Occam

Concurrency as the foundation of good programming
Overview

• Communicating Sequential Processes
• The Occam approach
• The Occam language
• Concurrent applications with Occam
• Occam for Linux – KRoC
• Occam Operating System
Occam

- Occam based on Communicating Sequential Processes (CSP) formalism developed by Tony Hoare, Oxford, UK, and an experimental language by David May, Bristol, UK
- Designed to have a formal semantics suitable for automatic program transformations
- Many groups investigated direct translation of Occam into hardware
Cosmetics

- Keywords are in CAPITAL letters
- Variables may include ’.’s
  - This.is.a.variable.name
- Scope is marked with indent spaces
  - No { } is uses
- Occam is line oriented
- Comments are anything following --
Occam Processes

- Not processes as we know them from operating systems
- More like procedures
- Or atomic blocks
- Think of them as structured actions
- EVERY LINE IS A PROCESS!!!
Structure
The structure of a program is a process with declarations preceding it.

<declares>
<process>

INT j:
SEQ
  j := 1
  j := j + 1
Types

- INT - Integers
- BOOL - Boolean
- BYTE - Characters
- REAL32 - Float
- REAL64 - Double
Arrays

• \[<\text{size}>]+ of <\text{type}> <\text{variable}>

• \[42\] of INT a:
  \[a[0] := 1\]

• \[42][42\] of REAL32 b:
  \[b[0][0] := 0\] (REAL32)
Assignments

• **Uses :=**
  – Just like Pascal does

• `<variable> := <expression>`

• `x := y + 1`
Boolean Expressions

• NOT a – True if a is false
• a AND b – True if both a and b are true
• a OR b – True if a or b is true
• a >< b – True if one is true and the other is false
Bit wise operators

- **NOT a** - Logical inversion of all bits
- **a \( \lor \) b** - Logical or
- **a \( \land \) b** - Logical and
- **a \( \oplus \) b** - Logical exclusive or
- **a \( \ll b \)** - Shift a b-bits left
- **a \( \gg b \)** - Shift a b-bits right
Comparisons

• = -- Equals
• > -- Larger than
• < -- Less than
• <> -- Not equal
• >= -- Larger than or equal
• <= -- Less than or equal
An Example

d:=(a \land (p < q)) \lor (b \land (p = q)) \lor (c \land (p > q))
Mathematical Expressions

• + -- Sum
• - -- Difference
• * -- Product
• / -- Quotient
• \ -- Remainder
Precedents

• Nasty surprise – there is no mathematical precedents rules!!! 😞
• All ”complex” formulas must be parameterized to make the expression non-ambivalent

\[-x := 2 \times y + 1 -- IS ILLEGAL\]
\[-x := (2 \times y) + 1 -- Is legal\]
Conditionals - IF

• IF <<condition> <expression>>+

• ONE condition must be true!!!
  – Otherwise the process stops

IF
  n < 0
  sign := -1
n = 0
  sign := 0
n > 0
  sign := 1

IF
  n < 0
  sign := -1
TRUE
  sign := 1
Interval IF

• IF
  i=0
  j:=2
  i=1
  j:=1
  i=2
  j:=0

• IF k = 0 FOR 2
  k=i
  j=2−k
CASE

- CASE is simply another for for IF – so it’s not directly related to case or switch in other languages
- CASE
  1
  j:=0
  2
  j:=1
  ELSE
  j:=0
- However CASE is much faster than IF
While

• \texttt{WHILE} boolean
  process
• \texttt{WHILE} \ a<0
  \ a:=1
Channels

• Channels connects processes and are the basis for the Communication part of the CSP implementation
• Channels are all rendez-vous
Channels

- Channels are of a given type
  - CHAN OF INT q:
  - CHAN OF ANY link:
  - \([n+1]\) CHAN OF ANY links:
Send !

