

Das SMV-System

Jens Lehmann

1. Juli 2003

Diese Ausarbeitung ist Teil des Proseminars „Modale Logik“ der Fakultät Informatik der TU Dresden im Sommersemester 2003. Es wird das SMV-System als Mittel zur Verifizierung von Eigenschaften von Hard- bzw. Softwaresystemen vorgestellt.

Inhaltsverzeichnis

1	Einleitung	3
2	Aufbau von SMV-Programmen	4
2.1	Grundlagen	4
2.2	Variablen und Datentypen	4
2.3	ASSIGN-Teil	5
2.4	DEFINE-Teil	6
2.5	Spezifikation	6
2.6	Module	6
2.7	synchrone und asynchrone Komposition	7
3	Fairness	8
4	Beispiele	10
4.1	gegenseitiger Ausschluss	10
4.1.1	Erklärung	10
4.1.2	Lösungsansatz	10
4.1.3	Verifizierung	12
4.2	Loyd-Puzzle	14
4.2.1	Erklärung des Spiels	14
4.2.2	Die Legende	14
4.2.3	SMV-Programm für ein kleineres Problem	14
4.2.4	Lösung des Loyd-Puzzle-Problems	17

1 Einleitung

Das SMV-System ist ein frei erhältlicher Modellüberprüfer. SMV steht für „Symbolic Model Verifier“. Es ist in der Programmiersprache C geschrieben und frei im Internet erhältlich (siehe z.B. [2] und [3]). Es gibt verschiedene Implementierungen des SMV-Systems, die sich erheblich unterscheiden. Ich habe die Carnegie-Mellon-Implementierung und die darauf basierende, aber umfangreichere Cadence-Berkeley-Version untersucht. Die hier aufgeführten Beispiele sollten in beiden Versionen lauffähig sein.

Mit Hilfe von SMV lässt sich in realistischer Zeit überprüfen, ob ein System eine Spezifikation erfüllt. Das System wird dabei durch ein Programm beschrieben. Die Spezifikation besteht aus einer Menge von Eigenschaften, die ebenfalls im Programm verankert sind.

Die Erfüllung der Spezifikation bedeutet, dass bei jedem theoretisch möglichen Verhalten das System die gewünschten Eigenschaften hat. Das ist ein Vorteil gegenüber Tests und Simulationen, die nur bei einem bestimmten Systemverhalten ermitteln, ob ein System die Spezifikation erfüllt.

Eingesetzt wird SMV bzw. weiterentwickelte Varianten dieses Systems zur Verifizierung von Hard- und Software.

2 Aufbau von SMV-Programmen

In den nächsten Kapiteln wird ein (keinesfalls vollständiger) Überblick über den Aufbau von SMV-Programmen gegeben.

2.1 Grundlagen

Ein SMV-Programm beschreibt ein Modell und die geforderten Spezifikationen. Ein Programm wird beim sogenannten main-Modul gestartet, so wie es ähnlich auch in Java und C der Fall ist. Damit man einen ersten Eindruck gewinnt, ein einfaches Beispiel:

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)
```

Im oberen Teil ist der Name des Moduls. Im VAR-Teil wird den im Programm benutzten Variablen ein Typ zugewiesen. In diesem Fall kann die Variable `request` boolesche Werte annehmen, während die Variable `state` die Werte `ready` und `busy` annehmen kann. Im ASSIGN-Teil werden Variablen initialisiert bzw. Variablen neue Werte zugewiesen. Der SPEC-Teil beschreibt die Spezifikation. In den nächsten Kapiteln wird genauer auf den Aufbau eines SMV-Programms eingegangen.

2.2 Variablen und Datentypen

Eine Variablenzuweisung hat die Form

```
<name> : <typ>;
```

wobei `<name>` der Name der Variable und `<typ>` der Variablentyp ist. Der einfachste Variablentyp ist `boolean`, d.h. die Variable kann nur die Werte 0 oder 1 annehmen. Weiterhin gibt es den Aufzählungstyp, bei dem in geschweiften Klammern alle möglichen Werte in einer kommagetrennten Liste stehen (siehe Variable `state` in Beispiel 2.1). Es ist auch möglich einer Variable einen bestimmten Bereich von ganzen Zahlen zuzuweisen. Zwischen kleinster und größter Zahl des Bereiches steht dabei `..`:

