

# Entwurf und Implementierung paralleler Programme

---

Prof. Dr. Rolf Hennicker

18.10.2004

# **Kapitel 8**

## **Lebendigkeitseigenschaften**

## 8.1 Der Begriff der Lebendigkeitseigenschaft

Lebendigkeitseigenschaften (“liveness properties”) drücken aus, dass während der Ausführung eines parallelen Programms (irgendwann) “etwas Gutes“ passiert.

Fortschrittseigenschaften sind spezielle Lebendigkeitseigenschaften (“progress properties”). Eine Fortschrittseigenschaft sichert zu, dass in jedem Ablauf eines Programms ab jedem Zeitpunkt noch irgendwann eine spezifizierte Aktion ausgeführt wird.

### **Beispiel (Einspurige Brücke):**

Irgendwann überquert jedes wartende Auto die Brücke.

## 8.2 Fortschrittseigenschaften

### Definition:

Sei  $F$  ein Identifikator und sei  $\{a_1, \dots, a_n\} \subseteq \text{Labels}$  eine Menge von Aktionen.  
Dann definiert

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

eine *Fortschrittseigenschaft*.

### Beispiel:

$$\text{COIN} = (\text{toss} \rightarrow \text{heads} \rightarrow \text{COIN} \\ | \text{toss} \rightarrow \text{tails} \rightarrow \text{COIN}).$$

### ***Faire Auswahl:***

Wenn eine Auswahl aus einer Menge von Transitionen unendlich oft durchgeführt wird, dann wird jede der möglichen Transitionen unendlich oft gewählt (und ausgeführt).

### **Beispiel:**

### ***Generelle Voraussetzung:***

Im Folgenden setzen wir immer faire Auswahl für FSP-Prozesse voraus, d.h. wir betrachten bei der Überprüfung von Fortschrittseigenschaften nur Aktionenfolgen, die sich durch faire Auswahl ergeben ("faire Abläufe").

**Definition:**

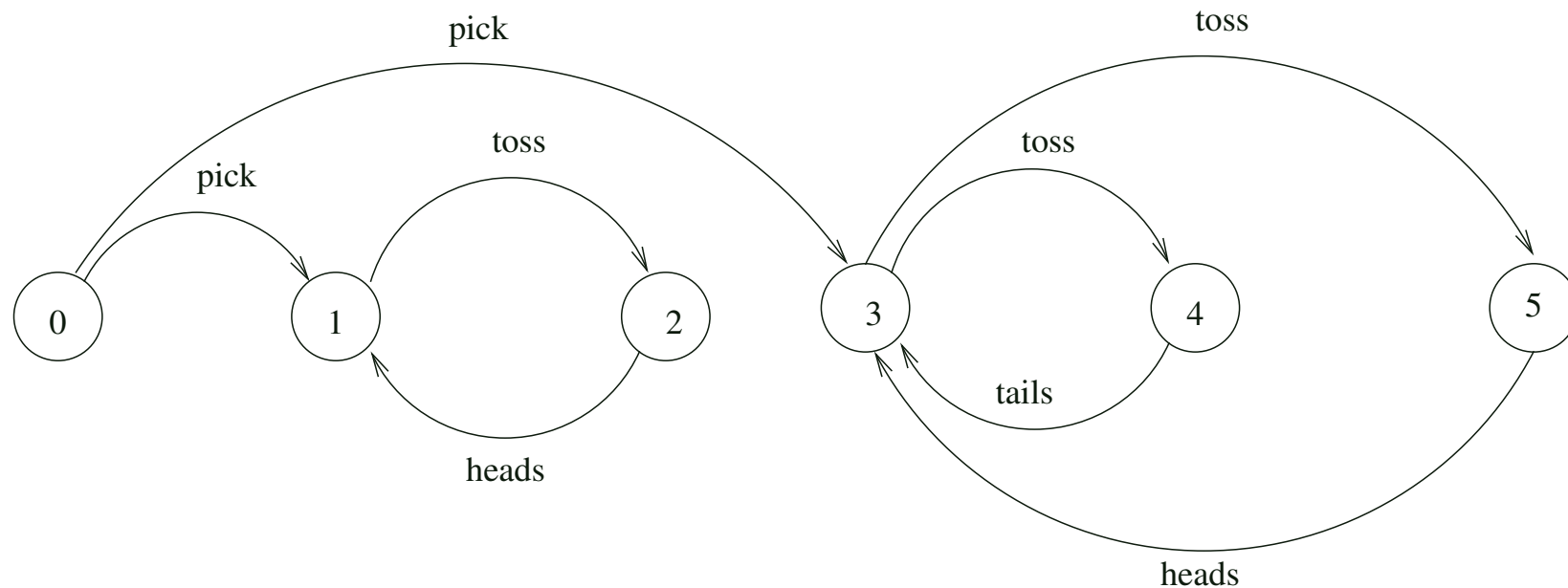
Sei  $P$  ein Prozess und progress  $F = \{a_1, \dots, a_n\}$  eine Fortschrittseigenschaft.  $P$  *erfüllt*  $F$ , wenn in jeder (fairen) Aktionsfolge (Ablauf) von  $P$  mindestens eine Aktion aus  $\{a_1, \dots, a_n\}$  unendlich oft vorkommt.

