ampel grün.

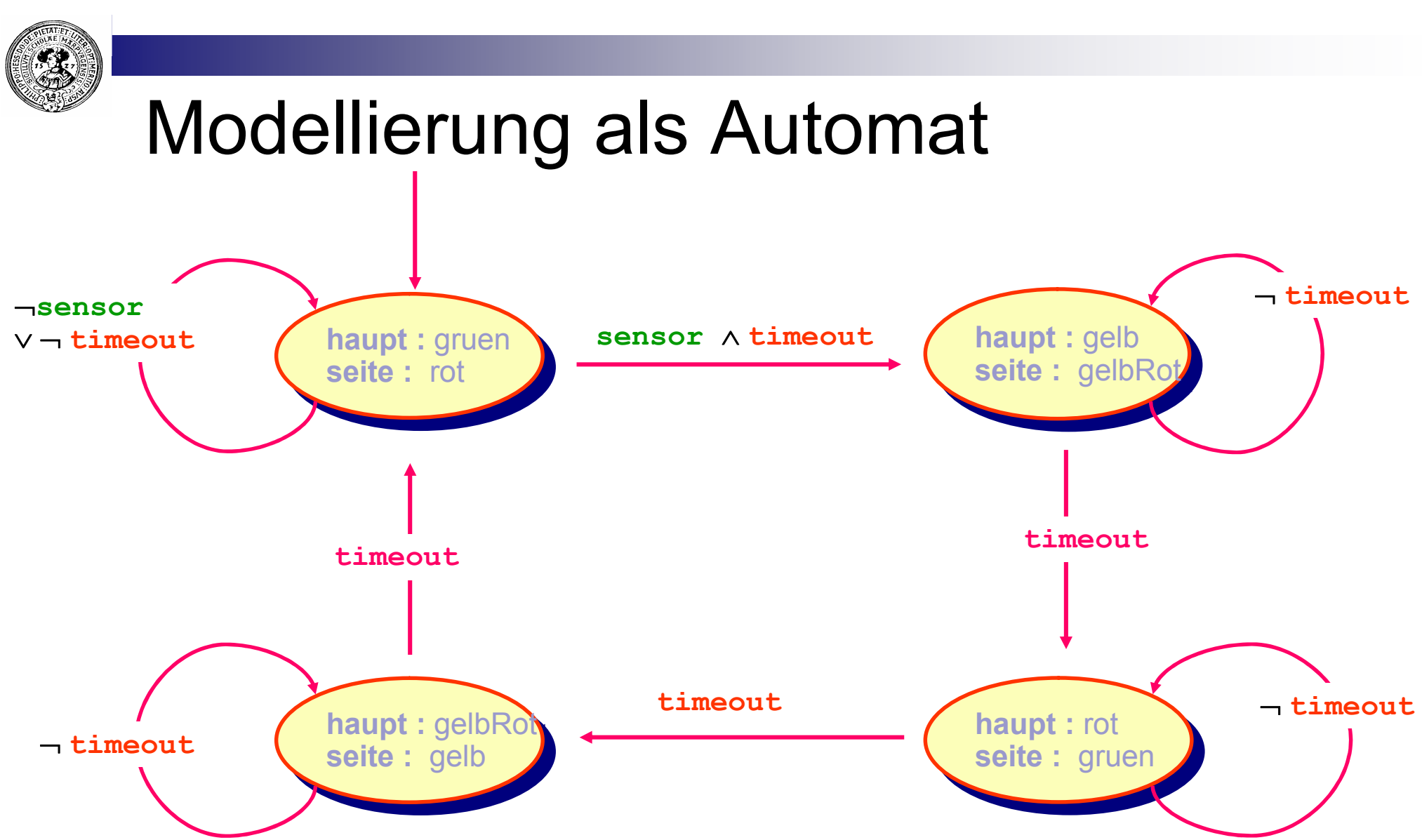


Steuerung durch ein C-Programm

```
typedef enum{rot,gruen,gelb,gelbRot} ampel;
extern bool sensor ;
extern bool timeout( );
extern void set_lights( );

ampel haupt, seite;
main(){
   haupt = gruen;
   seite = rot;
   while(1){ /* main loop */
     set_lights(haupt,seite);
     if (timeout()) {
        switch (haupt) {
          case rot : haupt = gelbRot ;
                     seite = gelb ;
                     break ;
```

```
          case gelbRot : haupt = gruen ;
                         seite = rot ;
                         break ;

          case gruen : if ( sensor ) {
                         haupt = gelb ;
                         seite = gelbRot;
                       } break ;

          case gelb : haupt = rot ;
                      seite = gruen ;
        } /*endswitch */
     } /* endif */
} /* endwhile */
}
```

Wie testen wir dieses Programm ? Können wir es verifizieren ?



Zu bestimmten Zeitpunkten kommt ein timeout-Signal.
Dann wird überprüft ob und wie die Ampeln geschaltet werden müssen.

