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

1 Introduction

XC [2] is a general purpose programming language, for expressing parallel algorithms. PiXC is a proposal to extend XC with mobility, using Milner's pi-calculus [12] as a model. Mobility enables the program function and data control flow to evolve in response to run-time input.

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 (extensions to) a compiler for PiXC that targets XMOS processors [1]

1.2 Version Control

0.1	Initial proposal for mobile channels and mobile processes (to target a dynamic Operating System).
0.2	Added initial bindings (mobile:b) and endpoint qualifiers (mobile:01) for mobile channels; added initial bindings for mobile processes. Took design patterns into a new section. Added section for semantics. Added section for case studies.
0.2.1	Reposition PiXC as a general purpose programming language (well-positioned to implement a dynamic Operating System).
0.3	Described protocols.

1.3 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].

Milner writes [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. Broadly, 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.

Mobility in the pi-calculus [12] can be used to express parallel algorithms, and Occam 2.1 [5] has been extended with elements of mobility in Occam-pi [9], so these ideas might be usefully added to XC [2]. PiXC makes programming with the Pi-calculus accessible and familiar, by modestly extending the XC [2] programming language. These extensions allow mobile processes and channels to be declared and have a language semantics.

1.3.1 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."

1.3.2 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 more widespread through a general purpose programming language that can express mobile algorithms, i.e. ones in which the data flow and control flow change dynamically in response to system input. 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.

1.3.3 Where we want to be

The aim is to bring the new hardware and theoretical advances to bear on a mobile programming language, 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. 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 expressing mobile algorithms in the pi-calculus to XC.

1.3.3.1 Fluid architecture

Pi is 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 also well-suited for general purpose programming and expressing algorithms; the dynamic communication of mobile channels and mobile processes through channels enables the function and data control flow 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 mobile context 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 master measures runtime performance for some application (we don't say how this measurement works) and decides to run 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 and a mobile driver process is run 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 Mobile context

Mobile context refers to the scope of 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

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 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 qualify the mobile context: p1::i, p1::t, p1::in and p1::out, which is not exposed as syntax the programmer would write.

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

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 qualify the mobile context: 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 context
    g( );       // compiler warning, function g not declared in mobile context
}
```

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).

2.2.1.3.5 Initial binding

Unless otherwise received, mobile processes are initially bound (as if they had been received) by passing mobile processes as parameters to process threads. Here *myThread* is a function that is run as a process (from a *par* statement).

```
void myThread( mobile p1 )
{
}
```

The compiler will check each mobile process is given at most one initial binding. If no initial binding is provided for a mobile process, the program may deadlock as no running thread would ever receive or be able to send the mobile; the compiler could issue a warning in this case. (This is made more clear in the case studies in section 5.1.2 .)

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 mobile process is sent as a unit. Here, *q0* is a function running in the source process context, where it is assumed *p1* is currently bound.

```
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, see 2.2.3) 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 )
```

```
{
    chan i, o;
    c :> p1( o, i );  // p1 is received along channel c;
                      // its state exists in the context of q1 process,
                      // bound to the local chanends,
                      // and p1 runs concurrently 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; this includes binding its input and output chanends to local channels.

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, which includes un-binding the local chanends. 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 master 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 and a channel (endpoint) is sent 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.

2.2.2.2.2 chanend

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.

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 Mobile context

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

2.2.2.3.1.1 Initial binding

Unless otherwise fixed, unbound chanend points are initially bound (as if they had been received) by further qualifying the function definition of a mobile channel with *mobile:b*.

```
void x2( chanend mobile:b c )    // c is initially bound on entry to x2
{                                // (but otherwise behaves as a mobile)
}
```

Each channel has exactly two endpoints that can be fixed or initially bound. The compiler will check that the allowed endpoints are initially bound or fixed.

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 x3( 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 x3
}
```

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 x4( chanend mobile c )
{
    c <: c;     // mobile channel endpoint c is sent along channel c,
                // its destructor unbinds c from the process context of x4
}
```

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 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, *y0*, 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.

2.2.3 Protocols

As mobile processes and mobile channels are developed and re-used, maintaining correct data exchange between processes can become a source of errors. A *protocol* can be used to specify the data structure that is communicated over a channel. This can be used with both mobile and plain (non-mobile) channels to enable compiler checking for data exchange. The idea is based on the protocol found in Occam 2.1 [5], or perhaps more like Pascal records.

2.2.3.1 Channel context

A protocol can be used in the context of a channel and chanends.

