CONCURRENT PROGRAMMING IN ML

Norman Ramsey

CS-TR-262-90

April 1990

Concurrent programming in ML

Norman Ramsey*
Department of Computer Science, Princeton University
35 Olden Street
Princeton, New Jersey 08544

April 26, 1990

Abstract

I have added concurrency to the functional language Standard ML. The model of concurrency is derived from CSP, but processes do not communicate directly; they interact by communicating over channels. Also, processes and channels can be created dynamically.

Three features of Standard ML—polymorphism, garbage collection, and modules—work exceptionally well with concurrency. Polymorphic functions that combine the concurrent primitives can be used in many programs. One can write processes that do not explicitly provide for their own termination; the garbage collector reclaims them when they can no longer communicate. Finally, one can use the Standard ML modules system to make small adjustments in the meanings of the primitives.

The implementation takes only 220 lines of Standard ML. It uses *call with current continuation*, `callcc`, to simulate concurrent execution on a sequential machine. `callcc` is implemented efficiently by the Standard ML of New Jersey compiler, which uses no runtime stack.

Introduction

I have added concurrent primitives to the functional language Standard ML. My model of concurrency is derived from CSP [Hoa78], but processes do not communicate directly; they interact by communicating over channels. Also, processes and channels can be created dynamically. Writing concurrent programs in ML has three advantages. One can write polymorphic functions that combine the concurrent primitives in generally useful ways; because the functions are polymorphic they can be used in many programs. One can write processes that do not explicitly provide for their own termination; the garbage collector reclaims them when they can no longer communicate. Finally, one can use the Standard ML modules system to make small adjustments in the meanings of the primitives.

The primitives are standard. **`begin`** creates a process; **`mkchan`** creates a channel. **`send`** sends a value on a channel; **`receive`** receives a value on a channel. **`select`** offers a choice between several

*Supported in part by a Fannie and John Hertz Foundation fellowship.

communications; these "potential communications" are first-class values and are created by the `SEND` and `RECEIVE` functions. `send` has a side effect—communication—while `SEND` returns a value; `select`ing the value returned by `SEND` has a side effect. All communication is synchronous.

These primitives are equivalent to those of the Pegasus Meta-Language [Rep88] (although developed independently); Below I show how to transform them into the Pegasus primitives. I have implemented two new primitives, `wait` and `WAIT`, which wait for input from or output to files to be possible. Using `wait` and `WAIT`, I have implemented functions that make it possible for users to treat file I/O just like communication on channels.

Using these primitives, it is easy to structure concurrent programs as collections of simple processes. Useful abstractions can be implemented as collections of channels managed by one or more processes. Some of the most useful are "plumbing"—functions that operate on small groups of processes and channels. Because the plumbing is polymorphic, it can be used in many programs.

Here is some of the plumbing I have found useful. `source(c,value)` supplies `value` endlessly on channel `c`. `sink c` accepts values endlessly from channel `c`. `copy(c1,c2)` connects `c1`'s output to `c2`'s input. `buffered_copy(c1,c2)` forms an *asynchronous* connection between `c1` and `c2`. `broadcast [c1,c2,...] value` sends `value` on each channel on the list `[c1,c2,...]`. `collect [c1,c2,....]` receives a value from each channel on the list, then returns the list of values. `fanout(src,[c1,c2,...])` supplies every channel on the list `[c1,c2,...]` with `src`'s output. I have implemented two versions of `fanout`, one in which all readers read at the same rate and one in which they can read at different rates.

I have used plumbing in two applications: a simulation of sequential circuits and a multiuser version of a "talk" program. I have also used it to duplicate an implementation of operations on power series [McI90].

It is sometimes convenient to make small changes in the semantics of the primitives. For example, one might want to restrict channels so that values could be sent in a single direction only; attempts to send values in the opposite direction would cause type errors, which would be detected at compile time. By using the Standard ML modules system, I can easily implement such changes without changing the implementation of the primitives; I give several examples below.

My implementation of the primitives requires 220 lines of Standard ML. It uses *call with current continuation* (`callcc`) to simulate concurrent execution on a sequential machine, essentially as described in [Wan80]. `callcc` can be implemented efficiently because SML-NJ uses no runtime stack [AJ89].

This paper shows the implementation of plumbing in terms of the concurrent primitives, and it sketches applications in which I have used the plumbing. It also gives implementations of different semantic changes to the primitives; the new semantics are derived from the old using the ML modules system. It gives brief descriptions of applications and implementation.

