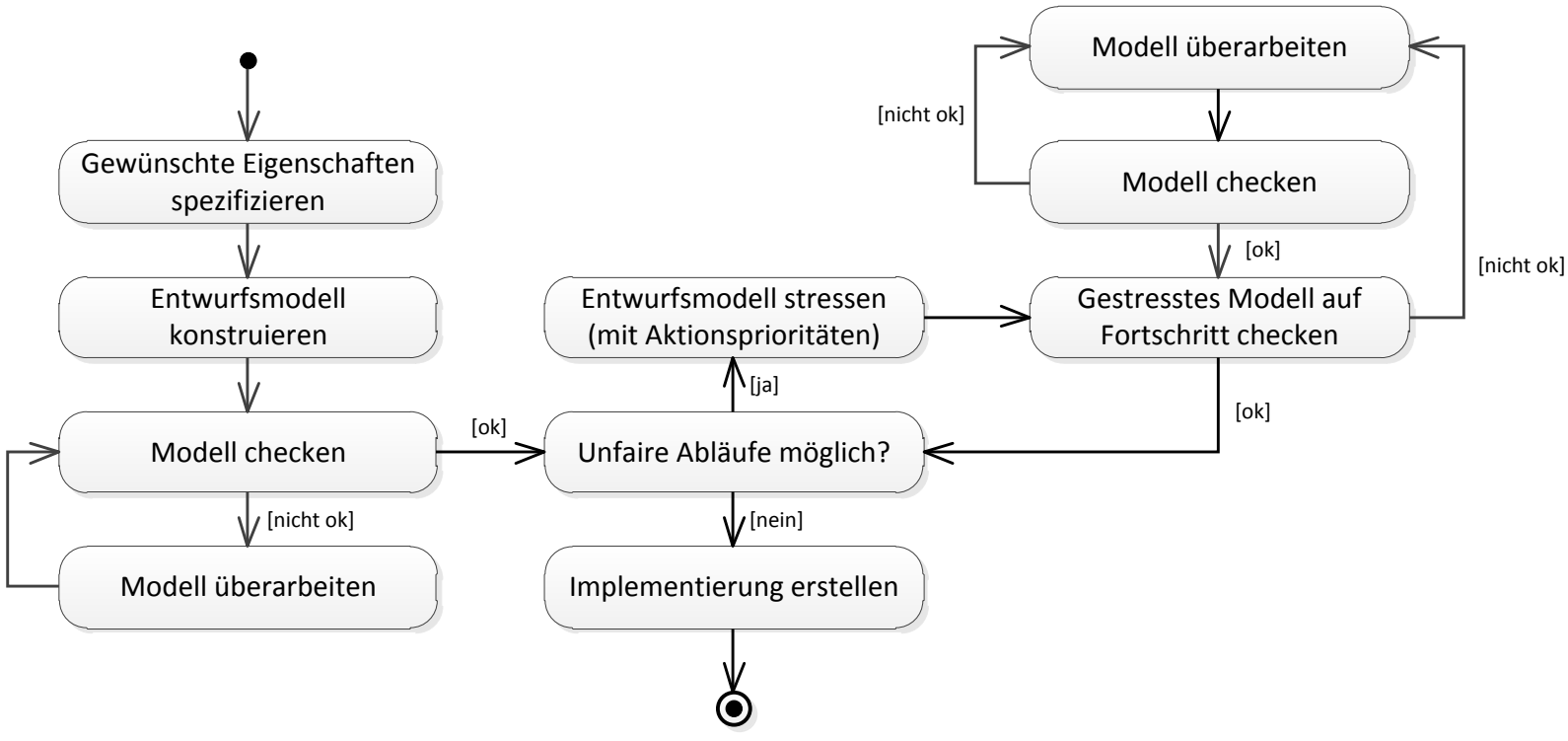
Entwurf und Implementierung paralleler Programme

Übungsblatt 10

Annabelle Klarl
(nach Folien von Dr. Sebastian Bauer, Dr. Philip Mayer)

8. Juli 2016

Entwicklungsmethodik unter Einbeziehung von Aktionsprioritäten



Folgende Eigenschaften sind zu überprüfen:
Deadlock, Sicherheitseigenschaften, Fortschrittseigenschaften

Aufgabe 1.

