Hochschule Bremerhaven
Prof. Dr. Thomas Umland

Bremerhaven, 17. Oktober 2024

Parallelprogrammierung

Aufgabenblatt 1

1.1

Verdeutlichen Sie sich die Funktionsweise des Odd Even-Algorithmus anhand der Fragen aus der Vorlesung (siehe Folie 1.19).

1.2

a) Implementieren Sie in Java eine Version des Odd Even Sort-Algorithmus *ohne* Verwendung von Nebenläufigkeit. Testen Sie Ihre Implementierung mit zufällig generierten Zahlenfolgen. Prüfen Sie das Resultat auf Korrektheit!

b) Wie würden Sie den Algorithmus nebenläufig implementieren? Was müssten Sie dabei beachten? Wo könnten Probleme (welche?) auftreten?

Hochschule Bremerhaven
Prof. Dr. Thomas Umland

Bremerhaven, 24. Oktober 2024

Parallelprogrammierung

Aufgabenblatt 2

2.1

Vollziehen Sie die Beispiele aus den Vorlesungsfolien zum Erzeugen bzw. Starten von Threads nach

- `Runnable`
- `Callable`
- `ExecutorService`
- `CompletableFuture`
- Parallel Streams
- Virtual Threads

2.2

Auf den Vorlesungsfolien wurde das Interface `ExecutorService` vorgestellt:

a) Experimentieren Sie ähnlich wie auf den Folien mit verschiedenen Instanzen von `ExecutorService` (z. B. „Cached Thread Pool“, „Fixed Thread Pool“, „Scheduled Thread Pool“, „Single Thread Executor“, „Virtual Threads Executor“, . . .).

Welches Verhalten lässt sich bei der Ausführung von `Runnable`- oder `Callable`-Instanzen beobachten? Wie viele Threads laufen wirklich gleichzeitig? Werden Thread-Instanzen wiederverwendet?

b) Erstellen Sie ein Java-Programm, dass eine rekursive Berechnung[1] entweder mit `RecursiveTask<V>` oder mit `RecursiveAction` durchführt.

Verwenden Sie zur Ausführung sowohl den Standard-`ForkJoinPool` als auch einen explizit erzeugten `ForkJoinPool` mit der dreifachen Parallelität des Standard-Pools.

c) Lässt sich im Aufgabenteil b „Work Stealing“ beobachten?

[1] Wenn Ihnen kein anderes Beispiel einfällt, können sie zum Beispiel den Quicksort-Algorithmus verwenden.

Hochschule Bremerhaven
Prof. Dr. Thomas Umland

Bremerhaven, 7. November 2024

Parallelprogrammierung

Aufgabenblatt 3

3.1

Auf den Folien wurde die Klasse `StrangeCounter` vorgestellt, die in dieser Aufgabe etwas modifiziert werden soll:

a) Definieren Sie ein Interface `CounterInterface`, das die Methoden `long get()`, `long incrementAndGet()` und `void check(long desired)` bereitstellt.[2]

Verwenden Sie für den Zähler `counter` jetzt eine Wrapper-Klasse `MyLong`, die das Interface `CounterInterface` implementiert und den Zählerstand intern als `private long` Variable enthält.[3]

Ändern Sie auch die Implementierung der Klasse `Incrementer`, so dass `counter` nicht mehr global vorhanden ist, sondern im Konstruktor übergeben wird:

```
public Incrementer(CountDownLatch start, CountDownLatch end,
                   CounterInterface counter) { ... }
```

b) Wie auf den Folien sollen wieder mehrere Instanzen von `Incrementer` nebenläufig ausgeführt und das Endergebnis auf Korrektheit überprüft werden.

Gibt es Unterschiede, wenn Sie die Threads direkt oder über einen `ExecutorService` erzeugen bzw. starten? Verwenden Sie verschiedene Implementierungen von `ExecutorService`, die Sie über die Factory-Methoden `Executors.newCachedThreadPool`, `Executors.newFixedThreadPool` und `Executors.newSingleThreadExecutor` erzeugen. Probieren Sie auch Virtual Threads aus.

Arbeitet Ihr Programm unter Verwendung der Wrapper-Klasse `MyLong` im Unterschied zu der Version auf den Folien immer korrekt? Warum?

c) Erweitern Sie `MyLong` zu `MyLongAtomic` und verwenden Sie dort statt einer `long`-Variablen eine Instanz der Klasse `java.util.concurrent.atomic.AtomicLong`.