The concurrent primitives

The creation and communication primitives are most basic. With their ML types, the creation primitives are

```
val begin : ('a -> 'b) -> 'a -> unit
val mkchan : unit -> '1a chan
```

`begin f a`[1] starts a process that applies `f` to `a`; `mkchan()` returns a fresh channel.[2] There is no need for destruction primitives because the vanilla garbage collector reclaims unreachable processes and channels. The communication primitives are

```
val send : 'a chan -> 'a -> unit
val receive : 'a chan -> 'a
```

`send c x` sends value `x` on channel `c`; `receive c` receives a value on channel `c` and returns it. Communication is synchronous: a `send` cannot take place until another process executes a matching `receive`, and vice versa.

The alternative function, `select`, is similar to that of CSP, but the alternatives are values, not language constructs. I have defined the type `'a communication`, the *potential communication*, to represent an alternative whose choice results in a value of type `'a`. The two basic potential communications are

```
val SEND : 'a chan -> 'a -> unit communication
val RECEIVE : 'a chan -> 'a communication
```

Evaluating `SEND c x` has no side effects by itself; values are not actually communicated until a potential communication is "actualized." This can take place when the value returned by `SEND c x` appears in a list of alternatives presented to the selection function:

```
val select : 'a communication list -> 'a
```

`select` examines a list of potential communications, waits until at least one of the potential communications can be actualized (which happens when some other process attempts a matching communication), then picks one such and actualizes it.

One can specify that additional computation should follow the actualization of a potential communication by using the `-->` operator. If `P` is a potential communication of type `'a communication` and `f` is a function of type `'a -> 'b`, then `P --> f` is a potential communication of type `'b communication`. `P --> f` is actualized by first actualizing `P`, then applying `f` to the result. (One can think of `f` as a continuation, or one can think of `P` as "guarding" the execution of `f`.) The type of `-->` is

```
val --> : 'a communication * ('a->'b) -> 'b communication
```

`wait` and `WAIT` interact with the I/O facilities provided by the operating system. Processes use `wait` or `WAIT` to guarantee that they can attempt I/O without blocking. If a process issued a system call that blocked, it could block all processes. A process can wait on one of three alternatives:

`wait (On_Read d)` waits for input to become ready on file descriptor `d`.

[1] The curried forms for `begin`, `send`, and `SEND` seem more convenient than the uncurried forms.

[2] The `'1a` indicates that the type of the channel must be a monotype; a similar restriction applies to reference types. Without such restrictions, one could use channels (or references) to subvert the type system.

`wait (On_Write d)`	waits for it to be possible to write on file descriptor `d`.
`wait (On_Exn d)`	waits until there is an exceptional condition pending on file descriptor `d`.

The relevant types are:

```
datatype wait = On_Read of int | On_Write of int | On_Exn of int
val wait : wait -> unit
val WAIT : wait -> unit communication
```

The best examples of the use of `wait` are in the routines that create channels that are connected to file descriptors. Here is a simplified version of a function that makes reading from a file look like receiving from a string channel. `dinchannel d` returns a channel; a process receiving from that channel will get the bytes read from file descriptor `d`.

```
fun dinchannel d =
    let val c = mkchan()
        val buffer = ByteArray.array(1024,0)   (* buffer for read(2) *)
        fun din(d,c) =
            let val rval = (wait (On_Read d); read(d,buf,bufsize))
                val data = (* extract only meaningful bytes *)
                        if rval<0 then raise IO_error
                        else substring(buf,0,rval)
            in  send c data;
                if rval > 0 then din(d,c) else ()  (*EOF*)
            end
    in  begin din (d,c);
        c
    end
```

The key fragment is

```
val rval = (wait (On_Read d); read(d,buf,bufsize))
```

which guarantees that `read`, a Unix system call, will not block.

Plumbing

When processes are connected by channels to form large graphs, `source` and `sink` can fix or ignore values at boundaries:

```
fun source (c,value) = (send c value; source (c,value))
fun sink c = (receive c; sink c)
```

`source (c,value)` supplies `value` endlessly on channel `c`. `sink c` accepts values endlessly from channel `c`. Because of garbage collection, these processes need not include code that provides for termination. Most processes can loop forever provided they communicate infinitely often; since communication is synchronous, such processes can become blocked when trying to communicate, and the garbage collector reclaims blocked processes when it reclaims the channels on which they are blocked. A channel is reclaimed when no unblocked process can reach it. In Occam 2, programmers must exercise considerable ingenuity to get groups of processes to terminate gracefully [Bur88]; in ML, programmers can just let processes block.