```
sub_range_var : 2..19;
```

Mit folgender Syntax lassen sich auch Arrays anlegen:

```
<name>: array <start>..<ende> of <typ>
```

Zum Beispiel würde `array_var: 0..7 of boolean` einen Array anlegen, dessen Index von 0 bis 7 läuft und dessen Werte vom Typ `boolean` sind. An der Stelle 0 hat der Array den Wert `array_var[0]`, Analog dazu lassen sich auch mehrdimensionale Arrays definieren.

2.3 ASSIGN-Teil

Das ist der Hauptteil des Programms.

Mit `init` können Variablen initialisiert werden. Dazu wird die Syntax

```
init(<name>) := <value>;
```

verwendet, wobei `<name>` der Name der Variable und `<value>` deren Startwert ist. Nicht initialisierte Variablen können jede mögliche Belegung (ihres Typs) annehmen.

Ein nächster potentieller Wert einer Variable kann folgendermaßen festgelegt werden:

```
next(<name>) := <value>;
```

Analog zur `init()`-Anweisung gilt auch hier, dass der nächste Wert einer Variable beliebig ist, wenn für diese Variable keine `next()`-Anweisung (oder ähnliche Anweisungen) existiert. `<value>` kann in beiden Fällen nicht nur ein Wert, sondern ein komplexeres Konstrukt sein. Wird einer Variable eine Menge von initialen oder nächsten Werten zugewiesen, so wählt das SMV-System nicht-deterministisch einen dieser Werte aus. Dadurch, dass eine Variable zu einem Zeit

Interessant ist noch das `case`-Konstrukt. Es wird mit `case` eingeleitet und mit `esac` abgeschlossen. Das Statement wird von oben nach unten abgearbeitet. Sobald ein Wert links des `:` wahr ist, gibt das Statement den Wert rechts vom Doppelpunkt zurück. Durch die `1` links vom Doppelpunkt in der letzten Zeile des `case`-Statements wird ein Standardfall angegeben, der ausgeführt wird, wenn alle anderen Ausdrücke falsch sind.

Beispiel 2.2 (case-Statement)

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

2.4 DEFINE-Teil

Mit Hilfe der `DEFINE`-Anweisung kann man einer Variable den aktuellen Wert eines Ausdrucks zuweisen.

Beispiel 2.3 (DEFINE)

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

Das dient dazu einfach auf den Wert dieses Ausdrucks zugreifen zu können. Das ist z.B. sinnvoll, wenn dieser Ausdruck häufig benutzt wird. Der Wert vergrößert den Zustandsraum nicht.

2.5 Spezifikation

Die Spezifikationen ist eine Menge von Formeln der CTL (Computational Tree Logic). Die Kenntnis von CTL wird vorausgesetzt und nicht näher beschrieben. Die Spezifikation ist eine Menge von Bedingungen, die bei jeder möglichen Programmausführung erfüllt sein müssen. Für jede dieser Bedingungen gibt das SMV-System aus, ob sie erfüllt ist oder nicht. Falls die Bedingung nicht erfüllt ist, dann wird ein Gegenbeispiel ausgegeben. Das Gegenbeispiel ist ein Ausführungspfad, der die Bedingung verletzt. Im Programmcode wird jede Bedingung mit dem Schlüsselwort `SPEC` gefolgt von der CTL-Formel notiert.

2.6 Module

Module helfen den Programmcode in mehrere kleine übersichtliche Teile zu zerlegen. Jedes SMV-Programm besteht aus mindestens einem Modul, dem `main`-Modul (wenn nicht bei der Ausführung explizit ein Startmodul angegeben wird). Jedes Modul wird mit dem Schlüsselwort `MODULE` und dem Namen des Moduls eingeleitet. Auf Variablen des Moduls kann, wie z.B. in in Java und C++ üblich, mit dem Punktoperator „.“ zugegriffen werden, ausführlich: `<modulname>.<variablenname>`.

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;
```

Module können Parameter entgegennehmen. In diesem Beispiel ist das der Parameter `carry_in`. Wenn der Parameterwert 0 ist, dann behält der Zähler seinen Wert bei, ansonsten invertiert er seinen Wert (beim Datentyp `boolean` werden `true` und `false` durch 0 und 1 repräsentiert).