Testen Sie wie in Aufgabenteil b wieder die Ausführung über direkte Threads und über `ExecutorService`.

Arbeitet Ihr Programm immer korrekt? Warum?

d) Nun soll der Zähler nur noch „modulo 16“ zählen – d. h. er durchläuft zyklisch die Werte $0, 1, 2, \ldots, 15$ und speichert auch nur diese ab![4] Trotzdem dürfen keine nebenläufig ausgeführten Zählimpulse verloren gehen (für Hinweise zur Lösung, siehe API-Dokumentation der Klasse `AtomicLong`)!

---

[2] `CounterInterface` wird für die Implementierung eines Zählers verwendet: `get` gibt den aktuellen Zählerstand zurück; `incrementAndGet` erhöht den aktuellen Zählerstand um 1 und gibt den neuen Zählerstand zurück; `check` vergleicht den aktuellen Zählerstand mit dem als Parameter übergebenen Sollwert und gibt bei einer Abweichung eine Meldung aus; `check` soll direkt im Interface als `default`-Methode implementiert werden.

[3] Außer der Kapselung der Methoden in der Wrapper-Klasse werden *keine* weiteren Maßnahmen getroffen.

[4] Der Zähler besitzt beispielsweise nach 35 Zählimpulsen den Wert 3. Beachte: Im Zähler dürfen nur die Werte $0, 1, 2, \ldots, 15$ gespeichert werden!

Erweitern Sie dazu `MyLongAtomic` zu `MyLongAtomicModulo` und verwenden intern wieder `AtomicLong` zum Speichern der Zählerwerte. Probieren Sie mehrere Lösungsansätze aus!

Testen Sie wie unter b und c wieder die nebenläufige Verwendung von `MyLongAtomicModulo`.

