Ludwig-Maximilians-Universität München
Institut für Informatik
Prof. Dr. R. Hennicker, A. Klarl, M. Busch

SoSe 2013
18.07.2013, 11:30–13:30 Uhr

Klausur: Lösungsvorschlag
Software Engineering für spezielle Anwendungsgebiete: Entwurf und Implementierung paralleler Programme

Nachname .....

Vorname .....

Matrikelnummer .....

Studienfach .....

Angestrebter Abschluss .....

Fachsemester .....

Hilfsmittel jeder Art sind nicht zugelassen. Schreiben Sie Ihren Namen und Ihre Matrikelnummer auf jedes Blatt. Die Klausurangabe mit allen Lösungen und zur Lösung verwendete Blätter sind in **jedem Fall nach der Klausur wieder abzugeben**.

Diese Klausur soll gewertet werden: ☐**JA** ☐**NEIN**
Hinweis: Wenn keines der beiden Felder angekreuzt ist, wird JA angenommen.

Mit meiner Unterschrift erkläre ich die Richtigkeit und Vollständigkeit der obigen Angaben.

.....

Nicht von dem/der Studierenden auszufüllen

1	2	3	4	5	Σ	Note
von 14	von 15	von 16	von 10	von 15	von 70	

Aufgabe 1 **FSP und schwache Bisimulation** **6 + 4 + 1 + 3 = 14 Punkte**

Ein Getränkeautomat bietet Tee und Kaffee zur Auswahl an. Der Automat ist zu Beginn voll gefüllt mit `FullTea` Tassen Tee und `FullCoffee` Tassen Kaffee. Solange Tee vorhanden ist, kann der blaue Knopf gedrückt werden (Aktion `blue`), woraufhin eine Tasse Tee ausgegeben wird (Aktion `tea`). Solange Kaffee vorhanden ist, kann der rote Knopf gedrückt werden (Aktion `red`), woraufhin eine Tasse Kaffee ausgegeben wird (Aktion `coffee`). Wenn kein Tee mehr vorhanden ist, kann der Automat wieder mit Tee vollständig aufgefüllt werden (Aktion `filltea`); wenn kein Kaffee mehr vorhanden ist, kann der Automat wieder mit Kaffee vollständig aufgefüllt werden (Aktion `fillcoffee`).

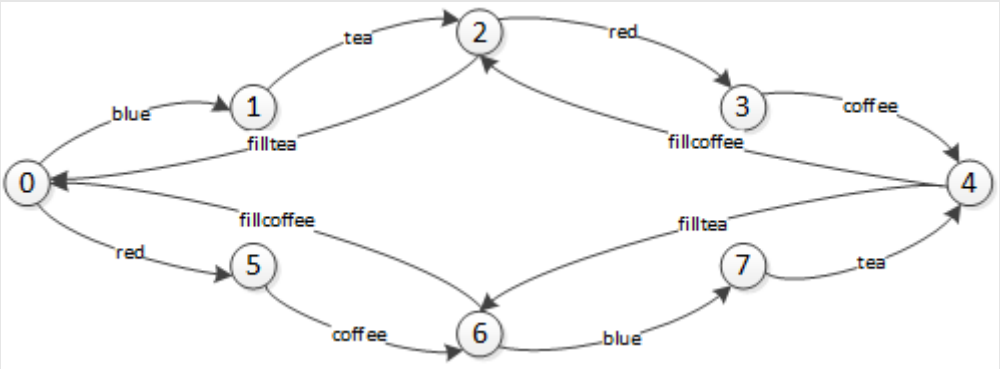
a) Modellieren Sie den Getränkeautomaten durch einen parametrisierten FSP-Prozess `DRINKS` mit den beiden Parametern `FullTea` und `FullCoffee`.