Gegeben sei das Leser- und Schreibersystem von Übungsblatt 9.

```
const Nread = 2
const Nwrite= 2
range Tread = 1..Nread
range Twrite = 1..Nwrite
const False = 0
const True  = 1
range Bool  = False..True

READER = (acquire->examine->release->READER).
WRITER = (acquire->modify->release->WRITER).

||RW = (reader[Tread]:READER || writer[Twrite]:WRITER).
```

```
RW_LOCK = RWL[0][False],
RWL[readers:0..Nread][writing:Bool] =
    ( when (!writing)
          reader[Tread].acquire -> RWL[readers+1][writing]
    | reader[Tread].release -> RWL[readers-1][writing]
    | when (readers==0 && !writing)
          writer[Twrite].acquire -> RWL[readers][True]
    | writer[Twrite].release -> RWL[readers][False]).

||RW_SYS = (RW || RW_LOCK).
```

RW_SYS erfüllt die Sicherheitseigenschaft SAFE_RW:

$$\text{RW\_SYS} \models \text{SAFE\_RW}$$

(Ein Schreiber darf nur dann auf den Datenbestand zugreifen, wenn kein anderer Schreiber und kein Leser auf den Datenbestand zugreift.)

(a) Welche Fortschrittseigenschaft soll das System erfüllen?

Aus Kap.8 der Vorlesung:
Fortschrittseigenschaften sind spezielle Lebendigkeitseigenschaften (progress properties"). Eine Fortschrittseigenschaft sichert zu, dass in jedem (fairen) Ablauf eines Programms ab jedem Zeitpunkt noch irgendwann eine spezifizierte Aktion ausgeführt wird.

Definition:

Sei F ein Name und sei $\{a_1, ..., a_n\} \subseteq$ Labels eine Menge von Aktionen ($n \geq 1$).
Dann definiert

$$\text{progress } F = \{a_1, ..., a_n\}$$

eine *Fortschrittseigenschaft*.

(a) Welche Fortschrittseigenschaft soll das System erfüllen?

- Es soll immer wieder ein Leser zugreifen können.
- Es soll immer wieder ein Schreiber zugreifen können.
- In FSP:

```
progress READ = {reader[Tread].acquire}
progress WRITE = {writer[Twrite].acquire}
```

- Stärker und präziser sind folgende Mengen von indizierten Fortschrittseigenschaften: *Jeder* Leser und *jeder* Schreiber soll immer wieder zugreifen können.

```
progress READ[i:Tread] = {reader[i].acquire}
progress WRITE[i:Twrite] = {writer[i].acquire}
```

Definition: Erfüllung einer Fortschrittseigenschaft

(aus Kap.8 der Vorlesung)

Sei Q ein Prozess und progress $F = \{a_1, ..., a_n\}$ eine Fortschrittseigenschaft. Q erfüllt F, geschrieben $Q \models F$, wenn in jedem fairen Ablauf von Q mindestens eine Aktion aus $\{a_1, ..., a_n\}$ unendlich oft vorkommt.

Idee zum Nachweis einer Fortschrittseigenschaft:

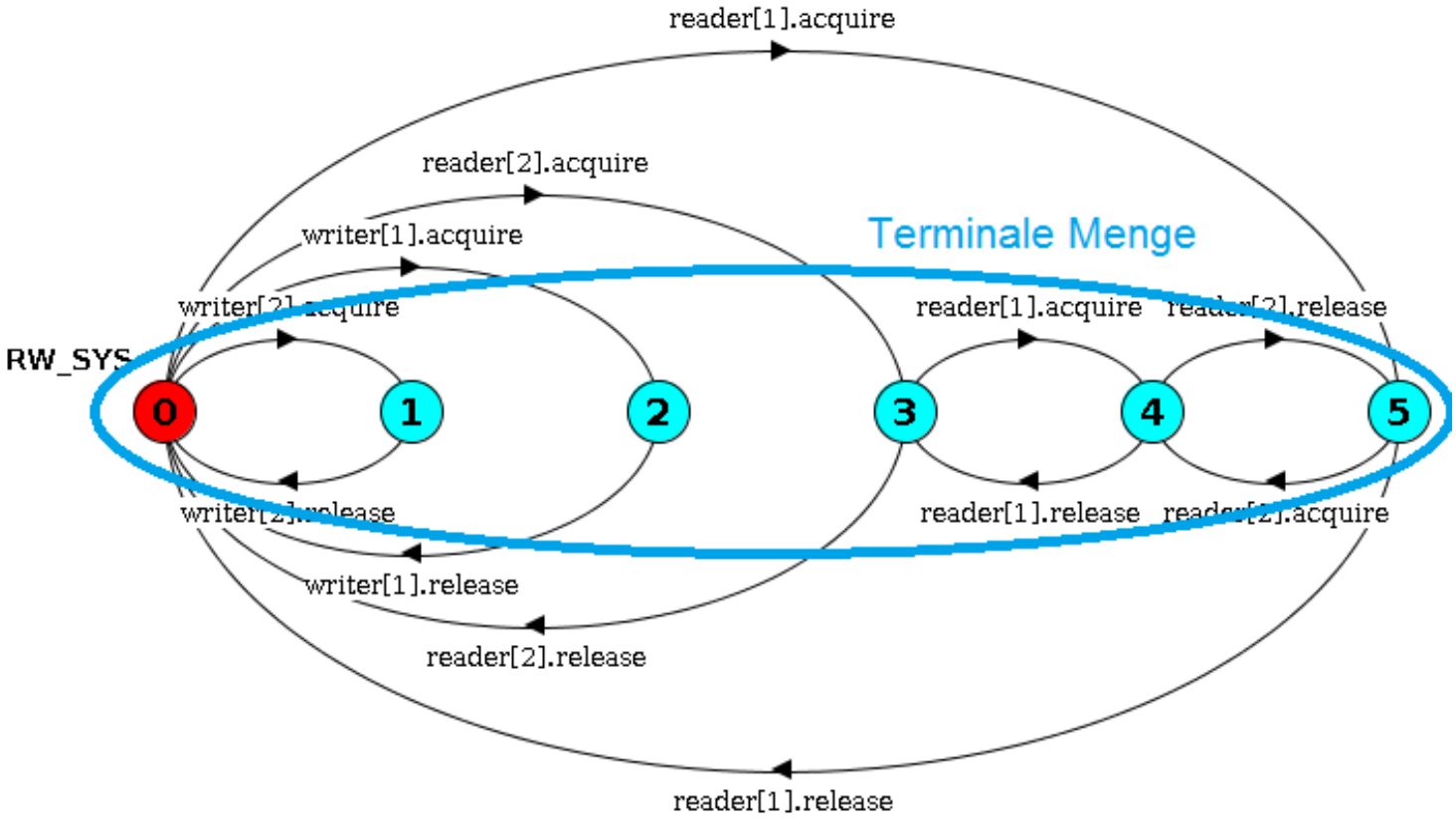
Suche “terminale“ Mengen von Zuständen und überprüfe, welche Aktionen dort möglich sind.