Übersetzung in ein SMV-Programm

Beginn der Datei
Ampel_timeout.smv

„--“ : Kommentar bis Zeilenende

MODULE = Komponente

Variablendeklarationen:
IVAR : input variables
VAR : variables

Aufzählungstyp

ASSIGN :
Beschreibung der Dynamik

init(<*Variable*>) := <*expr*> ;
setzt Anfangswerte

```
-- Die einfache Ampelsteuerung in NuSMV

MODULE main  -- Die Hauptkomponente
IVAR -- Unabhängige Variable = Signale
 sensor  : boolean;
 timeout : boolean;

VAR -- Zustandsvariable
 haupt : {rot,gruen,gelb,gelbRot};
 seite : {rot,gruen,gelb,gelbRot};

ASSIGN  -- die Dynamik des Systems
-- Anfangswerte:
   init(haupt) := gruen;
   init(seite) := rot;
```

Fortsetzung der SMV-Modellierung

- next beschreibt neuen Wert der Variablen
- Fallunterscheidung: case - esac:
  - TRUE (= 1) : default-Fall
  - Fehlt dieser: Wert =1 !!!

```
-- Zustandsübergänge:
   next(haupt) := case
      timeout & haupt=rot              : gelbRot;
      timeout & haupt=gelbRot          : gruen;
      timeout & haupt=gruen & sensor   : gelb;
      timeout & haupt=gelb              : rot;
      TRUE                             : haupt;
   esac; -- endcase

   next(seite) := case
       timeout & haupt=rot              : gelb;
      timeout & haupt=gelbRot          : rot;
      timeout & haupt=gruen & sensor   : gelbRot;
      timeout & haupt=gelb              : gruen;
      TRUE                             : seite;
   esac; -- endcase
```

Temporale Eigenschaften

Temporale Eigenschaften drücken wir mit den "Modalitäten"

G	always	(immer)
F	sometimes	(irgendwann)

aus.

$G \neg ((\text{haupt} = \text{gruen}) \wedge (\text{seite} = \text{gruen}))$

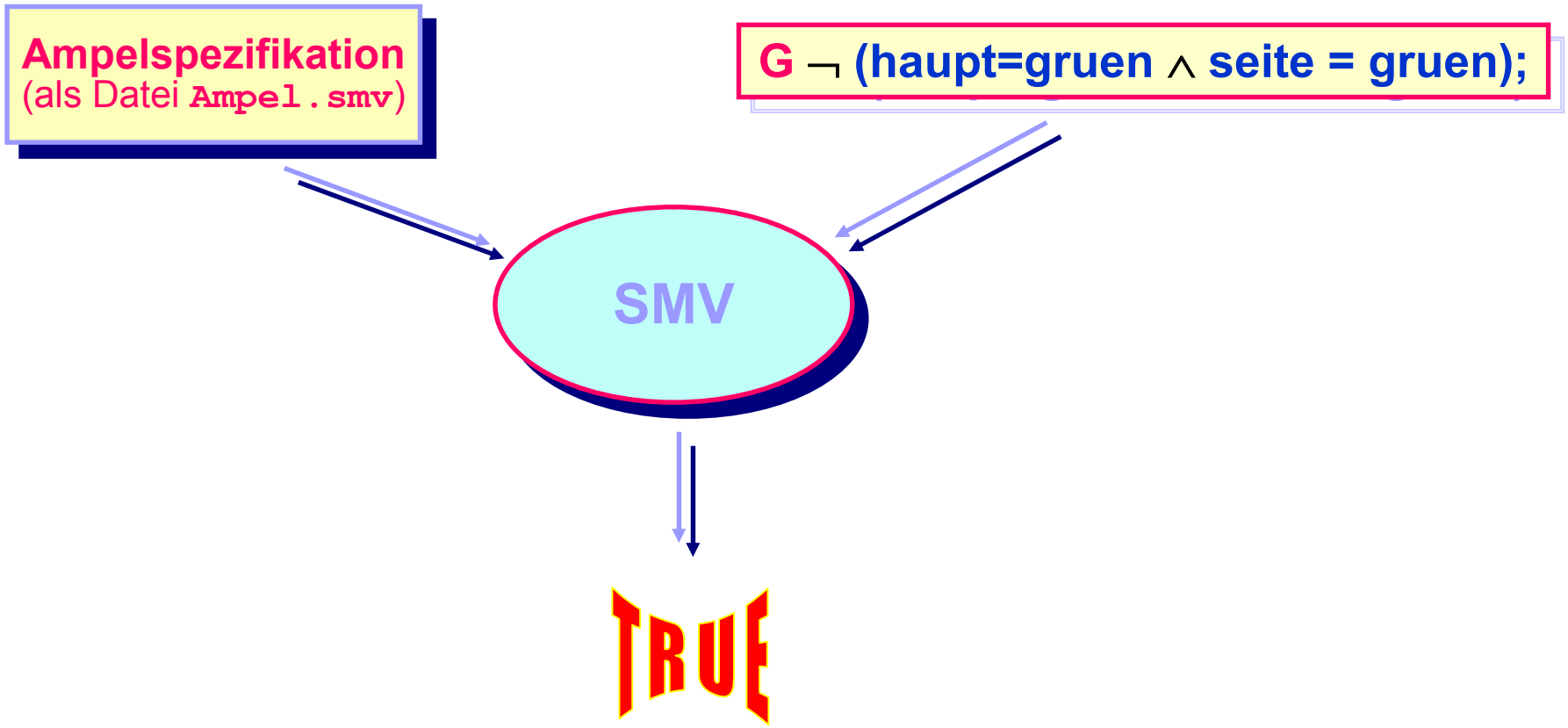
$G\ (\text{sensor} \Rightarrow F\ (\text{seite} = \text{gruen}))$

