A proposal for Pi-calculus extensions to XC (PiXC)

1 Introduction

PiXC is a proposal to extend the XC programming language [2] with mobility, using Milner's pi-calculus [12] as a model. These extensions should lead towards a dynamic Operating System Architecture for XMOS processors [1].

1.1 Aims

The aims of PiXC are:

- To use the pi-calculus [12] as a model to extend XC [2] with mobility
- To formalise a process-based Operating System paradigm on XMOS processors [1]

1.2 Summary

XMOS processors such as the XS-1 core [1] equip system designers with process-oriented hardware and software building blocks, not unlike the INMOS Transputer [7] cores which predated them. An XMOS XS-1 is a multi-core processor, each core with its own memory and i/o links [1]. The cores embed support for a number of services commonly provided by an operating system on general purpose processors, including multi-threading, communication, general purpose i/o, and timers. And like INMOS with Occam [5], XMOS provide a parallel programming language in XC [2] which can be considered the Operating System Language of XS-1 processors.

Using XC, an operating system for XS-1 processors is not a necessity in an embedded homogenous system. However, there are a number of Operating System technologies that could be applied to extend the services provided by XS-1 and XC, and those that could be applied to network with heterogenous systems. Mobility in the pi-calculus [12] can serve as a model for operating systems, and Occam 2.1 [5] has been extended with elements of mobility in Occam-pi [9], so these ideas might be usefully engineered into XC [2].

1.2.1 Operating System Architecture

In traditional operating systems, especially those targetting embedded systems, the executable unit is commonly statically linked so that the set of possible services is fixed during compilation, and resources like memory and CPU are shared among processes through virtualisation. In systems with an interface, libraries may be dynamically link-loaded at runtime to extend the set of kernel modules or device drivers. However, the concept of operation, the way people view and program for operating systems, largely remains fixed on a rigid architecture.

As Milner begins the introduction [12], "[pi is]...a calculus for analysing properties of concurrent communicating proceses, which may grow and shrink and move about". This fluidity takes the pi-Calculus to a level not reached by CCS, CSP or classical automata. In the broadest nutshell, pi is an extension to CCS that allows the very channels used for inter-process communication to be passed as communication elements. In communicating a channel, the effect is a dynamic re-mapping of the architecture.

1.2.2 Where we were

From Barron's concept for the transputer [6] that "a calculus is likely to take the form of a program language, which has primitive operations enabling communication between parallel processes", the relationship between Occam and CSP was exploited heavily by INMOS. From its beginning [3] Hoare's CSP postulated "...that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method." Over time CSP evolved into a model for concurrency [4] which tests processes against logical specifications (e.g. failures and trace semantics), when Occam took-up the software baton.

Independently of and at about the same time as CSP emerged, and in response to a critique of von-Neumann automata, CCS [11] surfaced with the assertion that "communication and concurrency are complementary notions" CCS theories of observational equivalence test process automata against each other at their communication ports. This provides a complementary viewpoint to testing logical specifications and properties, and is a good fit with Barron's ideas of a component [6] "The transputer focuses interest on the transfer of information across a boundary, rather than on the processing of information within that boundary."

1.2.3 Where we are

Aside from influential ideas, several of which have re-emerged in different forms, Occam has brought about neither significant developments in Operating System Architecture nor a compelling reason for industry to make the costly transition to parallel programming en masse. However, the ideas continue to evolve and nowadays the Occam mantel lies with KroC [8] and Occam-pi [9].

While the XMOS architecture makes it practical to use software to perform many functions that are traditionally implemented in hardware, transition to the new technology could be made easier still by providing an operating system that hides complexities of programming communicating processes, while bringing a comparable benefit to the way traditional operating systems hide virtualisation of resources. And it is with the pi-calculus [12] successor to CCS, where processes can migrate and channels themselves can be passed between processes, that a significant theoretical breakthrough has been made, so that instead of a static architecture we have a dynamic model.

