CSP
Communicating Sequential Processes

Einführung in die Syntax von CSP und Timed CSP

Aliki Tsiolakis
22.10.2002

CSP-Spezifikation `myfile.csp`

1. Typ-Deklarationen
2. Kanal-Deklarationen
3. Makro-Definitionen
4. Prozess-Spezifikationen
5. Verifikationsbedingungen (nur zur Verifikation)

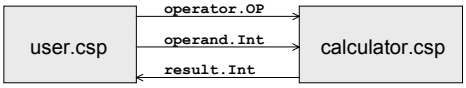
CSP: Typ-Deklarationen

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

CSP: Kanal-Deklarationen

- Kanal, der nur atomare Events repräsentiert
  ```
  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}
  ```


Input/Output-Kanäle (1)



Interface-Beschreibung

- Output-Kanäle
  ```
  channel operator : OP
  channel operand : Int
  ```
- Input-Kanal
  ```
  channel result : Int
  ```

Interface-Beschreibung

- Output-Kanäle
  ```
  channel result : Int
  ```
- Intput-Kanäle
  ```
  channel operator : OP
  channel operand : Int
  ```


Input/Output-Kanäle (2)

user.csp

```
pragma AM_OUTPUT
channel operator : OP
channel operand : Int

pragma AM_INPUT
channel result : Int
```

calculator.csp

```
pragma AM_OUTPUT
channel result : Int

pragma AM_INPUT
channel operator : OP
channel operand : Int
```

Error- und Warning-Kanäle

- Error-Kanal
  für schwerwiegende Fehler, die zum Abbruch des Test-Prozesses führen müssen/sollen
```
pragma AM_ERROR
channel error : { 0..9 }
```
- Warning-Kanal
  für "Fehler", die nicht zum Abbruch des Test-Prozesses führen sollen/müssen
```
pragma AM_WARNING
channel warning : { 0..5 }
channel warningLateEvent
```


Timer-Kanäle (1)

- Spezielle Kanäle um Timing-Bedingungen zu testen

1. setzen eines Timers
   "starten des Kurzzeitweckers"
2. Ablauf des Timers
   "klingeln des Kurzzeitweckers"
3. Stoppen des Timers
   "vor Ablauf des Weckers stoppen"

Timer-Kanäle (2)

- Setzen des Timers
```
pragma AM_SET_TIMER
channel setTimer : TIMERS
```
- Ablauf des Timers
```
pragma AM_ELAPSED_TIMER
channel elapsedTimer : TIMERS
```
- Stoppen des Timers
```
pragma AM_RESET_TIMER
channel resetTimer : TIMERS
```

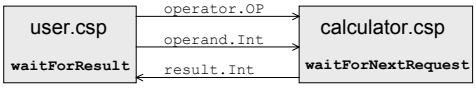

Setzen der Timer-Werte

- jeder Timer hat eine bestimmte ID, z.B. `5`
- Menge aller Timer einer Spezifikation
```
TIMERS = { 0..9 }
```
- jedem Timer (d.h. jeder ID) ist eine bestimmte Anzahl von Time-Ticks zugewiesen, dies geschieht ausserhalb der CSP-Spezifikation
```
0:100:-      -- Timer 0: 100ms
1:200:300    -- Timer 1: 200-300ms
2:1000:-     -- Timer 2: 1000ms = 1s
```


Interne Kanäle (1)

- Interne Kanäle werden für interne (oder versteckte) Events innerhalb einer CSP-Spezifikation verwendet
- Werden nicht zu anderen CSP-Spezifikationen geschickt (sind aber auch im *Test-Execution-Log*)

Interne Kanäle (2)



- Interner Kanal
```
pragma AM_INTERNAL
channel waitForResult
```
- Interner Kanal
```
pragma AM_INTERNAL
channel waitForNextRequest
```

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

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

(2) 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)

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

"Umsetzung" von Code

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 → (z.B.) Prozessparameter des entsprechenden Prozesses

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)
```