**Dynamic occam**

- **Introduction to Dynamic occam**
  - Motivation and Principles
- **Details**
  - Channel Ends and Direction Specifiers
  - Mobile Channel Structures (and SHARED Channels)
  - Dynamic Process Creation (FORK)
  - Extended Rendezvous
  - Process Priorities (32 levels now supported)
  - Extensions (parallel recursion, nested PROTOCOL definitions, ...)
- **Examples**
  - Dynamic Process Farms
  - Intercepting Channel Communications
  - Networked Channels
  - RMoX and occWeb
- **Summary**

**Motivation and Principles**

- **Motivation**
  - Classical occam embedded systems; hence pre-allocated memory (i.e. compile-time defined concurrency limits, array sizes and no recursion). *It's long been time to move on!*  
  - Remove static constraints (but retain as a voluntary option for use in hardware design and some embedded systems).  
  - Move towards general-purpose capability (because occam is too good to keep to ourselves).  
- **Principles for changes/extensions**
  - they must be useful and easy to use;  
  - they must be semantically sound and policed against misuse;  
  - they must have very light implementation (nano-memory and warp speed);  
  - they must be aligned with the core language (no semantic, safety or performance disturbance).  

**Channel Ends and Direction Specifiers**

An occam process may only use a channel parameter one-way (either for input or for output). That direction is specified (? or !), along with the structure of the messages carried – in this case, simple INTs. The compiler checks that channel usage within the body of the PROC conforms to its declared direction.
Channel Ends and Direction Specifiers

PROC integrate (CHAN INT in?, out!)

parallel implementation

Java (JCSP)

public class IntegrateInt implements CSProcess {
    private final ChannelInputInt in;
    private final ChannelOutputInt out;

    public IntegrateInt (ChannelInputInt in, ChannelOutputInt out) {
        this.in = in;
        this.out = out;
    }

    public void run () {
        ...
    }
}

Mobile Channel Structures

CHAN TYPE BUF.MGR
MOBILE RECORD
CHAN INT req?
CHAN MOBILE [BYTE] buf!
CHAN MOBILE [BYTE] ret?

req?
buf?
ret?

req?
buf?
ret?

Channel types declare a bundle of channels that will always be kept together. They are similar to the idea proposed for occam3, except that the ends of our bundles are mobile...
Mobile Channel Structures

... and we also specify the directions of the component channels ...

CHAN INT req?:    => requested buffer size
CHAN MOBILE []BYTE buf!:  => delivered array
CHAN MOBILE []BYTE ret?:  => returned array

... and [channel bundles, like atomic channels, have two ends which we call, arbitrarily, the "I" (or 'server') end and the 'I' (or 'client') end] ...

CHAN INT req?:    => requested buffer size
CHAN MOBILE []BYTE buf!:  => delivered array
CHAN MOBILE []BYTE ret?:  => returned array

For these mobile channel types, variables are declared only for their ends. Those ends are going to be independently mobile – not the channel as a whole.

BUF.MGR buf.cli,, buf.svr := MOBILE := MOBILE BUF.MGR

Those variables need to be given to separate parallel processes before it makes sense to use them – e.g:

BUF.MGR buf.cli, buf.svr := MOBILE BUF.MGR

However, it’s more flexible (and fun) to take advantage of their mobility.

Mobile channel-end variables may be assigned to each other and sent down channels – strong typing rules apply, of course. Recall, also, the basic rules of mobile assignment and communication: once assigned or communicated from, the mobile variable becomes undefined. It may not be used again until re-allocated, assigned or communicated to.
Mobile Channel Structures

CHAN BUF.MGR! cli.chan:
CHAN BUF.MGR? svr.chan:
PAR
  generator (cli.chan! svr.chan!)
  client (cli.chan?)
  server (svr.chan?)

BUF.MGR! buf.cli:
BUF.MGR? buf.svr:
SEQ
  buf.cli buf.svr := MOBILE BUF.MGR

BUF.MGR! buf.cli:
BUF.MGR? buf.svr:
SEQ
  buf.cli buf.svr := MOBILE BUF.MGR
  cli.chan ! buf.cli
  svr.chan ! buf.svr
  -- buf.cli and buf.svr are now undefined

PROC client (CHAN BUF.MGR! cli.chan?)
  BUF.MGR! cv:
  SEQ
  cli.chan ? cv
Shared Channel-Ends

\[
\text{PROC client.2 (SHARED BUF.MGR! s.buf.cli)}
\]
\[
\text{...}
\]
\[
\text{CLAIM s.buf.cli}
\]
\[
\text{MOBILE [BYT}e b:
\]
\[
\text{SEQ s.buf.cli[req] ! 42}
\]
\[
\text{s.buf.cli[buf] ? b}
\]\n\[
\text{... use b}
\]
\[
\text{s.buf.cli[req]} ! b
\]
\[
\text{...}
\]

Only s.buf.cli channels may be used within its CLAIM block and no nested CLAIMs.