To connect a reader and a writer one can use `copy`:

```
fun copy (in',out) = (send out (receive in'); copy(in',out))
```

(`in'` is used instead of `in` because `in` is a reserved word.) Because `copy` buffers one value, it creates one unit of delay in a pipeline. "Demand channels," which avoid delays, are explained below.

Imagine connecting one producer to many consumers. Assuming `broadcast` broadcasts a value to a list of channels, here is a `fanout` process:

```
fun fanout (in',outlist) =
    (broadcast outlist (receive in');
     fanout (in',outlist))
```

`fanout` is like `copy` except that `broadcast` replaces `send` and `outlist` is a list of channels instead of a single channel.

One can write functions that, given either the input or output end of `fanout`, create the opposite end. My strategy, which I have found generally useful, is to create a data structure composed of channels, start a process to manage the channels, then return the data structure. `replicas` makes `n` copies of the output from channel `c`, returning a list of channels:

```
fun replicas n c =                      (* given c, create l *)
    let fun mkchans 0 = []
          | mkchans n = mkchan()::mkchans(n-1)
        val chans = mkchans n           (* create a list of n channels *)
    in  begin fanout (c,chans);
        chans
    end
```

Given a list `l` of channels to which one wants to send identical streams of input, one can send a single stream to a "driver" channel `c`, returned by:

```
fun driver l =                          (* given l, create c *)
    let val c = mkchan()
    in  begin fanout (c,l);
        c
    end
```

To broadcast a value to a list of channels, one can send the value to one of the channels, then broadcast it to all the others. But one should avoid insisting that any particular channel receive the first message. By using `select` and `SEND`, `broadcast` can send the first message to any of the channels. n potential communications are needed to broadcast to n channels. Each is a function of a pair of the form (one channel, all the others). The function `openlist` produces such pairs. For example, `openlist` applied to a list of integers yields

```
openlist [1,2,3,...,n] =
  [ ( 1, [2,3,4,...,n] ),
    ( 2, [1,3,4,...,n] ),
        ...
    ( n, [1,2,3,...,n-1] )
  ]
```

`openlist` is polymorphic (type `'a list -> ('a * 'a list) list`).

`openlist`'s implementation could be given in an appendix.

Given `openlist`, here is `broadcast`:

```
fun broadcast [] msg = ()
  | broadcast channels msg = select
      (map  (fn(one,others)=>
               SEND one msg-->
                 fn()=>broadcast others msg)
            (openlist channels))
```

Broadcasting to an empty list of channels does nothing. To broadcast to a nonempty list, open the list with `openlist`, then use `map` to apply the function

```
fn(one,others)=> SEND one msg--> fn()=>broadcast others msg
```

to each of the resulting pairs. `map` returns a list of potential communications; each one, if actualized, would first send `msg` to the channel `one`, then apply `fn()=>broadcast others msg` to the result, broadcasting `msg` to all the others. `select` chooses a potential communication and actualizes it, which causes `broadcast` to be called recursively.[3] Because only one of the n lists computed by `openlist` is actually used, it is possible to write a slightly different version of `broadcast` that does not compute all the lists in advance, costing $O(n^2)$ instead of $O(n^3)$. `broadcast` can be implemented in $O(n)$ time by creating a tree of $n-1$ processes.

`broadcast` distributes values to neighbors; a similar function provides the inverse operation, collecting values from neighbors:

```
fun collectu [] = []
  | collectu inputs = select
      (map (fn(one,others)=>
```

[3] The recursive call to `broadcast` is placed inside a lambda-abstraction (`fn()=>...`) so that the program won't make the recursive call until some potential communication is actualized. In ML, the standard way to defer the evaluation of an expression `e` is to write "`fn()=>e`". Evaluation of `e` is deferred until the resulting function is applied to `()`.

```
                RECEIVE one-->
                    fn x=>x::collectu others)
            (openlist inputs))
```

collectu just like **broadcast** except for the difference in the potential communications. First a value is received from some channel, then values are collected from all the other channels, finally the first value received is put at the head of the list of collected values. In this version, the order of the collected values need not correspond to the order of channels in the list; order-preserving implementations can sort the values after receipt. Both **broadcast** and **collectu** have especially simple implementations because potential communications are not static constructs but are values that can be put in lists and manipulated.

