occam-pi Multiway Sync Models

Peter Welch (Kent University)

TUNA Project Meeting
York University
5th. December, 2005

(updated 12th. December, 2005)

This looks at directly modeling Steve Schneider's FDR script in occam-pi:

`clump1d2d3dtock-2005-12-06.fdr2`

A CELL process holds at most *one* platelet and imposes a speed limit:

```
channel tock
```

```
channel enterCell, exitCell

CELL  = enterCell -> tock -> FCELL []
        tock -> CELL

FCELL = exitCell -> CELL []
        tock -> FCELL
```

```
channel pass : {0..N}

AC (i) = {pass.i, pass.i+1, tock}
```

```
C (i) = CELL [[enterCell <- pass.i,
               exitCell <- pass.i+1]]
```

```
LINE = || i:{0..N-1} @ [AC (i)] C (i)
```

STICKYAB models the *sticking* of one platelet to another:

```
channel enterA, exitA, enterB
```

```
STICKAB = EMPTYA

EMPTYA  = enterA -> ALMOSTA []
          enterB -> EMPTYA []
          tock -> EMPTYA

ALMOSTA = tock -> FULLA []
          enterB -> ALMOSTA []
          exitA -> EMPTYA

FULLA   = exitA -> EMPTYA []
          enterB -> exitA -> EMPTYA []
          tock -> FULLA
```

delete this line and we won't need the **CELL** processes!

In 1-D, STICKYAB controls three adjacent cells (where “A” is cell “i” and “B” is cell “i+2”):

```
AF (i) = {pass.i, pass.i+1, pass.i+2, tock}
```

```
F (i) = STICKYAB [[enterA <- pass.i,
                   exitA <- pass.i+1,
                   enterB <- pass.i+2]]
```

```
FS = || i:{0..N-2} @ [AF (i)] F (i)
```

```
SYS1D = LINE |{|pass, tock|}| FS
```

What we are aiming for:

```
PROC sticky.ab (ALT.BARRIER enter.a, exit.a,
                enter.b, tock)

:
```

```
STICKAB = EMPTYA

EMPTYA  = enterA -> ALMOSTA []
          enterB -> EMPTYA []
          tock -> EMPTYA

ALMOSTA = tock -> FULLA []
          enterB -> ALMOSTA []
          exitA -> EMPTYA

FULLA   = exitA -> EMPTYA []
          enterB -> exitA -> EMPTYA []
          tock -> FULLA
```

```
PROC sticky.ab (ALT.BARRIER enter.a, exit.a,
                enter.b, tock)
  VAL INT EMPTY.A IS 0:
  VAL INT ALMOST.A IS 1:
  VAL INT FULL.A IS 2:
  INITIAL INT state IS EMPTY.A:
:
```

```
STICKAB = EMPTYA

EMPTYA  = enterA -> ALMOSTA []
          enterB -> EMPTYA []
          tock -> EMPTYA

ALMOSTA = tock -> FULLA []
          enterB -> ALMOSTA []
          exitA -> EMPTYA

FULLA   = exitA -> EMPTYA []
          enterB -> exitA -> EMPTYA []
          tock -> FULLA
```

```
PROC sticky.ab (ALT.BARRIER enter.a, exit.a,
                enter.b, tock)
  VAL INT EMPTY.A IS 0:
  VAL INT ALMOST.A IS 1:
  VAL INT FULL.A IS 2:
  INITIAL INT state IS EMPTY.A:
  WHILE TRUE
    CASE state
      ...  EMPTY.A case
      ...  ALMOST.A case
      ...  FULL.A case
:
```

```
STICKAB = EMPTYA

EMPTYA  = enterA -> ALMOSTA []
          enterB -> EMPTYA []
          tock -> EMPTYA

ALMOSTA = tock -> FULLA []
          enterB -> ALMOSTA []
          exitA -> EMPTYA

FULLA   = exitA -> EMPTYA []
          enterB -> exitA -> EMPTYA []
          tock -> FULLA
```

```
FULLA   = exitA -> EMPTYA []
          enterB -> exitA -> EMPTYA []
          tock -> FULLA
```

```
{{{  FULL.A case
FULL.A
  ALT
    SYNC exit.a
      state := EMPTY.A
    SYNC enter.b
      SEQ
        SYNC exit.a
        state := EMPTY.A
    SYNC tock
      SKIP

}}}
```