Instanzen der Module werden beim anlegen der Variablen gebildet z.B. `bit0 : counter_cell(1);`. Den aktuellen Zählerstand dieses Zählers kann man dann mit `bit0.value` abrufen.

2.7 synchrone und asynchrone Komposition

Normalerweise werden SMV-Module synchron angesteuert, d.h. es gibt eine globale Uhr und bei jeder Zeitweitschaltung wird jede Modulinstanz einmal angesteuert. Die Verifizierung von Hard- bzw. Software ist jedoch oft gerade für Fälle interessant, bei denen mehrere Prozesse parallel arbeiten und sich nicht vorhersagen lässt, welcher Prozess als nächstes aktiv wird. SMV bietet für diesen Fall das Schlüsselwort `process` an. Dieses wird beim instanzieren eines Moduls angegeben. Im Fall von Beispiel 2.4 wäre das der Aufruf `bit0 : process counter_cell(1);`. Ein Prozess im Sinne von SMV ist als eine Instanz eines Modules, welches mit dem Schlüsselwort `process` eingeführt wurde. Ein Programmschritt besteht dann darin nichtdeterministisch einen Prozess zu wählen und diesen auszuführen.

3 Fairness

Die im letzten Kapitel angesprochene Möglichkeit asynchron Prozesse auszuführen, führt auf ein Problem, welches an einem Beispiel demonstriert werden soll:

Listing 3.1 (Inverter)

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

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

Das Modul `inverter` nimmt einen booleschen Wert entgegen und negiert diesen. Im Hauptprogramm werden 3 `inverter`-Module instanziiert. Lässt man die `process`-Schlüsselwörter weg, dann läuft das Programm folgendermaßen ab:

Alle `output`-Werte werden mit 0 initialisiert. Im nächsten Schritt werden parallel alle Modulinstanzen abgearbeitet. Da `gate1`, `gate2` und `gate3` als Eingabe aus dem vorherigen Schritt 0 haben, werden alle gates auf 1 invertiert. Im nächsten Schritt werden alle Gates wieder 0 usw. Die Spezifikation sagt nur aus, dass `gate1.output` auf jedem möglichen Pfad min. einmal 0 und min. einmal 1 sein sollte. Diese Spezifikation ist also erfüllt. Man könnte die Spezifikation sogar noch erweitern und fordern, dass es auf jedem möglichen Pfad einen Zustand geben muss, bei dem alle `output`-Werte der Gates 0 bzw. 1 sind.

Beispiel 3.2 (erweiterte Spezifikation)

```
SPEC
  (AG AF (gate1.output & gate2.output & gate3.output))
  & (AG AF (!gate1.output & !gate2.output & !gate3.output))
```

Zurück zu Beispiel 3.1. Was passiert, wenn man die `process`-Schlüsselwörter nicht weglässt? Alle `output`-Werte wären anfangs 0, dann würde nichtdeterministisch ein Gate invertieren. Die Spezifikation fordert, dass `gate1` irgendwann den Wert 1 annimmt. In dem theoretisch möglichen Fall, dass Prozess `gate1` nie aufgerufen wird, ist die Spezifikation nicht erfüllt. SMV bietet die Möglichkeit anzugeben, dass der Prozessscheduler fair ist. Wenn man `FAIRNESS running` in ein Modul einfügt berücksichtigt SMV nur Pfade, in denen das gewählte Modul unendlich oft ausgeführt wird. In unserem Beispiel führt das dazu, dass

die Spezifikation erfüllt wird. Die erweiterte Spezifikation aus Beispiel 3.2 ist natürlich trotzdem nicht erfüllt, da der Scheduler die Möglichkeit hat die Gates so umzuschalten, dass sie nie alle den gleichen Wert haben.

In der FAIRNESS-Klausel können beliebige CTL-Formeln stehen. FAIRNESS ϕ bedeutet, dass jeder Pfad ignoriert wird, auf dem ϕ nicht unendlich oft wahr ist.