2.2.3.1.1 Channel protocol

A channel is qualified with a protocol by specifying a data structure. This provides a guide to the compiler for binding channel endpoints.

```
chan c0:int;          // c0 is a channel qualified with a simple protocol
```

A mobile channel can be similarly qualified with a protocol.

```
chan mobile c1:int;   // c1 is a mobile channel qualified with a protocol
```

The protocol data structure is a container, which is a sequence of data types. A simple protocol contains a single (scalar) data type, such as c0 and c1 above. A complex protocol contains more than one data type in a sequence.

```
chan c2:int:char;     // c2 is a channel which communicates a complex protocol
```

While the definition of a protocol shares similarities with a structure (or class), its elements are not named and cannot be communicated independently of each other.

2.2.3.1.2 Chanend protocol

A channel endpoint (mobile or otherwise) can be bound with a protocol.

```
void x0( chanend c:int )         // c is bound on entry to x0 by a protocol
{
}
```

The compiler shall perform static checks on the protocol data type consistency of a channel declaration, its binding to chanends, and its use within process and function scope.

2.2.3.2 Protocol types

A protocol is a type of sequence, named using the *protocol* keyword.

```
protocol p0          // p0 is a named protocol
{
    int:char;        // p0 is a sequence of int then char
}
```

Once a named protocol type is defined, it can be used to qualify a channel or a chanend.

```
chan c3:p0;          // c3 uses a complex protocol of type p0
```

2.2.3.2.1 Protocol case

A protocol can be constructed by embedding cases to form a named case protocol.

```
protocol p1          // p1 is a named case protocol
{
    case int:char;   // p1 is a sequence of either int then char,
    case int:int;    //    or int then int;
    case void;       //    or default cases
}
```

The void case supports default processing when no receiving pattern match is wanted; it cannot be sent.

2.2.3.2.2 Protocol array types

One or more variables of the same type may be combined to form an array.

```
protocol p2
{
    case unsigned char[10];      // p2 is a sequence of 10 unsigned chars,
    case int:char[2];            //    or an int then 2 chars
}
```

2.2.3.2.3 Ambiguity

The compiler may raise a warning or an error if a protocol definition is potentially ambiguous.

```
protocol p3
{
    case int:char;
    case int;        // compiler warning, potential protocol ambiguity
}
```

Static ambiguity occurs when more than one receive case cannot be uniquely decided. A protocol is unambiguous when outputting on a channel bound to a protocol.

2.2.3.3 Sending

Data is sent along channels bound to a protocol using the usual mechanism, with a small extension for sending sequences. The compiler will check the data type sequence used occurs in the channel protocol definition.

```
void x5( chanend mobile c:p3 )
{
    int v;
    char w;
    c <: v,w;   // v,w is a sequence of int:char, a case in protocol p3
    c <: w;     // compiler error, no such case in protocol
}
```

2.2.3.4 Receiving

Data is received along channels bound to a protocol using the channel input mechanism of XC. The compiler will check all possible protocol cases are provided for. For an unnamed protocol a select statement is not required; where one is required for a named case protocol.

```
void y2( chanend d:p2 )
{
    int v;
    unsigned char w[10];
    select
    {
          case d :> w;      // w matches unsigned char[10] found in p3
          case d :> v:w;    // v:w matches with the sequence int:char[2] in p3
    }
}
```

In the second case for function y2, type int:char[2] is found in p3 and this is allowed because v:w of type int:char[10] can contain the received data correctly. If the compiler enforces strict type checking then v:w would not match int:char[2], in both array dimension and unsigned-ness.

2.2.3.4.1 Ambiguous receive

```
void y3( chanend mobile e:p3 )
{
    int v;
    char w;
    select
    {
          case e :> v,w;    // v,w is a sequence of int:char found in p3
          case e :> v;      // int is found in p3, but possibly
    }     // compiler error, select statement with ambiguous protocol
}
```

In the case of y3, the compiler cannot determine if more data is expected after the leading int and therefore cannot distinguish the case.

2.2.3.4.2 Default receive

When a case protocol is defined with a void case, a receiving function can provide a default case for un-matched entries. This allows functions to match a subset of the protocol cases, and discard all others.

```
void y4( chanend mobile e:p1 )
{
    int v;
    char w;
    select
    {
          case e :> v,w;    // v and w contain input values
          case void
          {
                // handle all other input cases (no values known)
          }
    }
}
```

Without a default case, the compiler would generate an error for un-handled protocol cases in y4.

If within a source file the scope of a protocol is fully contained and any particular case is never sent, then an omitted receive case can be cast away silently or just issued with a warning. This is allowed because the suspect case will never be communicated.

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.

3.1 Mobile processes

Mobile processes allow services to migrate, such that the program control flow is changed. 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.

