

Das SMV-System

Jens Lehmann

1. Juli 2003

1. Einleitung

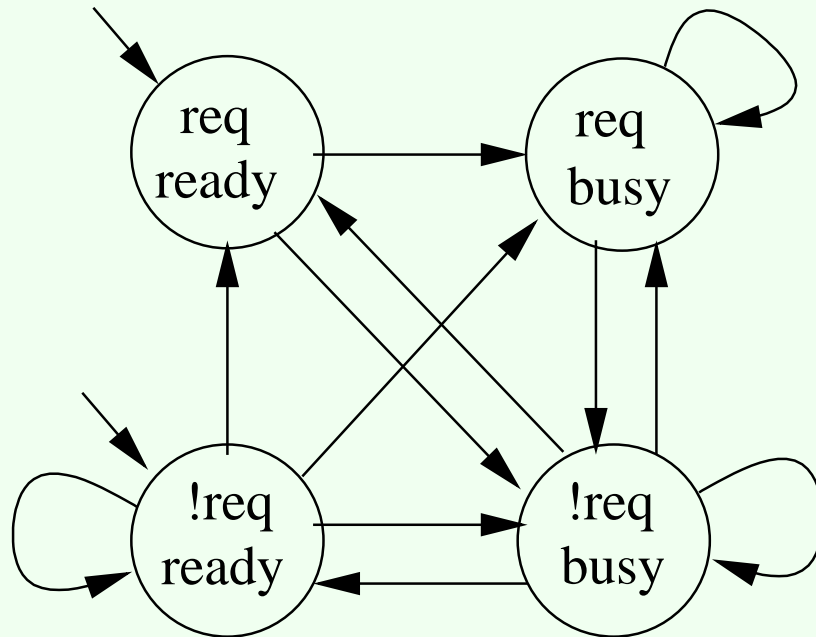
- frei erhältlicher Modellüberprüfer (siehe z.B. [2] und [3]).
- SMV überprüft in realistischer Zeit, ob ein Modell eine Spezifikation erfüllt
- Vorteil gegenüber Tests und Simulationen: für jedes mögliche Verhalten des Systems kann sichergestellt werden, dass es die Bedingungen erfüllt
- es gibt unterschiedliche Versionen des SMV-Systems, die sich im Funktionsumfang wesentlich unterscheiden

2. Aufbau von SMV-Programmen

2.1. Grundlagen

Listing 2.1 (einfaches SMV-Programm)

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



Modell, welches dem SMV-Programm entspricht

2.2. Variablen und Datentypen

- Variablen deklarieren: `<name> : <typ>;`
- Beispiel 1: `sub_range_var : 2..19;`
- Beispiel 2: `col : {green, blue};`
- Arrays: `<name>: array <start>..<ende> of <typ>`
- Beispiel: `array_var: 0..7 of boolean`
- im SMV-System entspricht `1` \rightarrow wahr und `0` \rightarrow falsch

2.3. ASSIGN-Teil

- Initialisieren von Variabeln: `init(<name>) := <value>;`
- potentiellen „nächsten“ Zustand der Variable festlegen:
`next(<name>) := <value>;`

Beispiel 2.2 (case-Statement)

```
next(bit0) :=
  case
    bit1           : 0;
    state = ready  : 0;
    1               : 1;
  esac;
```

2.4. DEFINE-Teil

- weist einem Identifikator den Wert eines Ausdrucks zu
- vergrößert den Zustandsraum nicht

Beispiel 2.3 (**DEFINE**)

DEFINE

```
foo := int0 + int1 + int2;
```

2.5. Spezifikation

- Menge von CTL-Formeln
- muss bei jeder möglichen Programmausführung wahr sein, damit sie erfüllt wird
- falls nicht erfüllt, wird ein Gegenbeispiel ausgegeben

2.6. Module

- helfen den Programmcode in kleine, übersichtliche Teile zu zerlegen
- Programm hat mindestens ein Modul (`main`)
- Programm wird (im Normalfall) bei `main` gestartet
- Instanziierung von Modulen im `VAR`-Teil
- Beispiel: `bit0 : counter_cell(1);`
- Zugriff auf Modulvariablen über Punktoperator z.B. `bit0.value`