Definition:

Sei Q ein Prozess mit $\text{lts}(Q) = (S, A, \Delta, q_0)$. Eine *terminale Menge* von Zuständen von Q ist eine nichtleere Teilmenge $T \subseteq S$, für die gilt:

1. Ist $s \in T$ und $(s, a, s') \in \Delta$, dann ist $s' \in T$ (d.h. T ist abgeschlossen unter Transitionen).
2. Jeder Zustand $s \in T$ ist (durch eine Aktionsfolge) von jedem anderen Zustand $s' \in T$ erreichbar.

Minimiertes LTS von RW_SYS mit terminaler Menge:

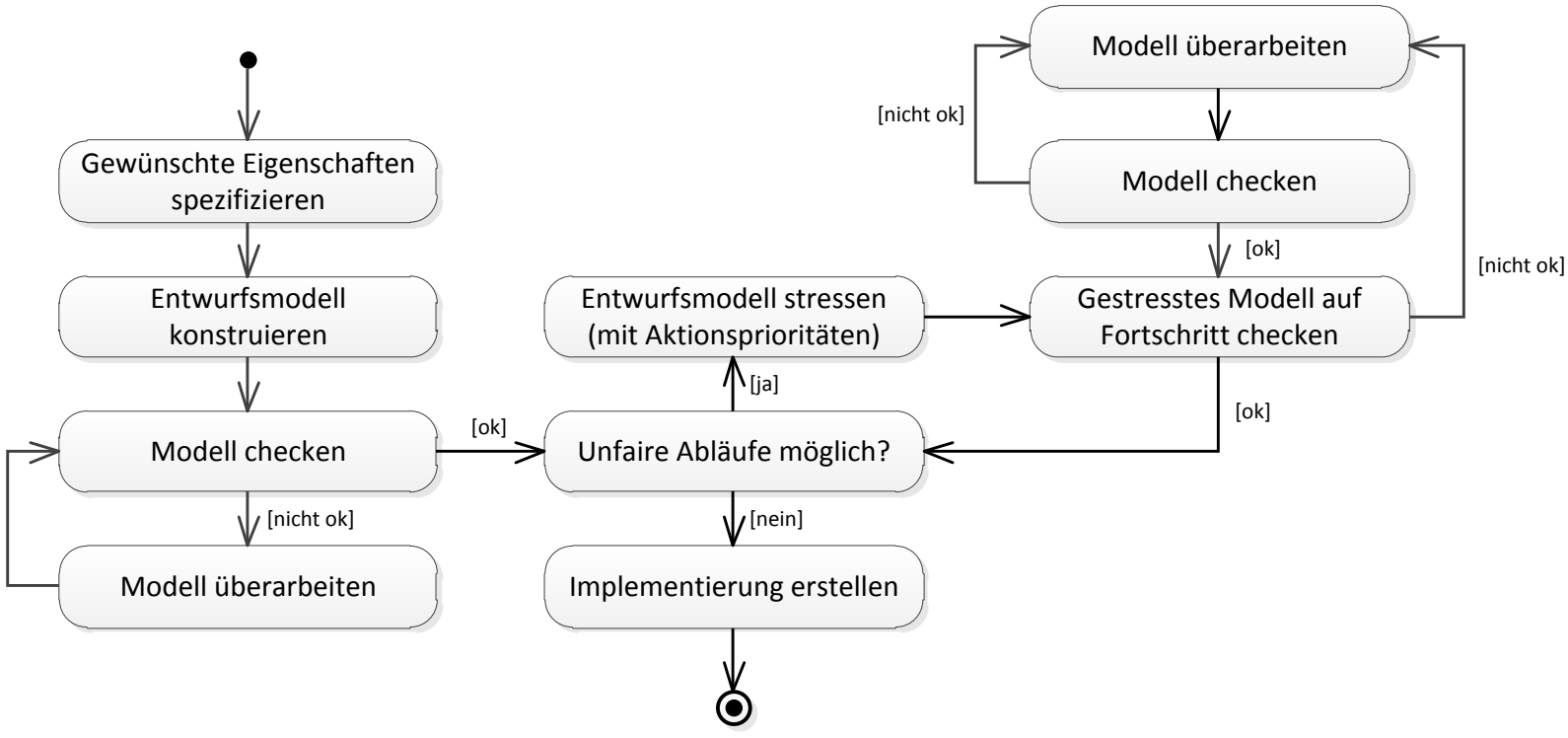


Es gilt:

$\texttt{RW\_SYS} \models \texttt{READ[i]}$ für $i = 1, 2$

$\texttt{RW\_SYS} \models \texttt{WRITE[i]}$ für $i = 1, 2$

Entwicklungsmethodik unter Einbeziehung von Aktionsprioritäten



Folgende Eigenschaften sind zu überprüfen:
Deadlock, Sicherheitseigenschaften, Fortschrittseigenschaften

(b) In welcher Situation kann es bei der Ausführung des Systems zu unfairen Abläufen kommen? Modellieren Sie diese Situation durch Einführung von Aktionsprioritäten!

- Zu unfairen Abläufen kann es kommen, wenn laufend "neue" Leser auf den Datenbestand zugreifen, während "alte" Leser den Zugriff noch nicht abgeschlossen haben (z.B. wegen besserer Prozessorauslastung).
- Damit können Schreiber nicht mehr auf den Datenbestand zugreifen.
- Modellierung durch Aktionsprioritäten in FSP:

```
||BUSYREADER = (RW_SYS)>>{reader[Tread].release}.
```

Freigeben des Datenbestands durch einen Leser (`reader[Tread].release`) wird also in `BUSYREADER` niedriger priorisiert.

(c) Welche Fortschrittseigenschaften sind nun verletzt?