Beispiel 3.3 (Fairness-Klauseln)

(1) FAIRNESS a=foo

(2) FAIRNESS a=start \rightarrow AF a=end

(1) sagt aus, dass a auf einem Pfad unendlich oft den Wert foo annimmt. (2) sagt aus, dass auf einem Pfad unendlich oft die CTL-Formel a=start \rightarrow AF a=end gültig sein muss.

4 Beispiele

Nach der Einführung und der Erklärung von Syntax und Semantik von SMV sollen jetzt zur Verdeutlichung praktische Beispiele gezeigt werden.

4.1 gegenseitiger Ausschluss

4.1.1 Erklärung

In modernen Betriebssystemen gibt es häufig den Fall, dass mehrere Prozesse sich eine Resource teilen müssen. Wird zum Beispiel eine Datei von einem Prozess zum Schreiben geöffnet, so kann kein zweiter Prozess ebenfalls in diese Datei schreiben, da das zu unvorhersagbaren Resultaten führen würde. Es ergibt sich also die Notwendigkeit sicherzustellen, dass eine Resource nur jeweils von maximal einem Prozess benutzt wird.

Bestimmte Teile im Code eines Prozesses werden deshalb als kritische Abschnitte identifiziert. Es darf sich nie mehr als ein Prozess innerhalb dieses kritischen Abschnitts befinden. Dieses Kriterium heißt „Sicherheit“. Da man das Kriterium einfach erfüllen könnte, indem man nie einen Prozess in den kritischen Abschnitt lässt, gibt es noch weitere sinnvolle Kriterien. Die genauen Hintergründe sprengen den Rahmen dieses Beispiels.

Hier ist die Liste der Anforderungen, die wir an ein Verfahren zur Zuteilung der Resource stellen wollen:

- *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

In Abbildung 1 ist ein Transitionssystem dargestellt, welches für zwei Prozesse den Schutz einer Resource gewährleisten soll. In einem Knoten steht „n“ dafür, dass der Prozess nicht im kritischen Abschnitt ist (non-critical). „t“ steht dafür, dass der Prozess anfragt den kritischen Abschnitt zu betreten (trying). „c“ steht dafür, dass der Prozess sich in einem kritischen Abschnitt befindet. In einem Knoten sind jeweils zwei der beschriebenen Buchstaben. Der erste Buchstabe steht für den ersten, der zweite Buchstabe für den zweiten Prozess.