1.2.4 Where we want to be

The aim is to bring the new hardware and theoretical advances to bear on a new Operating System Architecture, to ease the transition for designers and industry more widely into parallel system design and process-oriented software development, using XMOS processors.

Aside from an entry in Wikipedia, "[XMOS] Threads can also use Channels to communicate and synchronise allowing a CSP style of programming", there is scant detail relating to the formal CSP-basis of XC/XS-1. Whether one will emerge likely depends on any academic gain in "translating" to XC from Occam, if XMOS doesn't publish. And, like the theoretical foundation for Occam-pi [9] has evolved (by blending elements of the CSP, pi-Calculus and Petri-net models [10]), we can adapt a model best-suited for a dynamic XC-based operating system in the pi-calculus.

1.2.4.1 Fluid architecture

Pi is considered well-suited for modelling internets [12]; consider an internet (TCP/IP) server that accepts socket connection requests at a well-known address, and spawns a child process to serve each new connection concurrently. Pi is adept for modelling a dynamic operating system, whose very architecture changes dynamically to fulfil the state transitions of its processes. Mobility of channels and processes enables network topology to evolve in response to run-time events [10].

2 Constructs

Existing operational constructs in XC/XS-1 required to implement mobility are first identified, then new syntactic constructs for XC are proposed based upon Pi-calculus semantics.

2.1 Operational constructs

PiXC requires operational constructs in the XS-1 in order to migrate runnable processes and to allocate dynamic channel endpoints.

2.1.1 Runnable processes

The XS-1 instruction set provides for runnable processes through thread allocation and freeing, which is commonly accessed using the XC PAR construct. This operates quite distinctly from a *fork* in UNIX, and requires code for the process entry point to be statically linked. A process is run when code and data pointers are loaded into a set of registers, which are presented to the XS-1 scheduler. And a process terminates on return (or join), when its register set is freed.

2.1.2 Dynamic channels

Dynamic channels are those channel endpoint resources which are allocated to a process, and eventually freed, for which XS-1 instructions and registers are provided.

2.2 New constructs

New syntactic constructs are needed to program mobility in PiXC.

2.2.1 Mobile processes

Mobile processes are those which migrate, to run at different locations at different times.

To be mobile, rather than just invoked in the accepted way, a process has to preserve some state and provide various function entry points, which suggests a *class-like* structure operating entirely within the process. And to be safe, a process has to be sent through a channel to a new location.

2.2.1.1 Applications

Consider a dynamic master-slave scenario, where the operating system measures runtime performance for some application (we don't say how this measurement works) and decides to launch a mobile slave process to take some of the workload on a new processor.

Consider a dynamic device driver, where a runtime event is input to the operating system which decides to launch a mobile driver process to service it.

2.2.1.2 Declarations

A mobile process is a function declared with the *mobile* qualifier. This provides a guide to the compiler in scoping both a *mobile* code segment and process member variables.

```
mobile void p0( chanend out, chanend in )    // p0 is a qualified function
{
}
```

In general the *mobile* qualifier has no effect on the function body operational semantics, or on any nested function calls; however, there are certain restrictions on code and data scope which follow.

A mobile process cannot be invoked like a normal function.

```
int f( chanend top0, chanend fromp0 )
{
    p0( fromp0, top0 );     // compiler error, cannot invoke mobile process
}
```

Rather, before a mobile process is run, it first has to be received along a channel by a running process context (see 2.2.1.5), following which a new thread (runnable process) is scheduled.

2.2.1.3 Scope

Mobile process scope refers to static variables and functions accessed by a mobile process. The compilation must arrange storage for a mobile process to enable its member functions and data to reside in different XS-1 core memories at different times.

2.2.1.3.1 Member variables (static)

Process member variables maintain state across migrations. In particular they support mobile processes being communicated along channels, to resume at a new future location and context.

