Kapitel 7

Sicherheitseigenschaften

Prof. Dr. Rolf Hennicker

23.06.2016

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!“

   oder

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

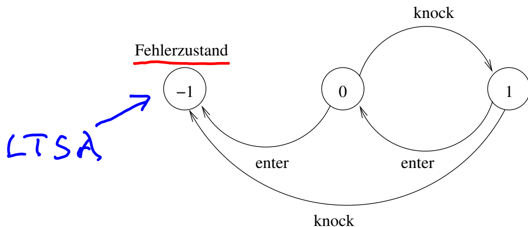
property P = E.

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

Beispiel:

property POLITE = (knock → enter → POLITE). αPOLITE = {knock, enter}

Im LTS eines Property-Prozesses werden die illegalen Abläufe offenbart.



Semantik von Property-Prozessen

Sei $\mathsf{T} = (S, A, \Delta, q_0)$ ein LTS (wie bisher).

Die Property-Vervollständigung von T ist definiert durch:

$$\mathsf{property}_{\mathsf{LTS}}(\mathsf{T}) =_{def} (S \cup \{\pi\}, A, \Delta, q_0)$$

wobei

- π ein (neuer) ausgezeichneter Fehlerzustand ist und
- $\Delta' = \Delta \cup \{(s, a, \pi) | s \in S, a \in \alpha T, \nexists s' \in S : (s, a, s') \in \Delta\}$.

Sei property P = E. eine Deklaration eines Property-Prozesses P.

Dann ist

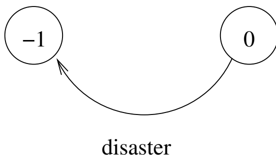
$$\mathsf{lts}(\mathsf{P}) =_{def} \begin{cases} \mathsf{property}_{\mathsf{LTS}}(\mathsf{lts}(\mathsf{E})) & \text{falls } P \notin \mathsf{FV}(\mathsf{E}) \\ \mathsf{property}_{\mathsf{LTS}}(\mathsf{lts}(\mathsf{rec}(\mathsf{P}{=}\mathsf{E}))) & \text{sonst} \end{cases}$$

Bemerkungen:

1. Fehlerzustände können explizit in FSP-Prozessausdrücken repräsentiert werden durch den vordefinierten (konstanten) Prozess ERROR.

2. Es ist auch möglich zu spezifizieren, dass eine bestimmte Aktion niemals in einem Ablauf vorkommen darf, z.B.

   property CALM = STOP + {disaster}. $\qquad$ αCALM = {disaster}



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 Aktionsfolgen:

Sei P ein Property-Prozess und w eine endliche oder unendliche Aktionsfolge über dem Alphabet αP. w ist eine *legale Aktionsfolge bzgl.* P, wenn kein endlicher Anfang von w im LTS von P zum Fehlerzustand führt.

z.B. property P = (a -> b -> P). $\alpha P = \{a, b\}$

w = abab legal

ababab... legal

ababb ab illegal

ababb abab... illegal

Ablaufeinschränkung:

Sei w eine endliche oder unendliche Aktionsfolge 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 $w|_{\alpha P}$ eine legale Aktionsfolge bzgl. P ist.

Interpretation:

- property P = E. ist eine Anforderungsspezifikation,
- Q ist ein Entwurfsmodell,
- Q ist korrekt bzgl. P, wenn gilt $Q \models P$.

Beispiele:

1. Sei
   αPOLITE
   VISITOR = (knock → enter → discuss → VISITOR).

   VISITOR erfüllt die Sicherheitseigenschaft POLITE.

   $k\,e\,d\,k\,e\,d \ldots |_{\alpha POLITE} = k\,e\,k\,e \ldots$ legale Aktionsfolge bzgl. POLITE

2. Sei
   WRONG_VISITOR =
   (knock → enter → discuss → enter → WRONG_VISITOR).

   WRONG_VISITOR erfüllt die Sicherheitseigenschaft POLITE nicht.

   $k\,e\,d\,e\,k\,e\,d\,e \ldots |_{\alpha POLITE} = k\,e\,e\,k\,e\,e \ldots$ illegale Aktionsfolge bzgl. POLITE

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

   Q erfüllt CALM genau dann, wenn in keinem Ablauf von Q die Aktion “disaster“ vorkommt.