## Beispiel (TWOCOIN):

TWOCOIN = (pick  $\rightarrow$  COIN | pick  $\rightarrow$  TRICK),

TRICK = (toss  $\rightarrow$  heads  $\rightarrow$  TRICK),

COIN = (toss  $\rightarrow$  heads  $\rightarrow$  COIN | toss  $\rightarrow$  tails  $\rightarrow$  COIN).



## 8.3 Nachweis von Fortschrittseigenschaften

### *Idee*

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

### **Definition:**

Sei  $P$  ein Prozess mit  $\text{Its}(P) = (S, A, \Delta, q)$ .

Eine *terminale Menge* von Zuständen von  $P$  ist eine 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 (d.h.  $s$  wird unendlich oft “besucht“).

### **Beispiel (TWOCOIN):**

**Satz:**

Sei  $P$  ein Prozess und sei  $\text{progress } F = \{a_1, \dots, a_n\}$  eine Fortschrittseigenschaft.  $P$  erfüllt  $F$  genau dann, wenn in jeder terminalen Menge  $T$  von Zuständen von  $P$  (mindestens) eine Transition mit einer Aktion aus  $\{a_1, \dots, a_n\}$  vorkommt.

### ***Automatisches Checken von Fortschrittseigenschaften:***

1. Konstruiere alle terminalen Mengen von Zuständen von  $P$ .
2. Falls es eine terminale Menge gibt, in der keine Aktion aus  $\{a_1, \dots, a_n\}$  vorkommt, ist  $F$  nicht erfüllt; ansonsten ist  $F$  erfüllt.

### **Beachte:**

Die Gültigkeit einer Fortschrittseigenschaft ist entscheidbar, da es im LTS von  $P$  nur endlich viele Zustände *und* endlich viele Transitionen gibt.

### **Beispiel (TWOCOIN):**

**Default-Analyse:**

Für alle Aktionen  $a$  im Alphabet eines Prozesses  $P$  wird überprüft, ob

$$\text{progress } F_a = \{a\}$$

von  $P$  erfüllt ist.

**Beispiel (Brücke):**

Die Default-Analyse zeigt, dass alle Fortschrittseigenschaften erfüllt sind.

*Grund:* In jedem *fairen* Ablauf wird jedes Auto unendlich oft über die Brücke fahren.

## 8.4 Aktionsprioritäten

In der Praxis können unter bestimmten Bedingungen auch gewisse unfaire Abläufe vorkommen. Dann ist Fortschritt nicht mehr gesichert, auch wenn alle Fortschrittseigenschaften gezeigt wurden.