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 :
            {
            }
      }
   }
```

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 i, chanend o )
   {
      int t = 1;                 // initially assume task 1
      c :> p1( o, i );           // 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( o, i );  // migrate mobile process p1
                        t = 0;      // exit loop
                  }
            }
      }
```

```
}
```

3.2 Mobile channels

Mobile channels allow data to be shared by processes such that the program data flow 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.

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. Note master initially binds mobile channel data, before it is able to send it; the compiler checks each chanend is fixed or initially bound once.

```
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:b 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.

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, dandc );
    slaveA( controlA, dandc );
    slaveB( controlB, dandc );
}

void selfsend(
  chanend A,
  chanend B,
  chanend dac0,             /* fix endpoint0 of dandc */
  chanend mobile:1b dac1    /* specify which endpoint, and initial bind */
  )
{
    A <: dac1;        // give mobile channel dandc[1] to slaveA
                      // process i/o on dandc[0] with slaveA
    dac0 :> dac1;     // receive mobile channel into dandc[1]
    B <: dac1;        // give mobile channel dandc[1] to slaveB
                      // process i/o on dandc[0] with slaveB
    dac0 :> dac1;     // receive mobile channel into dandc[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 dandc[0] and dandc[1] are identified, in this case by explicitly binding dac1 to chanend mobile:1 so that defacto the compiler will bind the remaining endpoint dandc[0] to dac0 which also happens to be fixed. The endpoint dac1 is also initially bound to *selfsend*. The processes *master*, *slaveA* and *slaveB* may reside on different processors.

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, data );
    slaveA( controlA, data );
    slaveB( controlB, data );
}

void pipeline( chanend A, chanend B, chanend D0, chanend mobile:b1 D1 )
{
    int v = 0;       // initially send server channel D[1] to slaveA
    for( ;; )
    {
        switch( v )
        {
            case 0:
            {
                A <: D1;    // give server channel D[1] to slaveA
            }
            break;
            case 1:
            {
                B <: D1;    // give server channel D[1] to slaveB
            }
            break;
        }

            // process i/o on D[0] with slave

        D0 :> v;    // identity of next slave to receive mobile channel
        D0 :> D1;   // 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 <: d;     // 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
    }
}
```

4 Semantics

4.1 Informal meaning

The semantics of PiXC can be informally presented, to better understand certain implementation issues.

4.1.1 Mobile context

The x::y mobile context for member variables and functions serves as a convenience to denote compiler semantics for mobile processes. It is not parsed as input syntax, and the programmer would not write it. The mobile context for processes does not extend beyond data and function encapsulation, constructors and destructors.

4.1.1.1 Minimal impact

Section A.3 of Programming XC [2], names and storage, helps to estimate the likely cost of extensions. PiXC should impose the minimal set of changes on XC, that do what is needed.

4.1.1.2 Non-explicit

It is not necessary to introduce (C++-style) classes or namespaces or to explicate a mobile context, and would increase what the XC compiler would have to do anew. Namespaces provide a context for symbols, and group related functions and data spread over several source files or libraries.

4.1.1.3 Issues

A mobile context is needed for the XC compiler to allocate storage space for mobile processes, which are long-lived and can migrate. A mobile context solves several implementation issues, including memory map, liveness, coherency and scalability, amongst others.

4.1.1.3.1 Memory Maps

The PiXC compiler should generate mobile process object code exactly once. During compilation global variables (or memory allocated to hold them) are fixed offset from dp (register 13). If a mobile context for member variables were omitted, then either:

- the memory maps of different processors would have to be made the same, in order for the common mobile process object code to access member variables,which is wasteful, or
- the object code would need to be compiled seperately for each processor, in order to access member variables in the different memory maps, which is also wasteful.

By using a mobile context, the compiler can offset member variables with respect to another register, say r11, per thread; or the compiler could treat member variables like far pointers, and go through a jump table; or another method could be used.

4.1.1.3.2 Liveness

A mobile process is constructed when it is received by a process. Similarly it is destructed when it is sent by a process. If a mobile context for member variables were omitted, each variable would need a 'lifetime' guard, which would need testing, on each and every access by a non-mobile process.

4.1.1.3.3 Coherency

When a process migrates from a source to a destination processor, its destination data members need to store the values already calculated in the source processor. If a mobile context for member variables were omitted, overwriting them when a mobile is received at the destination processor would appear as a corruption to other processes running on the destination processor.

And running on the new destination processor, changes written by the mobile process to global data would not be visible to other processes running on the original source processor.