Sicherheit

Lebendigkeit

Statt G und F verwendet man häufig auch [ ] „box“ und ◊ „diamond“.

Rolle eines Model checkers



Spezifikation in NuSMV

- **LTLSPEC**:

  Spezifikation einer zeitlichen Aussage in LTL:

  Lineare Temporale Logik

- Zeitliche Operatoren:
  - **G** : Immer
  - **F** : Irgendwann

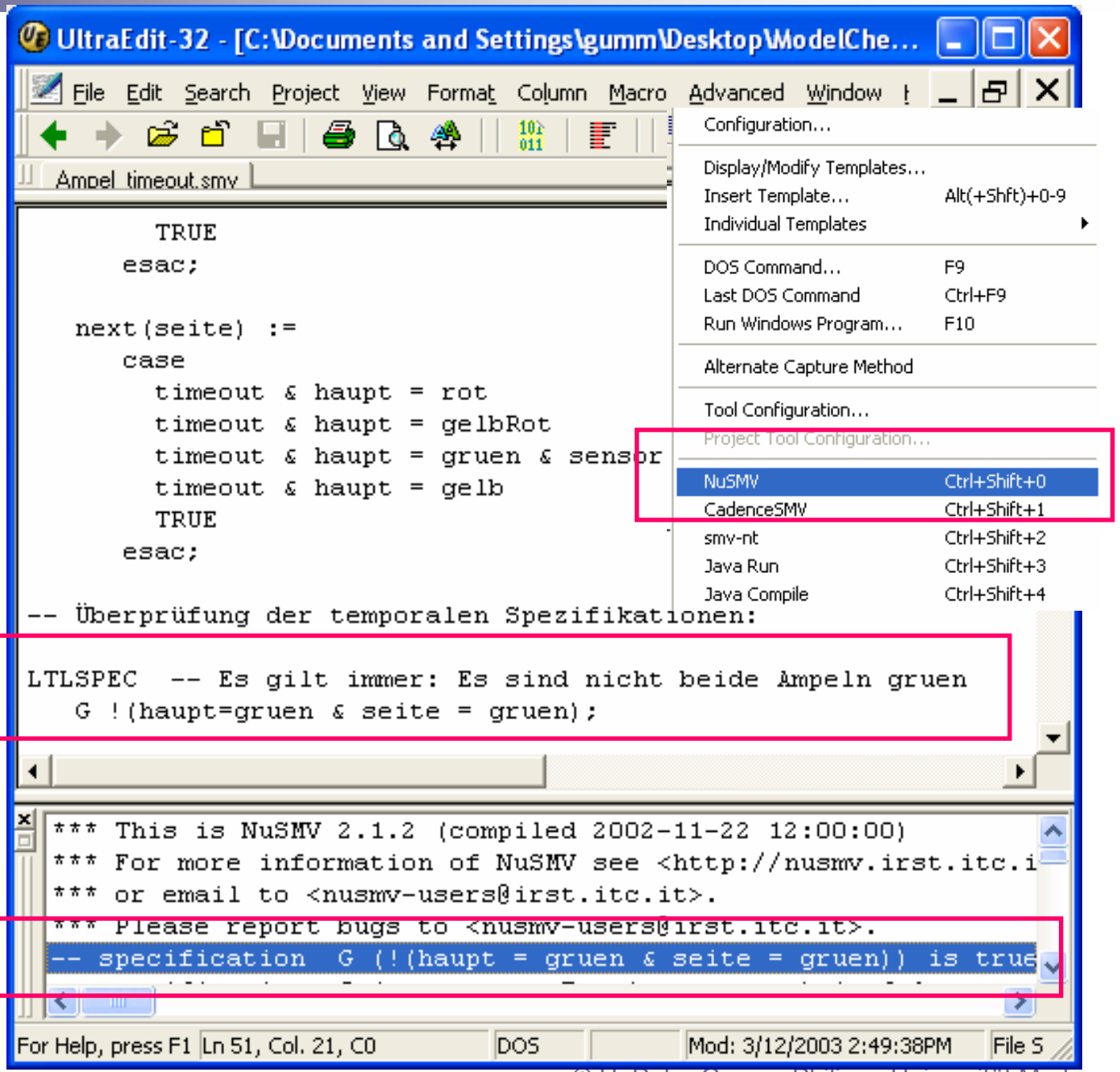
```
-- Nie beide Ampeln gleichzeitig auf gruen
LTLSPEC  -- Immer: Nicht beide Ampeln gruen
   G !(haupt=gruen & seite = gruen);
```