Aktionsprioritäten dienen dazu, solche Situationen zu modellieren.

### *Idee*

Das System wird unter "Stress" gesetzt  
(z.B. möglichst viele Autos gleichzeitig auf die Brücke).

Die Fortschrittseigenschaften werden unter dem gestressten Systemmodell erneut untersucht. (Falls sie nicht mehr gelten, müssen das Modell und ggf. das Programm geeignet modifiziert werden.)

## Prozesse mit Aktionsprioritäten

### Definition:

Sei  $E$  ein Prozessausdruck und  $a_1, \dots, a_n \in \alpha E$ .

#### 1. Hohe Priorität:

$$(E) \ll \{a_1, \dots, a_n\}$$

ist ein Prozessausdruck, in dem die Aktionen  $a_1, \dots, a_n$  *hohe Priorität* haben.

#### *Wirkung:*

Wo immer eine Auswahl im LTS von  $E$  vorkommt zwischen  $a \in \{a_1, \dots, a_n\}$  und  $b \notin \{a_1, \dots, a_n\}$  wird die Transition von  $b$  weggelassen.

#### 2. Niedrige Priorität:

$$(E) \gg \{a_1, \dots, a_n\}$$

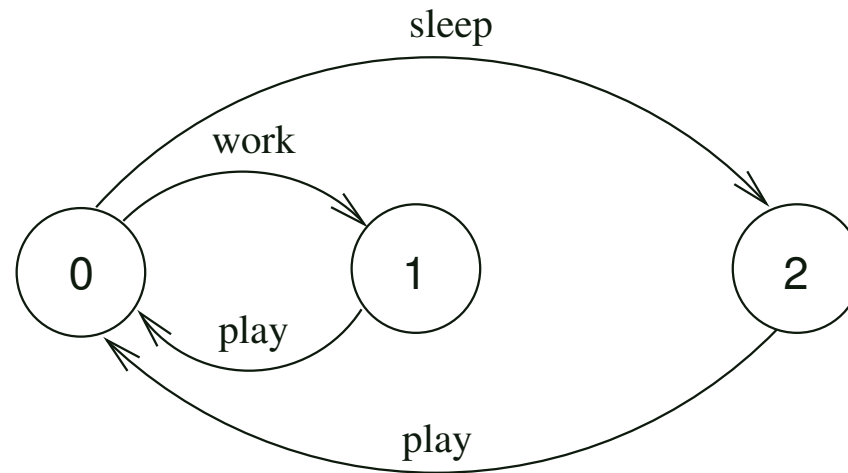
ist ein Prozessausdruck in dem die Aktionen  $a_1, \dots, a_n$  *niedrige Priorität* haben.

#### *Wirkung:*

Wo immer eine Auswahl im LTS von  $E$  vorkommt zwischen  $a \in \{a_1, \dots, a_n\}$  und  $b \notin \{a_1, \dots, a_n\}$  wird die Transition von  $a$  weggelassen.

**Beispiel:**

NORMAL = (work  $\rightarrow$  play  $\rightarrow$  NORMAL  
|| sleep  $\rightarrow$  play  $\rightarrow$  NORMAL).