Hochschule Bremerhaven
Prof. Dr. Thomas Umland

Bremerhaven, 14. November 2024

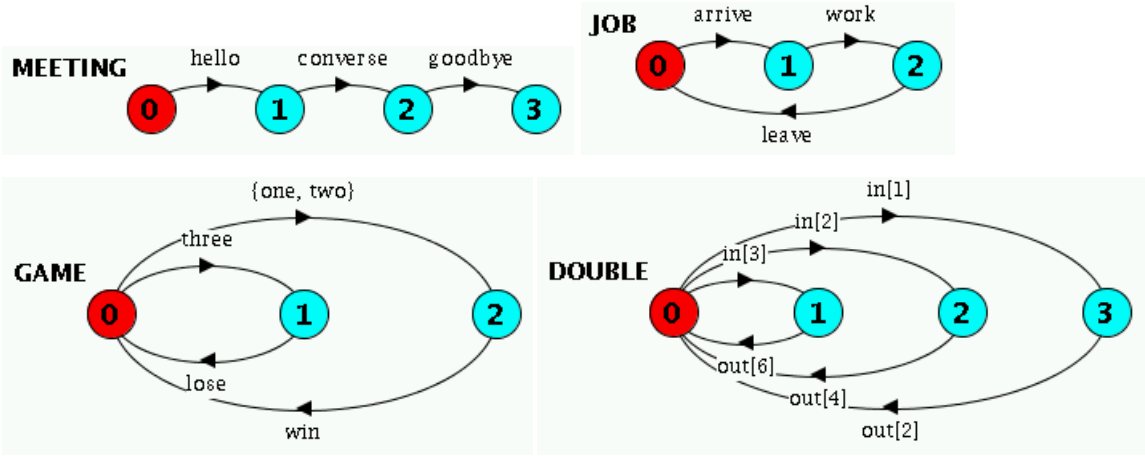
Parallelprogrammierung

Aufgabenblatt 4

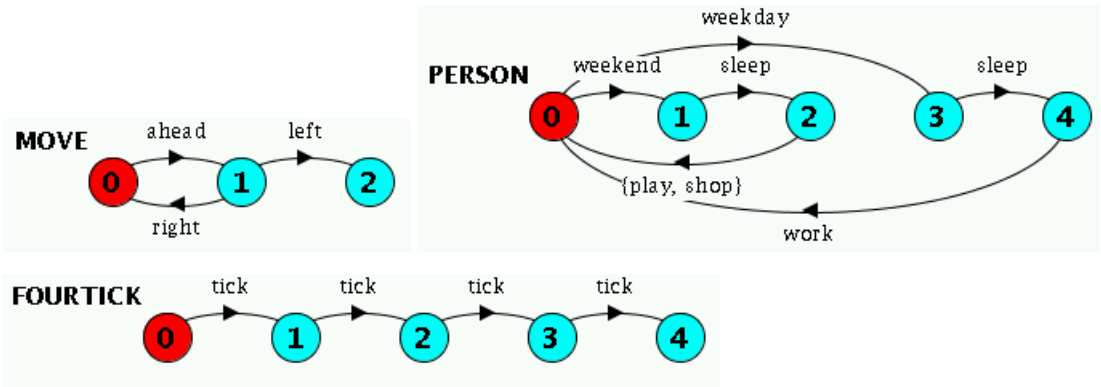
4.1

Geben Sie für jeden der sieben im folgenden als Zustandsgraphen dargestellten Prozesse die zugehörige algebraische FSP-Beschreibung an. Überprüfen Sie Ihre Lösung mit Hilfe des Analysetools LTSA und geben Sie einige Traces aus.

a)



b)



4.2

Ein Sensor misst den Wasserstand in einem Tank, wobei der Pegel die Werte $0, \ldots, 9$ annehmen kann (5 ist der Anfangspegel). In Abhängigkeit des Wasserstandes gibt der Sensor die Meldungen „zu niedrig“ (bei Pegel kleiner als 3), „zu hoch“ (bei Pegel größer als 7) oder „normal“ (sonst) aus. Modellieren Sie den Sensor als FSP SENSOR.

Hinweis: Das Alphabet von SENSOR ist $\{pegel[0..9], niedrig, hoch, normal\}$.

4.3

In einem Getränkeautomaten kostet ein Becher süße Brause 60 Cent. Der Automat nimmt 10-, 20- und 50-Cent-Münzen an und gibt gegebenenfalls Wechselgeld zurück. Beschreiben Sie den Getränkeautomaten algebraisch durch FSP und geben Sie den entsprechenden Zustandsgraphen an.

4.4

a) Zeigen Sie, dass die im Folgenden definierten Prozesse `S1` und `S2` das gleiche Verhalten zeigen:

```
P = (a -> b -> P).
Q = (c -> b -> Q).
||S1 = (P || Q).

S2 = ( a -> c -> b -> S2
     | c -> a -> b -> S2).
```

b) Finden Sie einen Prozess, der ohne die Verwendung der parallelen Komposition || definiert ist, aber das gleiche Verhalten wie der folgende Prozess `Fertigung` zeigt.

```
Prod1 = (prod1 -> fertig -> montiert -> Prod1).
Prod2 = (prod2 -> fertig -> montiert -> Prod2).
||Produktion = (Prod1 || Prod2).

Montage = (fertig -> montage -> montiert -> Montage).
||Fertigung = (Produktion || Montage).
```

4.5

Der Prozess `Speicher = (rein -> raus -> Speicher).` modelliert das Verhalten einer Speicherzelle, indem er zunächst eine `rein`- und dann eine `raus`-Aktionen ausführt.

Definieren Sie unter Verwendung von `Speicher` und mit Hilfe der parallelen Komposition einen Prozess `Schieber`, der das Verhalten eines Schieberegisters mit $N = 4$ internen Speicherzellen modelliert – d. h. das Schieberegister besitzt nach außen nur die Aktionen `rein` bzw. `raus`; im Innern des Schieberegisters werden die Daten von einer Zelle zur nächsten weitergeschoben.

Zeichnen Sie zuerst ein Strukturdiagramm und testen Sie dann Ihre Variante für $N \leq 4$ im Analysetool LTSA.

Hinweis: In FSP kann der Befehl `forall` sowohl in der parallelen Komposition als auch beim Relabelling verwendet werde.

4.6

a) Modifizieren Sie den Client-/Server-Prozess aus der Vorlesung so, dass mehr als ein Client den Dienst des Servers abrufen kann.

b) Ändern Sie Ihren Prozess aus dem ersten Aufgabenteil so, dass die `call`-Aktion der Clients statt mit einer Antwort des Servers auch mit einem Timeout beendet werden kann.