The Hoare/Dijkstra multiple-update solution requires a guard on sharedglobal data; accessing a global from parallel processes is guarded in XC at compile time, so only one thread is permitted to write to a global. A mobile context gives a dynamic alternative to this solution.

4.1.1.3.4 Scalability

PiXC should provide for the future when mobility is extended to heterogenous networks, to avoid the need to change or break existing programs or language features. Using a mobile context solves mobility problems generally, which should scale up to heterogenous processor networks.

5 Case Studies

Case studies provide more in-depth examples of mobile processes and channels.

5.1 Ethernet

The Ethernet case study extends the example in chapter 6.5 of XMOS Programming XC [2]. Changes to the extended source code are deliberately kept as few as possible, for ease of drawing a direct comparison.

This case study shows how 2 mobile process transmit clients can share the single ethernet driver without conflict, that mobile processes can act like a semaphore over multiple processors, and member variable state is maintained over mobility. First the text book [2] code is given (with corrections, and including a dummy program) and second extended versions are given; one with mobile processes and the other using mobile channels.

5.1.1 Basic program

```
/*
 * Programming XC chapter 6.5
 *
 * A single thread on an XS1 device can be used to implement a full duplex
 *  100Mbps Ethernet Media Independent Interface (MII) protocol [2]. This
 *  protocol implements the data transfer signals between the link layer and
 *  physical device (PHY).
 *
 * The signals are as follows:
 *   TXCLK is a free running 25MHz clock generated by the PHY.
 *   TXEN is a data valid signal driven high by the transmitter during frame
 *    transmission.
 *   TXD carries a nibble of data per clock period from the transmitter to
 *    the PHY.
 *   RXCLK is a free running clock generated by the PHY.
 *   RXDV is a data valid signal driven high by the PHY during frame
 *    transmission.
 *   RXD carries a nibble of data per clock period from the PHY to the
 *    receiver.
 *
 * The transmitter starts by sending a preamble of nibbles of value 0x5,
 *  followed by two nibbles of values 0x5 and 0xD. The data, which must be in
 *  the range of 64 to 1500 bytes, is then transmitted, least significant bit
 *  first, followed by four bytes containing a CRC.
 */

#include <xs1.h>


#define MORE 0
#define DONE 1


  /* The port TXD performs a 32-to-4 bit serialisation of data onto its pins.
   *  It is synchronised to the 1-bit port TXCLK and uses the 1-bit port TXEN
   *  as a ready-out strobe signal that is driven high whenever data is
   *  driven.  */
static port:32 out buffered TXD = XS1_PORT_4B;
static port out TXEN = XS1_PORT_1K;
static port in TXCLK = XS1_PORT_1J;
static port in TXER = XS1_PORT_1L;

  /* The port RXD performs a 4-to-32-bit deserialisation of data from its
   *  pins.
   *  It is synchronised to the 1-bit port RXCLK and uses the 1-bit port RXDV
   *  as a ready-in strobe signal that causes data to be sampled only when
   *  the strobe is high.
  */
static port:32 in buffered RXD = XS1_PORT_4A ;
static port in RXDV = XS1_PORT_1I ;
static port in RXCLK = XS1_PORT_1H ;
static port out RXER = XS1_PORT_1G;

static clock clktx = XS1_CLKBLK_1;
static clock clkrx = XS1_CLKBLK_2;

static int txerr = 0;
static int rxerr = 0;


/* In this configuration, the processor has only to output data once every
 *  eight clock periods and does not need to explicitly output the data valid
 *  signal.
 *  miiConfigTransmit defines and configures the ports in this way.
 */
static void miiConfigTransmit(
  clock clk,
  in port TXCLK,
  buffered out port:32 TXD,
  out port TXEN,
  in port TXER
  )
{
  configure_clock_src( clk, TXCLK );   /* TXCLK provides edges for clk */
  configure_out_port( TXD, clk, 0 );   /* TXD clocked by clk, initially 0 */
  configure_out_port( TXEN, clk, 0 );  /* TXEN clocked by clk, initially 0 */

    /* TXD to drive TXEN high whenever data is output; initial output 0;
     * the strobe remains 1 for sizeof( TXD ) clocks */
  configure_out_port_strobed_master( TXD, TXEN, clk, 0 );

  TXER :> txerr;  /* initial state of TXER */

  start_clock( clk );
}

/* In this configuration, the port can sample eight values before the data
 *   must be input by the processor, and the processor does not need to
 *   explicitly wait for the data valid signal.
 * miiConfigReceive defines and configures the ports in this way.
 */
static void miiConfigReceive(
  clock clk,
  in port RXCLK,
  buffered in port:32 RXD,
  in port RXDV,
  out port RXER
  )
{
  configure_clock_src( clk, RXCLK );    /* RXCLK provides edges for clk */
  configure_in_port( RXD, clk );        /* RXD clocked by clk */
  configure_in_port( RXDV, clk );       /* RXDV clocked by clk */

    /* RXD to drive RXDV high whenever data is output;
     * the strobe remains 1 for sizeof( TXD ) clocks */
  configure_in_port_strobed_slave( RXD, RXDV, clk );

  rxerr = 0;
  RXER <: rxerr;  /* clear RXER */

  start_clock( clk );
}


/* Configure the receive and transmit ports to the PHY device
 */
void miiConfigRxTx( void )
{