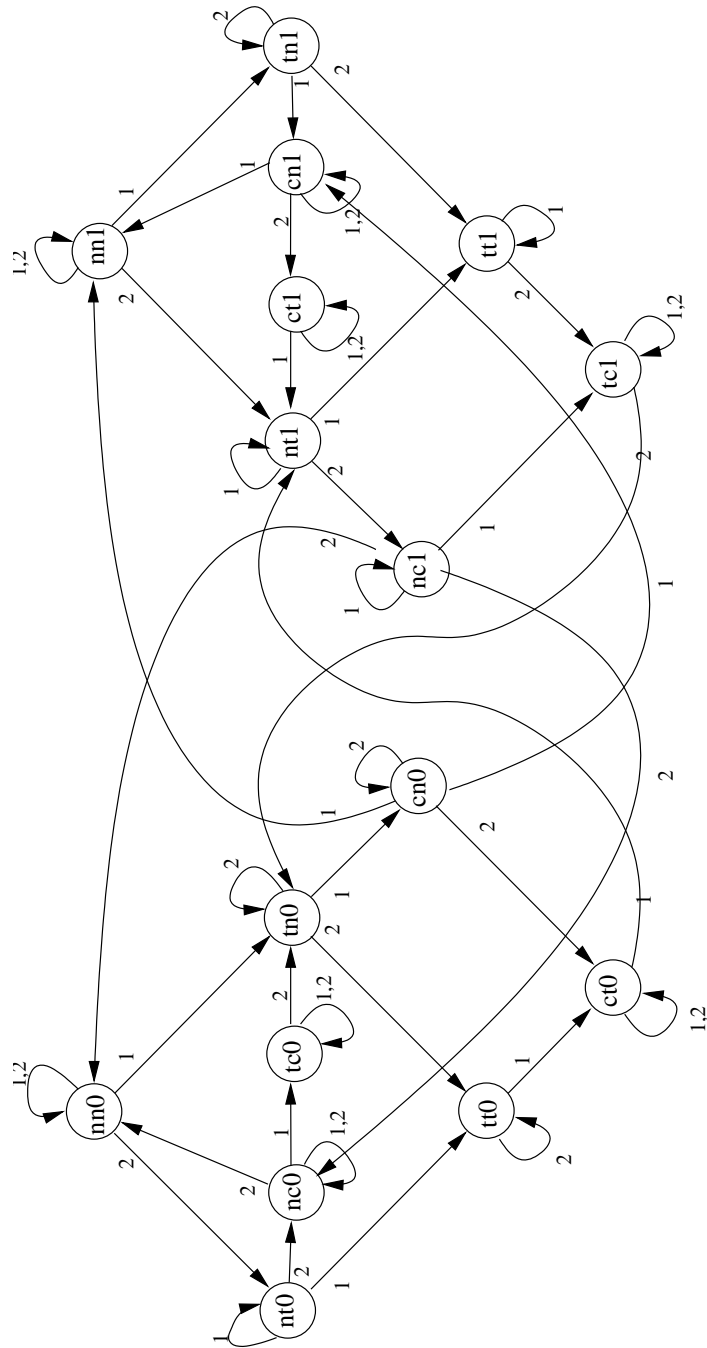


Abbildung 1: Transitionssystem gegenseitiger Ausschluss aus [1] S.187 (korrigiert)

Die Zahl dahinter ist die Variable `turn`. Diese dient dazu sicherzustellen, dass die Lebendigkeitsforderung erfüllt sein kann. Bei Zuständen, die sich nur durch die `turn`-Variable unterscheiden, sind die vom Zustand weg führenden Transitionen identisch bis auf die `turn`-Variable. Sie unterscheiden sich also nur durch die Art und Weise, wie sie entstanden sind. Würde man die Unterscheidung nicht einführen, so wäre es möglich, dass bei konkurrierenden Prozessen immer wieder ein Prozess bevorzugt wird. Die Zahlen über den Pfeilen geben an, welcher Prozess den Übergang macht.

4.1.3 Verifizierung

Abbildung 1 soll jetzt in ein SMV-Programm umgewandelt werden. Der Code dafür ist in Beispiel 4.1 abgebildet. Es wird ein Prozess-Modul erstellt, von dem 2 Instanzen gebildet werden. Um zu überprüfen, ob alle Forderungen erfüllt werden, werden diese in CTL-Formeln umgewandelt. Beiden Prozessen wird eine feste Variable `myturn` zugewiesen, die bestimmt, welcher Prozess im Konfliktfall den Vorrang hat. Über `other-st` kennt ein Prozess immer den Zustand des anderen Prozesses.

Im Modul `prc` sind dann folgende Zustandsübergänge formuliert, die mehreren Pfeilen im Diagramm entsprechen können:

- 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.

War ein Prozess gerade in seinem kritischen Abschnitt, so ist danach der andere Prozess am Zug. Auch hier ist die Klausel `FAIRNESS running` notwendig, da ansonsten ein Prozess „verhungern“ kann, weil er nie Prozessorzeit bekommt. Diese Klausel können wir einbauen, da der Scheduler des Betriebssystems dafür sorgt, dass ein Prozess nicht verhungert. Außerdem wollen wir fordern, dass ein Prozess nie unendlich lange in seinem kritischen Abschnitt bleibt. Das heisst, dass der Zustand `st` eines Prozesses nicht unendlich oft `c` ist. Deswegen fügen wir die Klausel `FAIRNESS !(st=c)` hinzu.