Lösung:

```
DRINKS(FullTea = 1, FullCoffee = 1) = D[FullTea][FullCoffee],
D[t:0..FullTea][c:0..FullCoffee] =
   ( when (t>0) blue -> tea -> D[t-1][c]
   | when (c>0) red -> coffee -> D[t][c-1]
   | when (t==0) filltea -> D[FullTea][c]
   | when (t==0) fillcoffee -> D[t][FullCoffee] ).
```

b) Betrachtet wird der Fall, in dem der Automat nur eine Tasse Tee und eine Tasse Kaffee fasst. Geben Sie das LTS des Prozesses `DRINKS(1,1)` an.

Lösung:
1 Punkt je Zyklus

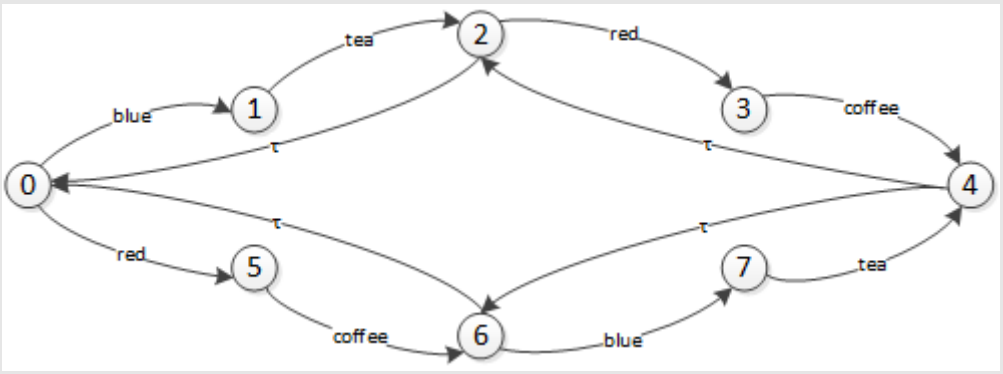


c) Unter den Voraussetzungen von Teil b) werden nun die Aktionen `filltea` und `fillcoffee` verborgen, d.h. wir betrachten den Prozess

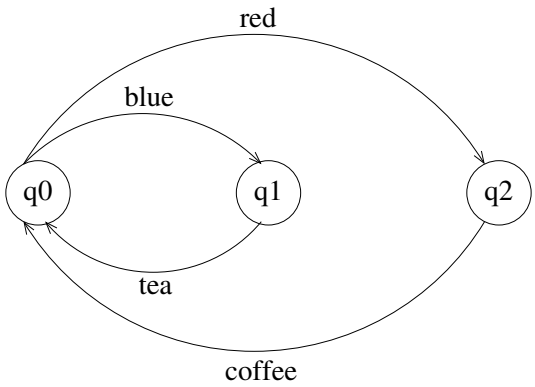
`DRINKS(1,1)`$\backslash\{$`filltea,fillcoffee`$\}$.

Geben Sie das LTS dieses Prozesses an.

Lösung:
1 Punkt auf Ersetzung durch τ



d) Nach Minimalisierung des LTSes aus Teil c) bzgl. beobachtbarer Äquivalenz ergibt sich folgendes LTS mit Anfangszustand `q0`:



Geben Sie eine schwache Bisimulationsrelation zwischen den Zuständen der beiden LTSe aus Teil c) und Teil d) an, die die beiden Anfangszustände enthält. (Ein Beweis, dass die angegebene Relation die Eigenschaften einer schwachen Bisimulation erfüllt, ist nicht nötig.)