In NuSMV

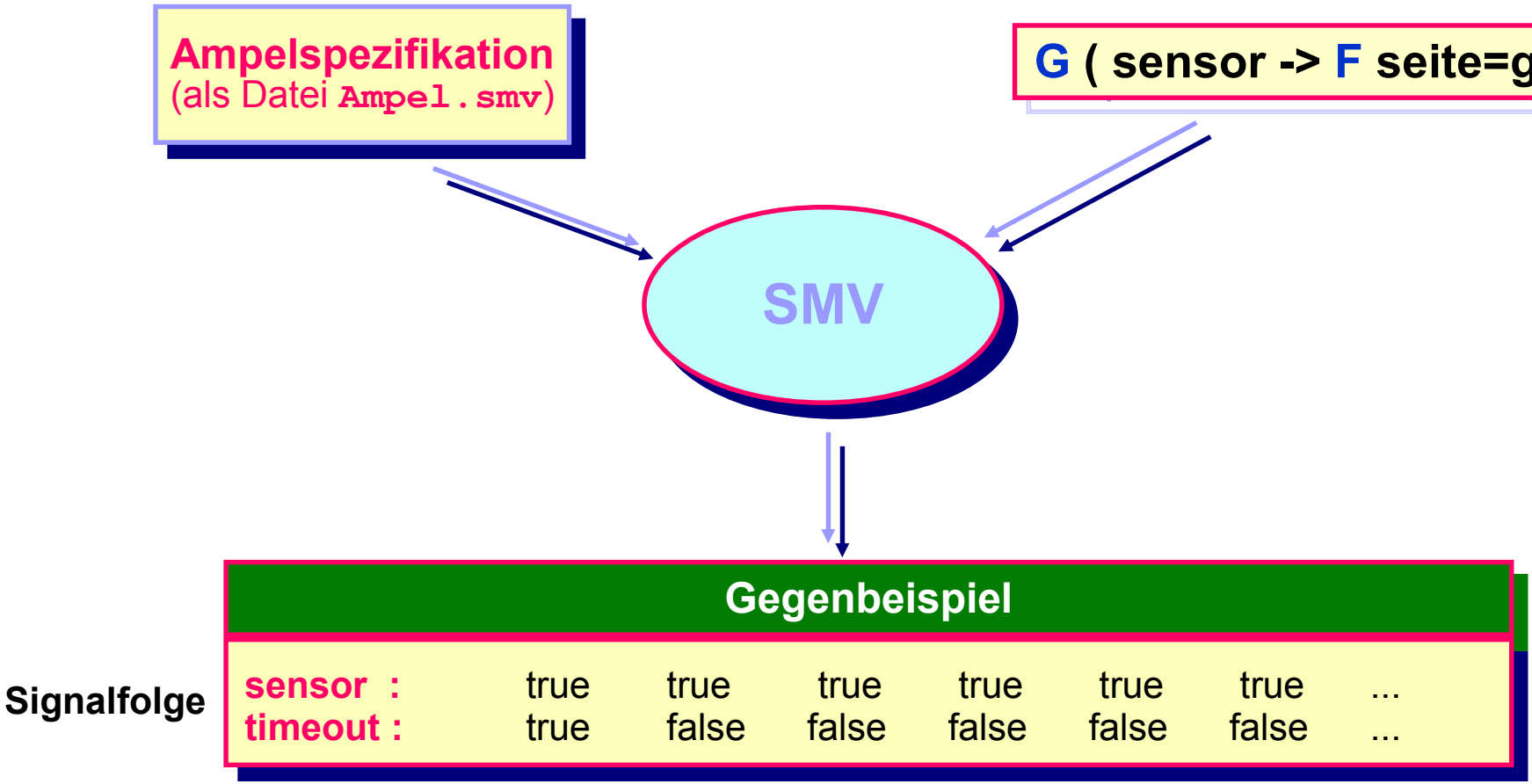
Aufruf aus UltraEdit:

Temporale Spezifikation:

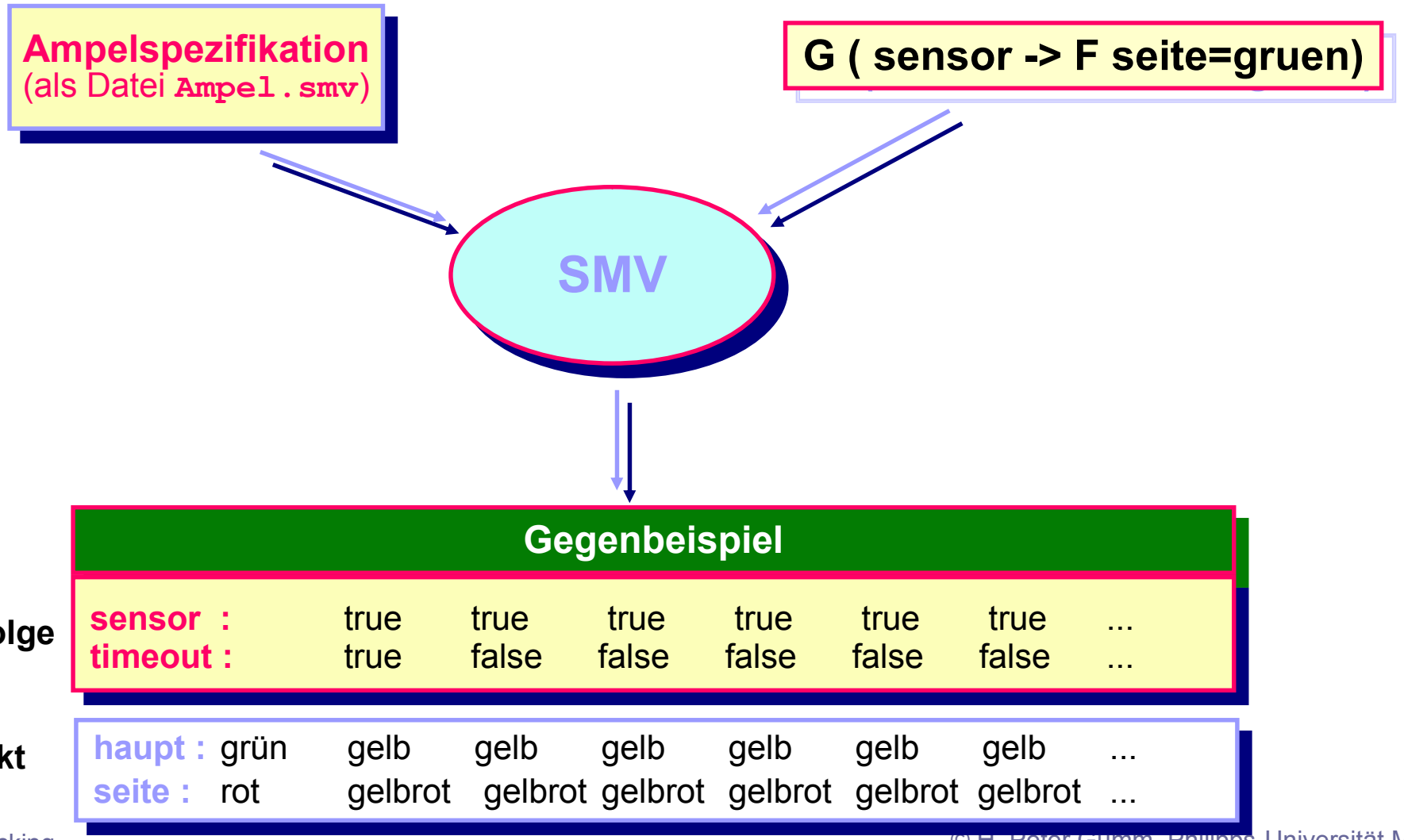
Ergebnis von NuSMV:



Rolle eines Model checkers



Rolle eines Model checkers



In NuSMV

```
-- Immer: Wenn der Sensor ein Auto meldet,
-- dann wird irgendwann die Ampel grün
LTLSPEC
   G (sensor -> F seite=gruen);
```

```
--  G ( sensor -> F seite = gruen) is false
-- as demonstrated by the execution sequence:
-> State 1.1 <-
    sensor = 1
    timeout = 1
    haupt = gruen
    seite = rot
-- loop starts here --
-> State 1.2 <-
    timeout = 0
    haupt = gelb
    seite = gelbRot
-> State 1.3 <-
```