broadcast and **fanout** both insist that the reader and writers all communicate at exactly the same rate. It is often useful to decouple processes so that communication can proceed at different rates in different processes. A producer can run arbitrarily far ahead of a consumer if the two are connected with a buffering process. This implementation maintains the buffer in a list, with the oldest value at the head. (A more efficient implementation would use a different data structure for the buffer.)

```
fun buffered_copy (in',out) =
    let fun holding [] = holding [receive in']
          | holding (buffer as h::t) =
              select [
                SEND out h  --> fn()=>holding t,
                RECEIVE in' --> fn new=>holding (buffer@[new])
              ]
    in  holding []
    end
```

Using **buffered_copy**, here is a function that returns a pair of channels over which communication is asynchronous:

```
fun buffered_pair() =
    let val pair = (mkchan(),mkchan())
    in  begin buffered_copy pair;
        pair
    end
```

It is useful to have a fanout process in which the readers are buffered. Such a process can connect a producer to many consumers while allowing the consumers to read at different rates. The producer may write as fast as the fastest consumer reads.

This implementation divides consumers into two groups. The consumers in the **fast** group are waiting for the next item from the producer. The consumers in the **slow** group have not yet read old items previously produced; a queue of such items is kept for each slow consumer. All the consumers in the **fast** group move to the **slow** group whenever a new value is received from the producer.

Consumers in the `slow` group move to the fast group when they have read all the items in their queues. A value can be received from the producer if and only if there is at least one `fast` consumer waiting to read it.

```
local
    fun holding(writer, fast, slow) = (* assumes fast@slow <> nil *)
        let val slowreads = map (fn((rdr,h::t),others)=>
                    SEND rdr h --> fn()=> case t of
                                nil => holding(writer,rdr::fast,others)
                              | _ => holding(writer,fast,(rdr,t)::others))
                  (openlist slow)
        in  case fast of
                nil => select slowreads
              | _ => select (
                       (RECEIVE writer --> fn x=>
                                 holding(writer,nil,
                                       map (fn c =>(c,[x])) fast @
                                       map (fn (c,l) => (c,l@[x])) slow))
                       ::slowreads)
        end
in
    fun ufanout (writer,nil) = sink writer
      | ufanout (writer,readers) = holding(writer,readers,nil)
end
```

Applications

Sequential circuits

It is easy to use channels and processes to simulate sequential circuits. I represent wires as channels and circuitry as processes. Latches and combinational logic are the two basic circuit elements. A latch holds a value, and has an input, an output, and a clock line:

```
fun latch (contents,input,output,clock) =
      (receive clock; send output contents;
       latch(receive input,input,output,clock)
```

A circuit simulation will block if a user creates a feedback loop that is not broken by any latch. Combinational logic has many inputs and one output; it computes some Boolean function `f`.

```
fun combine (inputs,output,f) =
      (send output (f (collect inputs));
       combine(inputs,output,f))
```

As an example, here is the combinational logic for an adder. The inputs are duplicated because I must define a separate process for each output bit:

```
    fun adder(a,b,carryin,sum,carryout) =
let val duplicate = replicas 2
    val [a1, a2] = duplicate a
    val [b1, b2] = duplicate b
    val [carry1, carry2] = duplicate carryin
    fun xor (x, y) = if x then not y else y
in  begin combine([a1,b1,carry1],sum,
  fn[a,b,c]=>xor(a,xor(b,c)));
    combine([a2,b2,carry2],carryout,
  fn[a,b,c]=>(a andalso b) orelse
     (b andalso c) orelse (c andalso a))
end
```

I have used `latch` and `combine` to write, in about 70 lines, a simulation of a Configurable Logic Block, the basic unit of the Xilinx programmable gate array [Xil88]. This simulation could be used to test and debug a circuit configuration before downloading it to the gate array.

Conference talk

Two people can use the Unix `talk` program to converse using the computer. Each user's screen is divided into two regions; characters typed locally appear in the upper region, and characters typed by the other user appear in the lower region. I have written a program that extends this facility to more than two users.

The implementation abstracts a terminal screen, or "tube," as a channel to which one can send certain kinds of messages: move cursor, print characters, and so on. I have written three implementations of this abstraction. The *hardware tube* is based on a file connected to a Unix `tty`; the `termcap` database is used to determine cursor motion. The *virtual tube* is a repositionable, resizable tube that occupies a window on the screen of another tube. The *named tube* is a virtual tube with a title bar. The three together require 180 lines of Standard ML.

