Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

A CSP Model for Mobile Channels

Peter H. WELCH and Frederick R.M. BARNES
Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, England

Abstract. CSP processes have a static view of their environment – a *fixed* set of events through which they synchronise with each other. In contrast, the π-calculus is based on the dynamic construction of events (channels) and their distribution over pre-existing channels. In this way, process networks can be constructed dynamically with processes acquiring new connectivity. For the construction of complex systems, such as Internet trading and the modeling of living organisms, such capabilities have an obvious attraction. The occam-π multiprocessing language is built upon classical occam, whose design and semantics are founded on CSP. To address the dynamics of complex systems, occam-π extensions enable the movement of channels (and multiway synchronisation barriers) through channels, with constraints in line with previous occam discipline for safe and efficient programming. This paper reconciles these extensions by building a formal (operational) semantics for mobile channels entirely within CSP. These semantics provide two benefits: formal analysis of occam-π systems using mobile channels and formal specification of implementation mechanisms for mobiles used by the occam-π compiler and run-time kernel.

Keywords. channels, processes, mobility, modeling, occam-π, CSP, π-calculus.

Introduction

The dynamic creation of channels and processes, together with their communication through channels, enables network topology to evolve in response to run-time events. Systems requiring this capability abound – for example, the modelling of complex biological phenomena and commercial Internet applications. Formal specification and analysis has been pioneered through Milner's π-calculus. Here, we present a model of channel mobility using Hoare's CSP and explain our motivation and the benefits obtained.

Mobile channels have been introduced into the occam-π multiprocessing language [1,2,3], whose classical design and semantics are founded on CSP [4,5,6]. CSP processes synchronise on fixed sets of events (so cannot dynamically acquire new connections) and that static nature cannot easily be relaxed. However, CSP allows infinite event sets and recursion, which gives us a lot of freedom when modeling.

This paper presents a CSP model for channel mobility that yields semantics that are both *operational* and *denotational*. The operational aspect provides a formal specification for all data structures and algorithms for a supporting run-time kernel (from which the occam-π kernel is derived). The denotational side preserves the *compositional* nature of occam-π components (no surprises when processes are networked in parallel, *what-you-see-is-what-you-get*). It also allows formal specification and analysis of occam-π systems and, so long as the number of mobile channels can be bounded and that bound is not too large, the application of automated model checkers (such as FDR [7]).

Section 1 reviews the mobile channel mechanisms of occam-π. Section 2 introduces the technique of modeling channels with processes, essential for the formal semantics of mobility presented here. Section 3 builds the kernel processes underlying the semantics. Section 4 maps occam-π code to the relevant synchronisations with the kernel. Finally, Section 5 draws conclusions and directions for future work.

1. Mobile Channels in occam-π

Mobile *channels*, along with mobile *data* and mobile *processes*, have been introduced into the occam-π multiprocessing language, a careful blend of classical (CSP-based) occam2.1 with the network dynamics of the π-calculus [8]. The *mobility* concept supported reflects the idea of movement: something *moves* from source to target, with the source losing it (unless explicitly marked for *sharing*). Mobile objects may also be *cloned* for distribution.

occam-π introduces channel *bundles*: a record structure of individual channels (fields) carrying different protocols (message structures) and operating in different directions. occam-π also separates the concept of channel (or channel bundle) *ends* from the channels (or bundles) themselves: processes see only one *end* of the external channels with which they interact with their environment. For mobile channels, it is the *channel-ends* that can be moved (by communication or assignment) – not the channels themselves. With the current compiler, channel mobility is implemented only for the new channel bundle types.

1.1 Mobile Channel Bundles

Channel types declare a *bundle* of channels that will always be kept together. They are similar to the idea proposed for occam3 [9], except that the ends of our bundles are (currently always declared to be) mobile, directions are specified for the individual channels, and the bundle has distinct *ends*.

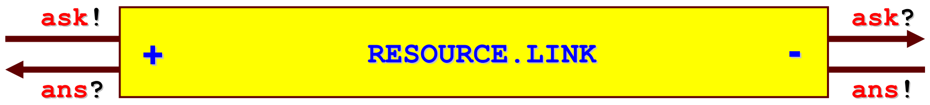

Figure 1: a channel bundle.

Figure 1 shows a typical bundle of channels supporting *client-server* communications. By convention, a *server* process takes the *negative* end of the bundle, receiving and answering questions from *client* processes sharing the *positive* end. The type is declared as follows:

```
CHAN TYPE RESOURCE.LINK
  MOBILE RECORD
    CHAN RESOURCE.ASK ask!:
    CHAN RESOURCE.ANS ans?:
:
```

Note that this declaration specifies field channel directions from the point of view of the positive end of the bundle. So, clients operate these channels in those declared directions (they *ask* questions and *receive* answers), whereas a server operates them the other way around (it *receives* questions and *delivers* answers).

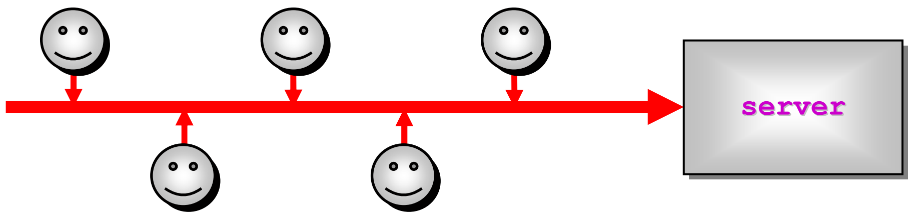

Figure 2: a client-server network.

