

# Entwurf und Implementierung paralleler Programme

---

Prof. Dr. Rolf Hennicker

11.12.2006

# **Kapitel 7**

## **Sicherheitseigenschaften**

## 7.1 Der Begriff der Sicherheitseigenschaft

Sicherheitseigenschaften (“safety properties”) drücken aus, dass während der Ausführung eines parallelen Programms “nichts Schlechtes“ passiert.

Sicherheitseigenschaften können formuliert werden durch:

1. Angabe der illegalen Abläufe: “Was nicht passieren darf!“
2. Angabe der legalen Abläufe: “Was passieren darf!“  
(aber deshalb noch nicht passieren muss)

Das Komplement der legalen Abläufe sind die illegalen Abläufe. Häufig ist es einfacher (und auch sicherer) die legalen Abläufe anzugeben. In FSP geschieht dies durch “Property-Prozesse“.

## 7.2 Property-Prozesse

### Definition:

Ist  $P$  ein Prozessidentifikator und  $E$  ein Prozessausdruck, dann ist

property  $P = E$ .

eine *Property-Prozessdeklaration*. Die Deklaration ist rekursiv, wenn  $P$  in dem Ausdruck  $E$  frei vorkommt, d.h.  $P \in FV(E)$ .

### Beispiel:

# Semantik von Property-Prozessen

## Bemerkungen:

1. Nur solche Property-Prozesse sind zulässig, deren LTS deterministisch ist.
2. Fehlerzustände können explizit in FSP-Ausdrücken repräsentiert werden durch den vordefinierten Prozess ERROR.
3. Es ist auch möglich zu spezifizieren, dass eine bestimmte Aktion niemals in einem Ablauf vorkommen darf.

## 7.3 Erfüllung von Sicherheitseigenschaften

### *Generelle Voraussetzung:*

Property-Prozesse sind *deterministisch* (d.h. es gibt keinen Zustand, von dem ausgehend es mehr als eine Transition mit derselben Aktion gibt) und sie enthalten keine verborgenen Aktionen.

### *Legale Abläufe:*

Sei  $P$  ein Property-Prozess und  $w$  ein Ablauf über dem Alphabet  $\alpha P$ .  
 $w$  ist ein *legaler Ablauf bzgl.  $P$* , wenn mit  $w$  im LTS von  $P$  der Fehlerzustand nicht erreicht wird, dh. genauer: wenn kein endlichen Anfang von  $w$  im LTS von  $P$  zum Fehlerzustand führt.

***Ablaufeinschränkung:***

Sei  $w$  ein Ablauf über einem Alphabet  $A$  und sei  $B \subseteq A$ . Entfernt man aus  $w$  alle Aktionen aus  $A$ , die nicht zu  $B$  gehören, so erhält man die *Einschränkung* von  $w$  auf  $B$ , bezeichnet mit  $w|_B$ .

**Definition (Erfüllung von Sicherheitseigenschaften):**

Sei  $P$  ein Property-Prozess und  $Q$  ein (gewöhnlicher) FSP-Prozess mit  $\alpha P \subseteq \alpha Q$ .  $Q$  *erfüllt* die Sicherheitseigenschaft  $P$ , geschrieben  $Q \models P$ , wenn für alle Abläufe  $w$  von  $Q$  die Einschränkung  $w|_{\alpha P}$  ein legaler Ablauf bzgl.  $P$  ist.

## Beispiele:

1. Sei

$VISITOR = (\text{knock} \rightarrow \text{enter} \rightarrow \text{discuss} \rightarrow VISITOR)$ .

$VISITOR$  erfüllt die Sicherheitseigenschaft  $POLITE$ .

2. Sei

$WRONG\_VISITOR = (\text{knock} \rightarrow \text{enter} \rightarrow \text{discuss} \rightarrow \text{enter} \rightarrow WRONG\_VISITOR)$ .

$WRONG\_VISITOR$  erfüllt die Sicherheitseigenschaft  $POLITE$  nicht.

3. Sei  $Q$  ein beliebiger FSP-Prozess und property  $CALM = STOP + \{\text{disaster}\}$ .

$Q + \{\text{disaster}\}$  erfüllt  $CALM$  genau dann, wenn in keinem Ablauf von  $Q$  die Aktion "disaster" vorkommt.

## 7.4 Nachweis von Sicherheitseigenschaften

**Beispiel (WRONG\_VISITOR):**

(WRONG\_VISITOR||POLITE) hat das LTS

**Beispiel (VISITOR):**

(VISITOR||POLITE) hat das LTS

**Intuition:**

Gegeben sei ein Prozess  $Q$  und ein Property-Prozess  $P$  mit  $\alpha P \subseteq \alpha Q$ . Sei  $w$  ein Ablauf von  $Q$ . Durch die parallele Komposition von  $Q$  mit dem Property-Prozess  $P$  wird jede Aktion in  $w$ , die auch im Alphabet von  $P$  vorkommt, im LTS von  $P$  in eindeutiger Weise (da  $P$  deterministisch) begleitet (durch Aktionssynchronisation). Erreicht man auf diese Weise den Fehlerzustand, dann ist  $w|_{\alpha P}$  ein illegaler Ablauf (bzgl.  $P$ ) und umgekehrt.