4.7

Ein Museum besitzt einen Eingang und einen Ausgang, die jeweils durch ein Drehkreuz gesichert und mit einer zentralen Steuerung verbunden sind. Zur Öffnungszeit gibt der Museumswärter an der zentralen Steuerung den Eingang ins Museum frei, woraufhin die beiden Drehkreuze entsperrt werden. Nach Ablauf der Besuchszeit sperrt er an der zentralen Steuerung den Eintritt weiterer Besucher. Nach der Schließung kann nur noch das Drehkreuz am Ausgang passiert werden (solange sich noch Besucher in den Räumen befinden). Da es sich um ein sehr kleines Museum handelt, erlaubt die Steuerung auch nur N Besuchern gleichzeitig, sich in den Räumen aufzuhalten.

Modellieren Sie das Museum durch die parallele Komposition der Prozesse `Eingang`, `Ausgang`, `Steuerung` und `Waerter`.

Hochschule Bremerhaven
Prof. Dr. Thomas Umland

Bremerhaven, 21. November 2024

Parallelprogrammierung

Aufgabenblatt 5

5.1

Gegeben sei die folgende Klasse:

```
public class HelloClass {
  private String greeting;
  public HelloClass(String name) {
    this.greeting = ">>> Hello " + name + "! <<<";
  }
  public void printNormal() {
    printIt(greeting);
  }
  public void printLower() {
    printIt(greeting.toLowerCase());
  }
  public void printUpper() {
    printIt(greeting.toUpperCase());
  }
  protected void printIt(String str) {
    for (int i = 0; i < str.length(); i++) {
      try {
        Thread.sleep(Math.round(Math.random()) * 100);
      } catch (Exception e) { }
      System.out.print(str.substring(i, i + 1));
    }
    System.out.println();
  }
}
```

Implementieren Sie zusätzlich die drei Threads `HelloNormalThread`, `HelloLowerThread` und `HelloUpperThread`. Diese Klassen bekommen im Konstruktor jeweils eine Instanz von `HelloClass` übergeben und führen in ihrer `run`-Methode die Methode `printNormal`, `printLower` bzw. `printUpper` der übergebenen `HelloClass`-Instanz aus.

Schreiben Sie jetzt noch eine weitere Klasse `HelloClassMain`, die nur eine `main`-Methode enthält, in der zunächst *eine* Instanz von `HelloClass` angelegt wird. Mit dieser werden dann drei Threads erzeugt (je eine Instanz von `HelloNormalThread`, `HelloLowerThread` bzw. `HelloUpperThread`) und nebenläufig gestartet.

a) Führen Sie `HelloClassMain` aus. Was wird auf der Konsole ausgegeben?

b) Durch welche Maßnahmen können Sie erreichen, dass die Ausgabe jedes Threads in einer eigenen Zeile ausgegeben wird? Erweitern Sie die Klasse `HelloClass` durch `HelloClassAtomic extends HelloClass` und setzen Sie dort ihre Maßnahmen um. Ersetzen Sie jetzt in `HelloClassMain` die Instanz von `HelloClass` durch `HelloClassAtomic`. Wie sieht die Ausgabe jetzt aus? Haben Ihre Maßnahmen gegriffen?

5.2

Das Reservierungssystem eines Theaters besteht aus einem zentralen Rechner, an den Reservierungsterminals angeschlossen sind. Die Reservierungsmitarbeiter können sich jederzeit auf ihrem Bildschirm die freien Plätze anzeigen lassen. Um einen Platz zu reservieren, wählt der Mitarbeiter einen Platz aus, gibt die Nummer an seinem Terminal ein und druckt das Ticket aus.

Konstruieren Sie ein solches Reservierungssystem, das eine freie Platzwahl von jedem Terminal erlaubt aber Doppelbelegungen der Plätze aufgrund gleichzeitiger Anforderung verschiedener Mitarbeiter ausschließt (Race Hazards). Modellieren Sie Ihr Reservierungssystem in FSP und zeigen Sie, dass bei Ihnen keine Doppelbelegungen vorkommen können.