  miiConfigReceive( clkrx, RXCLK, RXD, RXDV, RXER );
  miiConfigTransmit( clktx, TXCLK, TXD, TXEN, TXER );
}



/* miiTransmitFrame takes frame data from another thread and outputs it to
 *  the MII ports. For simplicity, the error signals and CRC are ignored.
 *
 * It first inputs from the channel c the size of the frame in bytes. It then
 *  outputs a 32-bit preamble to TXD, which is driven on the pins as nibbles
 *  over eight clock periods. On each iteration of the for loop, the next 32
 *  bits of data are then output to TXD for serialising onto the pins. This
 *  gives the processor enough time to get around the loop before the next
 *  block of data must be driven.
*/
void miiTransmitFrame(
  out buffered port:32 TXD,
  streaming chanend c
  )
{
  int numBytes;
  int tailBytes;
  int tailBits;
  int data;

    /* Input size of next packet */
  c :> numBytes;
  tailBytes =( numBytes / 4 );
  tailBits =( tailBytes * 8 );

    /* Output row of 0x5s followed by 0xD (little endian) */
  TXD <: 0xD5555555;

    /* Output complete 32-bit words for serialisation */
  for( int i=0; i <( numBytes - tailBytes ); i+=4 )
  {
    c :> data;    /* input data from client thread */
    TXD <: data;
  }

    /* Output partial 32-bits of data for serialisation */
  if( 0 != tailBits )
  {
    c :> data;    /* input data from client thread */

      /* Output the remaining bits of data that represent valid frame data */
    partout( TXD, tailBits, data );  /* __builtin partout( ) */
  }
}



/* miiReceiveFrame receives a single error-free frame and outputs it to
 *  another thread. For simplicity, the error signal and CRC are ignored.
 *
 * The processor waits for the last nibble of the preamble (0xD) to be
 *  sampled by the port RXD. The actual data is then received, which is in
 *  the range of 64 to 1500 bytes, least significant nibble first, followed
 *  by four bytes containing a CRC. On each iteration of the loop, it waits
 *  for either next eight nibbles of data to be sampled for input by RXD or
 *  for the data valid signal RXDV to go low.
 *
 * XS1 devices provide a single-entry buffer up to 32-bits wide and a 32-bit
 *  shift register, requiring up to 64 bits of data being input over two
 *  input statements once the data valid signal goes low.
*/
void miiReceiveFrame(
  in buffered port:32 RXD,
  in port RXDV,
  streaming chanend c
  )
{
  int dataValid;
  int data;
  int tail;          /* stores bit-count in last received part-word */

    /* Wait for start of frame */
  RXD when pinseq( 0xD ):> void;
  RXDV :> dataValid;

    /* Receive frame data /crc */
  do
  {
    select
    {
      case RXD :> data :
      {
          /* Input next 32 bits of data */
        c <: MORE;   /* indicate to receiving thread, more data follows */
        c <: data;   /* send next data word to receiving thread */
      }
      break;

      case RXDV when pinseq( 0 ):> dataValid :
      {
          /* An effect of using a port's serialisation and strobing
           *  capabilities together is that the ready-in signal may go low
           *  before a full transfer width's worth of data is received.
           * Input any bits remaining in port.
           */
        tail = endin( RXD );  /* __builtin endin( ) */

          /* send remaining data words to receiving thread */
        for( int byte =( tail >> 3 ); byte > 0; byte -= sizeof( int ))
        {
          RXD :> data;
          c <: MORE;
          c <: data;
        }

          /* indicate to receiving thread, end of data and how many bits are
           *  valid in the last word */
        c <: DONE ;
        c <: tail >> 3;
      }
      break ;
    }
  }
  while( 0 != dataValid );
}
```

```
/* Dummy program follows *************************************
 */

/* txClient dummy
 */
void txClient( chanend streaming c )
{
  static const int dataOut[ ]={ 'H', 'e', 'l', 'l', 'o' };
  static int runCount = 0;

  {
    c <:( sizeof( dataOut )/ sizeof( dataOut[ 0 ]));
    for( int i=0; i < sizeof( dataOut ); i++ )
    {
      c <: dataOut[ i ];
    }

    runCount++;
  }
}

/* rxClient dummy
 */
void rxClient( chanend streaming c )
{
  static int buf[ 1500 ];
  int todo;
  int validBits;

  for( int i=0; i < sizeof( buf ); i++ )
  {
    c :> todo;
    switch( todo )
    {
      case MORE :
      {
        c :> buf[ i ];
      }
      break;
```

```
        case DONE :
        {
          c :> validBits;
          i = sizeof( buf );  /* exit loop */
        }
        break;

        default :
        {
            /* protocol breakdown */
          i = sizeof( buf );  /* exit loop */
        }
        break;
      }
    }

    if( DONE != todo )
    {
      /* protocol failure, or ran out of buffer space */
    }
  }

  /* Run the drivers and dummy client threads in parallel
   */
  int main( void )
  {
    chan streaming txc;
    chan streaming rxc;


    miiConfigRxTx( );

    par
    {
      while( ! txerr )
      {
        par
        {
          miiTransmitFrame( TXD, txc );
          txClient( txc );
        }
      }
      while( ! rxerr )
      {
        par
        {
          miiReceiveFrame( RXD, RXDV, rxc );
          rxClient( rxc );
        }
      }
    }

    return( txerr + rxerr );
  }
```