There are no new syntactic qualifiers for mobile process member variables. The static keyword serves in the usual way, so that member variables are taken to lie within *mobile* process scope.

```
mobile void p1( chanend out, chanend in )
{
    mobile static int i = 0;      // i is a static int in mobile scope
    mobile static timer t;        // t is a static timer in mobile scope
}
```

The *mobile* qualifier on the static variables would not be written by programmers; it shows how the compiler needs to allocate storage for member variables, as if it had been written. The compiler may rename the member variables to fully qualify them: p1::i, p1::t, p1::in and p1::out.

2.2.1.3.2 Non-member variables

Variables nested within a mobile process without the static qualifier are just taken as temporary automatic (stack) variables.

2.2.1.3.3 Member functions (static)

Process member functions are those referenced within a *mobile* process scope. In particular they support mobile processes being communicated along channels, to run at a new future location.

There are no new syntactic qualifiers for mobile process member functions. The static keyword serves to declare member functions in a new way (for XC) within *mobile* process scope.

```
mobile void p2( chanend out, chanend in )
{
    mobile static int f( );       // f is a static function in mobile scope
}
```

The *mobile* qualifier on static function declarations would not be written by programmers; it shows how the compilation needs to allocate a code segment for member functions, as if it had been written. The compiler may rename member functions to fully qualify them: p2::f.

Member functions themselves are written as static functions at file level, with no further qualifiers.

```
mobile static int f( void )
{
}
```

The *mobile* qualifier would not be written by programmers; rather the compiler always assumes it is inherited if and only if a static function is referenced by a mobile process.

2.2.1.3.3.1 Nested member variables

Follows as 2.2.1.3.1 ; a *mobile* qualifier on static variables would not be written by programmers but is assumed by the compiler when a nested function inherits the *mobile* qualifier.

2.2.1.3.3.2 Nested member functions

Follows as 2.2.1.3.3 ; a *mobile* qualifier on static functions would not be written by programmers; but the referenced function should be declared within mobile process scope. The source code:

```
mobile void p3( chanend out, chanend in )    // p3 is a mobile process
{
    int i;                  // i is an automatic (stack) int
    static int f( );        // f is a static function in process scope
    static int g( );        // g is a static function in process scope
    i = f( );               // i holds the result of f of g
}
static int f( void )
{
    return( g( ));          // result is value of f of g
}
static int g( void )
{
    return( 0 );            // result is value of g
}
```

2.2.1.3.4 Non-member functions

There is no such thing as a non-member function. A non-static function cannot be called within *mobile* process scope, and results in a compilation error.

```
mobile void p4( chanend out, chanend in )
{
    int f( );   // compiler error, function needs static in mobile scope
    g( );       // compiler warning, function g not declared in mobile scope
}
```

The reason for this threefold: to allow the compiler to treat file-scope and process-scope in similar yet distinct ways; to avoid the compiler having to search through other compiled units to locate required code; and to prevent the need for multiple (and hence incoherent) copies of data. For the latter, consider if a function, *g*, is both called by a mobile process and also called elsewhere by a plain function, then it would be necessary to store two copies of *g* member variables, as the copy for the mobile process may be communicated into a remote core memory.

Note the caveat that library functions cannot be called within mobile process scope (unless the whole mobile process and its members are compiled into the same library unit).

2.2.1.4 Sending

A mobile process is declared precisely to enable its communication along a channel, for execution by a receiving process context as a new thread.

The top-level process is sent as a unit. Here, *q0* is a function running in the source process context.

```
void q0( chanend c )
{
    c <: p1;    // p1 is sent along channel c; its state ceases to exist in
                // the context of q0 process, and its thread is terminated
}
```

There is no new syntax, and the existing communication output primitive serves. When a mobile process is sent, a destructor may be called and its state goes out of context in *q0*.

At runtime the compiler will cause the (fully qualified) member variables to be sent as a single packet or structure (or Occam 2.1 style protocol) to the receiving process.

At buildtime, where it is communicated, the linker will link the object code for a mobile process with each processor's executable unit (so the code itself is not sent at runtime), and allocate space to receive the mobile process member variables.