1.2 Declaring, Allocating and Placing Ends of Mobile Channel Bundles

Variables are declared only to hold the ends of channel bundles – not the bundle as a whole. These ends are independently mobile. Bundle ends are allocated dynamically and in pairs. By default, a bundle end is *unshared*: it may only be connected to one process at a time. A bundle end may be declared as being *shared*: it may be connected to any number of parallel processes (which compete with each other to use). Resources for the bundles are automatically recovered when all references to them are lost.

Here is code that sets up an initial system (Figure 2) of many clients and one server:

```
RESOURCE.LINK- resource.server.end:
SHARED RESOURCE.LINK+ resource.client.end:
SEQ
  resource.server.end, resource.client.end := MOBILE RESOURCE.LINK
  PAR
    resource.server (resource.server.end, ...)
    PAR i = 0 FOR n.clients
      client (resource.client.end, ...)
:
```

where the server and client processes may have other connections (not shown in Figure 2). Note the polarity of the channel bundle types in the above declarations, indicating which end of the bundle is held by each variable.

1.3 Using and Moving Ends of Channel Bundles

1.3.1 Clients Holding Shared Positive Ends

This client process is aware that its link to the resource server is shared:

```
PROC client (SHARED RESOURCE.LINK+ resource.link,
             CHAN SHARED RESOURCE.LINK+ forward!, update?,
             ...)
  ...
:
```

In the above, `resource.link` is the (shared client) bundle end and the (classical) channels `forward!` and `update?` are for sending and receiving, respectively, new bundle ends.

To demonstrate use of this channel bundle, let us make the protocols used by its sub-channels more concrete:

```
PROTOCOL RESOURCE.ASK
  CASE
    size; INT
    deposit; RESOURCE
:

PROTOCOL RESOURCE.ANS IS RESOURCE:
```

where `RESOURCE` is some (expensive to allocate) mobile data structure. A client asks for a `RESOURCE` of a certain size (on its `ask!` sub-channel end) and duly receives one (on `ans?`):

```
CLAIM resource.link
  SEQ
    resource.link[ask] ! size; 42
    resource.link[ans] ? my.resource    -- RESOURCE
```

When the client has finished with the resource, it returns it back to the server:

```
CLAIM resource.link
  resource.link[ask] ! deposit; my.resource
```

Outside a **CLAIM**, a client may forward its end of the link to its server to another process:

```
forward ! resource.link
```

This dynamically introduces another client to the server. Because the bundle end is *shared*, the original client retains its link to the server.

When not in mid-transaction with its server (again, outside a **CLAIM** block), this client may choose to update its link:

```
update ? resource.link
```

It loses its original connection and is now the client of a (presumably) different server.

1.3.2 A Server Holding a Negative End

This server *pools* **RESOURCE** instances, retrieving suitable ones from its pool where possible when new ones are requested:

```
PROC resource.server (RESOURCE.LINK- resource.link,
                      CHAN RESOURCE.LINK- lend!, return?,
                      ...)
  ...  declare dynamic structures for managing the RESOURCE pool
  SEQ
    ...  initialise RESOURCE pool
    INITIAL BOOL running IS TRUE:
    WHILE running
      resource.link[ask] ? CASE
        INT n:
        size; n
          RESOURCE resource:
          SEQ
            IF
              ...  a suitably sized RESOURCE is in the pool
                ...  move it into the resource variable
              TRUE
                ...  dynamically allocate a new resource (of size 'n')
            resource.link[ans] ! resource
        RESOURCE resource:
        deposit; resource
          ...  deposit resource into the pool
:
```

At any time, this server may relinquish servicing its clients by forwarding its (exclusive) end of the link to another server:

```
lend ! resource.link
```

Because this link is *unshared*, the **resource.link** variable becomes *undefined* and this server can no longer service it – any attempt to do so will be trapped by the compiler. This server may do this if, for some reason, it cannot satisfy a client's request but the **forward** channel connects to a reserve server that can. To continue providing service, the forwarding had better be a loan – i.e. the other server returns it after satisfying the difficult request:

```
  return ? resource.link
```

Our server may now resume service to all its clients. The above server coding may need slight adjustment (e.g. if the reserve server has already supplied the resource on its behalf).

2. Modeling Channels with Processes

To provide a formal semantics of channel mobility in occam-π, we cannot model its channels with CSP *channels*. There are three challenges: the dynamics of construction, the dynamics of mobility (which demands varying synchronisation alphabets) and the concept of channel *ends* (which enforces correct sequences of direction – also unprotected within the π-calculus).

Instead, following the techniques developed for the formal modeling of mobile barriers (multiway synchronisations) in occam-π [10], we model mobile channels as *processes*. Each mobile channel process is constructed on demand and given a unique *index* number. Conventional channels, through which such a process is driven, model the two different *ends* of the mobile channel. Application processes *interleave* in their use of these *ends*, with that interleaving governed by possession of the relevant index. *Mobility* derives simply from communicating (and, then, forgetting) that index.

Let **P** be a process and **c** be an external (i.e. non-hidden) channel that **P** uses only for output. Further, assume **P** never uses **c** as the first action in a branch of an external choice (a constraint satisfied by all occam-π processes). Using the notion of *parallel introduction* (section 3.1.2 of [11]), all communications on channel **c** may be devolved to a *buddy* process, **ChanC**, with no change in semantics – i.e. that **P** is failures-divergences equivalent to the system (expressed in CSP$_M$, the machine-readable CSP syntax defined for FDR [7]):

```
  ( (P′; killC -> SKIP) [| {| writeC, ackC, killC |} |] ChanC )
  \ {| writeC, ackC, killC |}
```

where **writeC**, **ackC**, and **killC** are events chosen outside the alphabet of **P**, and where:

```
  ChanC = (writeC?x -> c!x -> ackC -> ChanC [] killC -> SKIP)
```

and where **P′** is obtained from **P** by delegating all communications (**c!a -> Q**) to its buddy process (**writeC!a -> ackC -> Q**).