Both Ends Shared

\[
\text{PROC server.2 (SHARED BUF.MGR? s.buf.svr)}
\]
\[
\text{...}
\]
\[
\text{CLAIM s.buf.svr}
\]
\[
\text{MOBILE [BYT}e b:
\]
\[
\text{INT s:
\]
\[
\text{SEQ s.buf.svr[req] ? s}
\]
\[
\text{b := MOBILE [s]BYT}
\]
\[
\text{s.buf.svr[buf] ? b}
\]\n\[
\text{... use b}
\]
\[
\text{s.buf.svr[req]} ! b
\]
\[
\text{...}
\]

Other channels and nested client CLAIMs may be used within a server CLAIM block.

Both Ends Shared

\[
\text{PROBLEM: once a client and server process have made their claims, they can do business across the shared channel bundle. Whilst this is happening, all other client and server processes are locked out from the communication resource.}
\]
\[
\text{SOLUTION: use the shared channel structure just to enable clients and servers to find each other and pass between them a private channel structure. Then, let go of the shared channel and transact business over the private links.}
\]
A client process makes both ends of a non-shared BUF.MGR channel and claims the shared channel. When successful, it sends the server-end of its BUF.MGR down the shared channel. This blocks until a server process claims its end of the shared channel and inputs that server-end.

Note that the client process, having output the server of its (unshared) BUF.MGR channel, no longer has that server-end and cannot use it or send it anywhere else. Only that client has the client-end and only the receiving server has the server-end.

Once that client and server finish their business, the server should return the server-end of the BUF.MGR channel back to the client, who may then reuse it to send to someone else. With a slightly modified definition of BUF.MGR, its server-end may be sent back down itself to the client.

Otherwise, it may have ceased to exist before the forked process terminates. Reference data must beSearch and declared global to the FORKING block.

Dynamic Process Creation

The PAR construct creates processes dynamically, but the creating process has to wait for them all to terminate before it can do anything else.

This is not always what we want! Many processes need to be able to fork off new processes (whose memory will need to be allocated at run-time) and carry on concurrently with them. Examples include web servers and operating systems.

But we are not operating a free-for-all heap in our new occam – strict aliasing control is maintained even for dynamically allocated structures. So, we must take care about memory referenced by long-lived forked processes.
Dynamic Process Creation

Outline of the front-end process farm handling incoming connections to the dynamic version of the occam web server.

PROC fe.farm (CHAN D.COMM? in?, SHARED C.COMM! to.sw)
  D.COMM? local:
  WORKING
  INITIAL INT c IS 0:
  WHILE TRUE
    SEQ
      IN ? local
      WORK fa.proc (c, local, to.sw)
      c := c + 1
    ...

PROC fa.proc (VAL INT n, D.COMM? in, SHARED C.COMM! to.sw)
  ...

Dynamic Process Farms (RMoX)

A pool.manager is responsible for a pool of workers who queue up to request work packets from a farmer.

The pool.manager must ensure that at least min.idle workers are always waiting to request new packets.

Each worker must keep the pool.manager informed as to whether it is working or idle. The pool.manager maintains a count of how many workers are idle and FORKS off new ones as the need arises.

Of course, this means the number of workers can never decrease – it can only ever keep growing. Limiting the number of idle workers to max.idle is left as an exercise.
Dynamic Process Farms (RMoX)

PROC farmer (W.IN? workers)
WHILE TRUE
MOBILE [M]BYTE packet:
SEQ
  manufacture work packet
  BOOL any
  workers[request] ? any
  workers[work] ? packet
;
PROC harvester (W.OUT? workers)
WHILE TRUE
MOBILE [M]BYTE packet:
SEQ
  workers[result] ? packet
  ... consume result packet
;
PROC worker (SHARED W.IN! in, SHARED W.OUT! out, SHARED SIGNAL! signal)
WHILE TRUE
MOBILE [M]BYTE packet:
SEQ
  CLAIM in
  in[request] ! TRUE
  in[work] ? packet
  CLAIM signal
  signal[count] ! –1 -- say we are working
  ... do the work
  CLAIM out
  out[result] ? packet -- hopefully, a modified one
  CLAIM signal
  signal[count] ! +1 -- say we are idle
;
PROC pool.manager (VAL INT min.idle, SHARED W.IN! in, SHARED W.OUT! out)
SHARED SIGNAL! signal.cli:
SIGNAL? signal.svr:
SEQ
  signal.cli, signal.svr := MOBILE SIGNAL
  FORGING
  INITIAL INT n.idle IS 0:
  WHILE TRUE
    SEQ
      (n.idle < min.idle) ==> FORK new workers
      INT n:
      SEQ
        signal.svr[count] ? n -- working/idle (-1/+1)
        n.idle := n.idle + n

Title goes here
Dynamic Process Farms (RMoX)

The dynamic management of process farms is one of the common design idioms used to support:

- RMoX ("Raw Metal occam ix")
  - an experimental operating system for general and real-time embedded applications, built exclusively on this extended CSP model and programmed (almost and eventually) entirely in occam.

Dynamic Process Farms (RMoX)

Extended Rendezvous

This is a convenience – and it’s free!