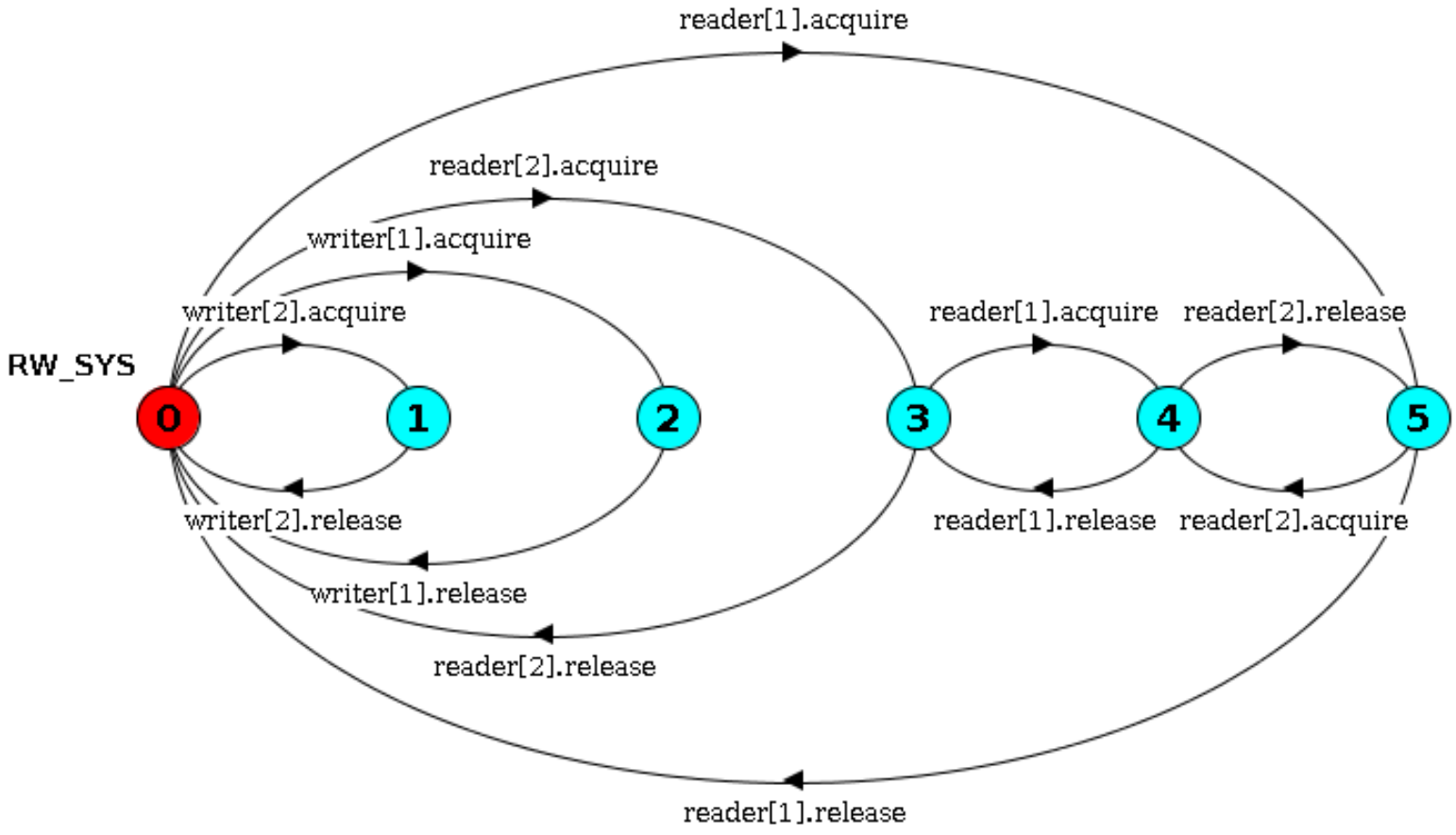
Beweisen Sie Ihre Aussage für den (vereinfachten) Fall, in dem bei den Prozessen `READER` und `WRITER` die Aktionen *examine* und *modify* verborgen sind.

```
READER = (acquire->examine->release->READER)\{examine}.