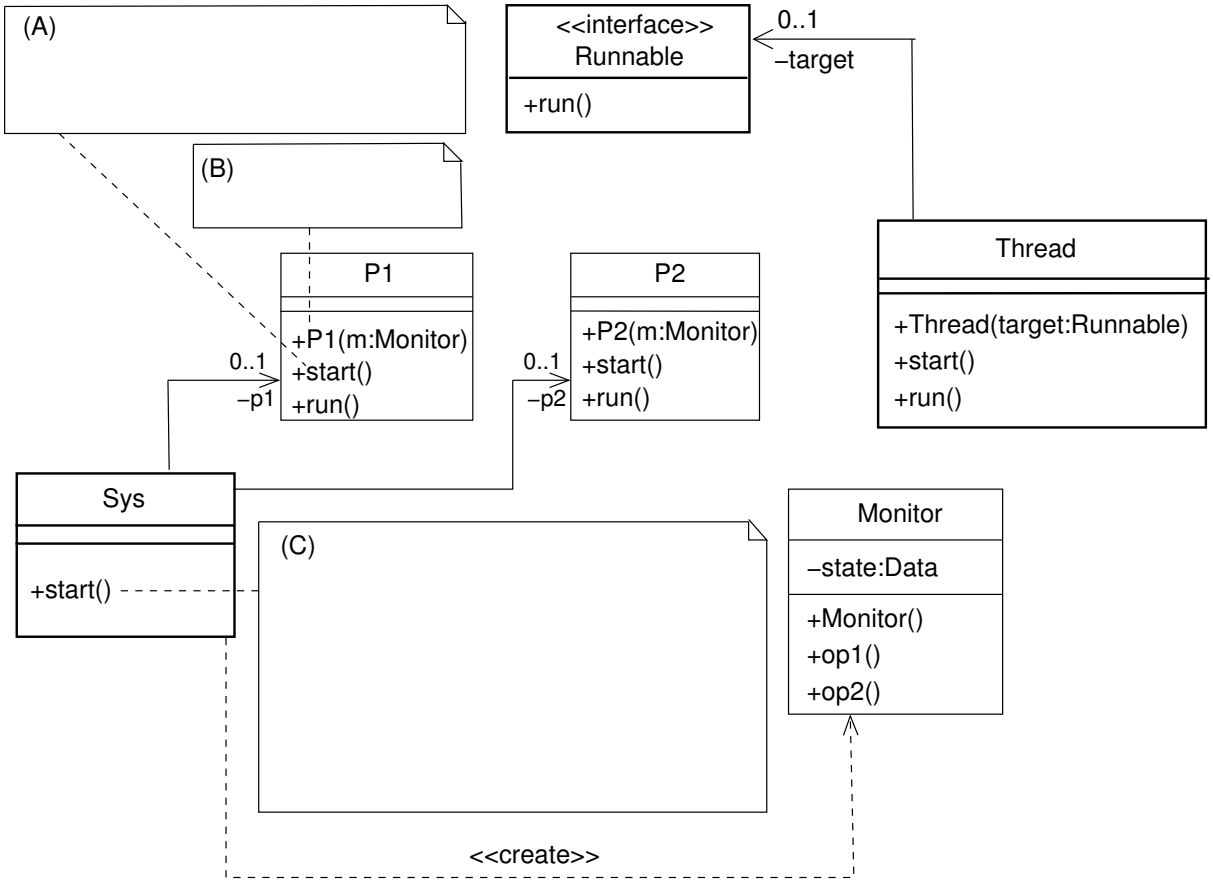
Lösung:
// 1 Punkt je Zustand im minimalisierten LTS
Relation: $\{(0, q_0), (2, q_0), (4, q_0), (6, q_0), (1, q_1), (7, q_1), (3, q_2), (5, q_2)\}$

Aufgabe 2 **Monitore** **3 + 5 + 7 = 15 Punkte**

Gegeben sei das in der Vorlesung angegebene Schema zur Modellierung von Synchronisationsbedingungen in FSP. Die Synchronisationsbedingungen für die beiden (aktiven) Prozesse `P1` und `P2` werden von einem `Monitor` überwacht.

```
P1 = (op1 -> P1).
P2 = (op2 -> P2).

MONITOR = MON[init],
MON[state:Data] =
   ( when (cond1) op1 ->  MON[nextState1]
   | when (cond2) op2 -> MON[nextState2] ).

||SYS = (P1 || P2 || MONITOR).
```

Das folgende Klassendiagramm zeigt ein Fragment des Schemas zur Implementierung von Synchronisationsbedingungen in Java.