2.2.1.5 Receiving

Until a mobile process is received (into a destination process context) no valid state exists for it. A mobile process is received by a destination process along a channel; the top-level unit as a whole is received. Here, *q1* is a function running in the receiving destination process context.

```
void q1( chanend c )
{
    c :> p1;    // p1 is received along channel c; its state exists in the
                // context of q1 process, and p1 runs as a new thread
}
```

There is no new syntax, and the existing communication input primitive serves. Once a mobile process is received, a constructor may be called and its state comes into context.

Rather than a free variable name, the input parameter is the bound mobile process name; this defines the data structure received, being the process member variables.

The *p1* state remains valid for the duration of the receiving process context, after which a mobile process destructor may be called. If *p1* should persist beyond the lifetime of the destination process context, it should be communicated to another process before this destination process terminates.

2.2.2 Mobile channels

Mobile channels are those which migrate, to bind with different endpoints at different times.

To be mobile, rather than just used in the accepted way, a channel has first to be emptied before migrating. And to be safe, a channel has to be sent from a source process through a channel, possibly itself, to a destination process. Once sent, the source process can no longer access the migrated channel; and the destination process can only access the migrated channel once received.

2.2.2.1 Applications

Consider a dynamic master-slave example, where the operating system measures runtime performance for some algorithm (we don't say how this measurement works) and decides to send a channel (endpoint) to a slave process on some other processor for it to take some of the workload.

Consider a dynamic device driver, where a runtime event is input to the operating system which decides to send a channel (endpoint) to a driver process for it to service the event.

2.2.2.2 Declarations

New syntax declares mobile channels and mobile chanends.

2.2.2.2.1 channel

A mobile channel is a channel declared with the *mobile* qualifier. This provides a guide to the compiler for allocating and binding channel endpoints.

```
chan mobile c;        // c is a qualified channel
```

A mobile channel can be used at global level by a sequential process, or, unlike a conventional channel, passed as an argument to multiple processes in a *PAR* statement. This allows the compiler to know which contexts the channel may bind in.

```
chan mobile data;
par
{
    procA( data );
    procB( data );
    procC( data );
}
```

In general the mobile qualifier has no effect on a channel semantics as a point-to-point pipe; rather it extends endpoints with variable-like semantics, allowing them to be dynamically (re-)bound.

Mobile endpoints are bound in two cases: when a mobile channel is input by a destination process; or when a processes fixes it, by omitting the mobile qualifier from its definition.

2.2.2.2.2 chanend

Channel endpoints are passed into functions as parameters. The *mobile* qualifier is used when declaring a mobile channel endpoint in a function scope.

```
void x0( chanend mobile c )      // c is not bound on entry to x0
{                                // (it may have been bound by the caller)
}
```

2.2.2.3 Scope

The *mobile* qualifier is used by the compiler to allocate storage space for channel endpoints, but not to bind them. The compiler may rename local channel to fully qualify them: x0::c.

2.2.2.3.1 constructor

The compiler may call channel (endpoint) constructor functions in two different cases: on entry to a function if the channel is fixed, and if *c* is input communicated.

```
void x1( chanend c )        // c is bound (fixed) on entry to x1
{
}
```

Without a *mobile* qualifier, x1::c is fixed (bound), even if the endpoint passed in is qualified.

2.2.2.3.2 destructor

The compiler may call channel (endpoint) destructor functions in two different cases: on exit from a function, where the channel endpoint may no longer be bound (once it passes out of scope); and if *c* is output communicated, where the channel is always unbound.

2.2.2.4 Sending

Mobile channels may be sent from a source process to a destination, communicated along a channel, to re-bind with the destination process.

```
void x2( chanend d, chanend mobile c )
{
    d <: c;     // mobile channel endpoint c is sent along channel d,
                // its destructor unbinds c from the process context of x2
}
```

There is no new syntax, and the existing communication output primitive serves. Once a channel is sent, an (endpoint) destructor may be called and the channel is unbound.

