Communicating Complex Systems

transforming formal models of complex systems into executable code
implementing complex functionality in and for occam-$\pi$

Joint work with Peter Welch (University of Kent)
and Fiona Polack (University of York)
Communicating Complex Systems

Contents

- Motivation
- A quick introduction to occam-π
- A simplified blood-clotting model
- Transformation into occam-π
- Multi-way synchronisation in occam-π
- Further developments
- Conclusions
We want to build complex systems, interested in emergent behaviour

- modelling biological systems — in this case, blood clotting (haemostatis)
- discovering the parameters of systems needed to do this: can we do it with 10 ‘processes’? a thousand? a million? what concurrency abstractions are needed?
- systems built from independent, interacting agents, operating within some environment — an approximation of the real-world

Need to be sure that our agents are safe in the implementation

- making sure that when we put hundreds of thousands together, the whole thing won’t deadlock, and that the results we observe are trustworthy
- use process calculi such as Circus [Woodcock et al.], CSP [Hoare] and the \( \pi \)-calculus [Milner] to formally reason about such systems
Motivation

Using occam-$\pi$ for the implementation

- CSP related semantics, with ideas of mobility from the $\pi$-calculus
- ultra-lightweight concurrency overheads — can handle up to a million simple processes on a modern desktop PC
- clean mapping from formal specification to implementation

Driving forward the development of occam-$\pi$

- our models make heavy use of **multi-way synchronisations**
- processes (agents) may make and withdraw offers of their own accord
- require an efficient and safe implementation for this, which previously did not exist in occam-$\pi$
A Quick Introduction to occam-π

- Systems are built from layered **networks** of communicating **processes**
  - linkage with single **channels** (synchronised, unbuffered point-to-point communication) and **bundles** of channels
  - extended to handle **shared** channels (communication still between two processes only) — also **barriers**, but only committed synchronisations
  - dynamic process creation, and more ...

Fred Barnes, August 2006
occam-π has been used successfully to create systems which could be described as ‘complex’

- operating-system components (the RMoX system)
- interactive games, a web-server, ...

Fred Barnes, August 2006
The Blood Clotting Model

We start with a simplified version of the models: 1-D pipeline of site processes
- a site is either empty or full, indicating that it contains a sticky platelet
- platelets are injected at one end and ‘flow’ down the pipeline
- when one bumps into another, they stick

Can experiment with the implementation by tweaking parameters such as the probability of a clot advancing and injection rate
The CSP model uses a series of SITE processes, synchronising on shared events to control progress.

\[
\text{EVENTS}(i) = \{ \text{pass}.i, \text{pass}.i+1, \text{pass}.i+2, \text{tock} \}
\]

\[
\text{SYSTEM} = \big| i: \{0..(N-1)\} \big| \text{EVENTS}(i) \ 	ext{SITE}(i)
\]

\[
\text{SITE}(i) = \text{EMPTY}(i)
\]

\[
\text{EMPTY}(i) =
\begin{align*}
\text{pass}.i & \rightarrow \text{ALMOST}(i) [ ] \\
\text{pass}.i+2 & \rightarrow \text{EMPTY}(i) [ ] \\
\text{tock} & \rightarrow \text{EMPTY}(i)
\end{align*}
\]

\[
\text{ALMOST}(i) =
\begin{align*}
\text{pass}.i+2 & \rightarrow \text{ALMOST}(i) [ ] \\
\text{tock} & \rightarrow \text{FULL}(i)
\end{align*}
\]

\[
\text{FULL}(i) =
\begin{align*}
\text{pass}.i+1 & \rightarrow \text{EMPTY}(i) [ ] \\
\text{pass}.i+2 & \rightarrow \text{pass}.i+1 \rightarrow \text{EMPTY}(i) [ ] \\
\text{tock} & \rightarrow \text{FULL}(i)
\end{align*}
\]
Almost a direct conversion, which could be done automatically

EVENTS(i) = \{pass.i, pass.i+1, pass.i+2, tock\}
SYSTEM = || i:{0..(N-1)}
\@
\{\text{EVENTS}(i)\} SITE(i)

SITE(i) = EMPTY(i)
EMPTY(i) =
  pass.i -> ALMOST(i) []
  pass.i+2 -> EMPTY(i) []
  tock -> EMPTY(i)

(further details given in the paper)
Implementations of this typically involve a **two-phase commit** protocol

- using a different **manager** process for each event
- managers receive offers at any time, counting down to zero
- when all enrolled processes have offered, manager starts to commit
- offers can be withdrawn at any time, including after zero has been reached

Offers, withdrawals and commits fly around in parallel, two-phase protocol is needed to secure the operation — moderately expensive

![Diagram of multiway synchronisation process]

- wait for offers
- signal commit-1
- wait for responses
- signal collapse
- signal commit-2
- all committed
- some withdrew
- \( c = 0 \)
When explaining the system to a colleague at Kent, he wondered why all manager processes weren’t integrated into a single, serial, process dealing with offers one at a time — we wondered too!