Signalfolge

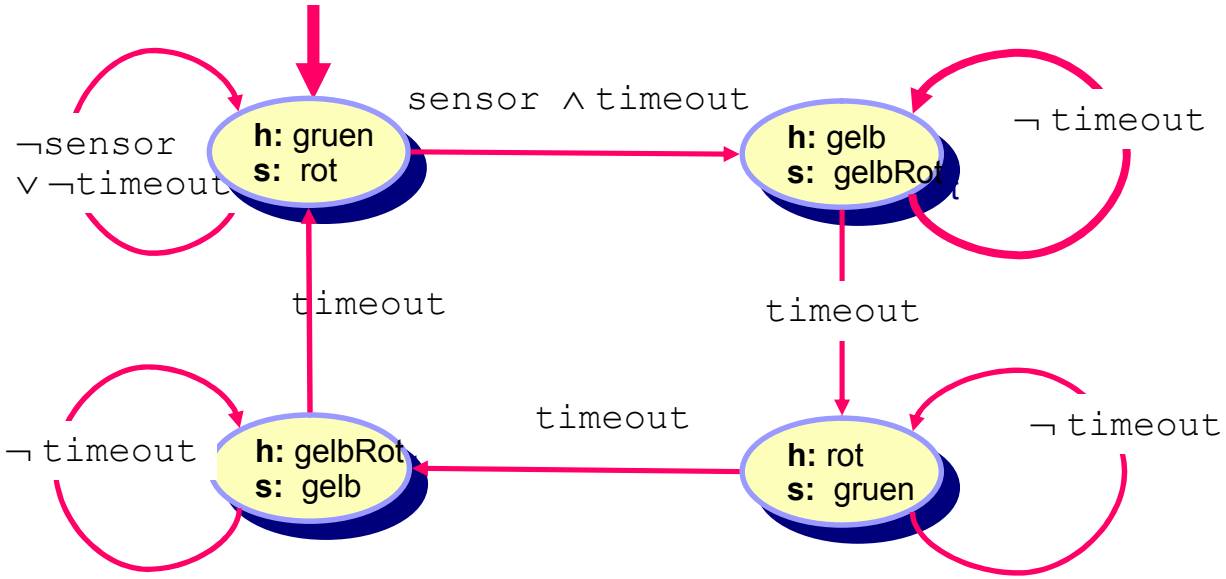
bewirkt

Gegenbeispiel

sensor :	true	true	true	...
timeout :	true	false	false	...
haupt :	grün	gelb	gelb	...
seite :	rot	gelbRot	gelbRot	

...

Analyse



$G\,(\,sensor \Rightarrow F\,(haupt = gruen\,))$

Gegenbeispiel

Input :	**sensor :**	true	true	true	true	true	true ...
	timeout :	true	false	false	false	false	false ...
bewirkt	**haupt :**	grün	gelb	gelb	gelb	gelb	gelb ...
	seite :	rot	gelbRot	gelbRot	gelbRot	gelbRot	gelbRot ...

Der nervöse Autofahrer

```
LTLSPEC --  Wir setzen voraus, dass immer wieder (= unendlich oft )
        --  ein timeout kommt
    G F timeout ->  G ( sensor -> F seite=gruen);
```

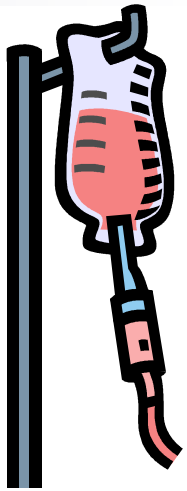
Gegenbeispiel von NuSMV:

```
...
-- loop starts here --

-> State 1.5 <-
     timeout = 0
     haupt = gruen
     seite = rot
-> State 1.6 <-
     sensor = 0
     timeout = 1
-> State 1.7 <-
     sensor = 1
     timeout = 0
```

Es handelt sich um einen hypernervösen Autofahrer, der immer vor- und zurückfährt.

Beim **timeout** ist er immer gerade nicht auf dem **sensor**.

Baldrian für den Fahrer



- Es gibt zwei Möglichkeiten, mit dem hypernervösen Autofahrer umzugehen
  - Wir garantieren nur dem ruhigen Autofahrer, dass er auch irgendwann grün bekommt
    - `G F timeout -> G (sensor & „Auto wartet“ -> F seite=grün)`
  - Wir modifizieren die Ampelsteuerung, so dass auch hypernervöse Autofahrer mal grün bekommen
    - Wir speichern das Sensorsignal in einer Variablen „request“ und löschen es erst bei „grün“ wieder

Temporaler Operator: Until

G(sensor ⇒
(sensor **until** sensor & timeout))

Zusätzliche Annahme

Definition

p **until** q : ⇔ p wahr **bis** q eintritt

Beispiel

p :	true	true	false	false	true	false	false	...
q :	false	false	true	true	true	false	true	...

p wird von q abgelöst

Im Beispiel gilt : **p until q,**
aber : **¬ G (p until q)**

Versuch einer Verbesserung

```
LTLSPEC -- sensor wartet auf timeout
      G( sensor -> sensor U sensor & timeout)
         -> G(sensor -> F (seite = gruen))
```

Ergebnis von NuSMV:

```
-- specification
-- G( sensor -> sensor U sensor & timeout) -> G(sensor -> F (seite=gruen))
-- is false as demonstrated by the following execution sequence
--
   ....
```

Die Analyse ergibt ...