5.1.2 Mobile processes

```
  /*
   * PiXC version of Programming XC chapter 6.5 using mobile processes
   *
   * ... rest of header as  5.1.1
   */

  #include <xs1.h>


  #define MORE 0
  #define DONE 1


    /* Because mobile processes names are bound on receiving, we use a simple
     *  protocol enum to allow a random selection for receipt.
     */
  typedef enum
  {
    CLIENTA = 0,
    CLIENTB = 1
  } CLIENT_ID;


  ... port definitions as  5.1.1
```

```
... clock definitions as  5.1.1

static int txerr = 0;
static int rxerr = 0;

... miiConfigTransmit as  5.1.1
... miiConfigReceive as  5.1.1
... miiConfigRxTx as  5.1.1
... miiTransmitFrame as  5.1.1
... miiReceiveFrame as  5.1.1


/* Dummy program follows *************************************
 */

/* txClient dummy
 *
 * Chanends txaIn and txaOut are fixed (or bound)
 */
mobile void txClientA( chanend txaOut, chanend txaIn )
{
  static const int dataOut[ ]={ 'H', 'e', 'l', 'l', 'o', ' ', 'A' };
  static int runCount = 0;  /* mobile process member variable */

  {
    txaOut <: sizeof( dataOut );
    for( int i=0; i < sizeof( dataOut ); i++ )
    {
      txaOut <: dataOut[ i ];
    }

    runCount++;  /* member variable maintained over mobility */
  }
}

/* txClient dummy
 *
 * Chanends txbIn and txbOut are fixed (or bound)
 */
mobile void txClientB( chanend txbOut, chanend txbIn )
```

```
{
  static const int dataOut[ ]={ 'H', 'e', 'l', 'l', 'o', ' ', 'B' };
  static int runCount = 0;  /* mobile process member variable */

  {
    txbOut <: sizeof( dataOut );
    for( int i=0; i < sizeof( dataOut ); i++ )
    {
      txbOut <: dataOut[ i ];
    }

    runCount++;  /* member variable maintained over mobility */
  }
}

/* txClient scheduler.
 *
 *  Randomly selects which transmit client is given access to the ethernet
 *   driver.
 *   Send the chosen mobile process (context) to main thread.
 *   Wait for its return.
 */
void txSched( chanend ch, mobile txClientA, mobile txClientB )
{
  timer t0;  /* timer, used as a random choice */
  int t;

  for( ;; )
  {
    t0 :> t;
    if( 0x1 & t )
    {
      ch <: CLIENTA;
      ch <: txClientA;  /* send mobile process (and lose its context) */
      ch :> txClientA;  /* receive mobile process bound name (and context) */
    }
    else
    {
      ch <: CLIENTB;
      ch <: txClientB;  /* send mobile process (and lose its context) */
```

```
        ch :> txClientB;  /* receive mobile process bound name (and context) */
      }
    }
  }

  /* rxClient dummy
   */
  void rxClient( chanend streaming c )
  {
    static int buf[ 1500 ];
    int todo;
    int validBits;

    for( int i=0; i < sizeof( buf ); i++ )
    {
      c :> todo;
      switch( todo )
      {
        case MORE :
        {
          c :> buf[ i ];
        }
        break;

        case DONE :
        {
          c :> validBits;
          i = sizeof( buf );  /* exit loop */
        }
        break;

        default :
        {
            /* protocol breakdown */
          i = sizeof( buf );  /* exit loop */
        }
        break;
      }
    }
    if( DONE != todo )
    {
      /* protocol failure, or ran out of buffer space */
    }
  }

  /* Run the drivers and dummy client threads in parallel
   */
  int main( void )
  {
    chan streaming rxc;
    chan sc;
    chan txaIn;
    chan txaOut;
    chan txbIn;
    chan txbOut;
    CLIENT_ID id;


    miiConfigRxTx( );

    par
    {
      txSched( sc, txclientA, txClientB );  /* initially bind  txclientA,  txclientB */

      while( ! txerr )
      {
          /* Receiving a mobile process (over a channel) is done into a bound
           *  name and not a free name. Therefore, we first receive and switch
           *  on a client_id.
           */
        sc :> id;
        switch( id )
        {
          case CLIENTA :
          {
              /* receive mobile process bound name (and its context),
               * bind its input and output channel-endpoints to a local
               * channel, and it is run as a parallel thread
               */
            sc :> txClientA( txaOut, txaIn );
            miiTransmitFrame( TXD, txaOut );
              /* send mobile process bound name (and lose its context),
               * unbind its input and output channel-endpoints from local
               * channels, and terminate its thread
               */
            sc <: txClientA( txaOut, txaIn );
          }
          break;

          case CLIENTB :
          {
              /* receive mobile process bound name (and its context),
               * bind its input and output channel-endpoints to a local
               * channel, and it is run as a parallel thread
               */
            sc :> txClientB( txbOut, txbIn );
            miiTransmitFrame( TXD, txbOut );
              /* send mobile process bound name (and lose its context),
               * unbind its input and output channel-endpoints from local
               * channels, and terminate its thread
               */
            sc <: txClientB( txbOut, txbIn );
          }
          break;
        }
      }

      while( ! rxerr )
      {
        par
        {
          miiReceiveFrame( RXD, RXDV, rxc );
          rxClient( rxc );
        }
      }
    }

    return( txerr + rxerr );
  }
```