Hinweis: Es reicht aus, wenn Sie eine feste Anzahl von Terminals bzw. Plätzen vorsehen (z. B. jeweils zwei).

5.3

Implementieren Sie Ihr Reservierungssystem aus Aufgabe 5.2 in Java.

a) Erstellen Sie zunächst eine korrekte Implementierung, in der keine Doppelbelegungen vorkommen können.

b) Verändern Sie Ihre Implementierung dann so, dass auch Race Hazards – d. h. Doppelbelegungen – möglich sind.

5.4

Rekursive Locks werden in Java dadurch realisiert, dass jeder Lock zählt, wie oft er durch denselben Thread gesperrt wurde. Erst wenn die Anzahl der Freigaben mit der Anzahl der Sperrungen übereinstimmt, kann das entsprechende Objekt durch einen anderen Thread gesperrt werden.

Gegeben seien die folgenden Deklarationen:

```
const N = 3
range C = 0..N  // Wertebereich eines Lock-Zaehlers
range P = 1..2  // Namen (Wertebereich) der Threads
```

Modellieren Sie die rekursiven Locks von Java als FSP `RecursiveLock` mit dem Alphabet `{acquire[p:P], release[p:P]}`, wobei `acquire[p]` den Lock für den Prozess `p` reserviert und `release[p:P]` ihn wieder freigibt.

Hochschule Bremerhaven
Prof. Dr. Thomas Umland

Bremerhaven, 28. November 2024

Parallelprogrammierung

Aufgabenblatt 6

6.1

In der Vorlesung wurde die Implementierung eines Parkplatzmodells mithilfe eines Monitors vorgestellt.

a) Testen Sie diese Implementierung für eine Parkplatzgröße von 5 Plätzen im Zusammenspiel mit den beiden Threads `Zufahrt` und `Ausfahrt`, die die Ein- bzw. Ausfahrt von jeweils 25 Fahrzeugen simulieren. Bauen Sie in diesen Threads zwischen den Ein- bzw. Ausfahrten der Fahrzeuge unterschiedlich große Wartezeiten durch `sleep` ein.

b) Kann in diesem konkreten Programm die Anweisung `notifyAll()` gefahrlos durch `notify()` ersetzt werden? Warum?

c) Was passiert, wenn die Anzahl der einfahrenden und die der ausfahrenden Fahrzeuge nicht übereinstimmt?

6.2

In der aus dem FSP-Modell abgeleiteten Javaimplementierung des beschränkten Puffers („bounded buffer“) aus der Vorlesung beginnen die `get`- und `put`-Methoden entsprechend den Transformationsregeln jeweils mit einer Anweisung der Form

`while (!` *wächter*`) wait();`

Diese Anweisung stellt sicher, dass der Methodenrumpf nur dann ausgeführt wird, wenn die Bedingung des „Wächters“ erfüllt ist.

Wie wirken sich die folgenden Änderungen dieser Programmzeile auf die Programmlogik aus?

a) `while (!` *wächter*`) wait(99);`

b) `while (!` *wächter*`) sleep(99);`

c) `while (!` *wächter*`) yield();`

d) `if (!` *wächter*`) wait();`

6.3

Eine Horde Kannibalen isst gemeinschaftlich von einem Büfett, auf dem bis zu M gekochte Missionare liegen. Wenn ein Kannibale Hunger hat, geht er zu dem Büfett und bedient sich. Es sei denn, das Büfett ist leer. In diesem Fall muss der Kannibale warten bis das Büfett wieder gefüllt wird. Wenn der Koch sieht, dass das Büfett leer ist, füllt er es mit M frisch gekochten Missionaren wieder auf. Das Verhalten der Kannibalen und des Kochs ist durch folgende Prozesse beschrieben:

```
const M = 5         // Kapazitaet des Bueffets
set K = {a, b, c}   // Menge der Kannibalen

Kannibale = (gehEssen -> binZufrieden -> Kannibale).

Koch    = (koche -> fuelleBueffet -> Koch).
```

Modellieren Sie auch das Büfett als FSP und implementieren Sie anschließend die Kannibalenspeisung in Java.

6.4

In FSP können *beliebig viele* Prozesse über eine *einzige* Aktion synchronisiert werden. Gehört eine Aktion `sync` zum Alphabet einer Menge von N Prozessen, so können sie alle erst dann weiterarbeiten, wenn jeder von ihnen diese gemeinsame `sync`-Aktion ausgeführt hat.

