Ideas from Communicating Sequential Processes

Prof. RKJ
CS 451 Lecture
2003
Background of the times

- Basic Structuring methods of computer programs
  1. Repetitive construct (while, ..)
  2. Alternative command (if, ...)
  3. Normal sequential program composition (,)

- Other important program structures
  subroutines
  Coroutines
  Classes
  Processes
  monitors
  Actors
  ...

Hoare’s contribution

Components on multiprocessors must be able to communicate with each other
What was existing: shared storage, semaphores, monitors, conditional critical regions...
Hoare suggested that
input and output are basic primitives of programming

Parallel composition of sequential processes is a fundamental program structuring method

Development of CSP (Communicating Sequential Processes Paradigm)
Contributions

Express parallel processes $p1 :: P1 \parallel p2 :: P2$

$P1$ and $P2$ can be composed together through input and output

Inputs can be used in guards

Guarded commands can be clubbed in an alternative statement through nondeterminism

Repeat alternative command
CSP Basic primitives

- Specifying parallel processes
- Communication between processes through input and output commands
- Guarded commands for obtaining nondeterminism
- Repetitive command
- Pattern matching feature for determining input messages
Specifying parallel processes

```
room::ROOM 11
fork(i:0..4)::FORK11
phil(0..4)::PHIL]
```

ROOM, FORK, PHIL provide the process specifications or command lists, and room, fork, and phil are the processes created.
Input and output commands

- **X?i**
  - Input a value from X and assign it to i

- **Y!10**
  - Output value 10 to Y

- **Sem!PO**
  - Output signal PO to process Sem

- **X(i)?VO**
  - Input a signal VO from the given process; refuse any other signal

- **X?(x,y)**
  - Input a pair of values and assign them to x and y respectively
Guarded commands and Alternative Command

Guarded commands $G \rightarrow CL$
that is, if guard $G$ is true, execute command list CL

$G_1 \rightarrow CL_1 | G_2 \rightarrow CL_2 | \ldots | G_n \rightarrow CL_n$
execute any one of the command list whose guards are
found to be true: Nondeterministic choice

IF all guards fail, alternative command fails
Repetitive command

- As many iterations of its constituent alternative command as possible

- $[ G_1 \rightarrow CL_1 | G_2 \rightarrow CL_2 | \ldots | G_n \rightarrow CL_n ]$

execute any one of the command list whose guards are found to be true: Nondeterministic choice and repeat

IF all source processes fail, alternative command fails, else processes wait for corresponding outputs
Try Bounded Buffer
Bounded Buffer

```plaintext
buffer:: int item; int buff[MAX]; ....

(Prod?item) and (buffernotfull) -> do insertion

(consumer?consume()) and (buffernotempty) -> remove an item;
  consumer: removedItem

prod:: PROD II consumer:: CONSUMER
```
Dining Philosophers

\[ \text{room}::\text{ROOM} \parallel \text{fork}[i:0..4]::\text{FORK} \parallel \text{phil}[i:0..4]::\text{PHIL} \]

1. ROOM

   \[ \text{int i, count; } \]

   \[ \text{ *[ } \text{(i:0..4)phil(i)?enter() and count<4 } \rightarrow \]

   \[ \text{ count=count+1 } \]

\[ \text{(i:0..4)phil(i)?exit() } \rightarrow \text{ count=count-1 } \]
Example 1

\[ X ::= \star \]

`c:char;`  

`west?c \rightarrow east!c`
Example 2

\[
X :: \ast \left[ \begin{array}{l}
c : \text{char}; \\
w \ast ? c \rightarrow [ \\
\quad c ! = \ast \rightarrow \text{east!} c \\
\quad c == \ast \rightarrow \text{west?} c; \\
\quad \left[ c == \ast \rightarrow \text{east!} \ast; \\
\quad \quad \text{east!} c; \\
\quad \quad c == \ast \rightarrow \text{east!} \ast^
\end{array} \right]
\]
\]

Assume that final character is not \ast

How do you deal with odd no. of \ast's at the end?
A subroutine

\[
\text{subr:: usr?(input parameters)} \rightarrow \text{...; X!(results)}
\]

\[
\text{usr:: ???????}
\]
A subroutine

```
[ subr:: usr?(input parameters) \rightarrow
  \ldots;
  usr!(results)
]
```

```
usr:: subr!(input params);
subr?(results);
```
Example 3

```plaintext
xx\_yz (i: 1..max) ::
  *[n:int;  
    xy\_yz(i-1)?n\rightarrow
    [n==0\rightarrow xy\_yz(i-1)!  
      n>0\rightarrow xy\_yz(i+1)!n-1
      r:int;  
      xy\_yz(i+1)?r;  
      xy\_yz(i-1)!(n*r)
    ]  
  ]  
  xy\_yz(0)::USER
```
Readings

CAR Hoare, Communicating Sequential Processes, CACM, August 1978