WRITER = (acquire->modify->release->WRITER)\{modify}.
```

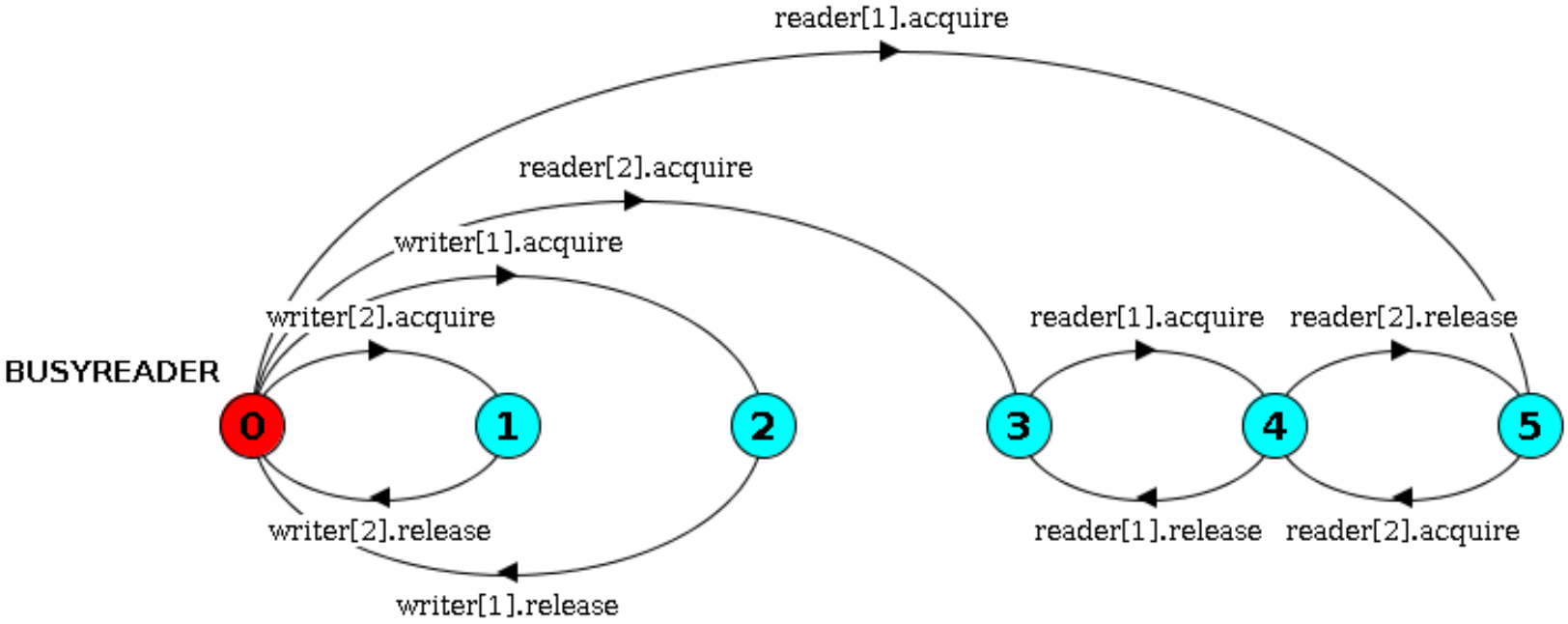
Wir berechnen zunächst das LTS von `RW_SYS` und ermitteln dann das LTS von `BUSYREADER`, durch Berücksichtigung der Aktionsprioritäten.

```
||BUSYREADER = (RW_SYS)>>{reader[Tread].release}.
```