Process **P′** completes a **writeC/ackC** sequence if, and only if, the communication **c!x** happens – and the **writeC/ackC** events are hidden (i.e. undetectable by an observer). Process **P** does not engage on **c** as the first event of an external choice; so neither does **P′** on **writeC**. This is a necessary constraint since, otherwise, the hiding of **writeC** would introduce non-determinism not previously present. Termination of the *buddy* process, **ChanC**, is handled with the addition of the (hidden) **killC** event – used only once, when/if the original process terminates.

Formally, the equivalence follows from the rich algebraic laws of CSP relating *event hiding*, *choice*, *sequential* and *parallel composition*, *prefix* and *SKIP* (outlined in section 4 of [11]).

Figure 3 applies this equivalence to transform a *channel*, **c**, connecting processes **P** and **Q** into a *process*, **ChanC′**. Process **P′** is defined above. For convenience, processes **Q′** and **ChanC′** are defined from **Q** and **ChanC** just by *renaming* their external (free) channel, **c**, as **readC** (where **readC** is an event chosen outside the alphabets of **P** and **Q**) We now have distinct terms to talk separately about the *writing-end* (**writeC**/**ackC**) and *reading-end* (**readC**) of our original channel **c**.

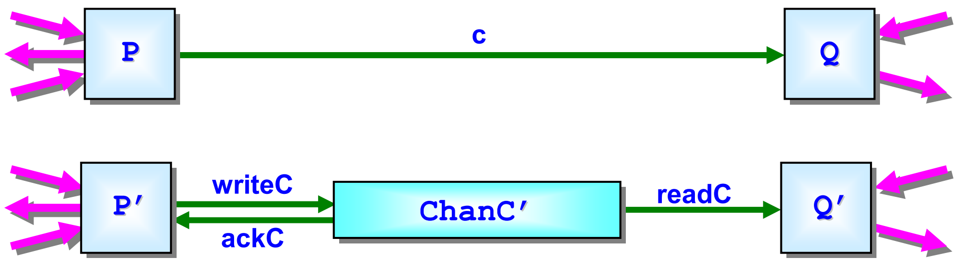

Figure 3: modeling a channel with a process.

Figure 3 applies this equivalence to transform a *channel*, **c**, connecting processes **P** and **Q** into a *process*, **ChanC′**, connecting processes **P′** and **Q′**. The process **P′** is defined above. For convenience, processes **Q′** and **ChanC′** are defined from **Q** and **ChanC** just by *renaming* their external channel, **c**, as **readC** (where **readC** is an event chosen outside the alphabets of **P** and **Q**). This gives us distinct terms with which we can talk separately about the *writing-end* (**writeC**/**ackC**) and *reading-end* (**readC**) of our original channel **c**.

3. A Kernel for Mobile Channels

We present semantics for the mobile channels of occam-π, addressing channel bundles, dynamic allocation, the separate identities of channel bundle ends, the use of the channels within the bundles, sharing and mobility.

3.1 Processes within a Channel Bundle

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

```
Bundle (c, fields) =
  (Refs (c, 2) [| {kill} |]
     (Mutex (c, positive) [| {kill} |]
        (Mutex (c, negative) [| {kill} |]
           Channels (c, fields)
        )
     )
  ) \ {kill}
```

where **c** is the unique index for the bundle, **fields** is the number of its channel fields and **positive** and **negative** are constants with values **0** and **1**, respectively. This is visualised in Figure 4, showing the channels independently engaged by the sub-processes.

The reference counting process is initialised to **2**, since one variable for each bundle *end* knows about it upon construction. This process engages in **enrol** and **resign** events for this bundle and is responsible for terminating its sibling processes should the count ever reach zero:

```
Refs (c, n) =
  enrol.c -> Refs (c, n + 1) []
  resign.c -> Refs (c, n - 1)                , if n > 0

Refs (c, 0) = kill -> SKIP
```