- parallel offers, countdowns, cancel messages and collapses disappear
- something is lost, however: some viable choices may never be made because of the way in which the oracle process operates — still legal choices, however

Processes make all their offers at once, oracle responds with selected index

- internally maintains counters for each event, synchronising when those reach zero — no second stage needed
Inside the Oracle

<table>
<thead>
<tr>
<th>events</th>
<th>counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>processes</td>
<td>offers</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0, 1</td>
</tr>
<tr>
<td>2</td>
<td>0, 1, 2</td>
</tr>
<tr>
<td>3</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>4</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>5</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>m-1</td>
<td>0, 1, ..., n-1</td>
</tr>
</tbody>
</table>

... and so on
Beyond the Oracle (1)

- There main drawback of the oracle technique is the explicit plumbing required
  - the ideal conversion uses the ALT language binding
- A modified BARRIER type is under construction in a new occam-π compiler
  - based on a decentralised version of the oracle logic
  - would provide a sufficient implementation for the ALT.BARRIER in the paper

Because a process may be waiting on multiple events, must force all disabling sequences to happen when synchronisation is complete (before any more multiway syncs can begin) — done with a simple global semaphore
  - and some extra logic in ALT enabling/disabling

*event structure*

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int ref_count</td>
<td>3</td>
</tr>
<tr>
<td>int enroll_count</td>
<td>2</td>
</tr>
<tr>
<td>int down_count</td>
<td>1</td>
</tr>
<tr>
<td>waitq_t *fptr</td>
<td></td>
</tr>
<tr>
<td>waitq_t *bptr</td>
<td></td>
</tr>
</tbody>
</table>

*wait-queue*

- waitq_t *next
- uint *wptr
- uint pri

'waitq' structures allocated dynamically as processes offer

(alting process)
Functional implementation of multiway synchronisations, but some drawbacks
- dynamic allocation and free of the wait-queue
- does not support **interleaving** or **partial synchronisation**

However, now implementing another one which does :-(
- requires some non-trivial compiler support to allocate data structures

```c
// barrier
int sets_enrolled 1
int sets_downcount 1
parbar_t *set_fp ptr
parbar_t *set_bptr

// par-barrier
parbar_t *next_set
parbar_t *prev_set
parbar_t *parent_set
bar_t *bar_link
int enroll_count 2
int sync_count 2
int down_count 1
procbar_t *q_fp tr
procbar_t *q_bptr

// proc-barrier
procbar_t *q_next
procbar_t *q_prev
parbar_t *pbar_link
uint *wptr
int alt_flags
```

sync-count specifies how many processes required to offer in this set
down-count may drop below zero, indicating surplus; does sync as it goes past
Beyond the Oracle (2)

- Implementation of this in the run-time system is mostly complete
  - still developing the compiler support for handling it:
    mostly allocating data structures in the right places for processes
  - will also be used by another mechanism in the same compiler:
    direct compilation of CSP (to be presented at CPA-2006)

- Provides a nice solution to the classic santa-claus problem, still thinking about
  the language binding:

```plaintext
PROC santa (BARRIER elves, reindeer)
WHILE TRUE
  PRI ALT
    elves
      ... meet with elves
    reindeer
      ... go deliver presents

PAR i = 0 FOR 9
  reindeer (r.sync)
PAR i = 0 FOR 10 INTERLEAVE e.sync(3)
  elf (e.sync)
```
Conclusions

- The work presents aspects of the occam-\(\pi\) simulation of the TUNA platelet case study, with a novel and fast mechanism for resolving CSP external choice between multiway synchronisation events from which any participant may withdraw its offer at any time.
  - efficient implementation of this in occam-\(\pi\), the developments of which enable the simulation of more complex clotting models involving interleaving.

- Using the behaviour of this model as a comparison for other occam-\(\pi\) implementations — already have efficient lazy simulations.

- Towards a better understanding of the rules and requirements for nanotech simulations.

- Beyond that, other biological, chemical and physical simulations.
Further Information

- This work was carried out as part of the EPSRC funded TUNA project (EP/C516966/1), between the Universities of York, Surrey and Kent:
  - http://www.cs.york.ac.uk/nature/tuna/index.htm

- Information about occam-π can be found at:
  - http://www.occam-pi.org/

- Information about the KRoC occam-π system and new compiler can be found at:
  - http://www.cs.kent.ac.uk/projects/ofa/kroc/
  - http://www.cs.kent.ac.uk/projects/ofa/nocc/

- The TUNA project team:
  - **York**: Ana Cavalcanti, Heather Turner, Susan Stepney, Jim Woodcock
  - **Surrey**: Helen Treharne, Steve Schneider
  - **Kent**: Fred Barnes, Peter Welch