• Send the value of a variable down a channel
• `<channel> ! <value>`
• `ch ! y + 1`
• A set of values can also be sent
  – `ch ! x; y; x + y`
Receive ?

- Receive from channel into a variable
- ? <variable>
- Ch ? X
- A set of values can also be received
  - ch ? x; y; z
SEQ

• A sequential block of code can be defined by a SEQ statement

• IF
  
  c < 2

  SEQ
  
  a:=1

  b:=2

  TRUE

  c:=3
PAR

• Parallel blocks of code can be defined using PAR
• After a PAR each line of code is an individual process
• IF
  c<2
  PAR
  a ! 1
  b ! 2
  TRUE
  PAR
  a ? x
  b ? y
What does this mean???

\[ a \ ? \ x \]
\[ a \ ! \ 2 \]

\[ x := 2 \]
ALT

• One statement from an block of code can be executed (non-deterministic) using the ALT

• ALT

  C1 ? x
  
  Cout ! x

  C2 ? x

  Cout ! x
PRI ALT

• Processes in an ALT may be prioritized and we can ask Occam to choose the highest priority in case more than one process is able to run

• PRI ALT
  
  C1 ? x
  Cout ! x
  C2 ? x
  Cout ! x
SEQ (again)

• Using replicators SEQ also doubles as a for loop constructor

• \[ \text{SEQ } i = 0 \text{ to } 10 \]
  \[ j := j + i \]
Timeout

• Timers can be used to timeout blocking operations
• Timers are a type like all others
• TIMER t:
Timeout

VAL one.sec IS 15625: -- ticks in second
INT begin.time:
TIMER time:
SEQ
  -- some code
  time ? begin.time
PRI ALT
  Ch ? y
    pl -- some process
  time ? AFTER begin.time + one.sec
SKIP
Skip

• The Occam instruction corresponding to NOP
• Simply used to make the process continue without any operation
• Typically used after an IF or a timeout
Stop

- **STOP** - do nothing and never terminate the process, i.e., never get to the next process.
Replicators

• An interval called a replicator can be added to the keywords
  – SEQ
  – PAR
  – IF
  – ALT
Replicated SEQ

SEQ i = 0 to 10
    ch ! i

Much like a FOR loop – only the loop variable is local to the scope
Replicated PAR

\[
\text{PAR } i = 0 \text{ to } 10 \\
\text{ch}[i] \! : \! 42
\]

This way we can send to 10 channels as they become ready
We only finish after they have all completed!
Replicated IF

IF $i = 0$ to $10$
\[ x[i] \div 2 = 0 \]
\[ \text{ch} ! x[i] \]

This way we can send all even values in down a channel
Replicated ALT

PRI ALT i = 1 FOR 10
  Ch ! y
    p1 -- some process
    time ? AFTER begin.time + (i*one.sec)
    SKIP

This way we can attempt a send 10 times before we finally give up
Procedures

• We can structure out programs further by using procedures
• Call by reference!
• PROC <name> {<params>}{<body>}:
  • PROC add.one(INT param)
    param:=param+1
    :
  :
Call by value

• Call by value can be forced by adding the VAL keyword before a parameter

• PROC add.one(VAL INT param)
  new.param:=param+1
  : -- this is now pointless:)
A Procedure Example

PROC Average ([]REAL32 Data, REAL32 Res)
SEQ
  Res := 0.0
  SEQ i = 0 for SIZE Data
    Res := Res + Data[i]
  Res := Res/(REAL32 ROUND (SIZE Data))
: 
Libraries

• Include header files
  – #INCLUDE "hostio.inc"

• Use libraries
  – #USE "hostio.lib"
End of part one

I hope😊
Hello World

PROC hello.world (CHAN OF BYTE keyboard, screen, error)

```plaintext
--{{
SEQ
  screen ! 'H'
  screen ! 'e'
  screen ! 'l'
  screen ! 'l'
  screen ! 'o'
  screen ! '
  screen ! 'W'
  screen ! 'o'
  screen ! 'r'
  screen ! 'l'
  screen ! 'd'
  screen ! '*c'
  screen ! '*n'
  --}}}
```
Hello World (2)