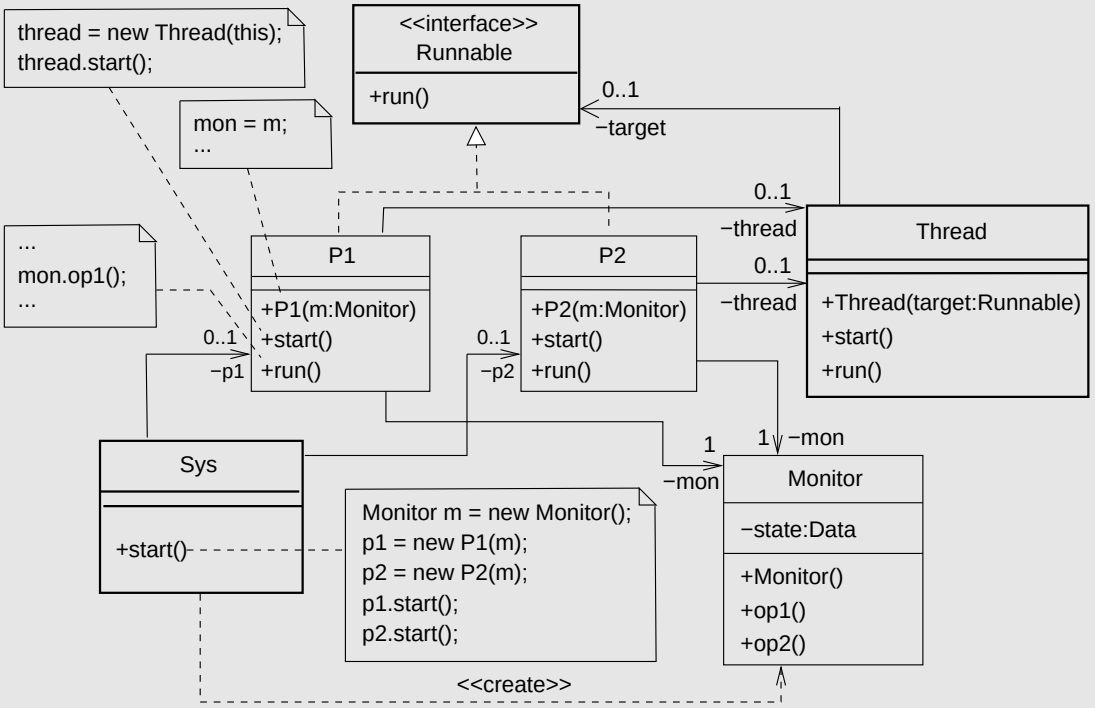


a) Tragen Sie die fehlenden Implementierungsbeziehungen (in Form von UML-Realisierungsbeziehungen) sowie die fehlenden gerichteten Assoziationen (mit Rollennamen und Multiplizitäten) in das Klassendiagramm ein.

Lösung:
1 Punkt für Realisierungsbeziehung,
1 Punkt für beide Assoziationen zu `Thread`,
1 Punkt für beide Assoziationen zu `Monitor`

b) Tragen Sie an den Stellen (A), (B) und (C) den Java-Code der jeweiligen Methoden bzw. Konstruktoren ein. Die Methode `start` der Klasse `P1` ist für die Erzeugung des Threads, der zur Ausführung der `run`-Methode von `P1` führt, verantwortlich.

Lösung:
1 Punkt für (A)
1 Punkt für (B)
3 Punkte für (C):
mit 1 Punkt Intialisierung von `Monitor`,
mit 1 Punkt Initialisierung von `P1` und `P2`,
mit 1 Punkt Aufruf von `start`



c) Der `Monitor` wird nun folgendermaßen konkretisiert:

```
MONITOR = MON[5],
MON[state:0..5] =
   ( when (state > 0) op1 -> MON[state-1]
   | when (state < 5) op2 -> MON[state+1] ).
```

Schreiben Sie eine Java-Klasse, die den Monitor implementiert. Dabei soll die in der Vorlesung angegebene Regel zur Implementierung von Synchronisationsbedingungen in Monitoren verwendet werden.

