# 2. Übungsblatt zur Vorlesung “Formale Methoden im Software Entwurf”

TECHNISCHE UNIVERSITÄT DARMSTADT

**Verifikation mit Spin**
**Diskussion der Aufgaben und Lösungen in den Tutorien: Kalenderwoche 45**

## Aufgabe 1 Verifikation: Zum Aufwärmen—Arrays

Auf dem vorherigen Übungsblatt haben Sie ein PROMELA Modell geschrieben, welches ein Array mit Werten zwischen 1 und 5 nicht-deterministisch füllt und danach das Produkt aus allen Arrayelementen berechnet. Verwenden Sie die Lösung unter `exercises/Array.pml` für diese Aufgabe. Sie können gerne im Anschluss, diese Aufgabe auch mit Ihrer eigenen Lösung durchspielen.

Verifizieren Sie mit Hilfe von Assertions, dass das Modell folgende Eigenschaften besitzt:

a) Das ausgegebene Produkt hat einen nicht-negativen Betrag.

b) Das ausgegebene Produkt ist größer oder gleich dem Wert jedes einzelnen Arrayelementes;

c) Formulieren und verfizieren Sie die Eigenschaft aus Punkt 2 mit nur einer Assertion und einem einzigen Arrayzugriff (sofern Ihre vorherige Lösung diese Einschränkung nicht sowieso schon erfüllt).

## Aufgabe 2 Verifikation des Log-in Systems

Für das vorherige Übungsblatt hatten Sie ein PROMELA Modell für ein einfaches Log-in Verfahren entwickelt. Verwenden Sie das Modell unter `exercises/Authentication.pml` für diese Aufgabe (einige vorher lokale Variablen wurden hier als globale Variablen deklariert).

Die PROMELA Implementierung des Log-in Systems soll auf die Einhaltung einiger gewünschter Eigenschaften geprüft werden.

### Aufgabe 2.1 Verifikation des Log-in Systems: Ungültige Endzustände

Starten Sie als erstes einen Verifikationsversuch auf ungültige Endzustände ohne an dem Modell etwas zu ändern. Der Verifikationsversuch mit Spin sollte fehlschlagen und ein Gegenbeispiel liefern.

Sehen Sie sich das Gegenbeispiel mit Hilfe des Simulationsmodus *Guided* an.

a) Erklären Sie die sich aus dem Gegenbeispiel ergebende Situation.

b) Ergänzen Sie das Modell in geeigneter und sinnvoller Weise, sodass die Verifikation nicht mehr fehlschlägt.

### Aufgabe 2.2 Verifikation von Eigenschaften: Assertions

Verifizieren Sie folgende Aussagen mit Hilfe von Assertions:

a) Nach dem Wechsel in den Zustand 1 ist die Anzahl der Fehlversuche 0.

b) Nach dem Wechsel in den Zustand 2 ist die Anzahl der Fehlversuche 3.

### Aufgabe 2.3 Verifikation von Invarianten des Systems mit Assertions

Das entwickelte System soll folgende Eigenschaft erfüllen:

*Das System befindet sich im Zustand* 2 *(Locked) genau dann, wenn der Log-in in drei (oder mehr) aufeinanderfolgenden Versuchen fehlgeschlagen ist.*

Diese Eigenschaft ist eine Invariante des System und soll zu jedem Zeitpunkt gelten.

a) Ergänzen Sie das Modell, sodass die obige Invariante mit Hilfe von Assertions ausgedrückt wird und ein Verifikationsversuch mit Spin unternommen werden kann.
*Hinweis/Herausforderung:* Es existiert auch eine Lösung, die keine Änderungen an der Implementierung des Authentication Prozesses benötigt.

b) Der Verifikationsversuch von 1 sollte fehlschlagen und Spin (mindestens) ein Gegenbeispiel erzeugen. Sehen Sie sich das Gegenbeispiel im Simulationsmodus *Guided* an und erklären Sie, weshalb die Invariante verletzt ist.

c) Ändern Sie das System (insbes. auch den Automaten) ab, sodass die obige Invariante gilt und verifizieren Sie diese.