Listing 4.1 (gegenseitiger Ausschluss)

```
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 ]]))

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)

```

4.2 Loyd-Puzzle

4.2.1 Erklärung des Spiels

Beim Loyd-Puzzle ist eine 4x4-Matrix gegeben. Jedes Feld, bis auf eins, enthält ein Plättchen mit einer bestimmten Zahl. Mit Hilfe des freien Feldes lassen sich die Plättchen verschieben und so neue Kombination erreichen. Ziel ist es eine bestimmte Anordnung der Zahlen zu bekommen. Normalerweise ist dabei keine Ausgangsmatrix vorgegeben. Das Loyd-Puzzle macht hier einen Unterschied und gibt eine feste Ausgangsmatrix vor, die sich zudem kaum von der Lösung unterscheidet. Lediglich die „14“ und die „15“ müssen vertauscht werden.

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	

Tabelle 1: Links das Ausgangsproblem und rechts die Lösung des Loyd-Puzzles

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. Jedemal 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

Bevor das Loyd-Puzzle-Problem gelöst werden soll, beschäftigen wir uns mit einem kleineren Problem. Statt einer 4x4-Matrix wird eine kleinere 3x3-Matrix mit 8 Plättchen verwendet. Tabelle 2 zeigt die Ausgangs- und Zielmatrix.

	1	2
3	4	5
6	7	8

8	7	6
5	4	3
2	1	

Tabelle 2: Links das Ausgangsproblem und rechts die Lösung des Puzzles

Es gibt viele Wege an dieses Problem heranzugehen. Ich habe mich dafür entschieden ein Modul `Field` für die Plättchen und ein Modul `FreiesField` für das freie Feld zu schreiben. Alle Plättchen werden mit einer Startposition initialisiert. Die Position setzt sich aus einer vertikalen und einer horizontalen Komponente zusammen.

Listing 4.2 (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);

```

Es stellt sich nun die Frage, wie ein Spiel überhaupt mit Hilfe des SMV-Systems gelöst werden kann, schließlich geht es scheinbar überhaupt nicht darum Eigenschaften eines Systems zu verifizieren. Wir machen uns zu Nutze, dass bei nicht erfüllten Spezifikationen immer ein Gegenbeispiel angegeben wird, welches die Spezifikation verletzt. Indem wir fordern, dass es keine Lösung des Puzzles gibt, zwingen wir das SMV-System dazu als Gegenbeispiel eine Lösung auszugeben.

Listing 4.3 (Spezifikation)

```

SPEC
  !(EF(
    (FF.v = 3 & FF.h = 3)
    & (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) ))

```

Die Bewegung der Plättchen wird im Modul `FreiesFeld` realisiert. In jedem Zug „bewegt“ sich das freie Feld in eine bestimmte Richtung. Die möglichen Richtungen sind `u` (unten), `o` (oben), `l` (links) und `r` (rechts). Die erlaubten Richtungen sind abhängig von der aktuellen Position des freien Feldes. Da das leere Feld am Anfang in der linken oberen Ecke ist, kann es sich von dort nur nach rechts oder unten bewegen. Die Positionsänderung des freien Feldes ist von dieser Richtung abhängig.

Die anderen Plättchen bewegen sich entsprechend dem freien Feld, deshalb ist das freie Feld ein Parameter für das Modul `Feld`. Jedes Plättchen prüft, ob das freie Feld sich auf die aktuelle Position des Plättchens bewegen will. Ist dies der Fall, dann nimmt das Plättchen im nächsten Zug die aktuelle Position des freien Feldes ein. Listing 4.4 zeigt den gesamten Quelltext.

Listing 4.4 (einfaches Puzzle)

```

MODULE main
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);
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) ))

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;

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

```

```

init(richtung) := anfangsrichtung;
-- festlegen in welche Richtung sich die freie Fläche bewegt
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};
    1 : richtung;
  esac;
next(v) :=
  case
    (richtung = u & v < 3) : v + 1;
    (richtung = o & v > 1) : v - 1;
    1 : v;
  esac;
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

Mit offensichtlichen Änderungen ergibt sich das Programm zur Lösung des Loyd-Puzzles. Hier wird jedoch ein Problem deutlich, was SMV oder ähnliche Systeme zu bewältigen haben. Die Anzahl der Zustände wächst sehr schnell, was dazu führt, dass das Programm zur Lösung des Loyd-Puzzles in dieser Form sehr viel Reosourcen und Zeit beansprucht. Terminiert das Programm dennoch erfolgreich, dann wird man eine Überraschung erleben: Die Spezifikation ist wahr! Das bedeutet, dass das Loyd-Puzzle unlösbar ist, was der Grund dafür war, warum über Jahrzehnte hinweg viele Leute an diesem Puzzle gescheitert sind.