Beispiel 2.4 (Modul eines Zählers mit Überlauf)

```
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;
```

2.7. synchrone und asynchrone Komposition

- Normalfall: synchrone SMV-Module mit globaler Uhr
- in Praxis häufig asynchrone Prozesse relevant
- SMV: Schlüsselwort `process` für asynchrone Module wird bei Instanziierung angegeben
- in jedem Programmschritt wird nichtdeterministisch ein Prozess zur Ausführung gewählt

3. Fairness

Listing 3.1 (Inverter)

```
MODULE main
VAR
  gate1: process inverter;
  gate2: process inverter;
  gate3: process inverter;
SPEC
  (AG AF gate1.output) & (AG AF !gate1.output)

MODULE inverter
VAR
  output : boolean;
ASSIGN
  init(output) := 0;
  next(output) := !output;
FAIRNESS running
```

Programmablauf ohne `process`:

- `output`-Werte werden mit 0 initialisiert
- im nächsten Schritt parallel alle Modulinstanzen abgearbeitet
- `gate1`, `gate2` und `gate3` auf 1 invertiert
- in jedem Schritt Invertierung aller Module
- Spezifikation erfüllt

Programmablauf mit `process`:

- `output`-Werte werden mit 0 initialisiert
- im nächsten Schritt wird nichtdeterministisch ein Modul zur Ausführung ausgewählt
- Problemfall: `gate1` wird nie zur Ausführung ausgewählt
- Folge: Spezifikation ist nicht erfüllt
- Lösung: Fairness-Klauseln

- SMV bietet Möglichkeit anzugeben, dass Prozess-Scheduler fair ist
- `FAIRNESS running` in einem Modul führt dazu, dass nur Pfade in denen die Modulinstanz unendlich oft ausgeführt wird, vom SMV-System berücksichtigt werden
- Folge: Prozess kann nicht „verhungern“
- Spezifikation aus Beispiel 3.1 ist erfüllt

- FAIRNESS-Klausel kann beliebige Formeln enthalten
- FAIRNESS ϕ : SMV-System ignoriert Pfade, auf denen ϕ nicht unendlich oft wahr ist

Beispiel 3.2 (Fairness-Klauseln)