PROC hello.world (CHAN OF BYTE keyboard, screen, error)
     --{{
        VAL []BYTE greeting IS "Hello World*c*n":
        SEQ i = 0 FOR SIZE greeting
            screen ! greeting[i]
        --}}

:
HELLO WORLD (3)

#USE "course.lib"

PROC hello.world (CHAN OF BYTE keyboard, screen, error)
    --{{
        out.string ("Hello World%c\n", 0, screen)
        --}}}
:

Echo

#include "consts.inc"

PROC echoing (CHAN OF BYTE keyboard, screen, error)
      --{{
      BYTE ch:
      SEQ
      ch := ' ' WHILE ch <> 'Z'
      SEQ
         keyboard ? ch
         screen ! ch
         screen ! FLUSH
         screen ! '*c'
         screen ! '*n'
      --}}}
      :

:
A simple FIFO buffer
A simple FIFO buffer

VAL INT N IS 4:
[N + 1] CHAN OF INT C:
PAR P = 0 FOR N
INT Value:
WHILE TRUE
SEQ
  C[P] ? Value
  C[P + 1] ! Value
Compile time limitations

INT N.PROC:
SEQ
   Ch ? N.PROC
   PAR P = 0 FOR N.PROC
par.process

INT N.PROC:
VAL INT MAX IS 100:
SEQ
   Ch ? N.PROC
   IF N > MAX
   STOP
   PAR P = 0 FOR N.PROC
par.process
Sieve in Occam

• Generate Prime numbers
  – With a lot of processes 😊
Sieve in Occam
Numbers

PROC Numbers(CHAN OF INT in, out)

INT i:
SEQ
i:=2
WHILE i <> EndToken
PRI ALT
in ? i
SKIP
TRUE & SKIP
SEQ
out ! i
i := i+1
:

:
EndStop

PROC EndSTOP(CHAN OF INT in, out)
  INT temp:
  SEQ
  in ? temp
  PAR
  out ! EndToken
  WHILE temp <> EndToken
    in ? temp

:
Filter

PROC Filter(CHAN OF INT left, right, down)
INT p, q:
SEQ
  left ? p
  q := 1
PAR
  down ! p
  WHILE q <> EndToken
  SEQ
    left ? q
    IF
      q = EndToken
      SKIP
      (q \ p) <> 0
      right ! q
      TRUE
      SKIP
    SKIP
  right ! EndToken
:
Concentrator

PROC Concentrator([]CHAN OF INT in,
                  CHAN OF INT out)

    INT p:
    SEQ i = 0 FOR SIZE in
    SEQ
        in [i] ? p
        out ! p
    :
Sieve

VAL INT N is 30:
PROC Generate(CHAN OF INT Primes)
    VAL INT EndToken IS 0:
    PROC Numbers(CHAN OF INT in, out)
    PROC EndSTOP(CHAN OF INT in, out)
    PROC Filter(CHAN OF INT, left, right, down)
    PROC Concentrator([]CHAN OF INT in,
        CHAN OF INT out)

[N+1]CHAN OF INT InterFilter:
[N]CHAN OF INT PC:
CHAN INT OK.STOP:
PAR
    Numbers(OK.STOP, InterFilter[0])
    PAR i = 0 FOR N
        Filter(InterFilter[i], InterFilter[i+1], PC[i])
    EndStop(InterFilter[N], OK.STOP)
    Concentrator(PC, Primes)
:


http://wotug.ukc.ac.uk/ocweb/
ocweb Performance
Occam for Linux

- **KRoC**
- Single threaded
- Easy to use
Occam OS

- KRoC runtime library ported to be an operating system in its own right
- Project a collaboration between
  - UKC
  - SDU
Conway's Game of Life