## Aufgabe 3 Kreuzungen

Modellieren Sie die folgende Situation in PROMELA:

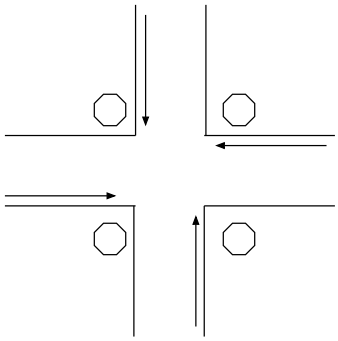

**Abbildung 1:** Eine einfache Kreuzung

Gegeben sei eine einfache Kreuzung (siehe Abbildung 1) von Nord nach Süd und von West nach Ost. Autos können in beide Richtungen fahren, jedoch nicht abbiegen. An allen vier Einmündungen steht ein Stoppschild. Autos dürfen in die Kreuzung einfahren, wenn kein Auto von rechts kommend ebenfalls in die Kreuzung einfahren will und kein Auto gerade auf der Kreuzung steht.

a) Modellieren Sie das Szenario für vier Autos (Nord-Süd, West-Ost, Süd-Nord und Ost-West). Verifizieren Sie, dass niemals zwei Autos gleichzeitig auf der Kreuzung sein können.

b) Spin sollte Ihnen beim Verifizieren einen ungültigen Endzustand anzeigen. Welche Situation beschreibt dieser? Wie sieht die Lösung für diese Situation im realen Leben aus?

c) Versuchen Sie Fehler in ihr Modell einzubauen (z.B. Einfahrt möglich, wenn ein anderes Auto auf der Kreuzung steht). Sehen Sie sich das erzeugte Gegenbeispiel an und vollziehen Sie den Fehler nach.

d) Führen Sie einige Sanity-Checks aus, um zu überprüfen, ob ihr Model sinnvoll ist. Überprüfen Sie z.B. das ihr Modell es erlaubt, das zwei Autos gleichzeitig an der Kreuzung warten u.ä.

Hinweise:

- **atomic** { condition –> statement1; ... statementK; } führt alle statements atomar aus, wenn die Bedingung wahr ist (und keines der Statements blockt). Ansonsten blockiert der Prozess solange bis die Bedingung wahr wird. (Details und ganze Wahrheit in der nächsten Vorlesung)

- Das Modell **proctype** P(**int** i) { ..... } **init** { **atomic** { **run** P(10); **run** P(35) } } startet zunächst mit dem **init** Prozess und startet 2 weitere Prozesse vom Typ P denen es als Argumente den Wert 10 bzw. 35 übergibt. Prozesse können auf ihre Argumente wie auf Variablen zugreifen.

- Hinweis für Fortgeschrittene: _pid ist eine 'read-only' prozess-lokale Variable, die die Prozessnummer enthält. Jeder Prozess kann auf diese Variable zugreifen. Siehe http://spinroot.com/spin/Man/_pid.html.

## Aufgabe 4 State Exploration

In der Datei `explode.pml` finden Sie ein kleines PROMELA Programm vor. Es besteht aus drei Prozessen. Zwei der Prozesse erhöhen bzw. erniedrigen den Wert der globalen Variablen `i`. Der dritte Prozess dient Verifikationszwecken und enthält eine Zusicherung deren Idee es ist, sicher zu stellen, dass `i` niemals den Wert 32767 erreicht.

a) Überlegen Sie sich, warum die Assertion in einigen Läufen verletzt wird. Wie sieht ein (möglichst kleines) Gegenbeispiel aus?

b) Laden Sie das PROMELA Programm mit iSpin und starten Sie einen Verifikationsversuch. Was ist die Ausgabe? Wieso widerspricht dies den Überlegungen aus der vorherigen Teilaufgabe? **Hinweis:** Sehen Sie sich die Ausgabe von spin genau an. Achten Sie insbesondere auf Fehlermeldungen.

c) Stellen Sie die Suchstrategie von `depth-first search` auf `breadth-first search` um. Starten Sie den Verifikationsversuch erneut. Dieses mal sollte das Ergebnis gemäß der ersten Teilaufgabe ausfallen. Sehen Sie sich die Ausgabe erneut an, sie finden die gleiche Fehlermeldung wie vorher. Warum ist das Ergebnis in diesem Fall trotzdem richtig?

d) Lassen Sie die Einstellung auf `breadth-first search` und öffnen Sie in iSpin im Verification Reiter, die „Advanced "Einstellungen. Sie können dort die maximale Suchtiefe einstellen (Standard: 10000). Wieviele Schritte benötigt, dass von Ihnen gefundene Gegenbeispiel. Setzen Sie die maximale Tiefe auf einen etwas niedrigeren Wert als die für Ihr Gegenbeispiel benötigten Schritte? Was beobachten Sie? Setzen Sie jetzt die Anzahl auf die Anzahl der Schritte für ihr Gegenbeispiel (leicht höher, da auch das Erzeugen der Prozesse in iSpin ein Schritt ist) und führen Sie die Verifikation erneut durch.