First-class Synchronous Operations in Standard ML*

J. H. Reppy

TR 89-1068
December 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This work was supported, in part, by the NSF and ONR under NSF grant CCR-85-14862.

First-class synchronous operations in Standard ML*

J.H. Reppy
Cornell University
jhr@cs.cornell.edu

December 20, 1989

Abstract

In [Reppy88], we introduced a new language mechanism, *first-class synchronous operations*, for synchronous message passing. In our approach, synchronous operations are represented by first-class values called *events*. Events can be combined in various ways, allowing a user to define new synchronization abstractions (e.g., remote procedure call), which have equal status with the built-in operations.

This paper describes this mechanism and presents a new implementation of events as part of a coroutine package for **Standard ML**. The coroutine package is written entirely in **SML**, using first-class continuations, and provides very light-weight processes. First-class continuations provide a natural way to represent events that closely follows an operational semantics for events.

1 Introduction

We have developed a coroutine package for **Standard ML** (**SML**)[HMM86,HMT88] that supports *first-class synchronous operations*[Reppy88]. This package has been implemented in the **SML** of New Jersey (**SML/NJ**) system[AM87] using *first-class continuations*, which are an experimental feature of **SML/NJ**[DM]. Because the **SML/NJ** implementation of continuations is very cheap, our coroutine package provides very light-weight threads.

The purpose of this paper is two-fold: to provide a guide to users and to describe the implementation. We assume a reasonable familiarity with **SML**. Section 2 describes the package and section 3 provides a number of examples and programming techniques. The implementation is described in section 4 and the source code is given in the appendices.

2 Concurrent SML

Concurrent **SML** is not a new language *per se*, but rather a set of concurrency primitives written in **SML**. The design of Concurrent **SML** has been heavily influenced by the author's

*This work was supported, in part, by the NSF and ONR under NSF grant CCR-85-14862

earlier work on **Pegasus** (done at AT&T Bell Laboratories)[RG86,Reppy88], which provides dynamic process creation, message passing on typed channels, and first-class synchronous operations. In this section we describe the standard features of Concurrent **SML**, then we motivate and describe events.

2.1 Processes and channels

The basic concurrency mechanisms of Concurrent **SML** (and **Pegasus**) are taken from **Amber**[Cardelli86]. Figure 1 gives the interface signature for these primitives. New processes

```
eqtype procid

val process : (unit -> unit) -> procid
val getpid : unit -> procid
val pid2string : procid -> string

type 'a chan

val channel : unit -> '1a chan
val send : ('a * 'a chan) -> unit
val accept : 'a chan -> 'a
```

Figure 1: Basic concurrency primitives

are dynamically created by applying `process` to a "`(unit -> unit)`" value. This creates a new thread to evaluate the argument and returns the `procid` of the new process. The function `getpid` returns the `procid` of current (i.e., calling) process; `pid2string` is used to make a printable string from a `procid`.

Processes communicate by synchronous message passing on typed channels. New channels are created by the `channel` function[1]. The `send` and `accept` functions provide synchronous message passing. When a process executes a `send` operation, it offers a message on a channel and waits until another process offers to `accept` the message; we say that the `send` and `accept` operations *matched*.

An important operation that most synchronous communication systems provide is a `select` operation. This operation allows a process to offer a number of communications simultaneously. The first operation that matches an operation by another process is selected. Although Concurrent **SML** does not provide a `select` operation, the mechanism described below provides the full power of `select`.

[1]The "`'1a`" in the result type of `channel` is a *weak type variable*. The result type must be weakly polymorphic to insure the soundness of the type system.

As an illustration of programming with channels and processes, consider figure 2, which is a stream-style program for computing prime numbers. We use "`int chan`" values to

```
(* from : int -> int chan *)
  fun from n = let
        val ch = channel ()
        fun count i = (send (i, ch); count (i+1))
        in
          process (fn () => (count n)); ch
        end

(* filter : (int * int chan) -> int chan *)
  fun filter (p, inCh) = let
        val outCh = channel ()
        fun loop () = let val i = accept inCh
              in
                if ((i mod p) <> 0) then send (i, outCh) else ();
                loop ()
              end
        in
          process (loop); outCh
        end

(* sieve : unit -> int chan *)
  fun sieve () = let
        val primes = channel ()
        fun loop ch = let val p = accept ch
              in
                send (p, primes);
                loop (filter (p, ch))
              end
        in
          process (fn () => (loop (from 2))); primes
        end
```

Figure 2: Sieve of Eratosthenes

represent streams of integers. The function `from` applied to n returns the infinite integer stream, $n, n+1, \ldots \|$ The function `filter` takes an integer p and a stream of integers and produces a stream with the multiples of p filtered out. These functions are used by `sieve` to produce a stream of prime numbers. Each time `sieve` finds a new prime, it adds another filter to the stream.

2.2 Events

The channel I/O mechanism described in the previous section is fairly standard, found in languages such as **CSP**[Hoare85], **occam**[INMOS83] and **Amber**[Cardelli86]. In Concurrent **SML**, however, `send` and `accept` are actually derived from the more general mechanism of *first-class synchronous operations*[Reppy88]. This mechanism addresses a number of problems with the conventional mechanism.