Lösung:

```
1  public class Monitor {
2     private int state = 5; // 1 Punkt
3
4     // 1 Punkt f"ur synchronized
5     public synchronized void op1() throws InterruptedException {
6        // je 1 Punkt f"ur while, condition und wait
7        while(!(state>0)) wait();
8        // auch ok: while(state==0) wait();
9        state--; // 1 Punkt
10       notifyAll(); // 1 Punkt
11    }
```

```
12
13     public synchronized void op2() throws InterruptedException {
14        while(!(state<5)) wait();
15        // auch ok: while(state==5) wait();
16        state++;
17        notifyAll();
18     }
19 }
```

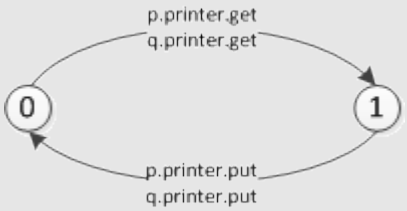
Aufgabe 3 **Deadlocks** **3 + 2 + 4 + 4 + 3 = 16 Punkte**

Es seien die folgenden FSP-Prozesse gegeben:

```
RESOURCE = (get -> put -> RESOURCE).
P = (printer.get -> scanner.get -> copy -> printer.put -> scanner.put -> P).
Q = (scanner.get -> printer.get -> copy -> scanner.put -> printer.put -> Q).
||SYS = (p:P || q:Q  || {p,q}::printer:RESOURCE || {p,q}::scanner:RESOURCE).
```

a) Geben Sie das Alphabet und das LTS des Prozesses {p,q}::printer:RESOURCE an.

Lösung:
Alphabet: { p.printer.get, p.printer.put, q.printer.get, q.printer.put } (1 Pkt)
LTS (2 Punkte):



b) Geben Sie zwei minimale Abläufe zu einem Deadlock des Prozesses SYS an.

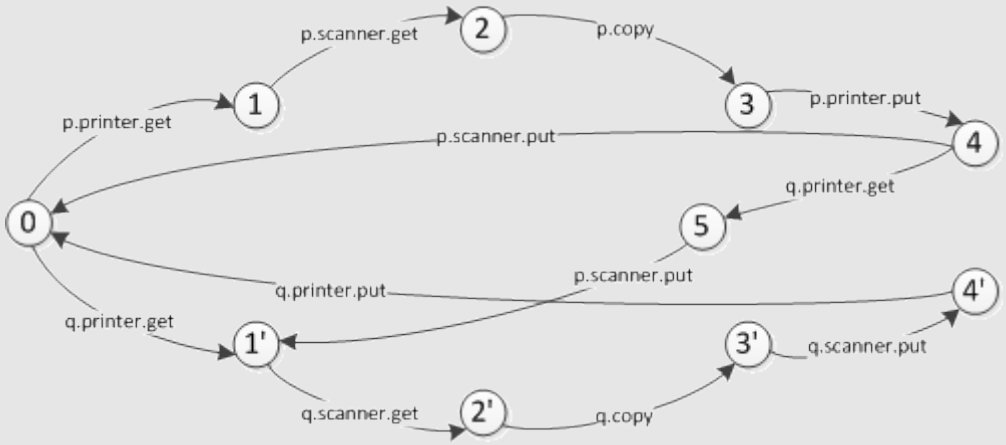
Lösung:
Trace 1: p.printer.get, q.scanner.get (1 Punkt)
Trace 2: q.scanner.get, p.printer.get (1 Punkt)

c) Das System SYS wird nun modifiziert, indem der Prozess Q durch den folgenden Prozess Q1 ersetzt wird:

```
Q1 = (printer.get -> scanner.get -> copy -> scanner.put -> printer.put -> Q1).
||SYS1 = (p:P || q:Q1  || {p,q}::printer:RESOURCE || {p,q}::scanner:RESOURCE).
```

Besitzt der Prozess SYS1 ein Deadlock? Begründen Sie Ihre Antwort, indem Sie das LTS von SYS1 angeben.

