

## Übungen zu Model Checking

### Aufgabe 0.1      Gegenseitiger Ausschluss - Dekker-Mutex-Algorithmus

Aus der Vorlesung kennen Sie die Problemstellung des „Gegenseitigen Ausschlusses“:

Es ist ein Algorithmus gesucht, welcher sicherstellt, dass zwei asynchron zu einander laufende Prozesse  $P_1, P_2$ , welche beide dieselbe Resource  $R$  benötigen, nie gleichzeitig auf  $R$  zugreifen.

Dabei wird nur angenommen, dass Lese- und Schreiboperationen atomar sind.

In der Vorlesung wurde Ihnen hierfür der Algorithmus von T.J. Dekker vorgestellt. Dieser lässt sich in Promela, der Eingabesprache von SPIN, wie folgt beschreiben:

```
1 // Zeilenkommentar
2 /*
3  Blockkommentar
4 */
5 //Definition globaler Variablen, auf welche jeder Prozess zugreifen kann
6 bit    turn;    //Variable mit genau einem bit Speichergröße, turn ∈ {0,1}
7 bool   flag [2]; //Array mit zwei Einträgen flag [0], flag [1] ∈ {true, false}
8 byte   cnt;    //8-bit Variable, cnt ∈ {0,1,...,255}
9
10 //Definition des Prozesstyps „dekker“ und gleichzeitig Instanzierung
11 //zweier Prozesse desselben Typs mittels active [2]
12 active [2] proctype dekker () {
13     pid    this, other;    //Lokale Variablen zum Speichern einer Process ID (PID)
14
15     this   = _pid;    //_pid gibt die PID des Prozesses zurück
16           //alle aktiven Prozesse sind, beginnend bei 0, durchnummeriert
17     other  = 1 - _pid; //Da nur zwei Prozesse aktiv sind, gilt {this, other} = {0,1}
18
19     again:
20         flag[this] = true;    //Zugriffsabsicht signalisieren
21
22     do    //Versuche Zugriffsrecht auf Resource zu erhalten
23         //Die do-Schleife wird solange durchlaufen,
24         //bis sie explizit mittels break verlassen wird
25         //Mittels :: werden mehrere 'geschützte' Optionen
26         //angegeben, wobei 'nichtdeterministisch' eine Anweisung
27         //ausgewählt wird, deren Guard erfüllt ist
28         :: flag[other] -> //Der andere Prozess will auch auf die Resource zugreifen
29           //flag[other] ist hier der Guard der folgenden if-Anweisung
30         if
31         :: turn == other -> //Falls der andere Prozess Vorfahrt hat,...
32           flag[this] = false; //Zugriffsabsicht zurückziehen
33           (turn == this) -> //Warte, bis dieser Prozess Vorfahrt hat
34           //Ausdrücke, welche sich zu true/false auswerten,
35           //blockieren den Prozess, bis sie erfüllt sind.
36           flag[this] = true //Zugriffsabsicht erneut äußern
37         :: else -> skip
38         fi //nächster Versuch; Sprung an den Anfang von do
39         :: else -> break //Der andere Prozesse hat keine Zugriffsabsicht geäußert
40           //es kann also auf Resource zugegriffen werden
41     od;
42
43     cnt++; //Betrete den geschützten Bereich
44     cnt--; //Verlasse den geschützten Bereich
45     turn = other; //”Fairness”: Gib dem anderen Prozess das Zugrecht
46     flag[this] = false;
47     goto again; }
```

- (a) Wie aus der Problemstellung hervorgeht, ist die wichtigste Eigenschaft, welche obiger Algorithmus erfüllen soll, dass sich nie beide Prozesse gleichzeitig in dem geschützten Bereich befinden.

Überprüfen Sie, ob diese Anforderung erfüllt ist:

- i) Starten Sie `xspin` (Doppelklick auf Symbol oder per Kommando-Zeile `./xspin`).
- ii) Übertragen Sie den Promela-Code in das nun geöffnete Textfenster.
- iii) Wählen Sie aus dem Menü „RUN“ den „LTL Property Manager“ aus.
- iv) Definieren Sie im Textfenster „Symbol Definitions“ die atomare Proposition „mutex“. Geben Sie hierfür in das Fenster „**#define** mutex ( cnt < 2 )“ ein.

Die Proposition „mutex“ ist dann genau in den Programzuständen wahr, in welchen die globale Variable „cnt“ den Wert 0 oder 1 hat.

- v) Tragen Sie nun in die Textzeile „Formula“ die entsprechende **LTL**-Formel ein, welche aussagt, dass zu jedem Zeitpunkt die Eigenschaft „mutex“ gelten soll.

Stellen Sie sicher, dass die Option „All Executions (desired behavior)“ ausgewählt ist.

*Hinweis:* SPIN verwendet `[]` bzw. `<>` für die Operatoren **G** bzw. **F**.

- vi) Klicken Sie nun auf den Knopf „Generate“, um den SPIN -Code für Ihre **LTL**-Formel (bzw. deren Negation) zu erzeugen.
- vii) Betätigen Sie dann den Knopf „Run Verification“, um die eigentliche Verifikation zu starten. In dem nun geöffneten Fenster sollten nur die Optionen „Exhaustive“ und „Blocks New Msgs“ ausgewählt sein.

Klicken Sie dann auf „Run“.

SPIN kompiliert nun den entsprechenden Model-Checker für den angegebenen Promela-Code und Ihre **LTL**-Formel und führt diesen anschließend aus. Den Statusmeldungen können Sie entnehmen, ob und wie viele Fehler SPIN gefunden hat.