2.2.2.4.1 Sending itself

A mobile channel may be sent from one process to another along itself, to bind at a new endpoint.

```
void x3( chanend mobile c )
{
    c <: c;     // mobile channel endpoint c is sent along channel c,
                // its destructor unbinds c from the process context of x3
}
```

There is no new syntax, and the existing communication output primitive serves. The compiler will cause the channel endpoint to be sent. Once a channel is sent, an (endpoint) destructor may be called and the channel is unbound.

Sending a channel along itself is useful if you consider a master-slave process relationship, in which the master sends a channel to a slave process for servicing and when that slave is done it returns the channel (see 2.2.3.2.2).

2.2.2.5 Receiving

Mobile channels may be received by a destination process from a source, communicated along channels, to bind at a new endpoint.

```
void y0( chanend d, chanend mobile e ) // e is not bound on entry to y0
{
    d :> e;     // mobile channel endpoint e is received along channel d,
                // when its constructor is called and e is bound
}
```

There is no new syntax, and the existing communication input primitive serves. The chanend *e* is within scope from the local function declaration, but unbound. The chanend *e* will be able to receive valid input only after it is itself received in a channel communication. On receipt, an (endpoint) constructor may be called and the channel *e* is bound; after which *e* is able to receive valid input.

2.2.2.5.1 Receiving itself

Any mobile channel (endpoint) can input to itself. The new input value overwrites the former value with immediate effect.

```
void y1( chanend d, chanend mobile e ) // e is not bound on entry to y1
{
    d :> e;     // mobile channel endpoint e is received on d and bound
    e :> e;     // mobile channel endpoint e is received on e and re-bound
    d :> d;     // compiler error, received channel is not mobile (or fixed)
}
```

This allows the compiler to treat channel-scope in a consistent way, in an extended variable-like semantics.

The intuitive appeal of this capability does not reduce the number of channels required in a system as *d :> e* is required the first time, so it may yet be deprecated or at least side-lined as a simple variable assignment (see 2.3.2).

2.2.3 Design patterns

Several design patterns can illustrate how to use mobile processes and channels. These also provide a further insight into the extended PiXC syntax and its meaning.

2.2.3.1 Mobile processes

With regard to a process-based Operating System paradigm, mobile processes allow services to migrate and for load-balancing across available hardware. But for such powerful concepts, they are actually fairly simple building blocks. A crucial concept of processes is statefulness, so that even after migrating (from one hardware processor to another) they can resume where they left off.

2.2.3.1.1 State machine pattern

The state machine pattern is used to enable a mobile process to resume execution after migrating. There are no special constructs required for a state machine; rather, a design pattern using a static variable with process-qualified scope serves.

```
mobile void p0( chanend out, chanend in )
{
    static int p0_state = 0;
    switch( p0_state )
    {
          case 0 :
          {
          }
          break;
          ...
          default :
          {
          }
    }
}
```

2.2.3.1.2 Roaming pattern

In the roaming pattern, mobile processes migrate between contexts as a function of their data- or event-driven processing. Here, *roam0* is a function in some process context.

```
void roam0( chanend c, chanend )
{
    int t = 1;              // initially assume task 1
    c :> p1;                // wait to receive mobile process p1
    for( ; t > 0; )
    {
                            // calculate next task, t, on p1 state
          switch( t )
          {
                case 1:
                {
                            // perform task t1, interacting with p1
                }
                break;
                ...
                default :
                {
                            // not a task for roam0 in this context
                      c <: p1;    // give mobile process p1 to new context
                      t = 0;      // exit loop
                }
```

```
            }
        }
    }
```

2.2.3.2 Mobile channels

With regard to a process-based Operating System paradigm, mobile channels allow data to be shared by processes in different turns that follows data event processing. Sharing data allows processes to be small and purposeful, handing over channels (and the data that passes through them) to other processes in turn. This is a powerful concept, in which data flows over a network of inter-connected processes, and yet mobile channels are fairly simple building blocks.