Listing 4.5 (Loyd-Puzzle)

```

-- Programm, welches zeigt, dass Loyd-Puzzle unlösbar ist

MODULE main
VAR
  FF : FreiTfeld; -- freier Platz
  F1 : Feld(1, 1, FF); -- Plättchen 1
  F2 : Feld(1, 2, FF); -- Plättchen 2

```

```

F3 : Feld(1, 3, FF); -- Plättchen 3
F4 : Feld(1, 4, FF); -- Plättchen 4
F5 : Feld(2, 1, FF); -- Plättchen 5
F6 : Feld(2, 2, FF); -- Plättchen 6
F7 : Feld(2, 3, FF); -- Plättchen 7
F8 : Feld(2, 4, FF); -- Plättchen 8
F9 : Feld(3, 1, FF); -- Plättchen 9
F10 : Feld(3, 2, FF); -- Plättchen 10
F11 : Feld(3, 3, FF); -- Plättchen 11
F12 : Feld(3, 4, FF); -- Plättchen 12
F13 : Feld(4, 1, FF); -- Plättchen 13
F14 : Feld(4, 2, FF); -- Plättchen 14
F15 : Feld(4, 3, FF); -- Plättchen 15

```

```

richtung : {l, r, u, o};

```

```

SPEC

```

```

!(EF( (F1.v = 1 & F1.h = 1)
      & (F2.v = 1 & F2.h = 2)
      & (F3.v = 1 & F3.h = 3)
      & (F4.v = 1 & F4.h = 4)
      & (F5.v = 2 & F5.h = 1)
      & (F6.v = 2 & F6.h = 2)
      & (F7.v = 2 & F7.h = 3)
      & (F8.v = 2 & F8.h = 4)
      & (F9.v = 3 & F9.h = 1)
      & (F10.v = 3 & F10.h = 2)
      & (F11.v = 3 & F11.h = 3)
      & (F12.v = 3 & F12.h = 4)
      & (F13.v = 4 & F13.h = 1)
      & (F14.v = 4 & F14.h = 3)
      & (F15.v = 4 & F15.h = 2) ))

```

```

MODULE Feld(vertical, horizontal, FF)

```

```

VAR

```

```

v : 1..4; -- vertikal
h : 1..4; -- 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;

MODULE Freifeld
VAR
    richtung : {l, r, u, o};
    v : 1..4;
    h : 1..4;
ASSIGN
    init(v) := 1;
    init(h) := 1;
    init(richtung) := {l, o};

-- Bewegungen der freien Fläche
next(v) :=
    case
        (richtung = u & v < 4) : v + 1;
        (richtung = o & v > 1) : v - 1;
    1
        : v;
    esac;
next(h) :=
    case
        (richtung = r & h < 4) : h + 1;
        (richtung = l & h > 1) : h - 1;
    1
        : h;
    esac;

-- möglichen nächsten Zug bestimmen
next(richtung) :=
    case
        (v = 1 & h = 1) : {u, r};
        (v = 1 & h = 2) : {u, r, l};
        (v = 1 & h = 3) : {u, r, l};
        (v = 1 & h = 4) : {u, l};
        (v = 2 & h = 1) : {o, u, r};
        (v = 2 & h = 2) : {o, u, r, l};
        (v = 2 & h = 3) : {o, u, r, l};
        (v = 2 & h = 4) : {o, u, l};
        (v = 3 & h = 1) : {o, u, r};
        (v = 3 & h = 2) : {o, u, r, l};
        (v = 3 & h = 3) : {o, u, r, l};
        (v = 3 & h = 4) : {o, u, l};
        (v = 4 & h = 1) : {o, r};
        (v = 4 & h = 2) : {o, r, l};
        (v = 4 & h = 3) : {o, r, l};
    
```

```
(v = 4 & h = 4) : {o, l};  
1 : richtung;  
esac;
```

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