A CSP Model for Mobile Channels
Peter Welch and Fred Barnes (University of Kent)

IFIP WG 2.4, Glasgow (18th. July, 2006)

Background ...

The dynamic construction of channels and processes, together with their communication through channels, enables network topology to evolve in response to run-time demands.

Systems needing this capability abound (modelling complex biological phenomena, commercial Internet applications, ...).

Formal specification and analysis was pioneered through Milner’s π-calculus. Here, we present a model of channel mobility based on Hoare’s CSP and say why ...

Why ... problems ... loophole ...

Mobile channels have been introduced into the occam-π multi-processing language, whose classical design and semantics are founded on CSP. That foundation is long established, pretty fundamental and has proven very successful.

CSP processes synchronise on fixed sets of events, so cannot dynamically acquire new connections (channels). That static nature cannot be relaxed – its semantics depends on it.

However, CSP allows infinite sets and recursion ... which gives us lots of freedom when modelling. 😊😊😊

Wins ...

The CSP channel mobility model yields a semantics that is both operational and denotational.

The operational aspect lays out a formal specification for all data structures and algorithms for a supporting run-time kernel (and from which that part of the occam-π kernel is derived).

The denotational side preserves the compositional nature of components (parallel composition brings no surprises). It also enables formal specification and analysis of applications and, if not too large, the use of automated model checkers (FDR).

Talk roadmap ...

Mobile channels in occam-π (review) ...
Modelling channels with processes ...
The CSP model (kernel processes) ...
The CSP model (application processes) ...
End note …
Mobile Channel Structures

Mobile Channel Structures

Mobile Channel Structures

Mobile Channel Structures

Channel types declare a bundle of channels that will always be kept together. They are similar to the idea proposed for ocaml3, except that the ends of our bundles are mobile...

... and we also specify the directions of the component channels ... from the point of view of the "F" end.

For these mobile channel bundles, variables are declared only for their ends. Those ends are independently mobile – not the bundle-as-a-whole.

They are allocated in pairs dynamically:

client.end, server.end == MOBILE BUF.MGR

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

MOBILE [][BYTE b:
  SEQ
  client.end[req] ? 42
  client.end[buf] ? b
  ... use b
  client.end[ret] ! b
]

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.

... and channel bundles, like *atomic* channels, have two ends which we call, arbitrarily, the ‘F’ (or ‘server’) end and the ‘I’ (or ‘client’) end...
Mobile Channel Structures

Generator

Client: cli.chan

Server: svr.chan

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

BUF.MGR! cli.chan:
BUF.MGR? svr.chan:
SEQ
  client.end server.end := MOBILE BUF.MGR

BUF.MGR! cli.chan:
BUF.MGR? svr.chan:
SEQ
  client.end server.end := MOBILE BUF.MGR
  cli.chan ! client.end

---

Client end, server end are now undefined

PROC client (CHAN BUF.MGR! cli.chan?)
BUF.MGR! client.end:
SEQ
  cli.chan ! client.end

PROC client (CHAN BUF.MGR! cli.chan?)
BUF.MGR! client.end:
SEQ
  cli.chan ! client.end

Title goes here
**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 processes are locked out from the communication resource.

**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 connection.

Set up a similar network, but with the shared channel type being **CARRY.BUF.MGR** (rather than **BUF.MGR**).

**CARRY.BUF.MGR** is a bundle of just one channel, along which **server-ends** of **BUF.MGR** bundles may be carried.
Both Ends Shared

A client process constructs both ends of an unshared BUF.MGR bundle and sends the shared end of its channel. When successful, it sends the server process claims its end of the shared channel. This blocks until a server process claims its end of the shared channel. This blocks until a server has the client-end and only the receiving server has the server-end. Note that the client process, having output the server-end of its (unshared) BUF.MGR channel, no longer has that channel and cannot use it or send it anywhere else. Only that client has the client-end and the receiving server has the server-end.

Talk roadmap ...

Mobile channels in occam-π (review) ...
Modelling channels with processes ...
The CSP model (kernel processes) ...
The CSP model (application processes) ...
End note ...

Mobile Channels – a CSP Model and Implementation

We can’t directly model occam-π mobile channel-ends as CSP channels because of the dynamics (mobility) and the concept of ends, which are unknown to CSP channels.

Instead, we model a mobile channel as a process, instrumented with conventional channels attached to either end. Each such process (i.e. mobile channel) is constructed on demand and given a unique index number. Mobility derives from simply communicating that index.
Channel $\rightarrow$ Process (occam-$\pi$)

**PROC** ChanC (CHAN INT writeC?, ackC?, readC?)
WHILE TRUE
INT x:
SEQ
writeC ? x
readC ! x
ackC ! 0
SEQ
writeC ! 42
ackC ? any
readC ? x
readC ? x

Given the occam-$\pi$ system constraints (unidirectional channels and no output guards), these systems are equivalent …

Channel $\rightarrow$ Process (CSP)

Given the occam-$\pi$ system constraints (unidirectional channels and no output guards), these systems are equivalent …

… except that ChanC really should terminate when $P'$ and $Q'$ terminate. This is easy to arrange and is handled properly later.

But the above model gives us a way to talk about channel ends, as opposed to just channels.
Mobile channels in \texttt{accom} (review) …

Modelling channels with processes …

The \textit{CSP} model (kernel processes) …

The \textit{CSP} model (application processes) …

End note …

---

A channel \textit{bundle} is a parallel collection of processes: one holding a \textit{reference count} (and responsible for termination), two \textit{semaphores} (one for each possibly shared end) and one \textit{channel process} for each field in the bundle.