The conference talk program presents to each user a screen that is tiled with named tubes (the title bars identify the participating users). The process that manages this screen keeps track of the participants and communicates with a central server. When a user wants to quit, this process sends "delete me" to the server. The server can notify this process that participants have been added or deleted. Data (characters typed by users) travels to and from the server.

The server distributes incoming data to all the participants and notifies current participants of add and delete requests. Add requests come from a "listener" process that spends most of its time idle, waiting for a connection to a Unix socket (the BSD Unix interprocess communication mechanism). When a connection is made, the server is notified, and the server spawns a new process that interacts with the new user until the participant data structure can be created.

The server, listener, and screen manager processes form a single 200-line ML program that runs as one Unix process. Each user types characters to a separate Unix process, running a 130-line C program that establishes the socket connection, queries the `termcap` data base, and then copies characters in both directions.

Writing the talk server in ML was easier than doing it in C using Unix interprocess communication (sockets). Sockets are asynchronous, not synchronous, and it is not easy to preserve both message ordering and message boundaries. In ML one can represent different kinds of messages using a disjoint union type; the compiler handles type checking and the details of the representation. Using sockets one can send only sequences of bytes; a programmer would either have to devise a representation of messages as byte sequences or pick some combination of `struct` and `union` to represent disjoint unions. In the latter case, the number of bytes constituting a message would have to be limited, since it is not meaningful to send a pointer (e.g. `char *`) over a socket.

The ML implementation is not perfect; its most annoying defect is that the program appears to halt for a few seconds every time it performs a major garbage collection [App89].

Modifying the primitives

I have used ML functors to map the primitives described above onto slightly different sets of primitives. I have restricted channels so that messages flow in one direction only; `mkchan()` then returns a pair: the input and output sides of the channel. I have restricted channels so that it is impossible to send a message until it is asked for; this is an abstraction of McIlroy's *demand channels* [McI90]. I have mapped my primitives to the primitives used in the Pegasus Meta-Language [Rep88]. Finally, I have added Boolean guards to potential communications, so that `select` statements can use combinations of Booleans and communications after the manner of CSP [Hoa78]. These four modifications are orthogonal; their implementations, which range in size from 6 to 27 lines, are given below. The brevity of these implementations suggests that the primitives shown above have wide applicability.

Unidirectional channels

Making channels unidirectional requires only one change to the types of the primitives; `mkchan` must be replaced with a function that returns a pair—the sending and receiving ends of a single channel. The unidirectional channels have the signature (I have omitted the signatures of several unchanged primitives):

```
signature UNICHAN = sig
    type 'a sourcechan                          (* receiving end of channel *)
    type 'a sinkchan                            (* sending end of channel *)
    val send : 'a sinkchan -> 'a -> unit        (* send on a channel *)
    val receive : 'a sourcechan -> 'a           (* receive from a channel *)
    val begin : ('a -> 'b) -> 'a -> unit
    val mkchans : unit -> '1a sinkchan * '1a sourcechan
```

```
                                        (* create fresh channel pair *)
    type 'b communication
    val SEND : 'a sinkchan -> 'a -> unit communication
    val RECEIVE : 'a sourcechan -> 'a communication
    ...
end
```

The implementation is trivial; the only thing that needs changing is `mkchan`:

```
functor UniChan (C:CHAN) : UNICHAN = struct
    open C
    type 'a sourcechan = 'a chan
    type 'a sinkchan = 'a chan
    fun mkchans() = let val c = mkchan() in (c,c) end
end
```

To hide the underlying representation (thereby preventing users from seeing that `'a sourcechan` and `'a sinkchan` are the same type), I use `abstraction`:

```
abstraction UC: UNICHAN = UniChan(SomeChan)
```

Demand channels

In a process like `copy`, above, a value must be accepted from the producer (and buffered) before it can be sent to the consumer. When many such processes are combined, it can be difficult to understand the timing properties the resulting, because each process may introduce a unit of delay. Important properties of an abstraction should be independent of the number of processes used to implement the abstraction, but this may be impossible where timing is important.

Another problem with ordinary channels occurs when it is expensive to compute a value to be sent on a channel. When the ultimate consumer of information has finished receiving data, unnecessary work has been done in computing the intermediate values that fill the pipeline.

