CSP-Spezifikation `myfile.csp`

1. Typ-Deklarationen
2. Kanal-Deklarationen
3. Prozess-Spezifikationen
4. Verifikationsbedingungen

CSP: Typ-Deklarationen

- vordefiniert: **`Bool = { true | false }`**
- konzeptuell vordefiniert: **`Int`**
  Verwendung: **`{ 3..5 }, {1,5,7}`**
- Selbstdefinierte Aufzählungstypen:
  `datatype myType = Gnu|Affe|Zebra`
- Abkürzungen: **`MyDef = { 7..21 }`**

CSP: Kanal-Deklarationen

- Kanal, der nur atomare Events repäsentiert
  `channel c1`
  `{| c1 |} = {c1}` **Menge aller Events, die über den Kanal c1 kommuniziert werden können**
- Kanal mit strukturierten Events
  `channel c2 : { 1..2,5 }`
  `{| c2 |} = {c2.1, c2.2, c2.5}`
- Kanal mit „mehrfach strukturierten“ Events
  `channel c3 : myType.{3..4}`
  `{| c3 |} = {c3.Gnu.3, c3.Gnu.4,`
  `c3.Affe.3, c3.Affe.4,`
  `c3.Zebra.3, c3.Zebra.4}`

CSP: Prozess-Spezifikationen (1)

- Prozess P: **`P =`** `<process body>`
- Prozess-System: **`P1 =`** `c1 -> c2.1 ->` **`P2`**
  `P2 =` `c2.5 ->` **`P1`**
- Rekursiver Prozess: **`P`** `= c1 ->` **`P`**
- Parametrisierte Prozesse: **`P = Px(1)`** (Initialisierung)
  `Px(v) =` ...
- Vordefinierte CSP-Prozesse:
  - CSP-Prozess, der kein Event zulässt/produziert **`STOP`**
  - CSP-Prozess für erfolgreiche Termination **`SKIP`**
  - Chaotischer Prozess über Eventmenge M **`CHAOS(M)`**

CSP: Prozess-Spezifikationen (2)

CSP-Operatoren

- Präfix-Operator `->`
- Alternativ-Operator (external choice) `[]`
  (internal choice) `|~|`
- Sequentielle Komposition `P1 ; P2`
- Boolscher Guard `b & (c1-> P)`
- Interleaving-Operator `P1 ||| P2`
- Parallel-Operator `P1 [|{|c1|}|] P2`
- Hiding-Operator `P \ {|c1|}`
- Renaming-Operator `P [[ c <- c' ]]`

CSP: Prozess-Spezifikationen (3)

- replicated external choice

```
M = {|c2|}                Q = c2.1 -> P
Q = [] x:M @(x-> P)           []
                              c2.2 -> P
                              []
                              c2.5 -> P
```

- Replicated internal choice

```
Q = |~| x:M @ (x->P)
```

- Replicated interleaving

```
Q = ||| x:{1..3} @ P(x)
```


CSP: Prozess-Spezifikationen (4)

- Zahlen

```
1+2   3-2   2*6   3/5   3%5   -2
x<y   x>y   x<=y  x>=y
x==y  x!=y
```

- Bool

```
true          false
b1 and b2     b2 or b1
b1==b2        b2!=b1
not b1
```


CSP: Prozess-Spezifikationen (5)

- Sequenzen

```
s = <>            t = <1,3,2,0>
null(s)=true      length(t)=4
head(t)=1         tail(t)=<3,2,0>
elem(1,t)=true    s^t=t
```

- Mengen

```
M={1..2}  N = {}  MyDef
empty(N)=true     card(M)=2
union(M,N)={1..2} inter(M,N)={}
member(3,M)=false
```

CSP: Prozess-Spezifikationen (6)

if-then-else

```
P(b1,b2) = if (b1 == b2)
           then c1 -> P(b1,false)        (Prozess)
           else c2.1 -> P(true,b2)       (Prozess)

P(b1,b2,b3) = (if (b1 and b2)
               then c1                   (Event)
               else (if (b2 and b3)
                     then c2.1           (Event)
                     else c2.5))         (Event)
              -> P(true, false, true)    (Prozess)
```


CSP: Prozess-Spezifikationen (7)

if-then-else (2)

```
P(b1) = (if (b1)
         then (c1->SKIP)          (Prozess)
         else STOP)               (Prozess)
         ; Q                      (Prozess)
```


Warum überhaupt?