Eine solche Synchronisation mehrerer Prozesse durch eine gemeinsame „Sperre“ oder „Barriere“ wird dementsprechend als *Barrier-Synchronisation* bezeichnet.

a) Die Java-Klasse `java.util.concurrent.CyclicBarrier` realisiert eine solche Barriere. Setzen Sie das folgende Modell eines Fertigungssystems mithilfe von `CyclicBarrier` in Java um:

```
ProdA = (prodA -> fertig -> montiert -> ProdA).
ProdB = (prodB -> fertig -> montiert -> ProdB).
ProdC = (prodC -> fertig -> montiert -> ProdC).
||Produktion = (ProdA || ProdB || ProdC).

Montage = (fertig -> montage -> montiert -> Montage).
||Fertigung = (Produktion || Montage).
```

In der Java-Implementierung soll jeder der Produktionsprozesse `ProdA`, `ProdB` und `ProdC` jeweils 20 Teile produzieren und danach terminieren. Verwenden Sie den Konstruktor mit der Signatur `CyclicBarrier(int parties)`.

b) Wie sieht eine Implementierung des Fertigungssystems aus, die den Konstruktor mit der Signatur `CyclicBarrier(int parties, Runnable barrierAction)` verwendet?

Hochschule Bremerhaven
Prof. Dr. Thomas Umland

Bremerhaven, 9. Januar 2025

Parallelprogrammierung

Aufgabenblatt 7

7.1

In der Vorlesung wurde die Implementierung einer Klasse `Semaphor` vorgestellt. Erfüllt die folgende Implementierung den gleichen Zweck? Warum?

```
public class Semaphor {
  private int wert;

  public Semaphor(int initial){
    this.wert = initial;  // Semaphor initialisieren
  }

  public synchronized void up() {
    wert++;
    notify();
  }

  public synchronized void down()
         throws InterruptedException {
    while (!(wert > 0)) { wait(); }
    wert--;
  }
}
```

7.2

Implementieren Sie eine Lösung des in der Vorlesung vorgestellten Problems der „Dinierenden Philosophen“ in Java:

> *Fünf Philosophen sitzen an einem runden Tisch. In der Mitte steht eine Schüssel mit Spagetti. Die Philosophen sind genügsam, daher verzichten sie auf Teller und besitzen auch nur fünf Gabeln, die sie zwischen sich legen. Somit hat jeder Philosoph rechts und links von sich je eine Gabel liegen.*
>
> *Zum Essen benutzt jeder Philosoph allerdings immer beide neben ihm liegenden Gabeln, so dass er eventuell warten muss, wenn er Essen möchte, weil eine Gabel gerade durch seinen Nachbarn verwendet wird.*
>
> *Der Ablauf des Abendessens wird durch die für jeden Philosophen immer wiederkehrende Abfolge der Aktionen „Denken“, „Essen“, „Denken“, „Essen“, „Denken“, ... bestimmt.*

a) Realisieren Sie zunächst eine „klassische“ Lösung, bei der jeder Philosoph erst die Gabel zu seiner Linken und dann die zu seiner Rechten aufnimmt und somit Verklemmungen (Deadlocks) auftreten können.

b) Verbessern Sie Ihr Programm aus Teil a) so, dass die für das Auftreten von Verklemmungen notwendige Bedingung der zyklischen Abhängigkeit durch Ordnen der Ressourcenanforderungen (Gabel mit der kleineren Nummer zuerst nehmen) ausgeschlossen wird.

Alternativ können auch die Philosophen mit einer geraden Nummer die linke und die mit ungerader Nummer die rechte Gabel zuerst nehmen.

c) Erstellen Sie eine weitere Deadlock-freie Variante Ihres Programmes aus Teil a), in der eine bereits belegte Ressource wieder freigegeben wird, falls nach einer gewissen Wartezeit (Timeout) die noch fehlende Ressource nicht belegt werden kann.

d) Wie könnten Sie sicherstellen, dass eine Ressource (Gabel) nur von dem Prozess freigegeben (abgelegt) werden kann, der sie auch belegt (aufgenommen) hat?

e) Eine der in der Vorlesung genannten Deadlock-Vermeidungsstrategien war die Einführung eines Butlers, der maximal vier Philosophen gleichzeitig an dem Tisch Platz nehmen lässt. Implementieren Sie diese Variante.

f) Wie könnte eine Modellierung der Butler-Variante aussehen? Weisen Sie mithilfe von LTSA nach, dass hier wirklich kein Deadlock auftritt.