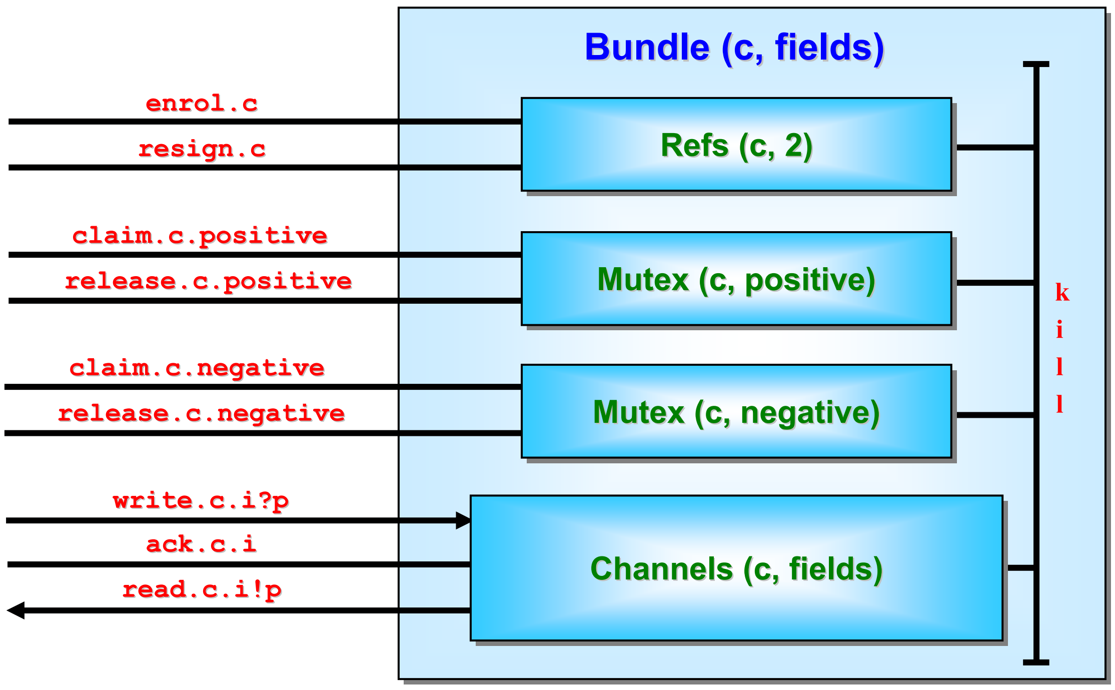

Figure 4: formal model of an occam-π channel bundle.

The *mutex* processes provide mutually exclusive locking for each end (**positive** or **negative**) of the bundle, engaging in **claim** and **release** events for this bundle. They can be terminated at any time by a **kill** signal (from the reference counting process). They enforce access to a bundle end by only one application process at a time. Strictly, they are only necessary for each end that is declared *shared*; their over-engineering here in this model simplifies it and is harmless. The coding is elementary and standard:

```
Mutex (c, x) =
  claim.c.x -> release.c.x -> Mutex (c, x)  []
  kill -> SKIP
```

where **x : {positive, negative}**.

The **channels** process (Figure 5) is a parallel collection of channel processes, one for each field, synchronising only on the termination signal:

```
Channels (c, fields) =
  [| {kill} |] i:{0..(fields - 1)} @ Chan (c, i)
```

where each channel process follows the pattern presented in Section 2:

```
Chan (c, i) =
  write.c.i?p -> read.c.i!p -> ack.c.i -> Chan (c, i) []
  kill -> SKIP
```

3.2 Dynamic Generation of Channel Bundles

Mobile channel bundle processes are generated upon request by the following server:

```
MC (c) =
  setMC?f -> getMC!c -> (Bundle (c, f) ||| MC (c + 1)) []
  noMoreBundles -> SKIP
```

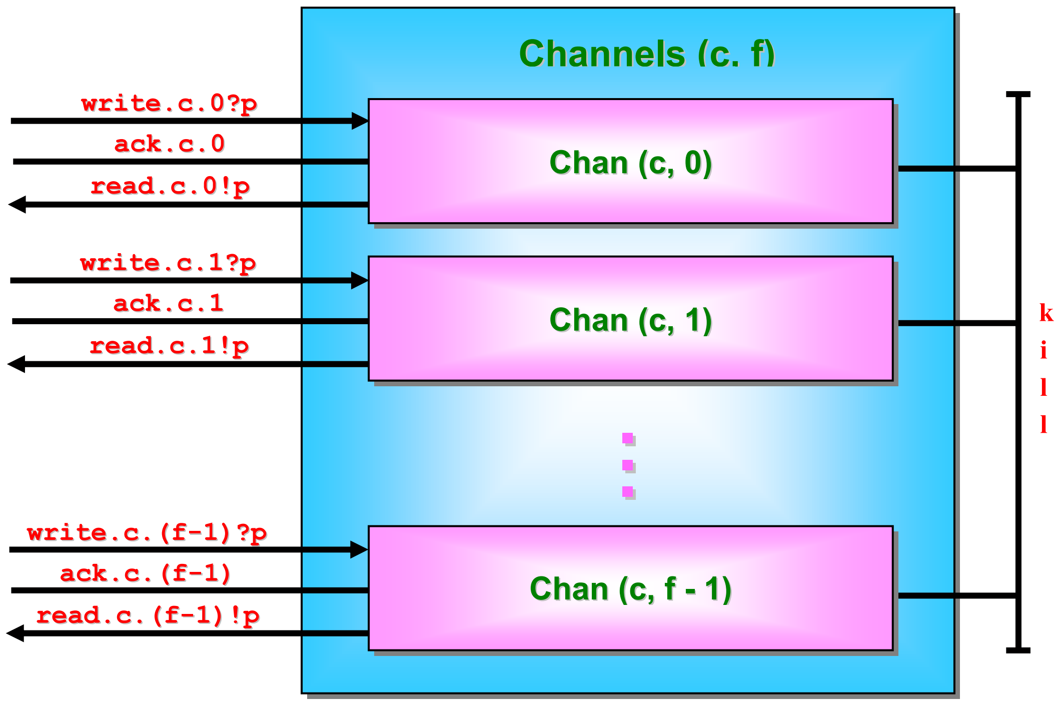

Figure 5: processes modelling the channels in a bundle.

Application processes will interleave on the **setMC** and **getMC** channels when constructing mobile channel bundles: the number of fields for the bundle is requested on **setMC** and a unique index number for the generated bundle is returned on **getMC**.