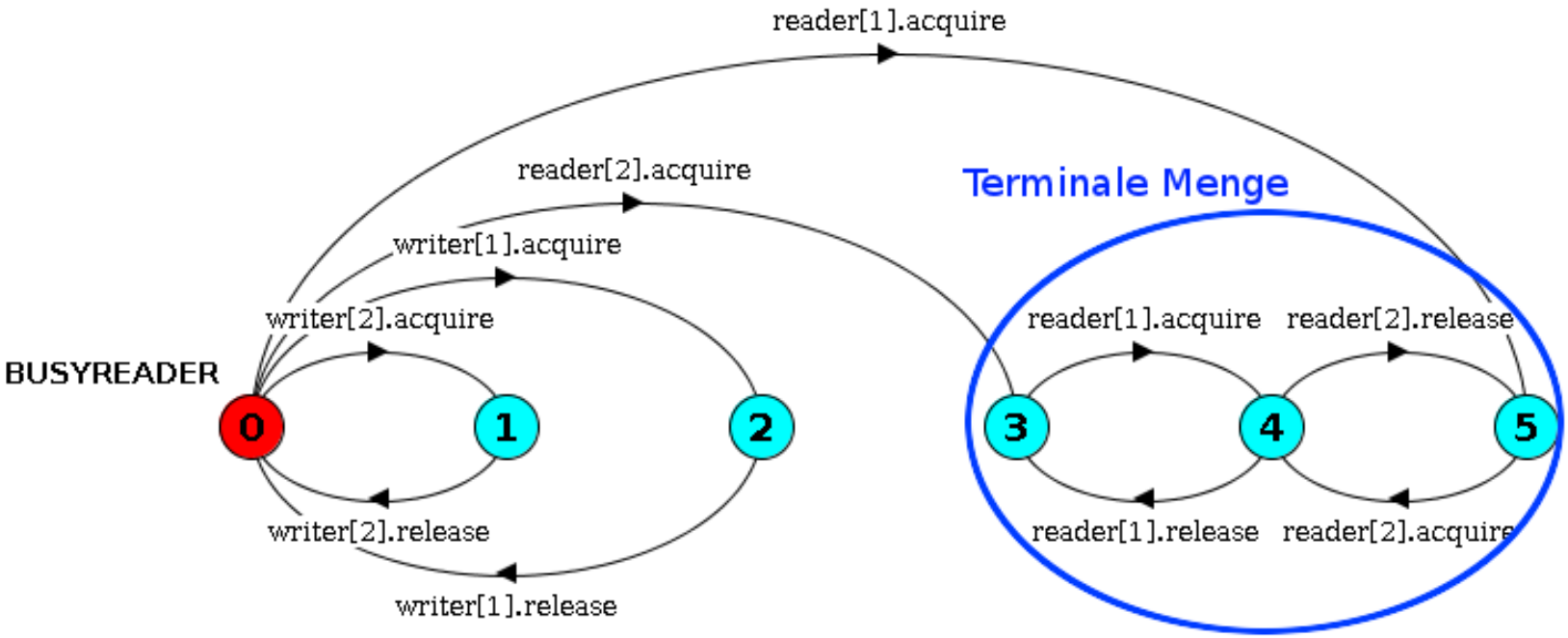
Minimiertes LTS von RW_SYS:



Minimiertes LTS von BUSYREADER:



Minimiertes LTS von BUSYREADER mit terminaler Menge:



- BUSYREADER $\models$ READ[i] für $i = 1, 2$
- BUSYREADER $\not\models$ WRITE[i] für $i = 1, 2$

Terminale Menge von Zuständen: $\{3, 4, 5\}$

Es kommt keine Transition mit `writer[i].acquire` vor, also ist WRITE[i] **nicht** erfüllt (für $i = 1, 2$).