- (1) FAIRNESS a=foo
- (2) FAIRNESS a=start \rightarrow AF a=end

4. Beispiele

4.1. gegenseitiger Ausschluss

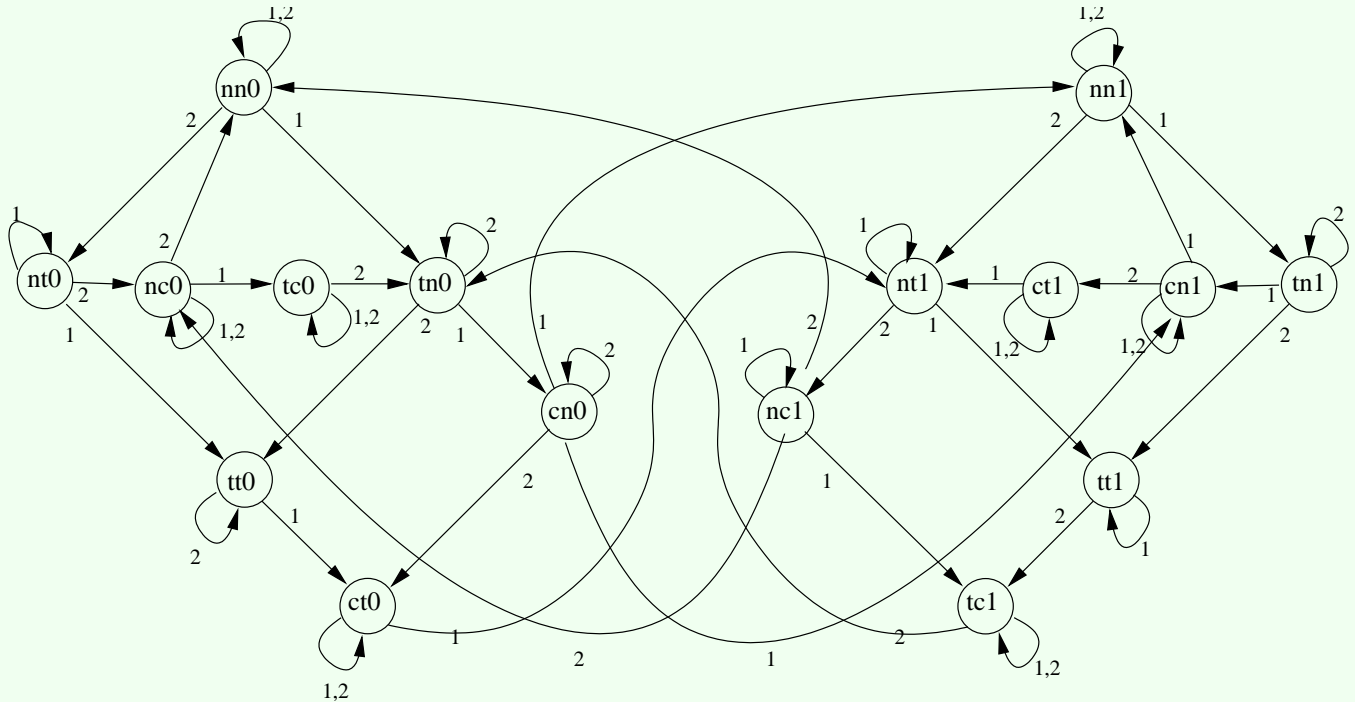
4.1.1. Erklärung

- in modernen Betriebssystemen, müssen sich oft mehrere Prozesse eine Resource teilen
- Beispiel: eine Datei kann nicht gleichzeitig von zwei Prozessen geschrieben werden
- solche Quellcodeteile werden als kritische Abschnitte markiert
- gesucht ist ein Verfahren, was die Verteilung einer Resource an zwei Prozesse nach bestimmten Kriterien gewährleistet

Anforderungen, an ein Verfahren zur Zuteilung der Resource:

- *Sicherheit*: Nur maximal ein Prozess befindet sich zu einem Zeitpunkt in seinem kritischen Abschnitt.
- *Lebendigkeit*: Wenn ein Prozess einen kritischen Abschnitt betreten möchte, wird ihm das (in der Zukunft) auch gewährt.
- *kein Blockieren*: Ein Prozess kann immer anfragen seinen kritischen Abschnitt zu betreten.
- *keine strenge Abarbeitungsreihenfolge*: Prozesse sollen ihre kritischen Abschnitte nicht in einer strengen Abarbeitungsreihenfolge betreten. Es soll z.B. möglich sein, dass ein Prozess zweimal hintereinander seinen kritischen Abschnitt betreten kann.

4.1.2. Lösungsansatz



Transitionssystem gegenseitiger Ausschluss

4.1.3. Verifizierung

- graphische Transitionssysteme lassen sich in SMV-Programme umwandeln
- Erstellung eines Prozessmoduls
- Prozess kann Zustände n , t und c annehmen
- analog zum Transitionssystem gibt es eine $turn$ -Variable
- Anforderungen werden als CTL-Formeln formuliert und bilden die Spezifikation

Listing 4.1 (gegenseitiger Ausschluss - main-Modul)

```
MODULE main
  VAR
    pr1 : process prc(pr2.st, turn, 0);
    pr2 : process prc(pr1.st, turn, 1);
    turn : boolean;
  ASSIGN
    init(turn) := 0;
  -- Sicherheit
  SPEC AG!((pr1.st = c) & (pr2.st = c))
  -- Lebendigkeit
  SPEC AG((pr1.st = t) -> AF (pr1.st = c))
  -- kein Blockieren
  SPEC AG((pr2.st = t) -> AF (pr2.st = c))
  -- beliebige Reihenfolge
  SPEC EF(pr1.st = c & E[pr1.st = c U
    (!pr1.st = c & E[! pr2.st = c U pr1.st = c ]))
```