---

\begin{verbatim}
Bundle (c, fields)

enroll.c
resign.c
claim.c.server
release.c.server
claim.c.client
release.c.client
write.c.i.p
ack.c.i
read.c.i.p

index number
%chains in bundle
\end{verbatim}

\begin{verbatim}
Bundle (c, fields)

enroll.c
resign.c
claim.c.server
release.c.server
claim.c.client
release.c.client
write.c.i.p
ack.c.i
read.c.i.p

%Refs (c, 2)

Semaphore (c, server)
Semaphore (c, client)
Channels (c, fields)

where server = 0 client = 1
\end{verbatim}

---

\begin{verbatim}
Refs (c, 0) = enroll.c \rightarrow \text{Refs (c, n + 1)} \quad \text{if}
resign.c \rightarrow \text{Refs (c, n = 1)}

Ref (c, 0) = kill \rightarrow \text{STOP}

terminate sibling
\end{verbatim}
Kernel Processes

Mobile channel bundle processes are generated as needed by the following server:

\[
\begin{align*}
\text{MCMC} (cc) &= \text{setMC} f \to \text{getMC} c \to \\
&\quad \langle \text{Bundle} (q, f) \rangle \quad \text{MC} (q + 1)) \\
&\quad \text{noMoreBundles} \to \text{SKIP}
\end{align*}
\]

The kernel starts this with index 1. We reserve index 0 for an undefined bundle.

For simplicity in this model, application processes sometimes try to resign from undefined bundles. To cope, we define:

\[
\begin{align*}
\text{undefined} &= 0 \\
\text{UndefinedBundle} &= \text{resign.undefined} \to \text{UndefinedBundle} [] \\
&\quad \text{noMoreBundles} \to \text{SKIP}
\end{align*}
\]

The mobile bundle kernel is:

\[
\text{MOBILE_BUNDLE_KERNEL} = \\
\text{MC} (i) \mid \text{noMoreBundles} \mid \text{UndefinedBundle}
\]

Let’s define:

\[
\text{kernel_chans} = \\
\quad \{\text{enrol}, \text{resign}, \text{claim}, \text{release}, \text{write}, \text{read}, \text{ack}, \\
\text{setMC}, \text{getMC}, \text{noMoreBundles} \}
\]

Here’s a diagram ...

Application and Kernel Processes

Then, if \( \text{APPLICATION_SYSTEM} \) is the \( \text{occam-\pi} \) application and \( \text{APPLICATION_SYSTEM}'' \) is the CSP modelling of its mobile channel bundle primitives \( \text{(in a minute)} \), the full model is:

\[
\langle \text{APPLICATION_SYSTEM''}, \text{noMoreBundles} \to \text{SKIP} \rangle \\
\text{kernel_chans} \\
\text{MOBILE_BUNDLE_KERNEL} \setminus \text{kernel_chans}
\]

Mobile channels in \( \text{occam-\pi} \) (review) …

Modelling channels with processes …

The CSP model (kernel processes) …

The CSP model (application processes) …

End note …
The paper describes how these map down to pure variables and assignment into variables. We express this using Circuits. Amongst other things, it adds the CSP model down to pure CSP. The paper describes how these map down to pure CSP.

We express this using Circuits. Amongst other things, it adds the CSP model down to pure CSP. The paper describes how these map down to pure CSP.

CSP

\[ P(x) \]

... just a reminder!

\[ P''((xx)) \]

\[ res pledge.xxx \]

\[ release.xxx \]

\[ claim.xxx \]

\[ release.xxx \]

\[ claim.xxx \]

... just a reminder!

... just a reminder!

... just a reminder!
Modelling ollcam-π Mobile Channels

**Sending an unshared mobile bundle end:**

\[ m ! cli \rightarrow (cli := undefined) \]

\[ m ! svr \rightarrow (svr := undefined) \]

**Sending a shared mobile bundle end:**

\[ m ! cli \rightarrow m!cli \rightarrow SKIP \]

\[ m ! svr \rightarrow m!svr \rightarrow SKIP \]

**Receiving a shared / unshared mobile bundle end:**

\[ m ? cli \rightarrow m!cli \rightarrow (cli := tmp) \]

\[ m ? svr \rightarrow m!svr \rightarrow (svr := tmp) \]

**Assignment of shared / unshared mobile bundle ends:**

\[ a := b \rightarrow (a := b); (b := undefined) \]

**Forking processes must take place within a FORKING block (by default, the whole system):**

\[ FORKING \]

\[ (((XX\'\'; done \rightarrow SKIP)) \mid \mid \mid FORK_P) \]

\[ \mid \mid \mid FORK_P \]

where:

\[ FORK_P \rightarrow \forall (P' := 0); \rightarrow (P := 0); \rightarrow (P := 0) \]

\[ \rightarrow (P := 0) \]

The done event ensures that the forking block, xx, cannot terminate until all processes forked in the block have also terminated.
Summary and the Future

We now have a complete model for mobile channels (and mobile barriers) in CSP, so we can apply formal reasoning to the design and analysis of occam-π systems. Model checking requires a (small) modification to constrain numbers to finite limits for particular systems!

The occam-π barrier, forking and mobile mechanisms seem to be delivering their promises. Performance is extremely lightweight. Run-time overheads are around 100 nanoseconds per operation (mobile bundle construction, communication and claiming, 3GHz Pentium 4) – even with poor cache behaviour in the application.

Applications like those for TUNA will be a strong testing ground for the mixing of the dynamic mechanisms of the π-calculus into CSP.

"A CSP Model for Mobile Channels", CPA 2008