## Beispiel (Brücke):

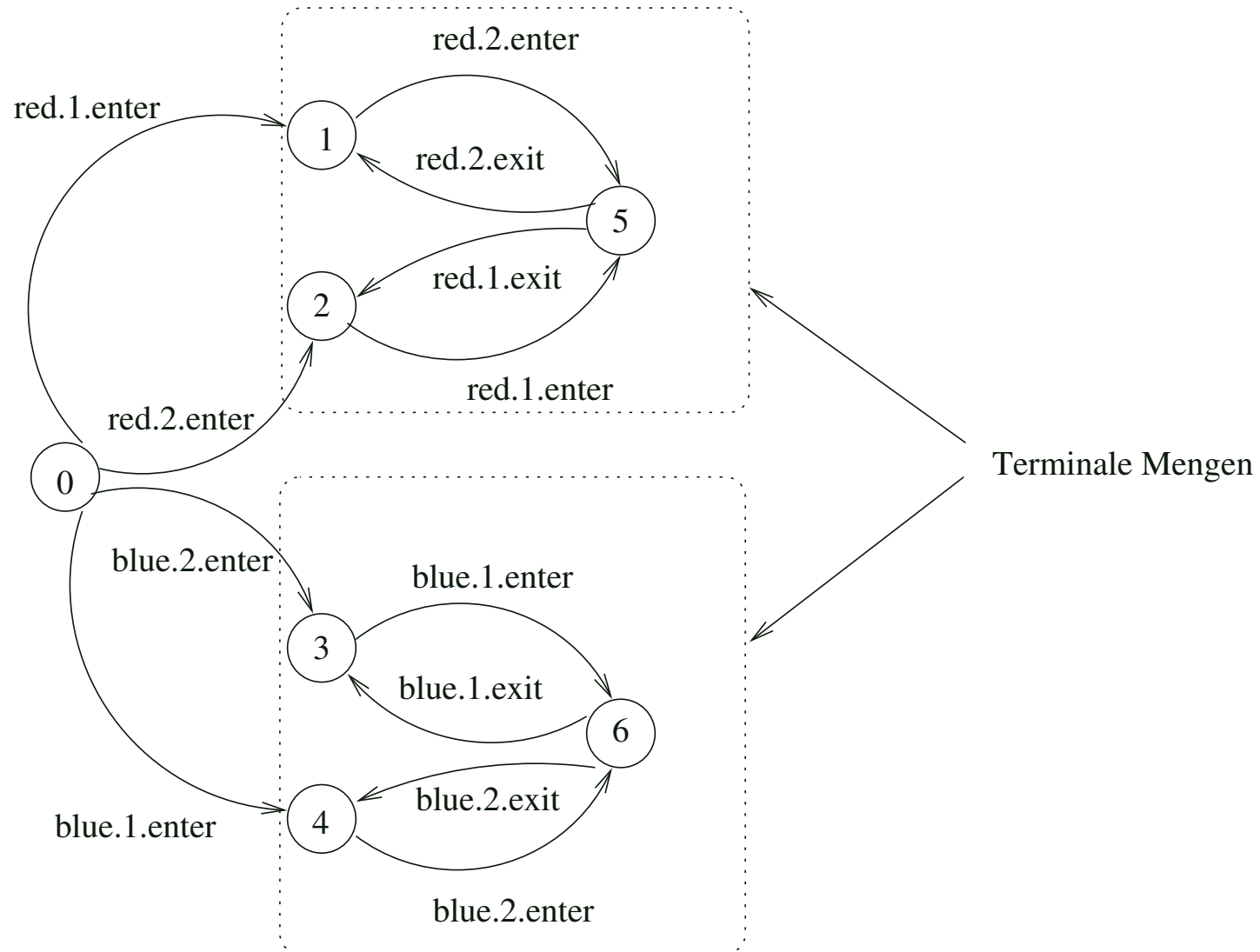
Die Defaultanalyse zeigt, dass alle Fortschrittseigenschaften erfüllt sind.

Welche unfairen Abläufe können in der Praxis (d.h. beim Programmablauf) auftreten?

1. *Möglichkeit:* Der Scheduler wählt nur rote oder nur blaue Autos: Nicht realistisch.
2. *Möglichkeit:* Der Prozessor soll möglichst gut ausgelastet werden, d.h. im Beispiel möglichst viele Autos auf die Brücke lassen.

## Modellierung:

## LTS von FULLBRIDGE für N=2:



## Erneute Fortschrittsanalyse:

progress BLUECROSS = {blue[ID].enter}.

progress REDCROSS = {red[ID].enter}.