(d) Revidieren Sie das Modell des Leser-Schreiber Systems so, dass alle Fortschrittseigenschaften gelten.

- Idee:
  Sobald ein Schreiber zugreifen möchte (`request`), darf kein weiterer Leser zugreifen.
- Überarbeitetes Modell:

```
READER = (acquire->examine->release->READER)\{examine}.
WRITER = (request->acquire->modify->release->WRITER)\{modify}.

||RW = (reader[Tread]:READER || writer[Twrite]:WRITER).
```

- (Fortsetzung)

```
RW_LOCK = RWL[0][False][0],
RWL[readers:0..Nread][writing:Bool][ww:0..Nwrite] =
   (when (!writing && ww==0)
      reader[Tread].acquire -> RWL[readers+1][writing][ww]
   |reader[Tread].release -> RWL[readers-1][writing][ww]
   |when (readers==0 && !writing)
      writer[Twrite].acquire -> RWL[readers][True][ww-1]
   |writer[Twrite].release -> RWL[readers][False][ww]
   |writer[Twrite].request -> RWL[readers][writing][ww+1]).

||RW_SYS = (RW || RW_LOCK).
||BUSYREADER = (RW_SYS)>>{reader[Tread].release}.
```

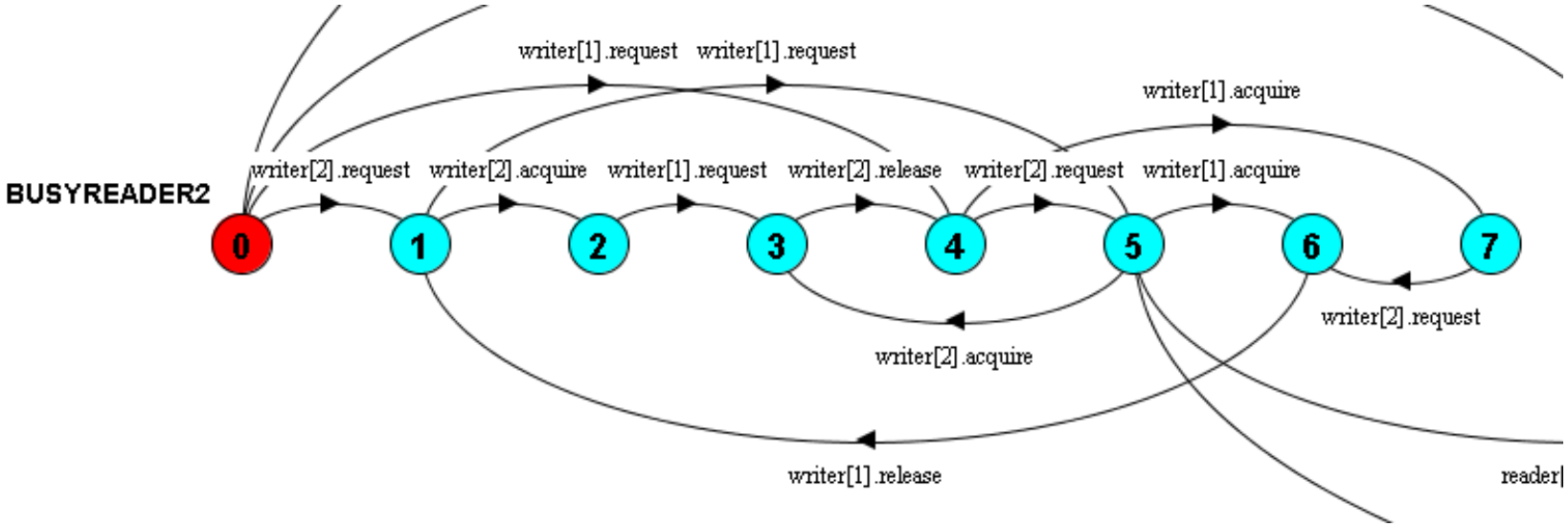
- (BUSYREADER $\models$ SAFE_RW)
- BUSYREADER $\models$ READ[i] für $i = 1, 2$
- BUSYREADER $\models$ WRITE[i] für $i = 1, 2$

(e) Kann es bei der Ausführung des revidierten Systemmodells nun zu anders gearteten Situationen kommen, in denen wiederum unfaire Abläufe möglich sind? Wie können diese ggf. modelliert und in einem neuerlichen Systemmodell verhindert werden?