2.2.3.2.1 Master-Slave pattern

In the case of a master-slave process arrangement, each master-slave share a *control* channel over which will be communicated a *data* channel. A seperate source process outputting on the *data* channel does not (need to) know which slave it is connected with. This pattern seperates control flow on fixed channels from the data flow over mobile channels.

```
chan controlA, controlB;
chan mobile data;
par
{
    source( data );
    master( controlA, controlB, data );
    slaveA( controlA, data );
    slaveB( controlB, data );
}
void master( chanend A, chanend B, chanend mobile D )
{
    A <: D;           // give data channel D endpoint to slaveA
    A :> D;           // take data channel D endpoint from slaveA
    B <: D;           // give data channel D endpoint to slaveB
    B :> D;           // take data channel D endpoint from slaveB
}
void slaveA( chanend ctrl, chanend mobile data )
{
    ctrl :> data;     // wait for a worker channel from master
                      // process i/o on data
    ctrl <: data;     // return data to master
}
void slaveB( chanend ctrl, chanend mobile data )
{
    ctrl :> data;     // wait for a worker channel from master
                      // process i/o on data
```

```
    ctrl <: data;     // return data to master
}
void source( chanend out )       // out is fixed
{
    // perform i/o on out
}
```

Processes *master*, *slaveA*, *slaveB* and *source* may reside on different processor cores and memories.

2.2.3.2.2 Self-Send pattern

A channel can be received, and when done with, sent along itself to the peer process. This pattern mixes both control flow and data flow on mobile channels. (However, it doesn't do away with the need for a control channel to send the initial mobile endpoint.)

```
chan controlA, controlB;
chan mobile dandc;
par
{
    selfsend( controlA, controlB, dandc );
    slaveA( controlA, dandc );
    slaveB( controlB, dandc );
}
void selfsend( chanend A, chanend B, chanend mobile dac )
{
    A <: dac;         // give mobile channel dac[1] to slaveA
                      // process i/o on dac[0] with slaveA
    dac :> dac;       // receive mobile channel into dac[1]
    B <: dac;         // give mobile channel dac[1] to slaveB
                      // process i/o on dac[0] with slaveB
    dac :> dac;       // receive mobile channel into dac[1]
}
void slaveA( chanend ctrl, chanend mobile dac )
{
    ctrl :> dac;      // wait for a mobile channel from master
                      // process i/o on dac
    dac <: dac;       // return dac
}
void slaveB( chanend mlink, chanend mobile c )
{
    ctrl :> dac;      // wait for a mobile channel from master
                      // process i/o on dac
    dac <: dac;       // return dac
}
```

The two endpoints s[0] and s[1] of *dac* are identified to aid the description of mobile channels; s[0] always occurs on the left-hand-side of both an input or an output statement, as the communicating channel; conversely, s[1] only occurs on the right-hand-side of both an input or an output statement, as the variable. The processes *master*, *slaveA* and *slaveB* may reside on different processors.

2.2.3.2.3 Pipeline pattern

The pipeline pattern is analogous to data passing through a processor pipeline. In this pattern for mobile channels, data passes through a set of processes by communicating a mobile channel between them. When it holds the mobile channel each process performs some function(s) on the data input from it, before deciding which process is next to receive the mobile channel. Unlike a static arrangement, the sequence of processes does not have to be pre-determined.

A variation on the master-slave pattern, prior to returning the mobile channel, each slave first sends a value along it to identify which slave process should be the next receiver.

```
chan controlA, controlB;
chan mobile data;
par
{
    pipeline( controlA, controlB, data );
    slaveA( controlA, data );
    slaveB( controlB, data );
}
void pipeline( chanend A, chanend B, chanend mobile D )
{
    int v = 0;        // initially send server channel D[1] to slaveA
    for( ;; )
    {
          switch( v )
          {
                case 0:
                {
                      A <: D;     // give server channel D[1] to slaveA
                }
                break;
                case 1:
                {
                      B <: D;     // give server channel D[1] to slaveB
                }
                break;
          }
                // process i/o on D[0] with slave
          D :> v;     // identity of next slave to receive mobile channel
          D :> D;     // receive mobile channel into D[1]
```

```
    }
}
void slaveA( chanend ctrl, chanend mobile d )
{
    int v = 1;  // default next process identity
    for( ;; )
    {
          ctrl :> d;  // wait for a worker channel from master
                      // process i/o on d
                      // calculate v, identity of next receiving process
          d <: v;     // send identity of next process in pipeline
          d <: c;     // return d
    }
}
void slaveB( chanend ctrl, chanend mobile d )
{
    int v = 0;  // default next process identity
    for( ;; )
    {
          ctrl :> d;  // wait for a worker channel from master
                      // process i/o on d
                      // calculate v, identity of next receiving process
          d <: v;     // send identity of next process in pipeline
          d <: d;     // return d
    }
}
```