Lösung:
Es tritt kein Deadlock auf. (1 Punkt)
LTS (je 1 Punkt für die Zyklen, 1 Punkt für Übergang zwischen den Zyklen):



d) In einer anderen Modifikation des Systems `SYS` werden die Prozesse `P` und `Q` durch die beiden folgenden Prozesse `P2` und `Q2` ersetzt:

```
P2 = (printer.get -> GETSCANNER),
GETSCANNER = ( scanner.get -> copy -> printer.put -> scanner.put -> P2
             | timeout -> printer.put -> P2).
Q2 = (scanner.get -> GETPRINTER),
GETPRINTER = ( printer.get -> copy -> scanner.put -> printer.put -> Q2
             | timeout -> scanner.put -> Q2).
||SYS2 = (p:P2 || q:Q2 || {p,q}::printer:RESOURCE || {p,q}::scanner:RESOURCE).
```

Erläutern Sie, warum der Prozess `SYS2`, im Gegensatz zum Prozess `SYS`, kein Deadlock hat! Welche der vier (in der Vorlesung angegebenen) notwendigen und hinreichenden Bedingungen für Deadlocks wurde hier verletzt?
Welchen Nachteil kann der Prozess `SYS2` jedoch haben, wenn wir beliebige Abläufe (auch unfaire) betrachten? Skizzieren Sie einen unendlichen Ablauf, der diesen Nachteil belegt!

Lösung:
Nach einer bestimmten Zeitspanne des Wartens werden die Ressourcen durch einen Timeout wieder freigegeben (1 Punkt). Es wurde die Bedingung "No Preemption"verletzt (1 Punkt).

Es besteht allerdings die Gefahr, dass kein echter Fortschritt passiert (1 Punkt).
Trace (1 Punkt): `p.printer.get`, `q.scanner.get`, `p.timeout`, `p.printer.put`, `p.printer.get`, `p.timeout`, ...

e) Wir betrachten weiterhin das System `SYS2` aus Teilaufgabe d).
Geben Sie in FSP-Syntax zwei Fortschrittseigenschaften an, so dass bei Erfüllung *beider* Fortschrittseigenschaften sichergestellt ist, dass in jedem fairen Ablauf von `SYS2` sowohl die Aktion `p.copy` als auch die Aktion `q.copy` unendlich oft vorkommen.
Erfüllt der Prozess `SYS2` beide Fortschrittsseigenschaften? Geben Sie eine informelle Begründung für Ihre Antwort an (Antworten ohne Begründung werden nicht gewertet).

Lösung:
`progress PCOPY = {p.copy}` (1 Punkt)
`progress QCOPY = {q.copy}` (1 Punkt)

Ja, `SYS2` erfüllt die beiden Fortschrittseigenschaften, da bei fairen Abläufen nicht erlaubt ist, dass ein Prozess ständig den `timeout`-Zweig wählt. (1 Punkt)

Aufgabe 4 **Sicherheitseigenschaften** **2 + 3 + 5 = 10 Punkte**

Gegeben seien ein Property-Prozess `P` sowie ein (gewöhnlicher) FSP-Prozess `Q` mit α`P` $\subseteq$ α`Q`.

a) Ergänzen Sie die folgende Definition einer legalen Aktionsfolge:

Sei `w` eine endliche oder unendliche Aktionsfolge über dem Alphabet α`P`.
`w` ist eine *legale Aktionsfolge bzgl.* `P`, wenn ...

Lösung:
... kein endlicher Anfang (1 Punkt) von `w` im LTS von P zum Fehlerzustand (1 Punkt) führt.

b) Ergänzen Sie die folgende Definition zur Erfüllung von Sicherheitseigenschaften:

Der Prozess `Q` *erfüllt* die Sicherheitseigenschaft `P`, geschrieben `Q` $\models$ `P`, wenn ...