This channel bundle generator will be started with index **1**. We reserve index **0** for an *undefined* bundle:

```
undefined = 0

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

Note that both the channel bundle generator, **MC**, and **UndefinedBundle** terminate on the **noMoreBundles** signal. That signal is generated when, and only when, the application is about to terminate. The **resign** signal, accepted but ignored by **UndefinedBundle**, is there to simplify some technical details in Section 4.

3.3 Mobile Channel Kernel

The *mobile channel kernel* consists of the generator and undefined bundle processes:

```
MOBILE_CHANNEL_KERNEL = MC (1) [| {noMoreBundles} |] UndefinedBundle
```

In addition to **noMoreBundles**, it engages (but does not syncrhonise internally) upon the following set of channels:

```
kernel_chans =
  {| enrol, resign, claim, release, write, read, ack,
     setMC, getMC, noMoreBundles |}
```

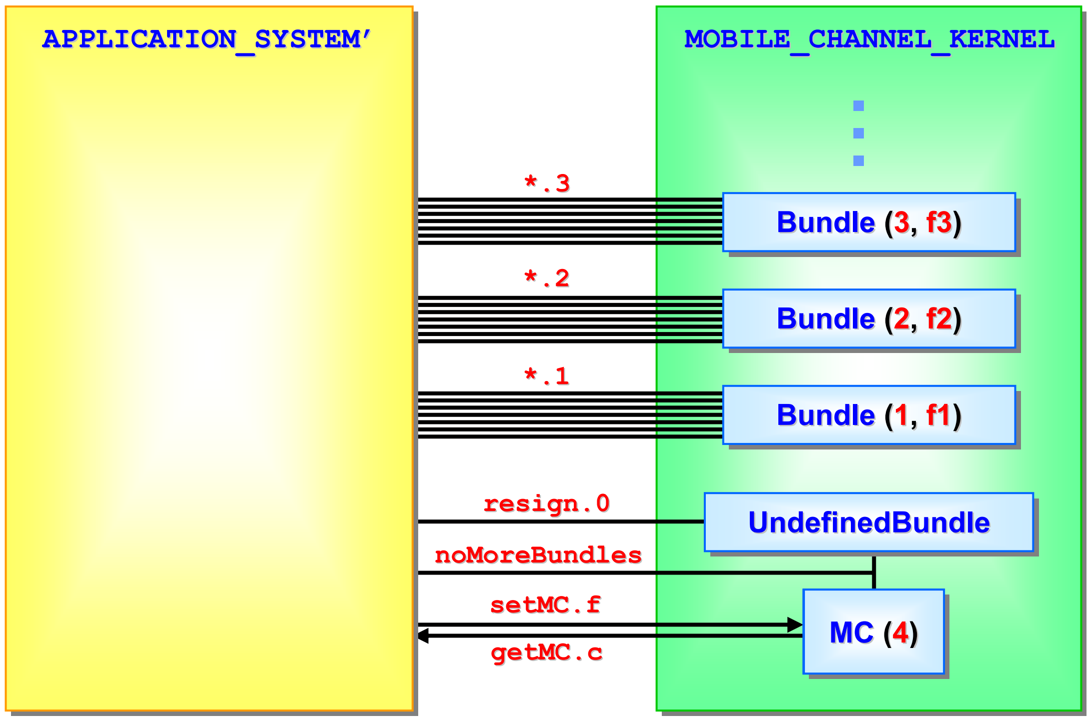

Figure 6: application system and support kernel
(after the construction of three mobile channel bundles).

If **APPLICATION_SYSTEM** is an occam-π application and **APPLICATION_SYSTEM′** is the CSP model of its use of mobile channel bundle primitives (Section 4), the full model is:

```
((APPLICATION_SYSTEM′; noMoreBundles -> SKIP)
 [| kernel_chans |]
 MOBILE_CHANNEL_KERNEL) \ kernel_chans
```

Figure 6 gives a visualisation after three mobile channels have been constructed.

4. The Semantics of occam-π Mobile Channels

This section defines the mapping from all occam-π mobile channel mechanisms to CSP. The mapping is simplified using some syntactic extensions from Circus [12], a CSP algebra combined with elements of Z for the formal specification of rich state transformations. The extensions used here are restricted to the declaration of *state variables*, their assignment as new primitive CSP processes and their use in expressions. These extensions have a trivial mapping down to pure CSP (which is described in section 2.3 of [10]).

The mappings will be presented by induction and example, using the definitions from Section 1 for concreteness. In the following, if **P** (or **P(x)**) is an occam-π process, then **P′** (or **P′(x)**) denotes its mapping into CSP.

4.1 Declaring Mobile Channel Bundle End Variables

All mobile channel bundle end variables – regardless of polarity, shared status, number of fields and channel protocols – are represented by variables holding a natural number index, unique for each *defined* bundle. Initially, they are all set to **undefined** (see Section 3.2).

Each of the following declarations:

```
RESOURCE.LINK+ x:                SHARED RESOURCE.LINK+ x:
P (x)                            P (x)

RESOURCE.LINK- x:                SHARED RESOURCE.LINK- x:
P (x)                            P (x)
```

$\rightsquigarrow$ (maps to)

```
Var x:N • x := undefined; P′(x); resign.x -> SKIP
```

Note that the mapped process resigns from the mobile bundle just before termination. We do not have to check whether the bundle variable is defined at this point, because of the definition and inclusion of the `UndefinedBundle` process in the kernel (Section 3.2).

4.2 Dynamic Construction of Mobile Bundles

Suppose `client` and `server` are opposite polarity `RESOURCE.LINK` variables, shared or unshared. To assign them to a freshly constructed mobile bundle, send the kernel the number of fields required (`#RESOURCE.LINK`), receive the index of channels to the bundle process created and assign this to the variables (not forgetting to resign from any bundles they may previously have been referencing):