Modellierung der Zustandsübergänge:

- Ist der aktuelle Prozess im Zustand n , dann bleibt das so oder der Prozess fragt an den kritischen Abschnitt zu betreten.
- Möchte der aktuelle Prozess in den kritischen Abschnitt und der andere Prozess ist nicht kritisch, dann darf er den kritischen Abschnitt betreten.
- Möchten beide Prozesse in den kritischen Abschnitt und ist der aktuelle Prozess „am Zug“, dann darf er den kritischen Abschnitt betreten.
- Ist der aktuelle Prozess im kritischen Abschnitt, dann bleibt er da oder verlässt ihn.
- In allen anderen Fällen behält der Prozess seinen Zustand.

Listing 4.2 (gegenseitiger Ausschluss - prc-Modul)

```
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-Kriterien:

- es soll ausgeschlossen werden, dass ein Prozess „verhungert“
- ein Prozess darf nicht für immer in seinem kritischen Abschnitt bleiben

Listing 4.3 (gegenseitiger Ausschluss - **FAIRNESS**-Klauseln)

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

4.2. Loyd-Puzzle

4.2.1. Erklärung des Spiels

1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

- 4x4-Matrix mit einem freien Feld
- 15 Plättchen sind mit Zahlen beschriftet
- über das freie Feld lassen sich Plättchen verschieben

4.2.2. Die Legende

Sam Loyd war ein amerikanischer Puzzle-Erfinder in der zweiten Hälfte des 19. Jahrhunderts. 1878 brachte er das eben erklärte Puzzle unter dem Name „Loyd’s Fifteen“ bzw. „Loyd’s Puzzle“ heraus. Loyd bot jedem \$1000 an, der eine Lösung des Puzzles präsentieren konnte. Eine Begeisterung für das Puzzle ging rund um die Welt, die bis zu Rubik’s Würfel 1980 nicht übertroffen wurde. Jedesmal schien die Lösung des Puzzles nah zu sein, aber niemand schaffte es eine Lösung zu präsentieren.

4.2.3. SMV-Programm für ein kleineres Problem

Vorerst soll kleineres Problem gelöst werden:

	1	2
3	4	5
6	7	8

8	7	6
5	4	3
2	1	

Strategie:

- Modul `Field` für Plättchen und Modul `FreiesField` für das freie Feld
- Plättchen werden mit Startposition initialisiert
- Position besteht aus einer vertikalen und einer horizontalen Komponente

Listing 4.4 (Initialisierung)

VAR

```
FF : FreiesFeld(1, 1, {r, u}); -- freies Feld
F1 : Feld(1, 2, FF); -- Ziffer 1 liegt auf Feld(1,2)
F2 : Feld(1, 3, FF); -- Ziffer 2 liegt auf Feld(1,3)
F3 : Feld(2, 1, FF); -- usw.
F4 : Feld(2, 2, FF);
F5 : Feld(2, 3, FF);
F6 : Feld(3, 1, FF);
F7 : Feld(3, 2, FF);
F8 : Feld(3, 3, FF);
```

Wie kann das SMV-System das Puzzle überhaupt lösen?

- Ausnutzung der Tatsache, dass bei einer nicht erfüllten Spezifikation immer ein Gegenbeispiel angegeben wird, welches die Spezifikation verletzt
- wir fordern, dass das Puzzle nicht in die gegebene Endkonfiguration überführt werden kann
- das ausgegebene Gegenbeispiel ist die Lösung des Problems

Listing 4.5 (Spezifikation)

SPEC