**Satz:**

Sei  $P$  ein Property-Prozess und  $Q$  ein (gewöhnlicher) FSP-Prozess mit  $\alpha P \subseteq \alpha Q$  und  $\pi \notin \text{Its}(Q)$ .  $Q$  *erfüllt* die Sicherheitseigenschaft  $P$  **genau dann**, wenn im LTS von  $(Q \parallel P)$  der Fehlerzustand *nicht* erreichbar ist (d.h. kein Ablauf vom Anfangszustand zum Fehlerzustand existiert).

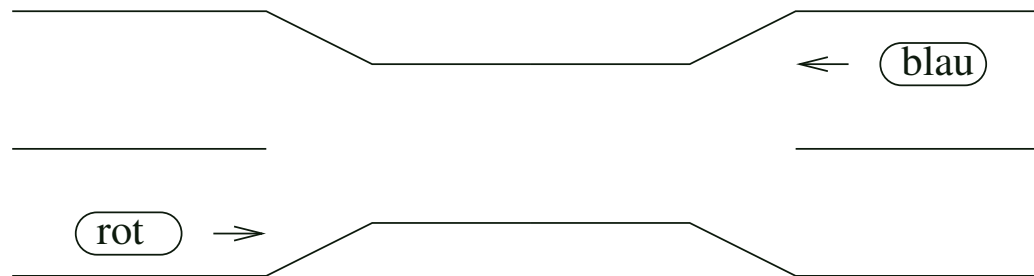
**Automatisches Checken von Sicherheitseigenschaften:**

Durch Breitensuche im LTS von  $(Q \parallel P)$  nach Erreichbarkeit des Fehlerzustands kann entschieden werden, ob ein Prozess  $Q$  eine Sicherheitseigenschaft  $P$  erfüllt oder nicht.

**Bemerkung:**

$Q$  erfüllt die Sicherheitseigenschaft  $P$  genau dann, wenn das LTS von  $(Q \parallel P)$  mit dem LTS von  $Q$  übereinstimmt.

## 7.5 Beispiel: Einspurige Brücke



### Sicherheitseigenschaft:

Rote und blaue Autos dürfen nicht gleichzeitig über die Brücke fahren.

### Modellierung der Grundkonzepte (des Problembereichs):

```
const N = 3 // Anzahl Autos pro Seite
range ID = 1..N // Autonummern
```

```
CAR = (enter → exit → CAR).
```

```
|| CONVOY = [ID]:CAR.
```

```
|| CARS = (red:CONVOY || blue:CONVOY).
```

## Modellierung der Sicherheitseigenschaft:

property ONEWAY = (red[ID].enter  $\rightarrow$  RED[1]  
| blue[ID].enter  $\rightarrow$  BLUE[1]),

RED[i:ID] =

BLUE[i:ID] =

ONEWAY besagt, dass wenn immer ein rotes (bzw. blaues) Auto auf der Brücke ist, nur ein rotes (bzw. blaues) Auto auf die Brücke fahren darf.

### Bemerkung:

ONEWAY modelliert eine Anforderung (bzgl. Sicherheit) und *keinen* Entwurf.

## Modellierung eines Entwurfs:

Zur Kontrolle der Brücke wird ein Monitor verwendet.

range  $T = 0..N$

BRIDGE = BRIDGE[0][0],

BRIDGE[nr:T][nb:T] =

## Modell des Gesamtsystems:

$\parallel \text{SYS} = (\text{CARS} \parallel \text{BRIDGE})$ .