SEQ
...
... wait for input but do not reschedule outputting process!
... rendezvous block
... The outputting process is unaware of the extended nature of the rendezvous
... reschedule outputting process only after the rendezvous block has terminated

Extended Rendezvous

Here is an informal operational semantics:

$$
\begin{align*}
\text{c} & \rightarrow 42 \\
\text{c} & \rightarrow x \\
\text{c} & \rightarrow \text{rendezvous block}
\end{align*}
$$

Extended Rendezvous

Not that it’s implemented that way!

- **No additional overheads** for normal channel communication.
- Implementation is very lightweight (approx. 30 cycles):
  - **no change** in outputting process code;
  - **new occam Virtual Machine (oVM)** instructions for "??".
- Solves a long-standing semantic anomaly of unhandled tags in variant protocols:
  - $((d \rightarrow \text{apple} ) \rightarrow (d \rightarrow \text{CASE banana}) = \text{STOP}$
A client process makes both ends of a non-shared BUF.MGR channel and claims the shared channel. When successful, it sends the server-end of its BUF.MGR down the shared channel. This blocks until a server process claims its end of the shared channel and inputs that server-end.

Once that client and server finish their business, the server should return the server-end of the BUF.MGR channel back to the client, who may then reuse it to send to someone else. With a slightly modified definition of BUF.MGR, its server-end may be sent down itself back to the client.

Intercept the sent BUF.MGR and forward our own.

Note: client and server processes are unchanged.

Intercept the sent BUF.MGR and forward our own.

FORK l.tap process and plug in loose ends.

client and server processes cannot detect the taps.
Extended Rendezvous Taps

PROC l.tap (BUF.MGR? in, BUF.MGR! out, SHARED LOG! log)
...  tap the req channel
...  tap the buf channel
{{}  tap the ret channel
WHILE TRUE
MOBILE []BYTE b:
in[ret] ?? b
out[ret] ! CLONE b
CLAIM log
log[report] ! returned; b

but this time, we want to stretch the shared (CARRY.BUF.MGR) channel over some communication network without changing the semantics of the system.

Networked Channel Structures

Note: client and server processes are unchanged ...

... and still detect no change in system semantics.
Networked Channel Structures

To set this up, the KRoC programmer (designer) only constructs the named network channel structure – the processes supporting the network are automatically forked and have no impact on system semantics.

Process Priority

- Currently, support for 32 levels of priority (0 = highest)
- Priorities are dynamic (not using PRI PAR)
  - but a process may only change its own priority;
  - which enables very low unit time overheads.
- Currently, priorities set by library routines:
  - PROC SETPRI (VAL INT p.absolute)
  - PROC SELPRI (VAL INT p.relative)
  - PROC INCPRI (VAL INT p.up)
  - PROC DECPRI (VAL INT p.down)
- A process may discover its own priority:
  - INT FUNCTION GETPRI ()
  - GETPRI does not damage the referential transparency of occam expressions.

Additional occam Extensions

- STEP size in replicators
- Fixing the transputer PRI ALT bug
  - Reversing the ALT disable sequence (as done by JCSP)
  - (PRI ALT, SKIP guards and pre-conditions
- Run-time computed PAR replicators
- Parallel Recursion
- RESULT Parameters and Abbreviations
- Nested PROTOCOL Definitions
- In-line Array Constructors
- Anonymous Channel Types
  - e.g. SHAREDCHAN BYTE screen!

Summary

- Everything available in KRoC 1.3.3 ☺☺☺
  - GPL (and some LGPL) open source
  - http://www.cs.ukc.ac.uk/projects/ofa/kroc/
  - occam is now directly applicable to a wide range of industrial/commercial practice:
    - embedded systems, safety-critical, real-time (of course) …
    - operating systems (RMoX), web servers (acctWeb) …
  - web farms, e-commerce, internet and parallel computing …
- Working on:
  - KRoC Network Edition (Mario Schweigler)
  - mobile processes (that carry state)
  - graphics/GUIs (again!)
- Can someone come up with a really good name?!!

URLs

- CSP
  - www.comlab.ox.ac.uk/archive/csp.html
- JCSP
  - www.cs.ukc.ac.uk/projects/ofa/jcsp/
- CTJ
  - www.rt.el.utwente.nl/javapp/
- KRoC
  - www.cs.ukc.ac.uk/projects/ofa/kroc/
- java-threads@ukc.ac.uk
  - www.cs.ukc.ac.uk/projects/ofa/java-threads/
- WoTUG
  - www.wotug.org/
Stop Press

JCSP Networking Edition
KRoC Commercial Support

www.quickstone.com

To get the dynamic capabilities presented in this talk, you need KRoC 1.3.3 or later.

The current (Linux/x86) on the KRoC website (www.cs.ukc.ac.uk/projects/ofa/kroc/) is 1.3.2. Pre-releases of 1.3.3 are available from the occam webserver pages (wotug.ukc.ac.uk/ocweb/), which links off the KRoC site.

Stop Press

A boot image of the RMoX demonstrator is available from the occam webserver pages (wotug.ukc.ac.uk/ocweb/), which links off the KRoC site.

To switch between the demo applications, use the Function keys, F1 through F6.