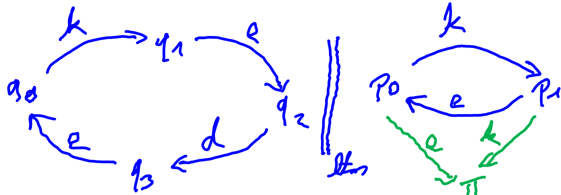
   $Q = (a \to a \to disaster \to b \to Q)$,

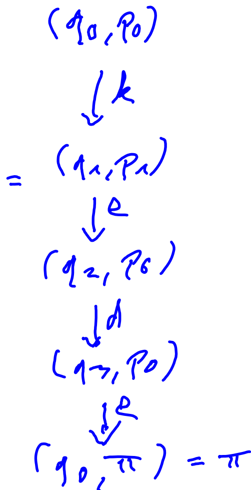
   $Q \not\models$ CALM   a a disaster b a a disaster b ...

Vorbemerkung: Bei der parallelen Komposition von LTSsen werden alle Zustände der Form (s,π) identifiziert mit dem Fehlerzustand π.

Beispiel (WRONG_VISITOR):

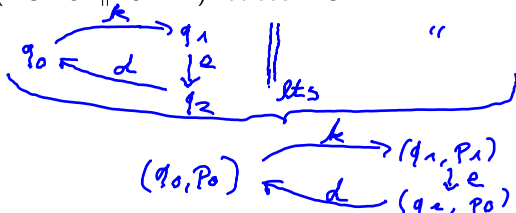
(WRONG_VISITOR||POLITE) hat das LTS





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 beliebiger 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 lts(P) deterministisch und vollständig) begleitet (durch Aktionssynchronisation). Erreicht man auf diese Weise den Fehlerzustand, dann ist $w|_{\alpha P}$ ein illegaler Ablauf (bzgl. P) und umgekehrt.

Modell

Satz 1:
Sei P ein Property-Prozess und Q ein (gewöhnlicher) FSP-Prozess mit $\alpha P \subseteq \alpha Q$ und $\pi \notin$ lts(Q). $Q \models P$ **genau dann, wenn** in lts($Q \parallel P$) der Fehlerzustand nicht erreichbar ist.

Automatisches Checken von Sicherheitseigenschaften: Model Checking

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.

vgl. Satz 1

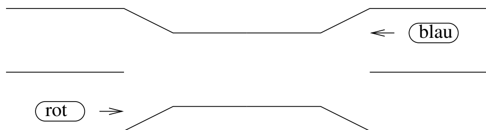
Entwicklungsmethodik für Programme mit Parallelität

1. Spezifikation der gewünschten Eigenschaften (Sicherheits- und Lebendigkeitseigenschaften; vgl. Kap. 8) mit FSP. Was?
2. Modellierung eines Entwurfs mit FSP. Wie?
3. Nachweis, dass das Entwurfsmodell deadlockfrei ist und die spezifizierten Eigenschaften erfüllt.
4. Java-Implementierung mit Threads und Monitoren gemäß des angegebenen Schemas zur Überführung von FSP-Entwurfsmodellen nach Java (vgl. Kap. 5).

Bemerkung:

Die Schritte 1 - 3 sind rein formal mit Korrektheitsnachweis im 3. Schritt. Der 4. Schritt is pragmatisch und nicht formal als korrekt bewiesen.

7.5 Beispiel: Einspurige Brücke



im Property-Prozess positiv zu formulieren

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
```

CAR = (enter → exit → CAR).

||CONVOY = [ID]:CAR. (also CONVOY = ([1]:CAR||[2]:CAR||[3]:CAR))

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

(also CARS = (red[1]:CAR|| ...||blue[3]:CAR))

Möglicher Ablauf: red[1].enter blue[3].enter ...

1. Spezifikation der Sicherheitseigenschaft:

Wir spezifizieren einen Property-Prozess ONEWAY, der besagt, dass wenn immer ein rotes (bzw. blaues) Auto auf der Brücke ist nur ein rotes (bzw. blaues) Auto auf die Brücke fahren darf.

```
property ONEWAY = (red[ID].enter → RED[1]
                  |blue[ID].enter → BLUE[1]),

RED[i:ID]   = (red[ID].enter → RED[i+1]
              |when (i==1) red[ID].exit → ONEWAY
              |when (i>1) red[ID].exit → RED[i-1]),

BLUE[i:ID]  = (blue[ID].enter → BLUE[i+1]
              |when (i==1) blue[ID].exit → ONEWAY
              |when (i>1) blue[ID].exit → BLUE[i-1]).
```

Bemerkung:

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

2. Modellierung eines Entwurfs:

Zur Kontrolle der Brücke wird ein Monitor verwendet.