What we can (just about) do:

using the “oracle” process …

```
PROC sticky.ab (VAL INT id,
     VAL INT enter.a, exit.a, enter.b, tock,
     SHARED CHAN ALT.SYNC.START to.oracle!,
     CHAN ALT.SYNC.FINISH from.oracle?)

:
```

```
STICKAB = EMPTYA

EMPTYA  = enterA -> ALMOSTA []
          enterB -> EMPTYA []
          tock -> EMPTYA

ALMOSTA = tock -> FULLA []
          enterB -> ALMOSTA []
          exitA -> EMPTYA

FULLA   = exitA -> EMPTYA []
          enterB -> exitA -> EMPTYA []
          tock -> FULLA
```

```
PROC sticky.ab (VAL INT id,
    VAL INT enter.a, exit.a, enter.b, tock,
    SHARED CHAN ALT.SYNC.START to.oracle!,
    CHAN ALT.SYNC.FINISH from.oracle?)
  ...  VAL INT EMPTY.A, ALMOST.A, FULL.A
  ...  VAL OFFER emtpy.a, almost.a, full.a, full.a.b
  INITIAL INT state IS EMPTY.A:

:
```

```
STICKAB = EMPTYA

EMPTYA  = enterA -> ALMOSTA []
          enterB -> EMPTYA []
          tock -> EMPTYA

ALMOSTA = tock -> FULLA []
          enterB -> ALMOSTA []
          exitA -> EMPTYA

FULLA   = exitA -> EMPTYA []
          enterB -> exitA -> EMPTYA []
          tock -> FULLA
```

```
PROC sticky.ab (VAL INT id,
    VAL INT enter.a, exit.a, enter.b, tock,
    SHARED CHAN ALT.SYNC.START to.oracle!,
    CHAN ALT.SYNC.FINISH from.oracle?)
  ...  VAL INT EMPTY.A, ALMOST.A, FULL.A
  ...  VAL OFFER emtpy.a, almost.a, full.a, full.a.b
  INITIAL INT state IS EMPTY.A:
  WHILE TRUE
    CASE state
      ...  EMPTY.A case
      ...  ALMOST.A case
      ...  FULL.A case
:
```

```
STICKAB = EMPTYA

EMPTYA  = enterA -> ALMOSTA []
          enterB -> EMPTYA []
          tock -> EMPTYA

ALMOSTA = tock -> FULLA []
          enterB -> ALMOSTA []
          exitA -> EMPTYA

FULLA   = exitA -> EMPTYA []
          enterB -> exitA -> EMPTYA []
          tock -> FULLA
```

```
{{{  VAL OFFER emtpy.a, almost.a, full.a, full.a.b
VAL OFFER empty.a IS [enter.a, enter.b, tock]:
VAL OFFER almost.a IS [tock, enter.b, exit.a]:
VAL OFFER full.a IS [exit.a, enter.b, tock]:
VAL OFFER full.a.b IS [exit.a]:
}}}
```

```
STICKAB = EMPTYA

EMPTYA  = enterA -> ALMOSTA []
          enterB -> EMPTYA []
          tock -> EMPTYA

ALMOSTA = tock -> FULLA []
          enterB -> ALMOSTA []
          exitA -> EMPTYA

FULLA   = exitA -> EMPTYA []
          enterB -> exitA -> EMPTYA []
          tock -> FULLA
```

```
PROC sticky.ab (VAL INT id,
    VAL INT enter.a, exit.a, enter.b, tock,
    SHARED CHAN ALT.SYNC.START to.oracle!,
    CHAN ALT.SYNC.FINISH from.oracle?)
  ...  VAL INT EMPTY.A, ALMOST.A, FULL.A
  ...  VAL OFFER emtpy.a, almost.a, full.a, full.a.b
  INITIAL INT state IS EMPTY.A:
  WHILE TRUE
    CASE state
      ...  EMPTY.A case
      ...  ALMOST.A case
      ...  FULL.A case
:
```

```
STICKAB = EMPTYA

EMPTYA  = enterA -> ALMOSTA []
          enterB -> EMPTYA []
          tock -> EMPTYA

ALMOSTA = tock -> FULLA []
          enterB -> ALMOSTA []
          exitA -> EMPTYA

FULLA   = exitA -> EMPTYA []
          enterB -> exitA -> EMPTYA []
          tock -> FULLA
```

```
FULLA = exitA -> EMPTYA []
        enterB -> exitA -> EMPTYA []
        tock -> FULLA
```

```
{{{  FULL.A case
FULL.A
  INT result:
  SEQ
    CLAIM to.oracle!
      to.oracle ! id; full.a
    from.oracle ? result; full.a
    CASE result
      ...  exit.a
      ...  enter.b
      ...  tock
}}}
```

[exit.a, enter.b, tock]

```
FULLA = exitA -> EMPTYA []
        enterB -> exitA -> EMPTYA []
        tock -> FULLA
```

```
{{{  exit.a
exit.a
  state := EMPTY.A
}}}
```

```
{{{  tock
tock
  SKIP
}}}
```

```
FULLA = exitA -> EMPTYA []
        enterB -> exitA -> EMPTYA []
        tock -> FULLA
```

```
{{{  enter.b
enter.b
  SEQ
    CLAIM to.oracle!
      to.oracle ! id; full.a.b
    from.oracle ? result; full.a.b
    state := EMPTY.A
}}}
```