Lösung:
... für alle Abläufe (1 Punkt) `w` von Q die Einschränkung $w|_{\alpha P}$ (1 Punkt) eine legale Aktionsfolge (1 Punkt) bzgl. P ist.

c) Seien `P` und `Q` wie oben mit α`P` $\subseteq$ α`Q`. Sei `R` ein weiterer (gewöhnlicher) FSP-Prozess mit α`P` $\subseteq$ α`R`. Es gelte`(Q||R)` $\models$ `P`. Gilt dann im Allgemeinen auch `Q` $\models$ `P`?
Beweisen Sie Ihre Antwort! Im positiven Fall ist die Definition der Erfüllung einer Sicherheitseigenschaft zu verwenden; im negativen Fall ist ein Gegenbeispiel anzugeben (Antworten ohne Beweis werden nicht gewertet).

Lösung:
Nein, `Q` $\models$ `P` gilt im Allgemeinen nicht (1 Punkt).

Gegenbeispiel:
`Q = a -> a -> STOP` (1 Punkt)
`R = a -> STOP` (1 Punkt)
`property P = a -> STOP` (1 Punkt)

Die Parallelschaltung `(Q || R)` erfüllt zwar die Sicherheiteigenschaft `P`, da `Q` das zweite `a` im Produkt nicht ausführen kann (1 Punkt). `Q` allein erfüllt die Sicherheitseigenschaft allerdings nicht.

Aufgabe 5 **Lebendigkeitseigenschaften** **2 + 7 + 6 = 15 Punkte**

Ein Parkhaus mit beschränkter Kapazität `MAX`, das potentiell von `N` Autos benutzt werden kann, wird folgendermaßen in FSP modelliert. Der Prozess `CONTROL` dient zur Kontrolle der Ein- und Ausfahrten. Im Gesamtsystem wird er mit dem aktuellen Parameter `2` (für ein Parkhaus mit zwei Plätzen) verwendet.

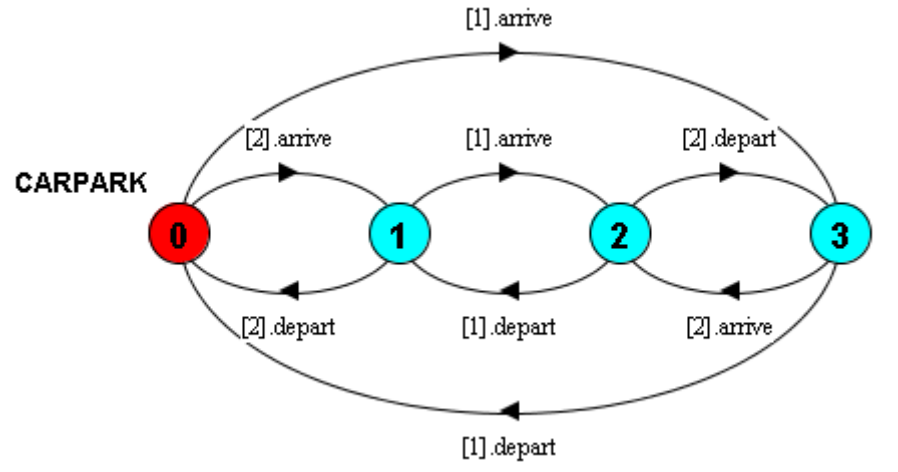
```
const N = 2
range ID = 1..N
CAR = (arrive -> depart -> CAR).
||CARS = [ID]:CAR.

CONTROL(MAX = 2) = CON[MAX],
CON[free:0..MAX] =
   ( when (free>0)   [ID].arrive -> CON[free-1]
   | when (free<MAX) [ID].depart -> CON[free+1]).

||CARPARK = (CARS || CONTROL(2)).
```

Wir betrachten folgende Fortschrittseigenschaften:

```
progress ARRIVE = {[ID].arrive}
progress DEPART = {[ID].depart}
progress ARRIVE1 = {[1].arrive}
progress DEPART1 = {[1].depart}
progress ARRIVE2 = {[2].arrive}
progress DEPART2 = {[2].depart}
```

a) Geben Sie das LTS des Prozesses `CARPARK` an.

