Making music with occam-$\pi$

Adam Sampson
ats@offog.org

University of Kent
http://www.cs.kent.ac.uk/
Here’s some work I did last year

Originally a fringe presentation at CPA-2006

An interesting application for process-oriented programming

But first, some background...
Electronic music

► ... would be more appropriately called *computational* music
► Generating and processing sound using mathematics
► Not new at all – electronic synthesiser date back to the 1940s
  ► Hammond Novachord, Ondioline, Theremin
Analogue becomes digital

- Originally done with analogue electronics (much like analogue computers)
- Early work with digital computers in the 1950s-60s
  - UNIVAC I (1951), Bell Labs (1962)
- Digital electronics adopted as soon as they became available
- Commercial microprocessor-based systems in the 1970s
  - Synclavier, Fairlight CMI
These days, we use microprocessors, DSPs, . . .

. . . or software on general-purpose computers (“soft synths”)

Some modern keyboards are actually PCs running Windows/Linux!

Interfaces and behaviours heavily influenced by the old analogue world
How does it work?

► Generate “pure” waveforms using oscillators
► ... or process sound from an existing instrument (e.g. voice, guitar)
► Apply operators to modify and combine waveforms
  ► Amplify, filter, mix, distort, modulate, delay ...
► Demo later!
Connecting audio signals between devices is easy

Sending control signals (“play note C-3 at volume 50”) is a bit more complex

MIDI was introduced in 1981

Reliable, low-speed serial links

Standard messages for things like:
  - Note on/off
  - Controller change (e.g. pitch bend, pedals)
  - Generic purpose data dumps (“sysex”)
Little boxes

- We tend to think of this in terms of connecting up boxes
- Literally, with modular synthesisers (from uber.tv):

![Modular synthesiser](image-url)
...all made out of ticky-tacky...

► ...and guitar effects (from guitargeek.com):
...and they all look just the same

► ... which means that software components are often described the same way (from the Roland D-110 manual):

Partial 1 (or 3) is mixed with the ring modulated sound of two Partialis (including Partial 1 or 3).

► Lots of software uses this notation to let you build software synths – Pd, Max/MSP, . . .

► Does this look familiar?
(from about 500 papers about occam-$\pi$– this one’s Mario’s)

We use the same approach when designing process-oriented programs

Boxes are processes; lines are channels
Why is this interesting?

- Like any research group, we’re always looking for applications...
- Fine-grained, high-performance concurrency
- Many potential users who think about problems like we do
- ... and are even using “our” notation
- Want to build reliable, scalable systems
- (Plus many of us are musicians already!)
First shot at building a synthesiser in occam-$\pi$

DATA TYPE SIGNAL IS [BLOCK.SIZE]REAL32:

Many simple components – oscillators, operators, input/output

- Most are direct equivalents of modular synth modules
- Most operators are < 10 lines of code

Can sequence music using occam-$\pi$ code:

```
out ! note; C.3; SQ
```

Supports MIDI input from real devices
A simple component

- Amplifier – just multiply all incoming numbers by a constant:

- Just like the CO631 examples:

```
PROC amp (CHAN SIGNAL in?,
          VAL REAL32 factor,
          CHAN SIGNAL out!)

WHILE TRUE
  SIGNAL s:
  SEQ
    in ? s
    out ! signal ([i = 0 FOR BLOCK.SIZE | s[i] * factor])
```

Making music with occam-π – p.14/31
Problems with OAK

- Completely static – must recompile to change layout or parameters
  - Makes it awkward to develop new sounds
- Not very efficient
  - Data is often copied
  - All processes run on every cycle
- Proved that the concept was workable, though...
Meanwhile, in experimental music...

- People have been creating sounds and music by writing software since the 1960s
- Increasingly important in the last 20 years
- ... but not normally done as part of a performance!
- Why not?
Live programming (2000 -)

- You don’t have to play an instrument to appreciate the performance
- Helps involve the audience more in the performance
  - Often a problem with electronic music
- More opportunities for improvisation – sounds as well as melodies
- Control video/lighting too
- Raises some interesting problems
Languages for live programming

- Must be highly expressive – make changes rapidly
- Must be possible to make incremental changes
- Control over when changes take effect
- Robust against programmer error
- Reliable – avoid glitches in the output and timing problems
- Needs both language and development environment support
- Notion of concurrency
- Existing examples: ChucK, fluxus (Scheme), feedback (Perl), . . .
LP from a POP perspective

- Kernel for lightweight concurrency – check
- Writing occam on the fly is right out!
  - So use the graphical notation the users already understand
  - Graphical process network editor – we’ve done this before
- We know how to build robust POP systems
  - Design component interfaces to support live rewiring
  - Apply design rules on the fly to ensure safety
Time for a demo!
Introducing the Live occam-π Visual Environment...
Proof-of-concept software – sorry if it all goes horribly wrong
What is LOVE?

- The second generation, after OAK
- Components can be created at runtime
- Dynamic, repluggable connections
- GUI – events, visualisation, changing settings
- Data copying is minimised
- Processes can sleep
Components in LOVE

- Same process, with wrappers to provide ports

Making music with occam-π – p.22/31
The code for that

```
PROC id, component (PROC.CTL? ctl)
PROC id (CHAN CHUNK in?, out!)
  WHILE TRUE
    CHUNK ch:
    SEQ
      in ? ch
      out ! ch
    :
    STREAM.WIRE? inw:
    STREAM.WIRE! inw.c:
    PORT.CTL? outp:
    PORT.CTL! outp.c:
    SEQ
      ctl[resp] ! reg, counts; 1; 1
      inw, inw.c := MOBILE STREAM.WIRE
      ctl[resp] ! reg, stream, in; inw.c
      outp, outp.c := MOBILE PORT.CTL
      ctl[resp] ! reg, stream, out; outp.c
      ctl[resp] ! reg, done
    
  CHAN CHUNK thru:
  PAR
    id (inw[c]?, thru!)
    stream, port (thru?, outp)
```
Replugging ports

- Input ports are mobile channels; sending end registered with a central manager process
- Output ports are buffer processes which broadcast to a set of channel ends
  - Manager has a (mobile) channel to each output port
  - Can connect, disconnect mobile channels
- MIDI and audio channels
The manager

- Starts and connects components dynamically in response to GUI events
- Enforces rules about which ports can connect to which
  - Type-checking
  - Avoid cycles
- Generic; does not know what audio is, just that it’s a type of port
Rolling your own GUI is bad, but for now...  
All based on vectors; scalable  
Hierachy of GUI components
  Window contains components, which contain buttons...  
Events filter down, draw lists filter back up  
Processes provided for standard GUI components (buttons, text boxes, sliders) and event filtering  
Seems to work well
The POP model is a natural fit for audio synthesis

... even within the constraints of live programming

We can use POP design rules to make it easier to build correct synthesis networks

Process-oriented programs are pretty : – )
► It’s pretty easy to make existing occam-π processes dynamically pluggable

► In conjunction with other work we’ve done (POPExplorer, etc.), this might lead toward a useful tool

► for teaching music to occam-π programmers?
► for teaching occam-π to musicians?
Future work

- Better synchronisation (see Carl’s work)
- Creating new components at runtime
  - Draw a network, drag a box around it
- Convert to occam code (and back?)
- Saving, deleting, . . .
- A better-designed GUI library
Thanks for listening!

- The code is available from here: http://offog.org/darcs/research/love/
- Any questions?