[exit.a]

exit.a

Let's use the CELL processes to synchronise visualisation (after a tock) of their state:

```
channel tock
```

```
channel enterCell, exitCell

CELL  = enterCell -> tock -> FCELL []
        tock -> CELL

FCELL = exitCell -> CELL []
        tock -> FCELL
```

```
channel pass : {0..N}

AC (i) = {pass.i, pass.i+1, tock}
```

```
C (i) = CELL [[enterCell <- pass.i,
               exitCell <- pass.i+1]]
```

```
LINE = || i:{0..N-1} @ [AC (i)] C (i)
```

For each **CELL** process, we add a **draw** event which *always* follows a **tock**:

```
channel tock
```

```
channel enterCell, exitCell

CELL  = enterCell -> tock -> FCELL []
        tock -> CELL

FCELL = exitCell -> CELL []
        tock -> FCELL
```

```
channel pass : {0..N}

AC (i) = {pass.i, pass.i+1, tock}
```

```
C (i) = CELL [[enterCell <- pass.i,
               exitCell <- pass.i+1]]
```

```
LINE = || i:{0..N-1} @ [AC (i)] C (i)
```

For each CELL process, we add a draw event which *always* follows a tock:

```
channel tock, draw
```

```
channel enterCell, exitCell

CELL  = enterCell -> tock -> draw -> FCELL []
        tock -> draw -> CELL

FCELL = exitCell -> CELL []
        tock -> draw -> FCELL
```

```
channel pass : {0..N}

AC (i) = {pass.i, pass.i+1, tock, draw}
```

```
C (i) = CELL [[enterCell <- pass.i,
               exitCell <- pass.i+1]]
```

```
LINE = || i:{0..N-1} @ [AC (i)] tock -> draw -> C (i)
```

Visualisation will take place between a tock and a draw – *when the cells can't change state:*

```
channel tock, draw
```

```
channel enterCell, exitCell

CELL  = enterCell -> tock -> draw -> FCELL []
        tock -> draw -> CELL

FCELL = exitCell -> CELL []
        tock -> draw -> FCELL
```

```
channel pass : {0..N}

AC (i) = {pass.i, pass.i+1, tock, draw}
```

```
C (i) = CELL [[enterCell <- pass.i,
               exitCell <- pass.i+1]]
```

```
LINE = || i:{0..N-1} @ [AC (i)] tock -> draw -> C (i)
```

A `DISPLAY` process also synchronises on `tock` and `draw`, rendering the *full* or *empty* states of the cells in between:

```
DISPLAY = (tock -> RENDER_CELLS); (draw -> DISPLAY)
```

```
RENDER_CELLS = ...
```

The `CELL` processes share *state variables* with `DISPLAY`, which observes them only during `RENDER_CELLS`.

These *state variables* could be easily modeled as processes with `load` and `store` channels.

Note also that *all* process *commit* to the draw event – *it is never offered as an option!*

```
channel enterCell, exitCell

CELL  = enterCell -> tock -> draw -> FCELL []
        tock -> draw -> CELL

FCELL = exitCell -> CELL []
        tock -> draw -> FCELL
```

```
C (i) = CELL [[enterCell <- pass.i,
               exitCell <- pass.i+1]]
```

```
LINE = || i:{0..N-1} @ [AC (i)] tock -> draw -> C (i)
```

```
DISPLAY = (tock -> RENDER_CELLS); (draw -> DISPLAY)
```

```
PROC cell (ALT.BARRIER enter, exit, tock,
           BARRIER draw, BYTE pixel)
  INITIAL BOOL empty IS TRUE:
  SEQ
    pixel := empty.colour
    SYNC tock
    SYNC draw
    WHILE TRUE
      IF
        empty
          ...  empty
        TRUE
          ...  full
:
```

pixel rendering takes place between these SYNCs

```
CELL  = enter -> tock -> draw -> FCELL
        []
        tock -> draw -> CELL

FCELL = exit -> CELL
        []
        tock -> draw -> FCELL
```

```
{{{  empty case
ALT
  SYNC enter
    SEQ
      pixel := full.colour
      empty := FALSE
      SYNC tock
      SYNC draw
  SYNC tock
    SYNC draw
}}}
```

***pixel rendering takes place between these* SYNCs**

***… and between these* SYNCs**

```
CELL  = enter -> tock -> draw -> FCELL
        []
        tock -> draw -> CELL

FCELL = exit -> CELL
        []
        tock -> draw -> FCELL
```

```
{{{  full case
ALT
  SYNC exit
    SEQ
      pixel := empty.colour
      empty := TRUE
  SYNC tock
    SYNC draw
}}}
```

pixel rendering takes place between these **SYNCs**

```
CELL  = enter -> tock -> draw -> FCELL
        []
        tock -> draw -> CELL

FCELL = exit -> CELL
        []
        tock -> draw -> FCELL
```

Alternatively, we could equip each `CELL` process with a `drawing` channel to which it would output its state (if changed) immediately following a `tock`.

The `DISPLAY` process could then monitor the `drawing` channels and promise to *service* them – i.e. always accept them. It would not have to engage with `tock`.

Many other scenarios for visualisation are possible …

The described method of giving each `CELL` process direct access to the pixel used for rendering may be the most efficient.