One can solve these problems by refusing to accept a value from a producer until a consumer has made a request for it. This has been implemented by making each channel a pair, half for requests and half for data [McI90], but I have implemented it more abstractly by providing three basic communication primitives instead of two. These are: "wait for a value to be requested," "satisfy a request," and "get a value." Requests can be satisfied at most once, and it's impossible to send a value before one is requested.

Here is the signature:

```
signature DCHAN = sig
    type 'a dchan
    type 'a request
    type 'b communication
    val mkchan : unit -> '1a dchan
```

```
    val get : '1a dchan -> '1a              (* get data on demand channel *)
    val dwait : 'a dchan -> 'a request      (* wait for a request *)
    val satisfy : 'a request -> 'a -> unit (* send val (after request recvd)*)
    val put : 'a dchan -> 'a -> unit        (* wait for request; send val*)
    val DWAIT : 'a dchan -> 'a request communication
    val SATISFY : 'a request -> 'a -> unit communication
    val GET : '1a dchan -> '1a communication
    val PUT : 'a dchan -> 'a -> unit communication
    ...
end
```

put, which complements get, is just a convenience; it is defined by

```
fun put dc x = (satisfy (dwait dc) x)
```

My implementation of demand channels is straightforward. Requests are channels, and demand channels are channels of requests:

```
functor DemandChannels (C:CHAN) : DCHAN = struct
    open C
    type 'a request = 'a chan
    type 'a dchan = 'a request chan

    fun get dc =
        let val c = mkchan()
        in  send dc c; receive c
        end
    fun dwait dc = receive dc
    val satisfy = send
    fun put dc x = (satisfy (dwait dc) x)
    fun DWAIT dc = RECEIVE dc
    val SATISFY = SEND
    fun GET dc =
        let val c = mkchan()
        in  SEND dc c --> fn()=>receive c
        end
    fun PUT dc x = DWAIT dc --> fn p=>satisfy p x
end
```

The implementation is slightly unsatisfying; an attempt to satisfy a request that has already been satisfied should raise an exception of some kind, instead of blocking silently. I see no way to do this without implementing these primitives directly.

Some changes have to be made to plumbing to make it work with demand channels, for example:

```
fun copy(in',out) = (satisfy (dwait out) (get in'); copy(in',out))
```

Because copying does not delay, it is possible to add to any channel a "noisy" **copy** that will announce traffic without changing timing properties:

```
fun noisy_copy(shout,in',out) =
    let val request = dwait out
        val x = get in'
    in  shout x; satisfy request x; noisy_copy(shout,in',out)
    end
```

where **shout** is assumed to announce the transmission of a value.

Synchronous events

In the Pegasus model, the communication primitives have no side effects; instead they create values called "synchronous events", which are similar to my potential communications. The **sync** primitive is the only primitive with a side effect; **sync e** "synchronizes with" the event **e** (similar to actualizing a potential communication) [Rep88]. I have used functors to transform my model of concurrency into the Pegasus model.

Here is a signature for the Pegasus model:

```
signature EVENTCHAN = sig
    type 'a chan
    type 'a event

    val SEND : 'a chan * 'a -> unit event
    val RECEIVE : 'a chan -> 'a event
    val BLOCK : 'a event
    val UNIT : unit event
    val choose : 'a event list -> 'a event
    infix 2 -->
    val --> : 'a event * ('a -> 'b) -> 'b event
    val sync : 'a event -> 'a
end
```

`BLOCK` (in Pegasus, **noevent**) is an event which can never be chosen; `sync(BLOCK)` causes a process to block forever. `UNIT` (in Pegasus, **anyevent**) is an event that can always be chosen, but does nothing; `sync(UNIT)` is equivalent to `()`.

I represent events as lists of potential communications (this corresponds to the canonical form of events described in [Rep88]). `SEND` and `RECEIVE` map in the straightforward way:

```
functor EventChan(C:CHAN) = struct
    type 'a chan = 'a C.chan
    type 'a event = 'a C.communication list
```

```
    fun SEND c x = [C.SEND c x]
    fun RECEIVE c = [C.RECEIVE c]
        ...
end
```

`sync` is `select`, and `choose` corresponds to list flattening, which combines a list of lists into a single list with the same elements:

```
fun flatten l =
    let fun f (newtail,nil) = newtail | f (newtail,h::t) = f(h@newtail,t)
    in  f(nil,rev l)
    end
val choose = flatten
val sync = C.select
```