```
range T = 0..N
BRIDGE = BRIDGE[0][0],
BRIDGE[nr:T][nb:T] =
        (when (nb==0) red[ID].enter → BRIDGE[nr+1][nb]
        | red[ID].exit → BRIDGE[nr-1][nb]
        | when (nr==0) blue[ID].enter → BRIDGE[nr][nb+1]
        | blue[ID].exit → BRIDGE[nr][nb-1]).
```

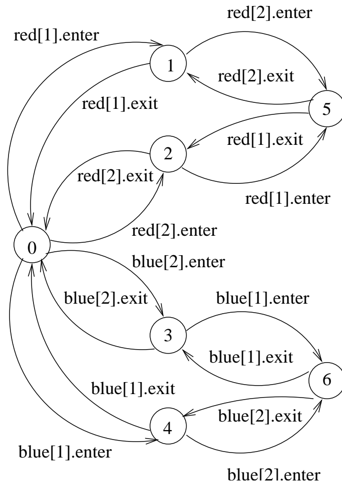
Modell des Gesamtsystems:

```
||SYS = (CARS || BRIDGE).
```

↑ überwacht bzw. koordiniert CARS so, dass keine illegalen Abläufe mehr möglich sind.

Nachweis → Folie 16.

LTS von SYS = (CARS ∥ BRIDGE) für N=2:



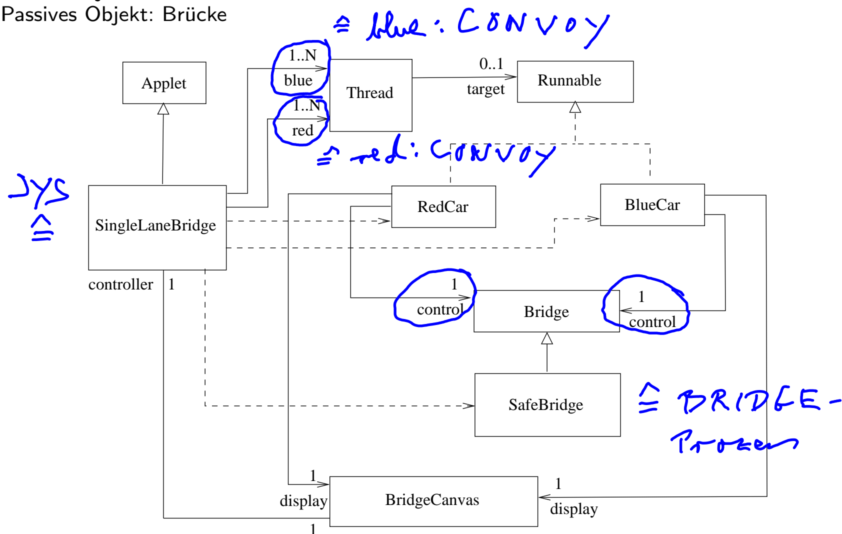
3. Nachweis der Sicherheitseigenschaft:

Wir wollen nachweisen: SYS $\models$ ONEWAY.

Dazu betrachten wir das LTS von (SYS $\|$ ONEWAY) und checken, dass der Fehlerzustand nicht erreichbar ist. (In der Tat stimmt das LTS von (SYS $\|$ ONEWAY) mit dem LTS von SYS überein.)

4. Implementierung in Java:

Aktive Objekte: rote und blaue Autos
Passives Objekt: Brücke



Java-Code (Auszug):

```
public class SingleLaneBridge extends Applet {
   BridgeCanvas display;
   int maxCar;  //N im FSP-Prozess
   ...
   Thread[] red;  // roter Convoy
   Thread[] blue; // blauer Convoy
   ...
   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();
      }
   }
}
```

ein gemeinsames Monitorobjekt

```
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++;
   //notify nicht nötig, da kein Thread auf die Erhöhung von nred warten wird
   }
   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; //Monitor
   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)); // nicht auf der Brücke
            control.redEnter();     //Aufruf der Monitoroperation
            while (display.moveRed(id)); // über die Brücke fahren
            control.redExit();        //Aufruf der Monitoroperation
         }
      } catch (InterruptedException e) {}
   }
}
```