```
client, server := MOBILE RESOURCE.LINK
```

$\rightsquigarrow$

```
setMC!#RESOURCE.LINK -> getMC?tmp ->
  ( resign.client -> (client := tmp) |||
    resign.server -> (server := tmp) )
```

Note that these processes interleave their use of the kernel channels. In the unlikely event that both variables were previously referencing opposite ends of the *same* bundle, the interleaved resignations are still handled correctly. As mentioned in Section 4.1, resigning from variables holding undefined references is also safe. We use *interleaving* in the above specification to give maximum freedom to the occam-π compiler/kernel in implementing these mechanisms.

4.3 Claiming a Shared Mobile Channel Bundle End

Suppose that `client` and `server` are `SHARED` bundle-ends. This time, we need to know their polarity. Suppose that `client` is positive and `server` is negative (which is the normal convention for a *client-server* relationship). An occam-π `CLAIM` obtains exclusive use of the bundle-end for the duration of the `CLAIM` block. So:

```
CLAIM client
  P
```

$\rightsquigarrow$

```
claim.client.positive -> P′; release.client.positive -> SKIP
```

and:

```
CLAIM server
  P
```

$\leadsto$

```
claim.server.negative -> P′; release server.negative -> SKIP
```

where these **claim** and **release** events synchronise with the relevant *mutex* processes in the relevant *channel bundle* processes in the kernel.

Of course, the occam-π language does not allow **CLAIM**s on *undefined* channel bundle variables and this rule is enforced statically by the compiler. So, we do not need to be concerned with such possibilities for these semantics (and this is why **claim** and **release** events do not have to play any part in the **UndefinedBundle** process of our CSP kernel).

4.4 Input/Output on a Channel in a Mobile Channel Bundle

Communications over channel fields in a bundle are modeled following the patterns given in Section 2.

Suppose **client** is a **RESOURCE.LINK+** variable and **server** is **RESOURCE.LINK-**; then communications on **client** follow the directions defined in the declaration:

```
CHAN TYPE RESOURCE.LINK
  MOBILE RECORD
    CHAN RESOURCE.ASK ask?:
    CHAN RESOURCE.ANS ans!:
:
```

but communications on **server** follow the opposite directions.

4.4.1 Positive Communications

We present the mapping for these first, since they are not complicated by the variant protocol defined for **RESOURCE.ASK**. For sending:

```
client[ask] ! size; n
```

$\leadsto$

```
write.client.ask!size.n -> ack.client.ask -> SKIP
```

where, in the mapped CSP, **ask** is just the field number (**0**) in the bundle. The values of **size** and **n** are, respectively, a constant specifying the message type being delivered and the resource size requested. In these semantics (of mobile channel communications), message contents are not relevant – they are only significant for the use made of them by application processes before and after communication. Similarly:

```
client[ask] ! deposit; r
```

$\leadsto$

```
write.client.ask!deposit.r -> ack.client.ask -> SKIP
```

On the other hand, for receiving:

```
client[ans] ? r
```

$\rightsquigarrow$

```
read.client.ans?tmp -> (r := tmp)
```

4.4.2 Negative Communications

The server side communications on **RESOURCE.LINK** are complicated slightly by the *variant* message structure in **RESOURCE.ASK**: the **CASE** input must be dealt with by testing the first (tag) part of the message. This is not significant for these semantics of mobile channel input, which are only concerned with synchronisations between application and kernel processes.

The mappings for sending and receiving do not depend on the polarity of the bundle-ends. Sending is the same as in Section 4.4.1, except that the field names are switched:

```
server[ans] ! r
```

$\rightsquigarrow$

```
write.server.ans!r -> ack.server.ans -> SKIP
```

Receiving is also as in Section 4.4.1, with switched field names. The complications of the variant protocol, used in this example, are not much of a distraction:

```
server[ask] ? CASE
  INT n:
  size; n
    P (n)
  RESOURCE r:
  deposit; r
    Q (r)
```

$\rightsquigarrow$

```
read.server.ask?tag.tmp ->
  if tag == size then P′(tmp) else Q′(tmp)
```

4.5 Sending Mobile Channel Bundle Ends

Sending a mobile channel bundle-end depends on whether it is shared. Assume that **client** and **server** are *unshared* and that we have suitable channels **m** and **n** carrying, respectively, **RESOURCE.LINK+** and **RESOURCE.LINK-**. Then:

```
m ! client
```

$\rightsquigarrow$

```
m!client -> (client := undefined)
```

which introduces the occam-π mobility semantics: *send-and-lose*. The mapping for sending **`server`** down channel **`n`** is identical, apart from name changes. These mappings are, of course, for classical (non-mobile) channels **`m`** and **`n`**. If these were themselves fields of a mobile channel bundle, the mappings from Section 4.4 would also be applied.

Assume, now, that **`client`** and **`server`** are *shared* and that we have suitable channels **`m`** and **`n`** carrying, respectively, **`SHARED RESOURCE.LINK+`** and **`SHARED RESOURCE.LINK-`**. The communication semantics are now different: the sending process *does not lose* a mobile, if it is shared, and the reference count on the mobile must be incremented (since the receiving process may now engage on it):

```
m ! client
```

$$\leadsto$$

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

Note: it is necessary that the *sender* of the mobile increments the reference count as part of the send, rather than the *receiver*. The receiver could only try to increment that count after receiving it – by which time the sender could have resigned from the bundle itself (perhaps through termination, see Section 4.1, or overwriting of its bundle variable, see below), which might reduce the reference count to zero, terminating the bundle process in the kernel and leaving the receiver deadlocked!

As before, the mapping for sending a *shared* **`server`** down channel **`n`** is identical to the above, apart from name changes. Again, these mappings are for classical (non-mobile) channels **`m`** and **`n`**. If these were themselves fields of a mobile channel bundle, the mappings from Section 4.4 would also be applied.

4.6 Receiving Mobile Channel Bundle Ends

Receiving a mobile channel bundle-end does *not* depend on whether it is shared. Assume that **`client`** and **`server`** are bundle-end variables, *shared* or *unshared*, and that we have suitable channels **`m`** and **`n`** for carrying their values. All that must be done is resign from the bundles currently referenced and assign the new references:

```
m ? client
```

$$\leadsto$$

```
m?tmp -> resign.client -> (client := tmp)
```

As before, the mapping for receiving a **`server`**, *shared* or *unshared*, from channel **`n`** is identical to the above, apart from name changes. Again, both these mappings are for classical (non-mobile) channels **`m`** and **`n`**. If these were themselves fields of a mobile channel bundle, the mappings from Section 4.4 would also be applied.

4.7 Assigning Mobile Channel Bundle Ends

Communication and assignment are intimately related in occam-π: an assignment has the same semantics as a communication of some value between variables in the *same* process.

Therefore, assignment of *unshared* mobiles leaves the source variable *undefined* and does not change the reference count on the assigned bundle – though the reference count on the bundle originally referenced by the target variable must decrement. Suppose `a` and `b` are *unshared* bundle-end variables of compatible type. Then:

```
a := b
```

$\rightsquigarrow$

```
resign.a -> (a := b); (b := undefined)
```

However, if `a` and `b` are *shared* bundle-end variables of compatible type. Then:

```
a := b
```

$\rightsquigarrow$

```
(enrol.b -> SKIP ||| resign.a -> SKIP); (a := b)
```

where, as in Section 4.2, we use interleaving to allow as much implementation freedom as possible.

4.8 Forking Processes with Mobile Channel Bundle End Parameters

Section 2 did not review the *forking* mechanism [1, 2, 3] of occam-π. However, passing the ends of mobile channel bundles to forked processes has been widely used in the complex system modelling being developed in our TUNA [13] and CoSMoS [14] projects – so, we do need a proper semantics for it.

Concurrency may be introduced into occam-π processes either by its classical **PAR** construct or by forking. A **PAR** construct does not terminate until *all* its concurrently running component processes terminate (and this maps to the CSP parallel operator). Often, a process needs to construct dynamically a new process, set it running concurrently with itself and continue – this is *forking*. Forking does not technically introduce anything that cannot be done with a **PAR** construct and recursion (which is precisely how its semantics are defined in [10]). However, forking does enable the more direct expression of certain idioms (e.g. a server loop that constructs processes to deal with new clients on demand and concurrently) and, pragmatically, has a more practical implementation (recursion in process components of a **PAR** construct makes an unbounded demand on memory, although a compiler may be able to optimize against that).

For the scope of this paper, we are only concerned about the semantics of passing the mobile ends of channel bundles to a forked process. For static arguments (e.g. data values or shared ends of classical channels), passing is by *communication* along a channel specific to the process type being forked.

Let `P(c)` be a process whose parameter is a mobile channel bundle end and let `forkP` be the channel, specific for `P`, connecting `P`-forking *application* processes to the (**PAR** recursive) `P`-*generator* process (Section 2.4.13 of [10]). The semantics of *forking* simply follows from the semantics of *sending* (Section 4.5 of this paper). If the argument is unshared, the forking process loses it:

```
FORK P (c)
```

$\rightsquigarrow$

```
forkP!c; (c := undefined)
```

More commonly, the argument will be shared, so that the forking process retains it:

```
FORK P (c)
```

$\rightsquigarrow$

```
enrol.c -> forkP!c -> SKIP
```

The details of the `P`-*generator* process at the receiving end of the `forkP` channel are the same as defined for the passing of mobile occam-π *barriers* in Section 2.4.13 of [10].

5. Conclusions and Future Work

The correct, flexible and efficient management of concurrency has growing importance with the advent of multicore processing elements. The mobile extensions to occam-π blend seamlessly into the classical language, maintaining the safety guarantees (e.g. against parallel race hazards), the simple communicating process programming model (which yields compositional semantics) and extreme lightness of overhead (memory and run-time).

This paper has presented a formal operational semantics, in terms of CSP, of all mobile channel mechanisms within occam-π. In turn, this leads to a denotational semantics (traces-failures-divergences [5, 6]) that enables formal reasoning about these aspects of occam-π systems. For use with the FDR model checker [7], the unbounded recursion of the mobile channel generator, `MC`, in Section 3 needs to be bounded – i.e. only a limited number of channel indices can be used, with their associated processes pooled for recycling (instead of terminated) when their reference count hits zero.

Also, these operational semantics specify crucial details of low-level code generation by occam-π compilers and the data structures and algorithms necessary for run-time support. The mobile channel *indices* generated by the real compilers are not the increasing sequence of natural numbers specified in Section 3. Actual memory addresses, dynamically allocated for the bundle structures, are used instead. This is *safe*, since these indices only need to be unique for each existing bundle, and *fast*, since the address gives direct access. Finally, the occam-π kernel is not actually implemented as the parallel collection specified by `MOBILE_CHANNEL_KERNEL` in Section 3, but by logic statically scheduled from it for serial use by the (extremely lightweight and multicore-safe) processes underlying occam-π.

We have not provided semantics for mobile channels in *arbitrary* CSP systems – only those conforming to the classical occam constraint of no output guards in external choice. That constraint was necessary for the modelling of (mobile) channels as processes, which in turn was necessary for the separate modelling of channel ends.

Without that separation, we could not allow *interleaving* of all application processes over the infinite enumeration of potential mobile channels: since if that were allowed, no application process would be able to synchronise with another to send a message down any mobile channel! With separate static channels modelling the *ends* of mobile channels, application processes synchronise with the *kernel* processes (modeling the mobile channels themselves) to communicate with each other and no buffering is introduced (which means that the fully synchronised semantics of CSP channel events are maintained). Application processes synchronise with each other over the *fixed* set of static channels (and other events) through which with they are connected (as normal) or synchronise with the kernel over the infinite, *but still fixed*, set of kernel channels. Either way, the synchronisation sets associated with all parallel operators remain fixed and we have a CSP for mobile channels.

Recently, fast algorithms for implementing external choice whose leading events are multiway synchronisations have been engineered [15, 16] and have been built into the JCSP library [17] and an experimental compiler for exercising CSP [18]. These can be simply applied to allow output guards.

Providing a CSP model for mobile channels both of whose ends can be used as guards in an external choice is an open question. One way may be to drop the separate modelling of channel ends (along with their supporting kernel processes) and divide application processes into two sets: those that receive mobile channel communications and those that send them. The processes in each set interleave with each other over the set of mobile (but actually normal) channels and the two process sets synchronise with each other over the fixed, but infinite, set of mobiles. This is elegant but not very flexible! Even if the division could be defined individually for each mobile channel, it still looks too constraining. In occam-π, mobile channel bundle ends can be passed to *any* processes with the right connections. Some serial processes may hold and use either end of the same channel bundle – though, hopefully, not at the same time! The semantics presented here for mobile channels without output guards has no such constraints.

Acknowledgements

This work started under the TUNA project (EPSRC EP/C516966/1) and is being applied as part of the CoSMoS project (EPSRC grants EP/E053505/1 and EP/E049419/1). We are indebted to all our colleagues in both these projects for their insights, motivation and encouragement. We are also very grateful to the anonymous referees of this paper and to everyone in the Concurrency sub-group at Kent for their many criticisms and corrections to so many crucial points of detail.

References

[1] P.H. Welch and F.R.M. Barnes. Communicating Mobile Processes: Introducing occam-π. In *'25 Years of CSP',* Lecture Notes in Computer Science vol. 3525, A. Abdallah, C. Jones, and J. Sanders, editors, Springer, pp. 175–210, Apr. 2005.

[2] F.R.M.Barnes and P.H.Welch. Prioritised Dynamic Communicating and Mobile Processes. *IEE Proceedings-Software*, 150(2):121-136, April 2003.

[3] F.R.M. Barnes and P.H. Welch. Prioritised dynamic communicating processes: Part 1. In James Pascoe et al., editors, *Communicating Process Architectures 2002*, volume 60 of *Concurrent Systems Engineering*, pp. 321-352, IOS Press, Amsterdam, The Netherlands, September 2002.

[4] C.A.R. Hoare. Communicating Sequential Processes. In *CACM*, vol. 21-8, pp. 666-677, August, 1978.

[5] C.A.R. Hoare. Communicating Sequential Processes. Prentice-Hall, ISBN 0-13-153271-5, 1985.

[6] A.W. Roscoe. The Theory and Practice of Concurrency. Prentice-Hall, ISBN 0-13-674409-5, 1997

[7] Formal Systems (Europe) Ltd. FDR2 Manual, **`www.fsel.com/documentation/fdr2/fdr2manual.pdf`**. May 2008.

[8] R. Milner. Communcating and Mobile Systems: the π-Calculus. Cambridge University Press, ISBN-13:9780521658690, 1999.

[9] Inmos Ltd. occam3 Draft Reference Manual, 1991.

[10] P.H. Welch and F.R.M. Barnes. Mobile Barriers for occam-π: Semantics, Implementation and Application. In J.F. Broenink et al., editors, *Communicating Process Architectures 2005*, volume 63 of *Concurrent Systems Engineering Series*, pp. 289-316, IOS Press, The Netherlands, September 2005.

[11] P.H. Welch and J.M.R Martin. Formal Analysis of Concurrent Java Systems. In A.W.P. Bakkers et al., editors, *Communicating Process Architectures 2000*, volume 58 of *Concurrent Systems Engineering*, pp. 275-301. WoTUG, IOS Press (Amsterdam), September 2000.

[12] J.C.P. Woodcock and A.L.C. Cavalcanti. The Semantics of Circus. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pp 184-203, Springer-Verlag, 2002.

[13] S. Stepney, J.C.P. Woodcock, F.A.C. Polack, A.L.C. Cavalcanti, S. Schreider, H.E. Treharne and P.H.Welch. TUNA: Theory Underpinning Nanotech Assemblers (Feasibility Study). EPSRC grant EP/C516966/1 (Universities of York, Surrey and Kent). Final report from **`www.cs.york.ac.uk/nature/tuna/`**, November, 2006.

[14] S. Stepney, F.A.C. Polack, J. Timmis, A.M. Tyrrell, M.A. Bates, P.H. Welch and F.R.M. Barnes. CoSMoS: Complex Systems Modelling and Simulation. EPSRC grant EP/E053505/1 (University of York) and EP/E049419/1 (University of Kent), October 2007 – September 2011.

[15] A.A. McEwan. Concurrent Program Development, DPhil Thesis, University of Oxford, 2006.

[16] P.H. Welch, F.R.M. Barnes, and F. Polack. Communicating Complex Systems. In *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS-2006)*, pp 107-117, Stanford University, IEEE Society Press, August 2006.

[17] P.H. Welch, N.C.C. Brown, J. Moores, K. Chalmers and B. Sputh. Integrating and Extending JCSP.In Alistair A. McEwan et al., editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering Series*, pp. 349-370, Amsterdam, The Netherlands, July 2007. IOS Press.

[18] F.R.M. Barnes. Compiling CSP. In P.H.Welch et al., editors, *Communicating Process Architectures 2006*, volume 64 of *Concurrent Systems Engineering Series*, pp. 377-388, Amsterdam, The Netherlands, September 2006. IOS Press.