I implement `BLOCK` and `UNIT` by creating special channels that are, respectively, never ready to communicate and always ready to communicate, To make `BLOCK` polymorphic (type `'a event`), I make its continuation raise an exception, which has type `'a`.

```
local
    val c = C.mkchan()
    exception Cant_Be_Raised
in
    val BLOCK = C.RECEIVE c --> fn () => raise Cant_Be_Raised
end

local
    val c = C.mkchan ()
    fun source c = (C.send c (); source c)
    val _ = C.begin' "source for UNIT" source c
in
    val UNIT = C.RECEIVE c
end
```

`BLOCK` and `UNIT` can be efficiently and elegantly added to the set of potential communications implemented in the kernel, but it is pleasant to know that they are not necessary.

Guarded potential communications

CSP alternatives can be guarded with Boolean expressions, potential communications, or combinations of both. Given `BLOCK` and `UNIT`, it is easy to extend potential communications, making it possible to put Boolean guards on the left, e.g. with `&&`:

```
infix 3 &&
val && : bool * 'a communication -> 'a communication
```

The implementation requires `BLOCK`:

```
fun && (guard, comm) = if guard then comm else BLOCK
```

The `&&` operator can be used with `UNIT` if one wants to guard an alternative with only a Boolean and not also a communication.

Implementing the primitives

Unlike the concurrent primitives in Pegasus and Newsqueak, which are implemented in C++ and C respectively [Rep88,Pik89a], my concurrent primitives are implemented completely in ML. They respect the ML type system and do not have access to the internal representations of data. The simulation of concurrency is as in [Wan80].

Functions of type `unit -> unit` represent processes whose executions are suspended; applying such a function to `()` resumes the process. Scheduling is non-preemptive; processes run until they attempt to communicate or to spawn a new process, or until they finish. Ready processes are stored in a priority queue, with pseudo-random priorities (thus simulating nondeterministic scheduling). Unready processes may be waiting for I/O, blocked trying to communicate on channels, or both. `resume_ready` removes a ready process from the ready queue and starts it. It also periodically asks the operating system (using the Unix `select` system call) whether I/O is ready, so that processes waiting for I/O can be moved to the ready queue.

Channels are pairs of queues. One queue holds processes blocked trying to send; the other holds processes blocked trying to receive. If one queue holds a blocked process, the other must be empty. A process blocked trying to receive a value of type `'a` can be represented by a continuation of type `'a cont`. Processes blocked trying to send have type `'a * unit cont`.

Processes can be waiting to communicate on more than one channel. When it becomes possible for such a process to communicate on one of the channels, it must remove itself from the queues of all the other channels. Implementing this procedure literally would be quite complicated, especially doing so without violating the ML type system. Instead, I have added a *dirty bit*, type `bool ref`, to the representation of a blocked process. Instances of a blocked process may appear on several queues, but all the instances share one dirty bit. The dirty bit is set as soon as any of the instances become unblocked. The primitives ignore (and discard) any instance of a blocked process whose dirty bit is set.[4] The same dirty bits are used by processes that are waiting for I/O, which enables one to mix `SEND`, `RECEIVE`, and `WAIT` in a single `select`.

`receive c` works by first attempting to remove a clean process from the queue of processes blocked trying to send on `c`. If that attempt fails, `receive` adds the current process to the queue of processes blocked trying to receive. It gets the appropriate continuation using `callcc` and creates a fresh dirty bit using `ref false`. `send` works analogously.

[4] This technique was subsequently used in a re-implementation of the Pegasus concurrency primitives in ML [Rep89].

Potential communications (such as are returned by `SEND` and `RECEIVE`) are values. When `select` is applied to a list of such values, it must first determine whether any can be actualized. If so, it picks one and actualizes it; otherwise, it blocks on all of them. A potential communication is represented by a triple of functions: one to determine whether a communication can be actualized, one to actualize it, and one to block on it. Checking whether a communication can be actualized attempts to remove a process from a queue, just as `send` and `receive` do. Actualization is implemented by calling `send` or `receive`. The blocking function adds an instances of the current process to a queue; it takes as parameters a continuation and a dirty bit.

When `select` is applied to a list of potential communications, it first polls each one to see if it can be actualized. If some communications can be actualized, `select` chooses one at random and actualizes it. Otherwise, `select` blocks the current process on all the potential communications simultaneously by creating a fresh dirty bit, and passing that bit (and a continuation) to the blocking function of each potential communication. After blocking it calls `resume_ready`.