7.3

a) In dem folgenden System kann ein Deadlock auftreten. Geben Sie einen Trace an, der zu einem Deadlock führt und identifizieren Sie die vier in der Vorlesung genannten Deadlock-Bedingungen:

```
Alice = (call.bob   -> wait.chris -> Alice).
Bob   = (call.chris -> wait.alice -> Bob).
Chris = (call.alice -> wait.bob   -> Chris).

||S = (Alice || Bob || Chris) /{call/wait}.
```

b) Die folgende Modellierung versucht die Verklemmungsgefahr auszuschalten, indem jeder Person die Möglichkeit einer „Auszeit“ während des ersten Anrufes eingeräumt wird. Ist die Verklemmungsgefahr wirklich gebannt? Begründen Sie Ihre Aussage.

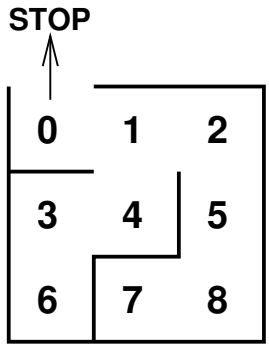
```
AliceT = (call.bob   -> wait.chris -> AliceT
         |timeout.alice -> wait.chris -> AliceT).
BobT   = (call.chris -> wait.alice -> BobT
         |timeout.bob -> wait.alice -> BobT).
ChrisT = (call.alice -> wait.bob   -> ChrisT
         |timeout.chris -> wait.bob -> ChrisT).

||ST = (AliceT || BobT || ChrisT) /{call/wait}.
```

7.4

a) Modellieren Sie die Bewegungsmöglichkeiten innerhalb des in der Abbildung dargestellten Labyrinths in FSP. Definieren Sie dabei den Ausgang explizit als Übergang in den Fehlerzustand `STOP` (die Übergänge zwischen den einzelnen Feldern können z. B. mithilfe der Himmelsrichtungen beschrieben werden):

```
Labyrinth(Start=8) = Feld[Start],
...
```

```
||Suche = Labyrinth(7).
```



Verwenden Sie jetzt den „Deadlock-Checker“ des Analysetools LTSA, um von einem beliebigen Feld automatisch einen Weg zum Ausgang (d. h. in FSP eine Aktionsfolge zum Zustand `STOP`) suchen zu lassen.

b) Wie können Sie Wege von verschiedenen Startpositionen gleichzeitig suchen lassen – z. B. von `Labyrinth(7)` und `Labyrinth(6)`?

(Hinweis: Verwenden Sie hierzu die parallele Komposition.)

7.5

In einer Banksoftware sind Kontodaten in einer Klasse `Konto` abgebildet. Die Abstraktion von Geldbeträgen erfolgt über die Klasse `Betrag`.

Zum Gutschreiben bzw. Abbuchen eines Betrags, besitzt `Konto` die Methoden `public synchronized void gutschreiben(Betrag betrag)` resp. `public synchronized void abbuchen(Betrag betrag)`.

Um zu überprüfen, ob das Abbuchen eines bestimmten Betrags im Rahmen des Überziehungskredits überhaupt möglich ist, ist eine interne Methode `protected synchronized boolean istGedeckt(Betrag betrag)` implementiert.

Zum Überweisen von Geldbeträgen zwischen zwei Konten der Bank wird schließlich folgende Methode verwendet:

```
public synchronized void ueberweisenNach (Konto nach,
                                          Betrag betrag) {
  if (istGedeckt(betrag){
    abbuchen(betrag);
    nach.gutschreiben(betrag);
  } else {
    // spielt hier keine Rolle
  }
}
```

Bewerten Sie diese Implementierung.

7.6

Versuchen Sie anhand eines selbst geschriebenen Java-Programms das Problem der Prioritätsumkehr nachzubilden bzw. nachzuweisen, dass die von Ihnen benutzte JVM Prioritätsvererbung zur Lösung des Problems einsetzt.