SPIN sollte keinen Fehler finden.

- (b) Eine weitere Forderung ist, dass immer, wenn ein Prozess auf die Resource zugreifen will, ihm dies auch schließlich gestattet wird.

Um diese Eigenschaft zu testen, ist die Variable `cnt` nicht ausreichend. Passen Sie den Promela-Code so an, dass Ihnen genügend Informationen zur Verfügung stehen, um entsprechende atomare Propositionen und die entsprechende **LTL**-Formel zu definieren.

Überprüfen Sie dann, wie in der ersten Teilaufgabe beschrieben, ob diese Eigenschaft erfüllt ist.

SPIN sollte Ihnen am Ende der Verifikation mitteilen, dass diese Anforderung nicht erfüllt ist, und sollte Ihnen nun mehrere Optionen anbieten, um den gefunden Fehler zu analysieren.

Wählen Sie die Option „Setup Guided Simulation“ und aktivieren Sie in dem nun geöffneten Fenster die Optionen „Time Sequence Panel“ und „Interleaved Steps“. Klicken Sie dann auf „Start“ und schließlich auf „Run“ (im Fenster „Simulation Output“).

Vollziehen Sie an Hand der Fenster „Time Sequence“ und „Data Values“ nach, dass auf Grund der asynchronen Ausführung SPIN auch Programmverläufe untersucht, in welchen nur noch einem der beiden Prozesse Rechenzeit zugestanden wird.

Starten Sie nun die Verifikation erneut, aktivieren Sie nun jedoch die Option „With Weak Fairness“, welche gerade dieses „unfaire“ Verhalten eines der beiden Prozesse ausschließt.

- (c) Unten stehend finden Sie die Promela-Version einer „optimierten“ Variante des Dekker-Algorithmus.

```
1  bool crit [2];
2  byte x,y,z;
3
4  active [2] proctype niceTry () {
5      pid this = _pid;
6      byte me = this + 1;
7  again:
8      x = me;
9      if
10     :: ( y == 0 || y == me ) -> skip
```

```

11     :: else -> goto again
12   fi;
13
14   z = me;
15   if
16     :: ( x == me ) -> skip
17     :: else -> goto again
18   fi;
19
20   y = me;
21   if
22     :: ( z == me ) -> skip
23     :: else -> goto again
24   fi;
25
26   crit [ this ] = true;
27   crit [ this ] = false;
28   goto again; }

```

Prüfen Sie, ob dieser Algorithmus die in den vorangegangenen Teilaufgaben geforderten Eigenschaften erfüllt, und versuchen Sie wieder mittels SPIN zu verstehen, wie der Fehler entsteht.

*Hinweis:* Sie können im Dialogfenster, welches Sie mit „Run Verification“ öffnen, unter „[Set Advanced Options]“ mittels der Option „Find Shortest Trail“ SPIN anweisen, schrittweise nach einem kürzesten Gegenbeispiel zu suchen.

- (d) Versuchen Sie nun selbst, den Algorithmus von Dekker zu vereinfachen. Überprüfen Sie z.B., ob auf eine der Schleifen „od ... do;“, „( turn == this ) -> ...“ oder „again: ... goto again;“ verzichtet bzw. durch ein „Nicht-Schleifen-Konstrukt“ ersetzt werden kann.
- (e) Ein sehr „kurzer“ Algorithmus für das Mutex-Problem wurde schließlich 1981 von G.L. Peterson vorgeschlagen:

```

1  bool turn, flag [2], crit [2];
2
3  active [2] proctype peterson() {
4    pid this, other;
5    this = _pid; other = 1 - this;
6  again:
7    flag [ this ] = true;
8    turn = this;
9    ( flag [ other ] == false || turn != this );
10   crit [ this ] = true;
11   crit [ this ] = false;
12   flag [ this ] = false;
13   goto again; }

```

Überprüfen Sie wiederum, ob auch dieser Algorithmus die geforderten Eigenschaften besitzt.

Testen Sie weiterhin, ob die Anweisung in Zeile 9 nicht zu ( flag [ other ] == false ) vereinfacht werden kann.

## Aufgabe 0.2

Auf der Vorlesungsseite finden Sie die Datei `museum.pml`. Diese soll ein System modellieren, in dem der Zugang in ein Museum geregelt wird. Es gibt  $N$  Prozesse, die Besucher darstellen, die das Museum von Zeit zu Zeit betreten möchten. Aus Platzgründen dürfen jedoch nur  $M$  von ihnen gleichzeitig im Gebäude sein ( $M < N$ ).

Das Museum will am Ende des Tages schließen. Dazu gibt es am Eingang eine Leuchttafel, die grün, gelb oder rot leuchten kann. Eintritt ist nur möglich, wenn die Tafel grün leuchtet. Gegen Ende des Tages schaltet eine Aufsicht die Tafel auf gelb um, woraufhin niemand mehr hereingelassen wird. Haben alle Besucher das Haus verlassen, wird auf rot umgeschaltet.

Geben Sie eine formale Spezifikation dieses Systems an, und prüfen Sie, ob `museum.pml` diese Spezifikation erfüllt. (Sie sollten zu dem Schluss kommen, dass dies nicht der Fall ist.) Überlegen Sie, wie man die Modellierung umgestalten könnte, so dass es geht. (Es gibt hierbei mehrere Möglichkeiten.) Versuchen Sie, das Modell entsprechend zu verändern.

## Quellen

Gerard J. Holzmann, *The Spain Model Checker*, Addison-Wesley