Beide Fortschrittseigenschaften werden von FULLBRIDGE nicht erfüllt.

## Revision des Modells:

### 1. *Versuch:*

Die Brücke lässt nur dann rote Autos auffahren, wenn kein blaues Auto auf der Brücke ist *und* wenn kein blaues Auto wartet. (Analog für blaue Autos!)

Ändere CAR in:

CAR = (request → enter → exit → CAR).

```
BRIDGE = BRIDGE[0][0][0][0],
BRIDGE[nr:T][nb:T][wr:T][wb:T] =
  (red[ID].request    -> BRIDGE[nr][nb][wr+1][wb]
  |when (nb==0 && wb==0)
      red[ID].enter   -> BRIDGE[nr+1][nb][wr-1][wb]
  |red[ID].exit       -> BRIDGE[nr-1][nb][wr][wb]
  |blue[ID].request   -> BRIDGE[nr][nb][wr][wb+1]
  |when (nr==0 && wr==0)
      blue[ID].enter  -> BRIDGE[nr][nb+1][wr][wb-1]
  |blue[ID].exit      -> BRIDGE[nr][nb-1][wr][wb]
  ).
```

Jetzt gibt es ein DEADLOCK. Ablauf dahin ist bei  $N=1$ :

red.1.request

blue.1.request

## 2. *Versuch:*

Wie Versuch 1, jedoch darf ein Auto auffahren, wenn seine Farbe an der Reihe ist, auch wenn andersfarbige Autos warten.

```

const True = 1
const False = 0
range B = False..True //bt=True: blue turn, bt=False: red turn
BRIDGE = BRIDGE[0][0][0][0][True],
BRIDGE[nr:T][nb:T][wr:T][wb:T][bt:B] =
  (red[ID].request    -> BRIDGE[nr][nb][wr+1][wb][bt]
  |when (nb==0 && (wb==0 || !bt))
      red[ID].enter   -> BRIDGE[nr+1][nb][wr-1][wb][bt]
  |red[ID].exit       -> BRIDGE[nr-1][nb][wr][wb][True]
  |blue[ID].request   -> BRIDGE[nr][nb][wr][wb+1][bt]
  |when (nr==0 && (wr==0 || bt))
      blue[ID].enter  -> BRIDGE[nr][nb+1][wr][wb-1][bt]
  |blue[ID].exit      -> BRIDGE[nr][nb-1][wr][wb][False]
  ).

```

**Möglicher Ablauf** (Blau ist an der Reihe):

```
red.1.request  
blue.1.request  
blue.1.enter  
blue.2.request  
blue.2.enter  
blue.1.exit  
blue.1.request  
blue.2.exit  
red.1.enter  
...
```

Die Fortschrittsanalyse zeigt, dass FULLBRIDGE nun alle Fortschrittseigenschaften erfüllt.

## Erneute Implementierung:

Diese erfolgt gemäß des revidierten Systemmodells.

```
class FairBridge extends Bridge {
    private int nred = 0;    // count of red cars on the bridge
    private int nblue = 0;  // count of blue cars on the bridge
    private int waitblue = 0; // count of waiting blue cars
    private int waitred = 0; // count of waiting red cars
    private boolean blueturn = true; // asymmetric system

    synchronized void redEnter() throws InterruptedException {
        waitred++;
        while (nblue>0 || (waitblue>0 && blueturn)) wait();
        waitred--;
        nred++;
    }

    synchronized void redExit() {
        nred--;
        blueturn = true;
        if (nred == 0) notifyAll();
    }
}
```

```
synchronized void blueEnter() throws InterruptedException {
    waitblue++;
    while (nred>0 || (waitred>0 && !blueturn)) wait();
    waitblue--;
    nblue++;
}

synchronized void blueExit() {
    nblue--;
    blueturn = false;
    if (nblue == 0) notifyAll();
}
}
```