```
!(EF(      (F1.v = 3 & F1.h = 2)
           & (F2.v = 3 & F2.h = 1)
           & (F3.v = 2 & F3.h = 3)
           & (F4.v = 2 & F4.h = 2)
           & (F5.v = 2 & F5.h = 1)
           & (F6.v = 1 & F6.h = 3)
           & (F7.v = 1 & F7.h = 2)
           & (F8.v = 1 & F8.h = 1) ))
```

Implementierung:

- Bewegung der Plättchen wird im Modul `FreiesFeld` realisiert
- in jedem Zug „bewegt“ sich das freie Feld in eine bestimmte Richtung
- erlaubte Richtungen abhängig von der aktuellen Position des freien Feldes
- andere Plättchen bewegen sich entsprechend dem freien Feld
- bewegt sich das freie Feld auf die Position eines Plättchens, dann nimmt das Plättchen im nächsten Zug die aktuelle Position des freien Feldes ein

Listing 4.6 (Feld-Modul)

```
MODULE Feld(vertical, horizontal, FF)
VAR
  v : 1..3; -- vertikal
  h : 1..3; -- horizontal
ASSIGN
  init(v) := vertical;
  init(h) := horizontal;
  next(v) :=
    case
      (FF.richtung=u & v=FF.v + 1 & h=FF.h)
      | (FF.richtung=o & v=FF.v - 1 & h=FF.h) : FF.v;
      1                                         : v;
    esac;
  next(h) :=
    case
      (FF.richtung=l & v=FF.v & h=FF.h - 1)
      | (FF.richtung=r & v=FF.v & h=FF.h + 1) : FF.h;
      1                                         : h;
    esac;
```

Listing 4.7 (FreiesFeld-Modul 1)

```
MODULE FreiesFeld(vertikal, horizontal, anfangsrichtung)
VAR
  richtung : {l, r, u, o};
  v : 1..3;
  h : 1..3;
ASSIGN
  init(richtung) := anfangsrichtung;
  next(richtung) :=
    case
      (v = 1 & h = 1) : {u, r};
      (v = 1 & h = 2) : {u, r, l};
      (v = 1 & h = 3) : {u, l};
      (v = 2 & h = 1) : {o, u, r};
      (v = 2 & h = 2) : {o, u, r, l};
      (v = 2 & h = 3) : {o, u, l};
      (v = 3 & h = 1) : {o, r};
      (v = 3 & h = 2) : {o, l, r};
      (v = 3 & h = 3) : {o, l};
    esac;
```

Listing 4.8 (FreiesFeld-Modul 2)

```
init(v) := vertikal;
next(v) :=
  case
    (richtung = u & v < 3) : v + 1;
    (richtung = o & v > 1) : v - 1;
    1                        : v;
  esac;
init(h) := horizontal;
next(h) :=
  case
    (richtung = r & h < 3) : h + 1;
    (richtung = l & h > 1) : h - 1;
    1                        : h;
  esac;
```

4.2.4. Lösung des Loyd-Puzzle-Problems

- analog zu dem kleineren Puzzle lässt sich auch das Loyd-Puzzle-Problem beschreiben
- durch State-Explosion-Problem terminiert das Programm auf „normalen“ Rechnern in dieser Implementierungsvariante nicht, da es zuviel Zeit und Ressourcen beansprucht
- würde es terminieren, so gibt es eine Überraschung: die Spezifikation ist wahr und das Loyd-Puzzle somit unlösbar, was erklärt warum über Jahrzehnte hinweg viele Leute an diesem Problem gescheitert sind

Ende

Literatur

- [1] [HuthRyan00] *Logic in Computer Science*, M. Huth & M. Ryan, Cambridge University Press, 2000
- [2] <http://www-2.cs.cmu.edu/~modelcheck/smv.html>, Download des SMV-Systems (Original Carnegie-Mellon)
- [3] <http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/>, Download des Cadence-Berkeley-SMV-Systems
- [4] <http://www-cad.eecs.berkeley.edu/~kenmcmil/tutorial.ps>, SMV-Tutorial, K. L. McMillan, Cadence Berkeley Labs, 1999
- [5] SMV-Manual, K. L. McMillan, 1992-2001 (liegt [2] bei)
- [6] <http://www-cad.eecs.berkeley.edu/~kenmcmil/language.ps>, SMV-Sprachbeschreibung, K. L. McMillan, Cadence Berkeley Labs, 1999