- Bestimmte Eigenschaften einer Implementierung bzw. eines Algorithmus sollen nachgewiesen werden, z.B.
  - Deadlockfreiheit
  - Livelockfreiheit
  - Implementierung/Algorithmus verhält sich wie von dem Protokoll X / Algorithmus A / allg. Spezifikation Y /... gefordert

Situation 1

Vorhandene Implementierung (Code) in C, C++, Pascal, Java, ...

z.B. Peterson´s Algorithmus

Zum Nachweis der gewünschten Eigenschaften wird erstellt

1. konkretes System (Implementierungsmodell)
2. abstraktes System (Referenzmodell)
3. Verifikationsbedingungen

Situation 1: Konkretes System

Möglichst direkte Umsetzung der Implementierungsentscheidungen

- (globale) Variablen → Variablen-Prozesse
- Prozeduren/Funktionen → Prozesse
- Schleifen → Rekursive Prozesse mit Abbruchbedingungen
- Variablenzugriff → `read/write`-Kanäle des Variablen-Prozesses
- Prozedur/Funktions-parameter → Prozessparameter des entsprechenden Prozesses

Situation 1: Abstraktes System

- Referenz-CSP-Prozess, der „offensichtlich“ die gewünschten Eigenschaften besitzt
  - z.B., mutual exclusion

```
N = { 0..1 }
ASYS = [] x:N @ enter.x -> leave.x -> ASYS
```

- Idealisiertes Modell der Spezifikation

Situation 1: Verifikationsbedingungen (1)

- Deadlockfreiheit

```
assert SYS :[deadlock free [F]]
```

- Livelockfreiheit

```
assert SYS :[livelock free [FD]]
```


Situation 1: Verifikationsbedingungen (2)

- Konkretes System führt nur Traces aus, die auch im abstrakten System möglich sind (Implementierung ist korrekt)

```
assert ASYS [T= SYS
```

  Wenn im konkreten System eine Menge M von neuen Events eingeführt wurden, muss das abstrakte System um diese erweitert werden:

```
ASYS = ASYS_old ||| RUN(M)
RUN(M) = [] e:M @ e -> RUN(M)
```

Situation 1: Verifikationsbedingungen (3)

- Jede Folge von Events, die das abstrakte System erzeugen kann, ist auch im konkreten System legal

```
assert SYS [T= ASYS
```

  Wenn im konkreten System eine Menge M von „Implementierungs“-Events eingeführt wurde, so müssen diese vorher versteckt werden:

```
SYS = SYS_old \ M
```

  Vorsicht bei der Verwendung des Hiding-Operators!

Situation 2

Umgangssprachlicher Algorithmus

(kein Code)

z.B. *Cigaret Smoker Problem*

Zum Nachweis der gewünschten Eigenschaften wird erstellt

1. abstraktes System
2. konkretes System
3. Verifikationsbedingungen

Situation 2: Abstraktes System

- Modellierung der allgemeinen Eigenschaften des Systems, die nachgewiesen werden sollen
- Keine unnötigen „Implementierungsdetails“

Situation 2: Konkretes System (1)

Möglichst direkte Umsetzung der Algorithmus-Idee

Beispiele:

- „In einem Raum sitzen drei Raucher und ein Verkäufer.“ →
```
Raucher1
Raucher2
Raucher3
Verkäufer
```
- „Menge von Zutaten“ → `ZUTATEN = {X1,X2,X3}`
- „V. wählt beliebig eine Zutat aus“ →
```
[]  x:ZUTATEN @ ...
|~| x:ZUTATEN @ ...
```
- „X weckt Y auf“ →
```
X [| {|wecke|} |] Y
X = ..-> wecke.Y -> X
Y = wecke.Y -> .. -> Y
```

Situation 2: Konkretes System (2)

- Wenn der Algorithmus später implementiert werden soll, muss darauf geachtet werden, dass eine direkte Umsetzung in C/C++/Java/Pascal/...-Code möglich ist.

Situation 2: Verifikationsbedingungen

- Siehe Situation 1

Variablen-Prozesse

Boolsche Variable **b:**

```
channel read_b, write_b : Bool
VAR_B(v) = read_b!v -> VAR_B(v)
           write_b?w -> VAR_B(w)
```

1. Lesender Zugriff von einem anderen Prozess
   ```
   P = read_b?b -> Q(b)
   ```
2. Schreibender Zugriff von einem anderen Prozess
   ```
   R = write_b!true -> S
   ```
3. Gesamtsystem
   ```
   SYS = (P ||| R)
         [|{|read_b, write_b|}|] VAR_B(false)
   ```


Schleifen-Prozesse

```
void my_f () {

  printf(„hallo“);
  int x = 0;
  while (x<5) {
    x++;
  }

 printf(„world“);
}
```

```
channel hello, world

MY_F = hello ->
       WHILE(0);
       world ->
       SKIP

WHILE(x) =
    if (x<5)
    then WHILE(x+1)
    else SKIP
```

Parametrisierte Prozess

```
Q = P(1,1,1)
P(x,y,z) = if (x<2)
           then P(x+1,y,z)
           else (if (x>2)
                 then P(x-1,y,z)
                 else P(x+1,x,z) )

P(1,1,1), P(2,1,1), P(3,2,1), P(2,2,1)
```


Beispiele von CSP-Spezifikationen

- generell alle Beispiele aus VL/UE
- heute:
  - Strict Alternation
  - Cigaret Smoker Problem
  - total ausgenutzter Ringpuffer – Algorithmus 1
  - total ausgenutzter Ringpuffer – Algorithmus 2