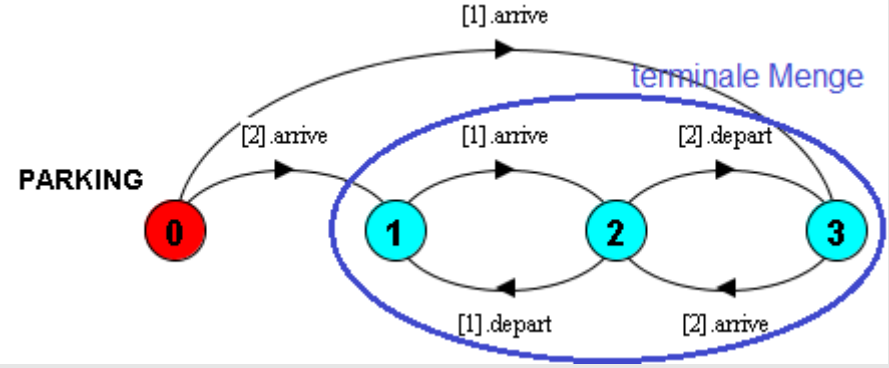
Lösung:
2 Punkte



b) Das Verlassen des Parkhauses soll für alle Autos niedrige Priorität haben. Modellieren Sie diesen Sachverhalt durch einen geeigneten FSP-Prozess `PARKING` und geben Sie das LTS von `PARKING` an. Untersuchen Sie unter Angabe der terminalen Mengen, welche der obigen Fortschrittseigenschaften der Prozess `PARKING` erfüllt.

Lösung:
`||PARKING = (CARPARK) >> {[ID].depart}.`
(1 Punkt für >>, 1 Punkt für `{[ID].depart}`)

LTS (3 Punkte: 1 Punkt pro weggelassener Transition, 1 Punkt für terminale Menge):

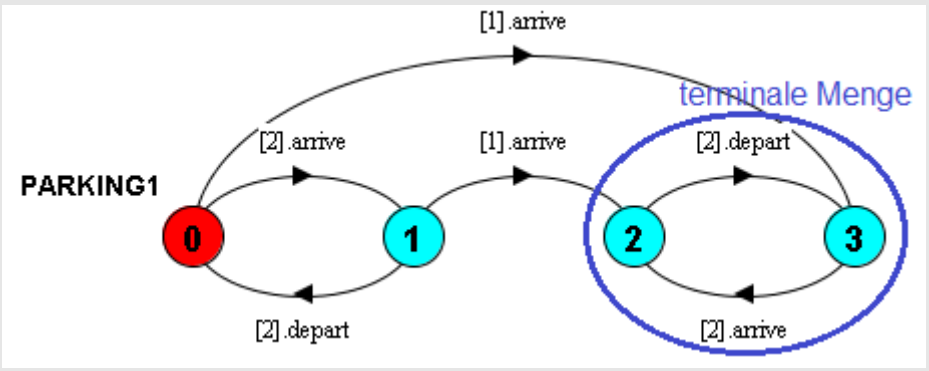


Der Prozess erfüllt alle Lebendigkeitseigenschaften (2 Punkte).

c) Das Verlassen der Parkhauses soll nun nur für das Auto `[1]:CAR` niedrige Priorität haben. Modellieren Sie diesen Sachverhalt durch einen geeigneten FSP-Prozess `PARKING1` und geben Sie das LTS von `PARKING1` an. Untersuchen Sie wieder unter Angabe der terminalen Mengen, welche der obigen Fortschrittseigenschaften der Prozess `PARKING1` erfüllt.

Lösung:
`||PARKING1 = (CARPARK) >> {[1].depart}.` (1 Punkt)

LTS (3 Punkte: 2 Punkte für weggelassene Transition, 1 Punkt für terminale Menge):



Der Prozess erfüllt alle Lebendigkeitseigenschaften außer `ARRIVE1` und `DEPART1` (2 Punkte).