2.3 Extensions

The mobile process extensions described are simplistic, and could be extended in several ways.

2.3.1 Scope

The file scope static declaration for process member variables and process member functions could be relaxed if the compiler/linker will search through a collection of source files and check for reference calls [hard].

Alternatively and perhaps less pleasingly (as the language is XC and not XC++), programmers could write fully qualified names for process member variables and functions, along the lines outlined in the previous sections: p0::x [easier].

2.3.2 Assignment

Mobile channel endpoints are declared in the definition of a containing function; this is for consistency with XC. They extend endpoints with a variable-like semantics. And values are

assigned (to a mobile endpoint) as the variable component of a channel input expression.

If chanend was made a proper data type, an equivalent variable-like semantics for mobile chanend binding could be provided through assignment.

```
void swapchan( chanend control, chanend mobile d1, chanend mobile d2 )
{
    chanend mobile c;       // a local variable of type chanend, unbound
    control :> d1;          // bind d1 endpoint by input (constructor)
    control :> d2;          // bind d2 endpoint by input (constructor)
    c = d2;                 // temporary, bind c by assignment
    d2 = d1;                // re-bind d2 by assignment
    d1 = c;                 // re-bind d1 by assignment
}
```

This is safe provided all channel bindings were input to the function through a communication expression.

2.3.3 Arrays

If the mobile process defines a proper data type then arrays of processes could be declared, sharing a single definition. For example,

```
mobile void p[8]( void );
```

This provides an array of alike mobile processes, each with independent (member variables) state, able to roam the processor network looking for work assignment.

2.3.4 Heterogenous networks

By sending process member variables only, and not code, the mobile model extends to heterogenous processor networks (in which a distinct compiled unit exists for each processor type, e.g. XS-1, ARM7). The requirement for operating in heterogenous processor networks is to transmit the data in a uniform layout, for which XML could be used.

3 Conclusions

PiXC is a proposal to extend XC with mobility and to formalise process-oriented semantics, leading towards a dynamic Operating System Architecture for XMOS processors. Syntax and a meaning for mobile processes and mobile channels is given. Certain syntax and semantics are omitted (and others are probably waiting discovery), and those given could yet change in future drafts.

References

[1] XMOS XS1 Architecture, May D., 2008

[2] Programming XC on XMOS devices, Watt D., 2009

[3] Communicating Sequential Processes, Hoare C.A.R., C.ACM 21(8) pp:666-677, 1978.

[4] Communicating Sequential Processes, Hoare C.A.R., Prentice Hall, 1985.

[5] Occam Programming Manual, INMOS Limited, Prentice Hall, 1984. [See also Occam2, 1988]

[6] The Transputer, Barron I., 1978.

[7] Transputer Reference Manual, INMOS Limited, Prentice Hall, 1988.

[8] Kent Relocatable Occam Compiler, University of Kent

[9] Occam-pi, RmoX, Occam-pi and RMoX, University of Kent

[10] Communicating Mobile Processes, Communicating Mobile Channels, University of Kent

[11] Communication and Concurrency, Milner R., Prentice Hall, 1989. [see also Calculus of Communicating Systems, Milner A.J.R., Springer Verlag LNCS 92, 1980.]

[12] pi-Calculus, Milner R., Cambridge, 1999.