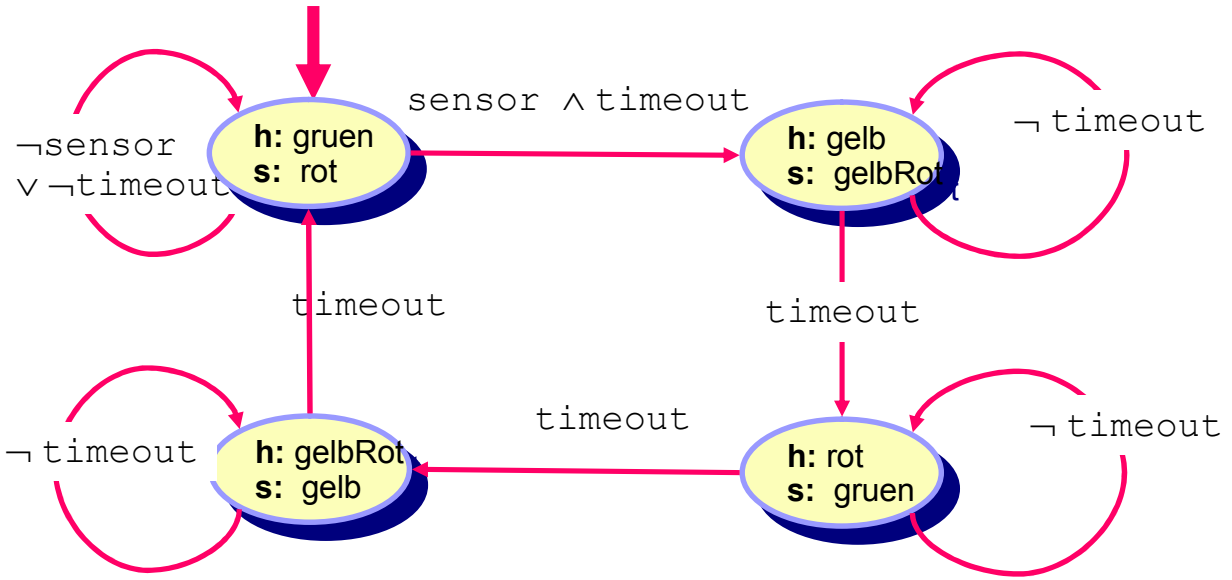
sensor müsste nach dem `timeout` noch einen Takt warten ...

Analyse

```
G( sensor -> sensor U sensor  & timeout)
          -> G(sensor -> F (seite = gruen))
```

Gegenbeispiel von NuSMV

Input :								
sensor :	true	true	true	true	true	false	false ...	
timeout :	true	true	true	true	true	true	true ...	
bewirkt								
haupt :	grün	gelb	rot	gelbRo	grün	grün	grün ...	
seite :	rot	gelbRot	gruen	gelb	rot	rot	rot ...	



Einige Autos kommen. seite wird grün, dann gelb, danach kommt noch ein Auto. Es löst den sensor aus und setzt sofort wieder zurück.

Zunächst wird die Hauptampel grün. Wenn jetzt der sensor zurückgesetzt wird und bleibt, so bleibt das System im Zustand (h:grün, s: rot)

Temporaler Operator X - next time

Definition

X p ist jetzt wahr : ⇔ p ist zum nächsten Zeitpunkt wahr

Beispiel

p :	false	true	false	false	true	false	false	...

Hier ist X p wahr

Hier ist X p falsch

Zusätzliche Annahme

```
LTLSPEC -- sensor wartet noch 1 Takt nach timeout
      G( sensor -> sensor U (timeout & X sensor ))
          -> G(sensor -> F (seite = gruen))
```

Versuch einer Verbesserung

```
LTLSPEC -- sensor wartet auf timeout
      G( sensor -> sensor U timeout & X sensor  )
         -> G(sensor -> F (seite = gruen))
```

Ergebnis von NuSMV:

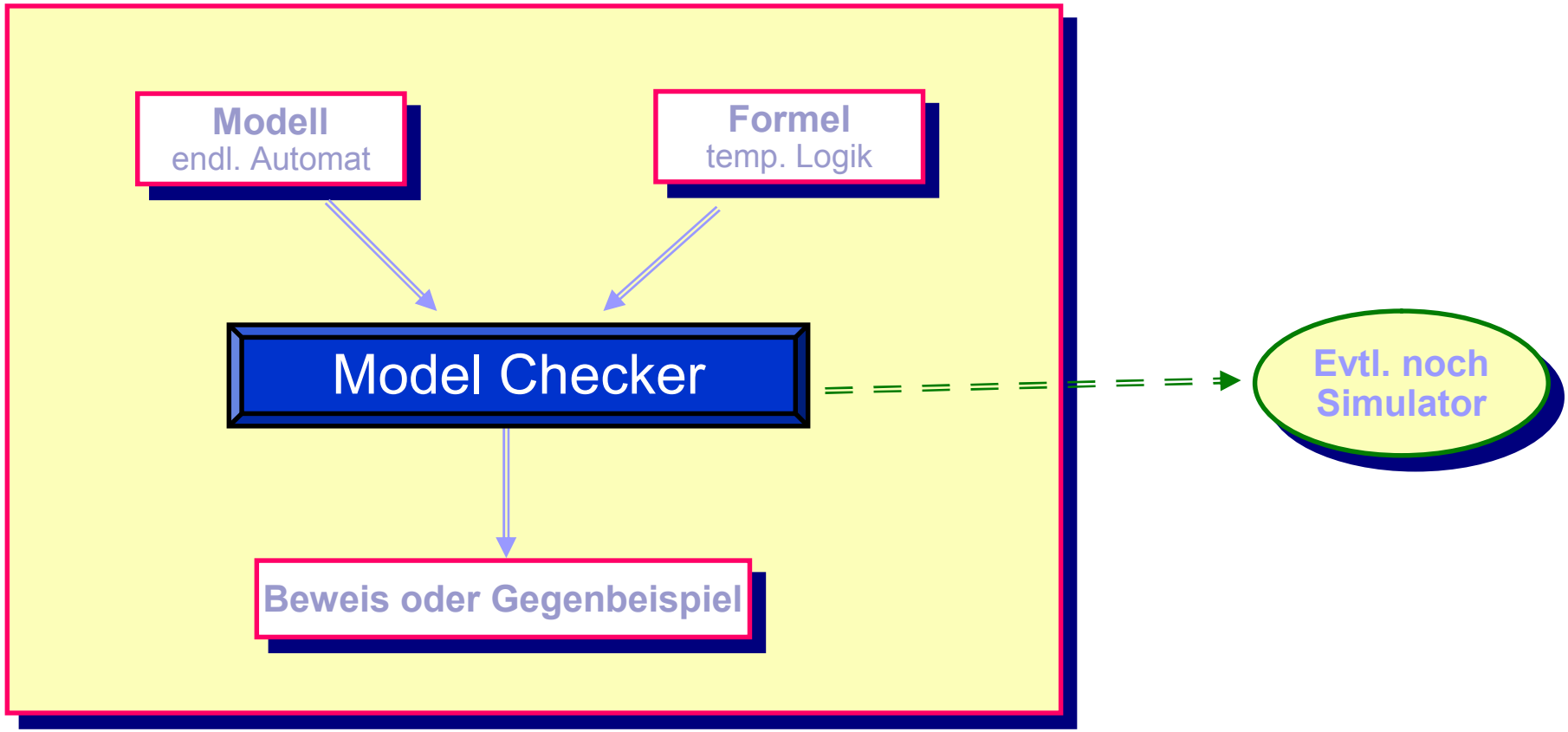
```
-- specification ( G (sensor -> ((sensor U timeout) &  X sensor))
      ->  G (sensor ->  F seite = gruen)) is true
```

C-Programm für Ampel mit timer

```
typedef enum {rot,gruen,gelb,gelbRot }
   ampel;
extern bool sensor ;
extern void setTimer( );
extern void set_lights( );
ampel haupt, seite ;

main(){
   haupt = gruen;
   seite = rot;
   setTimer(10);
   while(1){ /* main loop */
        set_lights(haupt,seite);
        if (timeout()) {
                switch (haupt) {
                case rot :
                        haupt = gelbRot ;
                        seite = gelb ;
                        setTimer(1);
                        break ;
```

```
                case gelbRot :
                        haupt = gruen ;
                        seite = rot ;
                        setTimer(10);
                        break ;
                case gruen :
                        if (sensor ) {
                                haupt = gelb ;
                                seite = gelbRot;
                        }
                        setTimer(1);
                        break ;
                case gelb :
                        haupt = rot ;
                        seite = gruen ;
                        setTimer(6);
                } /*endswitch */
        } /* endif */
} /* endwhile */  }
```

Zusammenfassung: Model Checking



Wichtig: Beweis, bzw. Gegenbeispiel entstehen „auf Knopfdruck"

Hausaufgaben

- 1. Installieren Sie sich NuSMV und bringen Sie die Beispiele der Vorlesung zum Laufen
- 2. Modifizieren Sie das Ampel-Beispiel, indem Sie eine Variable „request" einführen
  - Es soll neben der Sicherheitseigenschaft, u.a. auch gelten (mit served := seite=gruen) :
    - (G F timeout) -> G (sensor -> F served)
- 3. Führen Sie einen timer ein, der das timeout generiert.
  - Jede neue Ampelphase stellt den timer, um zu garantieren, dass die Phase eine bestimmte Zeitdauer erhalten bleibt.
    - Die Grünphase mind. 10 ticks, die Rotphase mindestens 6 ticks, gelb und gelbRot je 1 tick.
  - Der timer wird in jedem Schritt erniedrigt. Definieren Sie „timeout" als „timer=0".
  - Es sollen u.a. gelten:
    - Die Sicherheitseigenschaft,
    - G F timeout ,
    - G (sensor -> F served).
- 4. Ersetzen Sie request durch eine numerische Variable. Ziel ist es, einfacher bestimmen zu können, wie lange ein Auto schon wartet.
  - Versuchen Sie folgende – oder ähnliche Eigenschaften zu garantieren
    - G (request < 20) -- Ein Auto wartet nicht länger als 20 ticks
    - G (request >= 15 -> served | X served) -- Wartet es 15 ticks, so bekommt es spätestens zum nächsten tick gruen.
    - G (request =0 -> (request=0 U sensor)
    - G (request >0 -> (request >0 U served)