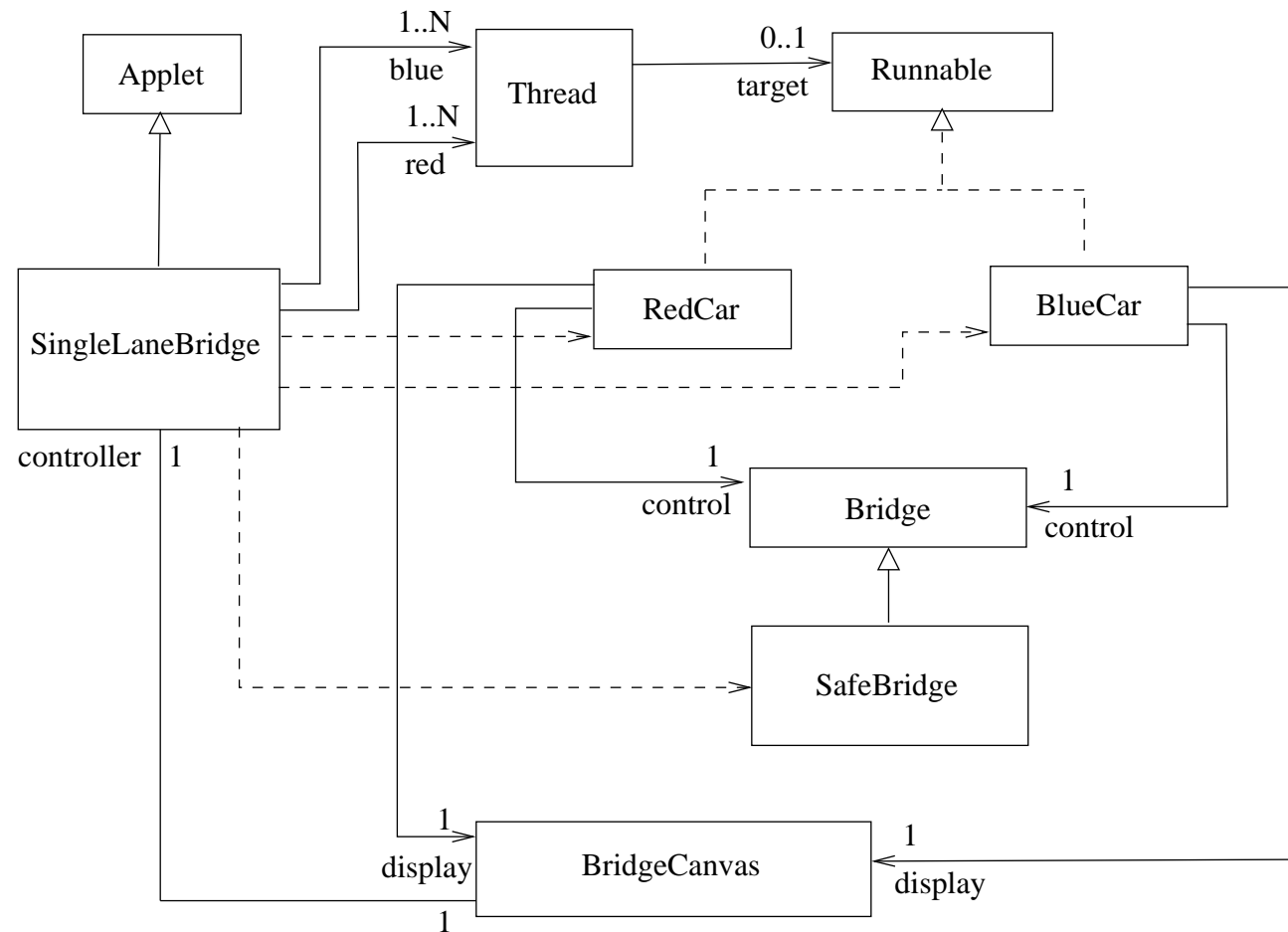
## Nachweis der Sicherheitseigenschaft:

Wir betrachten das LTS von  $(\text{SYS} \parallel \text{ONEWAY})$  und checken, dass der Fehlerzustand nicht erreichbar ist.

## Implementierung in Java

Aktive Objekte: rote und blaue Autos

Passives Objekt: Brücke



## Java-Code (Auszug):

```
public class SingleLaneBridge extends Applet {
    BridgeCanvas display;
    int maxCar;
    ...
    Thread[] red; // Convoy
    Thread[] blue;
    ...
    public void start() {
        red = new Thread[maxCar];
        blue = new Thread[maxCar];
        Bridge b = new SafeBridge();
        ...
        for (int i=0; i<maxCar; i++) {
            red[i] = new Thread(new RedCar(b,display,i));
            blue[i] = new Thread(new BlueCar(b,display,i));
            red[i].start();
            blue[i].start();
        }
    }
}
```

```
class Bridge { // unsafe
    synchronized void redEnter() { }
    synchronized void redExit() { }
    synchronized void blueEnter(){ }
    synchronized void blueExit() { }
}
```

```
class SafeBridge extends Bridge {
    private int nred = 0; // red cars on bridge
    private int nblue = 0;

    synchronized void redEnter() throws InterruptedException {
        while (nblue>0) wait();
        nred++;
    }
    synchronized void redExit() {
        nred--;
        if (nred==0) notifyAll();
    }
    synchronized void blueEnter() throws InterruptedException {
        while (nred>0) wait();
        nblue++;
    }
    synchronized void blueExit() {
        nblue--;
        if (nblue==0) notifyAll();
    }
}
```

```
class RedCar implements Runnable {
    BridgeCanvas display;
    Bridge control;
    int id;

    RedCar(Bridge b, BridgeCanvas d, int id) {
        control = b;
        display = d;
        this.id = id;
    }

    public void run() {
        try {
            while (true) {
                while (!display.moveRed(id)); // not on bridge
                control.redEnter();
                while (display.moveRed(id)); // move over bridge
                control.redExit();
            }
        } catch (InterruptedException e) {}
    }
}
```