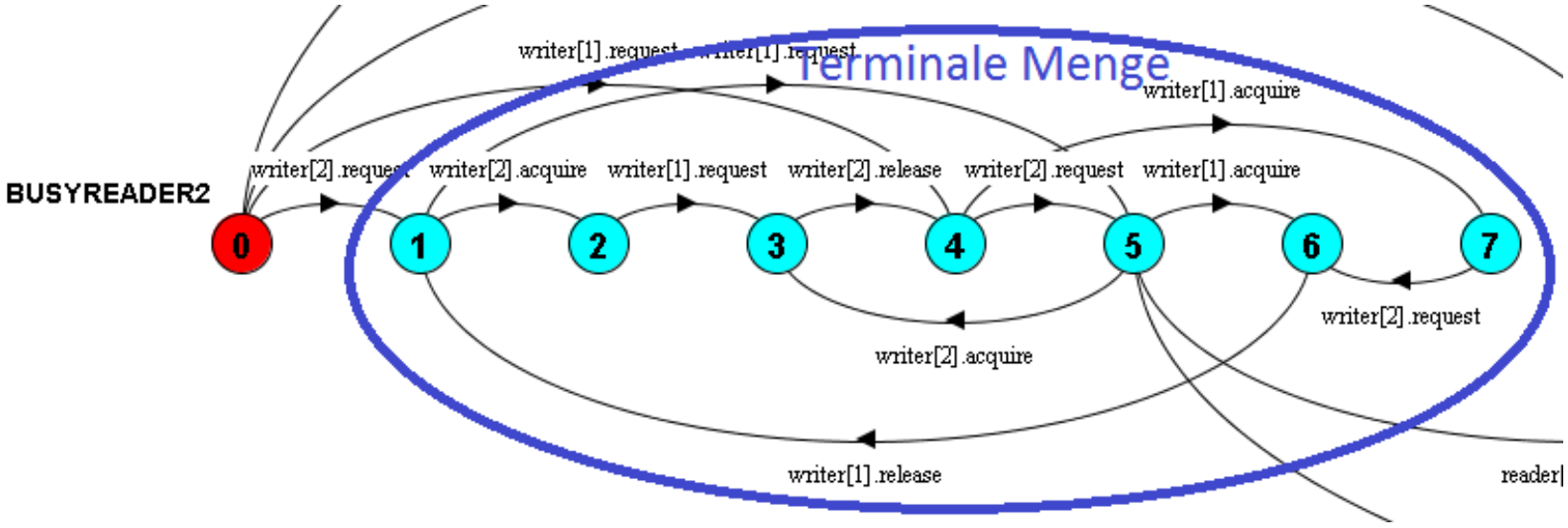
- Bei vielen Schreibern werden möglicherweise viele "request" durchgeführt, während andere Schreiber noch schreiben.
- Leser können dann nicht mehr auf den Datenbestand zugreifen.
- Modellierung durch:

```
||BUSYREADER2 = (RW_SYS)>>{reader[Tread].release,
                           writer[Twrite].release}.
```

Minimiertes LTS von BUSYREADER2:



Minimiertes LTS von BUSYREADER2 mit terminaler Menge:



- BUSYREADER2 $\models$ WRITE[i] für $i = 1, 2$
- BUSYREADER2 $\not\models$ READ[i] für $i = 1, 2$
  Terminale Menge von Zuständen: $\{1, 2, 3, 4, 5, 6, 7\}$

  Es kommt keine Transition mit reader[i].acquire vor, also ist READ[i] **nicht** erfüllt (für $i = 1, 2$).

Wie kann das Modell nochmals überarbeitet werden?

- Idee: Erweitere RW_LOCK um eine Berechtigungsmarke
- rturn ist genau dann True, wenn die Leser die Berechtigung zum Zugreifen haben.
- Nochmal überarbeitetes Modell für

```
RW_LOCK = RWL[0][False][0][False],
RWL[readers:0..Nread][writing:Bool][ww:0..Nwrite][rturn:Bool] =
  (when (!writing && (ww==0||rturn))
      reader[Tread].acquire -> RWL[readers+1][writing][ww][rturn]
  |reader[Tread].release -> RWL[readers-1][writing][ww][False]
  |when (readers==0 && !writing && !rturn)
      writer[Twrite].acquire -> RWL[readers][True][ww-1][rturn]
  |writer[Twrite].release -> RWL[readers][False][ww][True]
  |writer[Twrite].request -> RWL[readers][writing][ww+1][rturn]).

||BUSYREADER2 = (RW_SYS)>>{reader[Tread].release,
                           writer[Twrite].release}.
```

- BUSYREADER2 $\models$ READ[i] für $i = 1, 2$
- BUSYREADER2 $\models$ WRITE[i] für $i = 1, 2$

Aufgabe 2.

Implementieren Sie die beiden Varianten des Leser-Schreiber Modells aus Aufgabe 1, Teil (d) und (e) in Java.