Modelchecking mit Spin

Vorbereitung

Auf Github ist im Repository `https://github.com/fh-wedel/FSV-2023` ein Codespace so konfiguriert, dass Sie `spin` ohne weitere Installation (in der Kommandozeilen-Version) verwenden können.

Wenn Sie `spin` auf Ihrem System installieren wollen, dann folgen Sie bitte den unten unter 1. stehenden Schritten. Wenn Sie den Codespace nutzen, können Sie 1. überspringen.

1. **Spin Modelchecker installieren**
   Um Spin zu verwenden benötigen Sie einen funktionsfähigen GCC. Unter Windows ist die Verwendung des Linux Subsystem for Windows empfohlen.

   Unter Linux oder Mac OS lässt sich Spin am einfachsten aus der Quellcode-Distribution installieren:

   (a) Clonen Sie bitte das Spin-Reqository unter `https://github.com/nimble-code/Spin` und übersetzen Sie Spin mit dem Kommando `make`. Platzieren Sie bitte das resultierende Executable `Src/spin` in in ein Verzeichnis das nach Kommandos durchsucht wird (oder erweitern Sie `PATH` um das `Src`-Verzeichnis).

   Alternativ oder für Windows verwenden Sie das `spin`-Executable aus den vorkompilierten Archiven für Ihre Plattform aus dem `Bin`-Verzeichnis des Spin-Repositories.

   Optional:

   (b) Clonen Sie bitte auch das Repostory der (in Java geschriebenen) graphischen Entwicklungsumgebung `jspin 5.0` von `https://github.com/motib/jspin.git`. JSpin wird mit `java -jar jspin.jar` gestartet. Möglicherweise bevorzugen Sie jSpin vor der Kommandozeile.

2. **Minimax-Counter**
   Wir wollen zunächst das Minimax-Counter-Beispiel der Vorlesung nachvollziehen (siehe Codespace oder Moodle)

   (a) **Modell verstehen**
   Sehen Sie sich bitte `minimax-counter-check.pml` an.

   (b) **Modell checken**
   Lassen Sie das Verifizierer-Programm `pan.c` erzeugen:

   ```
   $ spin -a mininmax-counter-check.pml
   ```

   Übersetzen Sie es bitte:

   ```
   $ gcc -o pan pan.c
   ```

   und lassen es laufen:

   ```
   $ ./pan
   ```

(c) **Fehler lokalisieren**
Lassen Sie Spin das Modell erneut, geführt durch den Trail `mininmax-counter-check.pml.trail` ausführen, um die Stelle des Fehlers zu finden.

```
$ spin -t -p mininmax-counter-check.pml
```

(d) **Minimax-Counter-Korrektur**
Korrigieren Sie bitte die `Minimax-Counter`-Spezifikation und führen Sie bitte das Modelchecking aus Aufgabe 2b erneut durch. Wie ändert sich die Ausgabe?

3. **Dinierende Philosophen**
Bitte spezifizieren Sie das Problem der dinierenden Philosophen[1] in Promela und überprüfen Sie, ob Ihr Modell

(a) frei von Deadlocks ist und wenn nicht, lokalisieren Sie bitte den Fehler,

(b) niemals zwei Philosophen gleichzeitig die selben Gabeln nehmen

Orientieren Sie sich bitte an folgendem Promela-Gerüst:

```
#define NUM_PHIL 4

proctype phil(int id) {
do
  ::
  printf("thinking\n");
  /* ... */
  printf("eating\n");
  /* ... */
od
}

init {
int i = 0;

do
:: i >= NUM_PHIL -> break
:: else -> run phil(i);
           i++
od
}
```

Tipp: Modellierung der Gabeln

Wie lassen sich die Gabeln der Philosophen modellieren?

Eine Möglichkeit ist, die Gabeln als Array zu modellieren, jede Gabel entspricht einem Array-Element, in dem festgehalten wird, wie viele Philosophen diese Gabel genommen haben.

[1] Sie sitzen im Kreis, zwischen je zwei Philosophen liegt eine Gabel, jeder braucht zwei Gabeln zum Essen, Philosophen denken und essen abwechselnd.

```
int fork[ NUM_PHIL ]; // fork[i]=0: Gabel liegt;
                      // fork[i]=n >0: Gabel von n Philosophen genommen
```

Aufnehmen und Ablegen der Gabeln können dann durch geeignete Veränderung des passenden Array-Elements erreicht werden.

Mehr über Promela

Promela als Spezifikationssprache hat einige Eigenschaften, die gegenüber Programmiersprachen bemerkenswert sind:

- Nebenläufige Prozesse
  Promela erlaubt es, Prozesse mit **proctype** zu definieren. Jeder Prozess besitzt einen eigenen sequentiellen (nichtdeterministischen) Programmfluss, der mit `run` Prozesstypname in Gang gesetzt wird. Prozesse können über globale Variablen (und über Kanäle) kommunizieren.

- Ausführbarkeit von Anweisungen
  Promela–Anweisungen können *ausführbar* oder *blockiert* sein, d. h. die Programmausführung kann an einer Anweisung stoppen und erst nach Eintreten von bestimmten Bedingungen fortgesetzt werden. Boolesche Audrücke (Bedingungen) sind nur dann ausführbar, wenn sie wahr sind und blockieren, wenn sie falsch sind. Beispielsweise lässt sich

  ```
  if
  :: (a<10) -> printf("A ist kleiner als zehn.\n");
  fi
  ```

  daher auch kürzer (a<10) -> printf("A ist kleiner als zehn.\n"); schreiben[2]. Tatsächlich ist `->` nur eine andere Notation für `;` — eingesetzt, um bedingte Ausführung zu verdeutlichen.

- Atomare Ausführung von Sequenzen
  Die Ausführung von Folgen von Anweisungen eines Prozesses kann durch die Ausführung von Anweisungen anderer Prozesse unterbrochen werden (*Interleaving*, *Scheduling*). Mit `atomic{stmt1, stmt2,... }` kann die atomare Ausführung einer Sequenz (ohne Interleaving) erzwungen werden.

- Zusicherungen
  Mit `assert(Bedingung)` kann sichergestellt werden, dass die angegebene Bedingung erfüllt ist. Dies wird z. B. eingesetzt, um in einem eigenen Check-Prozess auf das dauerhaft Einhalten gewünschter Systembedingungen zu prüfen.

[2] `if` wird nur benötigt, falls es mehrere Alternativen gibt, aus denen (nichtdeterministisch) ausgewählt werden soll.