Extending the implementation to support preemptive scheduling would require extra support from the runtime system. One would need to be able to define an ML interrupt handler (with access to the continuation representing the process state at the time of the interrupt), and the data structures in the scheduler and in channels would need to be protected with semaphores. The timer interrupt handler could then cause a process switch. Extending such an implementation to parallel hardware should be straightforward.

Performance

To measure the extra cost involved in using channels in ML, instead of, for example, ML functions, I wrote a process that received integers and kept a running sum of all integers received. I compared this to an "accumulator" function that kept a running sum in an assignable integer cell (`int ref`). The accumulator process is

```
fun accumulator () =
    let val c = mkchan()
        fun accum (n:int) = accum (n+receive c)
    in  begin' "accumulator" accum 0;
        c
    end
```

The accumulator function is

```
fun faccumulator () =
    let val a = ref 0
        fun accum (n:int) = a := !a + n
    in  accum
    end;
```

In the benchmarks, the functions `send (accumulator())` and `faccumulator()` were called repeatedly with argument 1.

I tested the communicating version with my standard implementation of the primitives and also with implementations I modified for increased speed. The first modification uses restricted channels whose queues hold at most one blocked process. Many programs work unchanged with such channels, and these channels can be changed into general channels using one piece of plumbing. The second modification replaces the nondeterministic scheduler with one that keeps ready processes on a stack; it is unfair but fast (like the one in [Wan80]).

To find the cost of communication, I measured the amount of user time required to send 10,000 integers on a VAX 8650 running 4.3 BSD Unix, then repeated this measurement several times. Similarly, I called the accumulator function 1,000,000 times. The times given here are for a single communication or function call; CPU is user time spent outside the garbage collector, and GC is time spent in the garbage collector. The VAX 8650 is rated at 5 MIPS.

Algorithm	CPU	GC	Total	Rel. CPU	Rel. Total
Standard	177 μsec	34 (16%)	211 μsec	1.00	1.00
No queues	151	35 (19%)	186	.85	.87
Unfair sched	142	30 (17%)	172	.80	.82
Unfair & NQ	115	30 (21%)	145	.65	.69
Function call	5	0	5	.03	.02

The large garbage collection overhead comes from updating the reference list [App89]. It could be reduced substantially by using virtual memory techniques [Sha87]. The factor of 35 disparity between the costs of communication and function call is consistent with the observation that a `send/receive` pair requires 27 function calls on average.

This implementation should help a programmer find a concise and elegant way to structure an application as a set of communicating processes. The ML modules system can be used to make small changes in the primitives until the most convenient expression is found. The implementation is slow because it is general and it is written entirely in ML. If speed is important, the most useful version of the primitives can be implemented in C (or assembler) and linked with the Standard ML of New Jersey runtime system [App90].

References

[AJ89] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *16th ACM Symposium on the Principles of Programming Languages*, pages 293–302, January 1989.

[App89] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice/Experience*, 19(2):171–183, February 1989.

[App90] Andrew W. Appel. A runtime system. To appear in *Lisp & Symbolic Computation*, 1990.

[AS83] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *Computing Surveys*, 15(1):1–43, March 1983.

[Bur88] Alan Burns. *Programming in occam 2.* Addison-Wesley, 1988.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[McI90] M. Douglas McIlroy. Squinting at power series. To appear in *Software—Practice & Experience*, 1990.

[Pik89a] Rob Pike. Lectures on the implementation of Newsqueak. Given at Princeton University, April 1989.

[Pik89b] Rob Pike. Newsqueak: A language for communicating with mice. Technical Report 143, AT&T Bell Laboratories, Computing Science Dept, April 1989.

[Rep88] John H. Reppy. Synchronous operations as first-class values. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 250–259, June 1988.

[Rep89] John H. Reppy. First-class synchronous operations in Standard ML. Technical Report TR 89-1068, Cornell University Department of Computer Science, December 1989.

[Sha87] Robert A. Shaw. Improving garbage collector performance in virtual memory. Technical Report CSL-TR-87-323, Stanford University Computer Systems Laboratory, March 1987.

[Wan80] Mitchell Wand. Continuation-based multiprocessing. In *Proceedings of the 1980 LISP Conference*, pages 19–28, 1980.

[Xil88] Xilinx, Inc. *The Programmable Gate Array Data Book*, 1988.