5.1.3 Mobile channels

```
/*
 * PiXC version of Programming XC chapter 6.5 with mobile channels
 *
 * ... rest of header as  5.1.1
 */

#include <xs1.h>


#define MORE 0
#define DONE 1

#define mobile  /* _ONLY_ to produce more insightful PiXC compilation reports  */


  /* Because mobile processes names are bound on receiving, we use a simple
   *  protocol enum to allow a random selection for receipt.
   */
typedef enum
{
  CLIENTA = 0,
  CLIENTB = 1
} CLIENT_ID;


... port definitions as  5.1.1
... clock definitions as  5.1.1

static int txerr = 0;
static int rxerr = 0;

... miiConfigTransmit as  5.1.1
... miiConfigReceive as  5.1.1
... miiConfigRxTx as  5.1.1
... miiTransmitFrame as  5.1.1
... miiReceiveFrame as  5.1.1


/* Dummy program follows *************************************
```

```
 */

/* txClient dummy with fixed channels
 */
void txClientA( chanend txaOut, chanend txaIn )
{
  static const int dataOut[ ]={ 'H', 'e', 'l', 'l', 'o', ' ', 'A' };
  static int runCount = 0;  /* mobile process member variable */

  {
    txaOut <: sizeof( dataOut );
    for( int i=0; i < sizeof( dataOut ); i++ )
    {
      txaOut <: dataOut[ i ];
    }

    runCount++;
  }
}

/* txClient dummy with fixed channels
 */
void txClientB( chanend txbOut, chanend txbIn )
{
  static const int dataOut[ ]={ 'H', 'e', 'l', 'l', 'o', ' ', 'B' };
  static int runCount = 0;  /* mobile process member variable */

  {
    txbOut <: sizeof( dataOut );
    for( int i=0; i < sizeof( dataOut ); i++ )
    {
      txbOut <: dataOut[ i ];
    }

    runCount++;
  }
}

/* txClient scheduler.
 *
 *  Randomly selects which transmit client is given access to the ethernet
 *   driver .
 *   Initially bound to txaIn, txaOut, txbIn, txbOut.
 *   Send the chosen mobile process (context) to main thread.
 *   Wait for its return.
 */
void txSched(
  chanend ch,
  chanend mobile:b txaIn,   /* initially bind txaIn endpoint */
  chanend mobile:b txaOut,  /* initially bind txaOut endpoint */
  chanend mobile:b txbIn,   /* initially bind txbIn endpoint */
  chanend mobile:b txbOut   /* initially bind txbOut endpoint */
  )
{
  timer t0;  /* timer, used as a random choice */
  int t;

  for( ;; )
  {
    t0 :> t;
    if( 0x1 & t )
    {
      ch <: CLIENTA;
      ch <: txaIn;      /* send mobile channel (and lose chanend) */
      ch <: txaOut;     /* send mobile channel (and lose chanend) */
      ch :> txaIn;      /* receive mobile channel (and collect chanend) */
      ch :> txaOut;     /* receive mobile channel (and collect chanend) */
    }
    else
    {
      ch <: CLIENTB;
      ch <: txbIn;      /* send mobile channel (and lose chanend) */
      ch <: txbOut;     /* send mobile channel (and lose chanend) */
      ch :> txbIn;      /* receive mobile channel (and collect chanend) */
      ch :> txbOut;     /* receive mobile channel (and collect chanend) */
    }
  }
}

/* rxClient dummy
```

```
 */
void rxClient( chanend streaming c )
{
  static int buf[ 1500 ];
  int todo;
  int validBits;

  for( int i=0; i < sizeof( buf ); i++ )
  {
    c :> todo;
    switch( todo )
    {
      case MORE :
      {
        c :> buf[ i ];
      }
      break;

      case DONE :
      {
        c :> validBits;
        i = sizeof( buf );  /* exit loop */
      }
      break;

      default :
      {
          /* protocol breakdown */
        i = sizeof( buf );  /* exit loop */
      }
      break;
    }
