A CSP Model for Mobile Channels

Peter Welch and Fred Barnes
(University of Kent)

IFIP WG 2.4, Glasgow (18th. July, 2006)
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 though Milner’s *π-calculus*. Here, we present a model of channel mobility based on Hoare’s *CSP* and say why …
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. 😊😊😊
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**).
The correct, flexible, scaleable and efficient management of concurrency has growing importance with the advent of multi-core processing elements (and for other reasons).

The mobile extensions to occam-π blend seamlessly into the classical occam language, maintaining safety guarantees (e.g. against parallel race hazards), the simple communicating process model (\textit{what-you-see-is-what-you-get} compositional semantics) and extreme lightness of concurrency overheads (\textbf{32 bytes} per process, \textbf{low tens of nanoseconds} for channel communication/synchronisation/context-switch – combined).
Talk roadmap ...

- Mobile channels in \textit{occam-}\pi\textsuperscript{-}\ (review) ...
- Modelling channels with processes ...
  - The \textbf{CSP} model (kernel processes) ...
  - The \textbf{CSP} model (application processes) ...
- End note ...
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

CHAN TYPE BUF.MGR

MOBILE RECORD

CHAN INT req?: -- requested resource size
CHAN MOBILE []BYTE buf!: -- delivered resource
CHAN MOBILE []BYTE ret?: -- returned resource

... and channel bundles, like atomic channels, have two ends which we call, arbitrarily, the “?” (or “server”) end and the “!” (or “client”) end ...
Mobile Channel Structures

CHAN TYPE BUF.MGR

MOBILE RECORD

CHAN INT req?: -- requested resource size
CHAN MOBILE []BYTE buf!: -- delivered resource
CHAN MOBILE []BYTE ret?: -- returned resource

... and we also specify the directions of the component channels ... from the point of view of the “?” end.
Mobile Channel Structures

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

BUF.MGR! client.end:  -- “client”-end variable
BUF.MGR? server.end:  -- “server”-end variable

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:

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

MOBILE [][]BYTE b:
INT s:
SEQ
  server.end[req] ? s
  b := MOBILE [s]BYTE
  server.end[buf] ! b
  server.end[ret] ? b
```
Mobile Channel Structures

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?)
Mobile Channel Structures

```
BUF.MGR! client.end:
BUF.MGR? server.end:
SEQ
    client.end, server.end := MOBILE BUF.MGR
```
Mobile Channel Structures

BUF.MGR!  client.end:
BUF.MGR?  server.end:
SEQ
  client.end, server.end := MOBILE BUF.MGR
cli.chan ! client.end
Mobile Channel Structures

BUF.MGR! client.end:
BUF.MGR? server.end:
SEQ
client.end, server.end := MOBILE BUF.MGR
cli.chan ! client.end
svr.chan ! server.end
-- client.end and server.end are now undefined
Mobile Channel Structures

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

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

PROC client (CHAN BUF.MGR! cli.chan?)
  BUF.MGR! client.end:
  SEQ
    cli.chan ? client.end
    real.client (client.end)
  :
Mobile Channel Structures

PROC server (CHAN BUF.MGR? svr.chan?)
BUF.MGR? server.end:
SEQ
Mobile Channel Structures

PROC server (CHAN BUF.MGR? svr.chan?)
BUF.MGR? server.end:
SEQ
    svr.chan ? server.end
Mobile Channel Structures

PROC server (CHAN BUF.MGR? svr.chan?)
  BUF.MGR? server.end:
  SEQ
    svr.chan ? server.end
    real.server (server.end)
  :

cli.chan (BUF.MGR!) -> generator -> svr.chan (BUF.MGR?)

real.client ! BUF.MGR ? real.server

8-Nov-11 Copyright P.H.Welch
Mobile Channel Structures

PROC real.client (BUF.MGR! client.end)
   ...
   :
PROC real.server (BUF.MGR? server.end)
   ...
   :
Mobile Channel Structures

```plaintext
PROC real.client (BUF.MGR! client.end)
  ...
  :

PROC real.server (BUF.MGR? server.end)
  ...
  :
```
**Shared Channel-Ends**

```
SHARED BUF.MGR! shared.client.end: -- "client"-end variable
BUF.MGR? server.end: -- "server"-end variable
SEQ
    shared.client.end, server.end := MOBILE BUF.MGR
PAR
    PAR i = 0 FOR n.clients
    client.2 (shared.client.end)
    server (server.end)
```

*n.clients* may be run-time computed
Shared Channel-Ends

PROC client.2 (SHARED BUF.MGR! shared.client.end)

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

Only shared.client.end channels may be used within this client CLAIM block (and no nested CLAIMs)

shared.client.end may not be used outside of a CLAIM block
Both Ends Shared

SHARED BUF.MGR! shared.client.end: -- "client"-end variable
SHARED BUF.MGR? shared.server.end: -- "server"-end variable

SEQ

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

PAR

PAR i = 0 FOR n.clients
  client.2 (shared.client.end)
PAR i = 0 FOR n.servers
  server.2 (shared.server.end)

n.clients/n.servers may be run-time computed
Both Ends Shared

PROC server.2 (SHARED BUF.MGR? shared.server.end)
...
  CLAIM shared.server.end
    MOBILE []BYTE b:
    INT s:
    SEQ
      shared.server.end[req] ? s
      b := MOBILE [s]BYTE
      shared.server.end[buf] ! b
      shared.server.end[ret] ? b
  ...

shared.server.end may not be used outside of a CLAIM block

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

**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 \textit{CARRY.BUF.MGR} (rather than \textit{BUF.MGR}).

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

CHAN TYPE CARRY.BUF.MGR
MOBILE RECORD
CHAN BUF.MGR? svr?:
:

A **client** process constructs both ends of an *unshared* BUF.MGR bundle and **claims** the shared CARRY.BUF.MGR 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 CARRY.BUF.MGR and inputs that **server-end**.
Both Ends Shared

CHAN TYPE CARRY.BUF.MGR
MOBILE RECORD
CHAN BUF.MGR? svr?:

Note that the client process, having output the server-end 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. 😊
Talk roadmap …

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

Modelling channels with processes …

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

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

End note …
Mobile Channels – a CSP Model and Implementation

We can’t directly model \texttt{occam-\pi} 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 $(\text{occam-}\pi)$

\[
\begin{array}{c}
\text{P} \quad \text{Q} \\
\text{c ! 42} \\
\text{c ? x}
\end{array}
\]
**Channel \rightarrow Process (occam-π)**

**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
Channel $\rightarrow$ Process (CSP)

Channel $\rightarrow$ Process (CSP)

- process $P$ only ever \textit{writes} on $c$;
- process $Q$ only ever \textit{reads} on $c$;
- no output guards (e.g. $(c!42 \rightarrow P2)$ is never part of an external choice, []).

![Diagram showing the relationship between processes $P$ and $Q$ through channel $c$.]

- $c!42 \rightarrow P2$
- $c?x \rightarrow Q2(x)$

\( (P \mid\{c\} \cup X \mid Q) \setminus \{c\} \)
Channel → Process (CSP)

\[(P' \mid X \mid Q') \mid \{|writeC, readC, ackC|\} \mid ChanC\]
Given the *occam*-π system constraints (*uni-directional 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.
Talk roadmap …

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

Modelling channels with processes …

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

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

End note …
Mobile Channel Structures (CSP)

CHAN TYPE BUF.MGR

MOBILE RECORD
CHAN INT req?: -- requested resource size
CHAN MOBILE []BYTE buf!: -- delivered resource
CHAN MOBILE []BYTE ret?: -- returned resource

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

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

Index number

#chans in bundle
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

Bundle (c, fields)

Refs (c, 2)

Semaphore (c, server)

Semaphore (c, client)

Channels (c, fields)

where server = 0     client = 1
\[\text{Refs (c, n)} = \]
\[
\text{enroll.c} \rightarrow \text{Refs (c, n + 1)} \quad [
\]
\[
\text{resign.c} \rightarrow \text{Refs (c, n - 1)}
\]

\[\text{Refs (c, 0)} = \text{kill} \rightarrow \text{SKIP}\]
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

Bundle (c, fields)

Refs (c, 2)

Semaphore (c, server)

Semaphore (c, client)

Channels (c, fields)

where

server = 0       client = 1
Semaphore (c, x) =
  claim.c.x -> release.c.x -> Semaphore (c, x) []
  kill -> SKIP

where
  x : {server, client}
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
Channels (c, fields)

write.c.0.p
ack.c.0
read.c.0.p

write.c.1.p
ack.c.1
read.c.1.p

write.c.(f-1).p
ack.c.(f-1)
read.c.(f-1).p
Channels (c, fields)

write.c.0.p
ack.c.0
read.c.0.p
write.c.1.p
ack.c.1
read.c.1.p
write.c.(f-1).p
ack.c.(f-1)
read.c.(f-1).p

Chan (c, 0)
Chan (c, 1)
Chan (c, fields – 1)
Channels (c, fields) =
    |\{kill\}| \{Chan (c, i) | i = 0..(fields - 1)\}
Chan (c, i) =
    write.c.i?p -> read.c.i!p -> ack.c.i -> Chan (c, i)
    []
    kill -> SKIP
Mobile **channel bundle** processes are generated as needed by the following server:

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:

```
undefined = 0

UndefinedBundle =
    resign.undefined -> UndefinedBundle []
    noMoreBundles -> SKIP
```
Kernel Processes

The *mobile bundle* kernel is:

\[
\text{MOBILE\_BUNDLE\_KERNEL} = \\
\text{MC (1) | \{noMoreBundles\} | UndefinedBundle}
\]

Let’s define:

\[
\text{kernel\_chans} = \\
\{ | enrol, resign, claim, release, write, read, ack, \\
setMC, getMC, noMoreBundles | \}
\]
Then, if \texttt{APPLICATION\_SYSTEM} is the \texttt{occam-\pi} application and \texttt{APPLICATION\_SYSTEM'} is the \texttt{CSP} modelling of its mobile channel bundle primitives \textit{(in a minute)}, the full model is:

\[
\text{(APPLICATION\_SYSTEM'; noMoreBundles \rightarrow SKIP) } \\
\mid \text{kernel\_chans} \mid \\
\text{MOBILE\_BUNDLE\_KERNEL)} \setminus \text{kernel\_chans}
\]

Here’s a diagram …
Application and Kernel Processes

APPLICATION_SYSTEM'

MOBILE_CHANNEL KERNEL

... ...

Bundle (3, f3)
Bundle (2, f2)
Bundle (1, f1)
UndefinedBundle

resign.0
noMoreBundles
setMC.f
getMC.c
MC (4)
Talk roadmap ...

Mobile channels in \textit{occam-\pi} (review) ...

Modelling channels with processes ...

The \textbf{CSP} model (kernel processes) ...

The \textbf{CSP} model (application processes) ...

End note ...
Modelling $\text{occam-\pi}$ Mobile Channels

CHAN TYPE BUF.MGR

MOBILE RECORD

CHAN INT req?: \hspace{1cm} -- requested resource size
CHAN MOBILE [ ]BYTE buf!: \hspace{1cm} -- delivered resource
CHAN MOBILE [ ]BYTE ret?: \hspace{1cm} -- returned resource

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

We express this using Circus. Amongst other things, it adds variables and assignment into CSP, which we find convenient. The paper¹ describes how these map down to pure CSP.

where \( P'(x) \) is the CSP model of \( P(x) \).

¹ "A CSP Model for Mobile Channels", CPA 2008
Modelling \textit{occam-}\pi\ Mobile Channels

\begin{itemize}
  \item \texttt{BUF.MGR! cli:}
  \item \texttt{BUF.MGR? svr:}
  \item \texttt{assume}
  \item \texttt{BUF.MGR! cli:}
  \item \texttt{BUF.MGR? svr:}
\end{itemize}

Dynamic construction of \textit{mobile bundles}:

\texttt{cli, svr := MOBILE BUF.MGR}

application process sends the number of fields for the bundle …

... and is returned its index number

\texttt{setMC!nf(BUF.MGR) \rightarrow getMC?tmp \rightarrow ((cli := tmp) || (svr := tmp))}
Claiming a **SHARED mobile bundle end**:

\[
\text{CLAIM} \quad \text{cli} \quad P \quad \sim \quad \text{claim.cli.client} \rightarrow P' \;
\text{release.cli.client} \rightarrow \text{SKIP} \\
\text{CLAIM} \quad \text{svr} \quad P \quad \sim \quad \text{claim.svr.server} \rightarrow P' \;
\text{release.svr.server} \rightarrow \text{SKIP}
\]
Modelling occam-$\pi$ Mobile Channels

Input and output on a channel field of a *mobile bundle*:

- **cli[req]** ! n \implies \text{write.cli.0!n} \rightarrow \text{ack.cli.0} \rightarrow \text{SKIP}

- **cli[buf]** ? b \implies \text{read.cli.1?tmp} \rightarrow (b := \text{tmp})

**CHAN TYPE** BUF.MGR

**MOBILE RECORD**

- **CHAN INT** req?: -- requested resource size
- **CHAN MOBILE []BYTE** buf!: -- delivered resource
- **CHAN MOBILE []BYTE** ret?: -- returned resource
Modelling occam-$\pi$ Mobile Channels

Input and output on a channel field of a mobile bundle:

\[
\text{svr[req] ? n} \quad \sim \quad \text{read.svr.0?tmp} \rightarrow (n := \text{tmp})
\]

\[
\text{svr[buf] ! b} \quad \sim \quad \text{write.svr.1!b} \rightarrow \text{ack.svr.1} \rightarrow \text{SKIP}
\]

\begin{verbatim}
CHAN TYPE BUF.MGR
  MOBILE RECORD
    CHAN INT req?:: -- requested resource size
    CHAN MOBILE []BYTE buf!: -- delivered resource
    CHAN MOBILE []BYTE ret?: -- returned resource

: 
\end{verbatim}
Modelling occam\(\pi\) Mobile Channels

Sending an **unshared mobile bundle end**:

\[
\begin{align*}
\text{m!cli} & \rightarrow (\text{cli} := \text{undefined}) \\
\text{n!svr} & \rightarrow (\text{svr} := \text{undefined})
\end{align*}
\]
Sending a **shared mobile bundle end**:

```
CHAN SHARED BUF.MGR! m

m ! cli
```

```
enroll.cli -> m!cli -> SKIP
```

```
CHAN SHARED BUF.MGR? n

n ! svr
```

```
enroll.svr -> n!svr -> SKIP
```
Modelling \textit{occam-\pi} Mobile Channels

Receiving a \textit{shared} / \textit{unshared} mobile bundle end:

\begin{align*}
\text{CHAN BUF.MGR! } m & \quad \text{or} \quad \text{CHAN SHARED BUF.MGR! } m \\
\text{CHAN BUF.MGR? } n & \quad \text{or} \quad \text{CHAN SHARED BUF.MGR? } n
\end{align*}

\begin{align*}
m ? \text{cli} & \sim \quad m?tmp \rightarrow \text{resign.cli} \rightarrow (\text{cli} := \text{tmp}) \\
n ? \text{svr} & \sim \quad n?tmp \rightarrow \text{resign.svr} \rightarrow (\text{svr} := \text{tmp})
\end{align*}
Assignment of shared / unshared mobile bundle ends:

\[ a := b \]

- \( \text{resign}.a \rightarrow (a := b); \ (b := \text{undefined}) \)

or

\[ \text{BUF.MGR!} \ a, b: \]

\[ \text{SHARED} \ \text{BUF.MGR!} \ a, b: \]

\[ (\text{enroll}.b \rightarrow \text{SKIP} ||| \text{resign}.a \rightarrow \text{SKIP}); \ (a := b) \]

or

\[ \text{BUF.MGR?} \ a, b: \]

\[ \text{SHARED} \ \text{BUF.MGR?} \ a, b: \]
occam-π generates parallel processes either with a `FORK` or a `PAR`. Arguments are always communicated to a `FORK`:

- `FORK P (c)`
- `forkP!c -> (c := undefined)`

or

- `BUF.MGR! c:`
- `BUF.MGR? c:`

- `FORK P (c)`
- `enroll.c -> forkP!c -> SKIP`

or

- `SHARED BUF.MGR! c:`
- `SHARED BUF.MGR? c:`
Forking processes must take place within a `FORKING` block (by default, the whole system):

\[
((X'; \text{done} \rightarrow \text{SKIP}) \\
\{\{\text{forkP, done}\}\}) \\
\text{FORK_P}
\]

\[
) \setminus \{\{\text{forkP, done}\}\}
\]

where:

\[
\text{FORK_P} = \text{forkP?c} \rightarrow ((P'(c); \text{resign.c} \rightarrow \text{done} \rightarrow \text{SKIP}) \\
\{\text{done}\} | \text{FORK_P}) []
\]

\[
\text{done} \rightarrow \text{SKIP}
\]

The `done` event ensures that the forking block, \(X\), cannot terminate until all processes forked in the block have also terminated.
Application and Kernel Processes

APPLICATION_SYSTEM

MOBILE_CHANNEL_KERNEL

*.1
Bundle (1, f1)

*.2
Bundle (2, f2)

*.3
Bundle (3, f3)

resign.0

noMoreBundles

setMC.f

getMC.c

UNDEFINED_CHAN

MC (4)
Talk roadmap ...

Mobile channels in occam-$\pi$ (review) ...
Modelling channels with processes ...
The CSP model (kernel processes) ...
The CSP model (application processes) ...
End note ...
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 systems. Model checking requires a (small) modification to constrain numbers to finite limits for particular systems!

The occam-\(\pi\) 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\(^2\) will be a strong testing ground for the mixing of the dynamic mechanisms of the \(\pi\)-calculus into CSP.

\(^2\) “A CSP Model for Mobile Channels”, CPA 2008