  }

  if( DONE != todo )
  {
    /* protocol failure, or ran out of buffer space */
  }
}
```

```
/* Run the drivers and dummy mobile process client threads in parallel
 */
int main( void )
{
  chan sc;
  chan streaming rxc;
  CLIENT_ID id;
  chan mobile txaIn;  /* mobile chanend un-bound */
  chan mobile txaOut; /* mobile chanend un-bound */
  chan mobile txbIn;  /* mobile chanend un-bound */
  chan mobile txbOut; /* mobile chanend un-bound */


  miiConfigRxTx( );

  par
  {
    txClientA( txaOut, txaIn );  /* mobile chanend fixed in definition */

    txClientB( txbOut, txbIn );  /* mobile chanend fixed in definition */

      /* mobile chanends initially bound in definition of txSched */
    txSched( sc, txaIn, txaOut, txbIn, txbOut );

    while( ! txerr )
    {
        /* Receiving a mobile process (over a channel) is done into a bound
         *  name and not a free name. Therefore, we first receive and switch
         *  on a client_id.
         */
      sc :> id;
      switch( id )
      {
        case CLIENTA :
        {
          sc :> txaIn;      /* receive mobile chanend (and bind context) */
          sc :> txaOut;     /* receive mobile chanend (and bind context) */

          miiTransmitFrame( TXD, txaOut );
          sc <: txaIn;      /* send mobile chanend (and unbind context) */
          sc <: txaOut;     /* send mobile chanend (and unbind context) */
        }
        break;

        case CLIENTB :
        {
          sc :> txbIn;      /* receive mobile chanend (and bind context) */
          sc :> txbOut;     /* receive mobile chanend (and bind context) */

          miiTransmitFrame( TXD, txbOut );

          sc <: txbIn;      /* send mobile chanend (and unbind context) */
          sc <: txbOut;     /* send mobile chanend (and unbind context) */
        }
        break;
      }
    }

    while( ! rxerr )
    {
      par
      {
        miiReceiveFrame( RXD, RXDV, rxc );
        rxClient( rxc );
      }
    }
  }

  return( txerr + rxerr );
}
```

6 Extensions

The mobile process extensions could be further extended in several ways.

6.1 Mobile context

The file-scope static declaration which serves as the mobile context 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].

Or mobile contexts could be incorporated with namespaces. Alternatively and perhaps less pleasingly (as the base 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].

6.2 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.

6.3 Process 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]( chanend out, chanend in );
```

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

6.4 Channel arrays

With the introduction of protocols, channel arrays become meaningful. It is not yet clear exactly how they may prove useful, however.

6.5 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.

6.6 Mobile algorithms

As a general purpose programming language, PiXC can be used to express mobile algorithms; the design patterns given are such an example. The dynamic communication of mobile channels and mobile processes through channels enables the function and data control flow to evolve in response to run-time events. This opens up a sub-branch of computer science research.

7 Conclusions

PiXC is a proposal to extend XC with mobility and to formalise (extensions to) a compiler for PiXC that targets 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.