The example from the previous section provides a good illustration of a major problem with conventional message-passing mechanisms. We are implementing the abstraction of integer streams using channels, but the representation is not hidden. Thus there is nothing to stop the user from inserting an arbitrary value into the stream of primes. The obvious solution is to package the channel in a function

```
(* sieve : unit -> (unit -> int) *)
  fun sieve () = let
        val primes = channel ()
        fun loop ch = ...
        in
          process (fn () => (loop (from 2)));
          (fn () => (accept primes))
        end
```

Unfortunately, this creates another problem: we have hidden the synchronization aspect of "getting the next prime." Thus there is no way to get the effect of a `select` operation on the stream of primes. We are forced to choose between abstraction and flexibility. Our solution to this is to make synchronous operations first-class, which allows us to have our cake and eat it too.

Consider the meanings of `send` and `accept`: a process executing a `send` waits until another process is ready to receive and then transmits the message; a process executing an `accept` waits until another process offers a message and then receives it. Both of these operations can be characterized as "wait until the operation can be completed and then do it." This is the intuition behind first-class synchronous operations; we separate the waiting (i.e., synchronization) from the actual operation. We call these operations *event* values. Figure 3 gives the interface of this mechanism.

More formally, a "τ `event`" value is a synchronous operation that, upon synchronization, returns a value of type τ. The functions `receive` and `transmit` are used to build events that describe channel I/O operations. For example, the definition

```
val evt = transmit ("hello world", ch)
```

```
type 'a event

exception Sync

val sync : 'a event -> 'a

val choose : 'a event list -> 'a event
val wrap   : ('a event * ('a -> 'b)) -> 'b event

val transmit : ('a * 'a chan) -> unit event
val receive  : 'a chan -> 'a event
val wait : procid -> unit event

val noevent : 'a event
val rdyevent : unit event
val anyevent : unit event
```

Figure 3: Events

binds `evt` to an event value describing the operation of sending the string `"hello world"` on the channel ch. To actually send the message we apply the `sync` operation to `evt`. It follows that the standard channel I/O operations are easily defined using events

```
fun send (x, ch) = sync (transmit (x, ch))
fun accept ch = sync (receive ch)
```

We can use events to provide an abstract interface to the stream of primes without hiding the synchronization aspect of the abstraction.

```
(* sieve : unit -> int event *)
  fun sieve () = let
        val primes = channel ()
        fun loop ch = ...
        in
          process (fn () => (loop (from 2)));
          receive primes
        end
```

The power of the event type comes from the `choose` and `wrap` operations, which allow events to be combined to form new synchronization abstractions. The `choose` operation builds an event value for the non-deterministic selection from a list of events. The `wrap` operation provides a way to bind a *wrapper function* to an event. The wrapper is applied to the result of the event it wraps; the following rule illustrates this. A **CSP**-style select mechanism can be implemented using `choose` and `wrap`. For example, the following expression will either read an integer from c1 and square it, or will read an integer from c2 and add 10 to it[2]

```
sync (choose [
  wrap (receive c1, (fn i => (i*i))),
  wrap (receive c2, (fn i => (i + 10)))])
```

The function wait builds an event for synchronizing on the death of a process. There are three base event values that have special semantics. The value noevent, which is equivalent to "choose []", is never satisfied. The value rdyevent is always immediately satisfied. The value anyevent is similar to rdyevent, except that when synchronizing on a choice of events, anyevent values have lower priority than the other events. This property can be used to implement polling. For example, the function

```
fun pollChan ch = sync (choose [
      wrap (receive ch, (fn x => SOME x)),
      wrap (anyevent, (fn () => NONE))])
```

returns NONE if there is no message waiting on ch, otherwise it returns the message[3]. Note that rdyevent would not work here, because in the situation where there is a message waiting on ch, there is a possibility that the rdyevent will be selected.

2.3 Semantics

The semantics of events are given informally in [Reppy88] in terms of the *canonical event form.* A canonical event has the form

$$\mathtt{choose}[\mathtt{wrap}(bev_1,\ f_1),\ \ldots,\ \mathtt{wrap}(bev_n,\ f_n)]$$

where the bev_i are base events. A collection of rewrite rules are given in [Reppy88], which map event values to equivalent canonical event values.

The meaning of applying sync to a canonical event is given by: first, poll the bev_i looking for immediately satisfiable events and choose one if available; second, if there are no immediately satisfiable events, then suspend the process until one of the events is satisfied; finally, apply the corresponding wrapper function to the result of the satisfied event and return it as the result.

[2] The **SML** expression "[$e_1, ..., e_n$]" evaluates to a list of n elements.
[3] This is using the **SML** option type, defined as "**datatype 'a option = NONE | SOME 'a**"

3 Programming with events

Events provide a powerful mechanism for implementing new synchronization and communication abstractions. In this section we first give some small examples of programming techniques using events and then give a complete example of the implementation of a new abstraction.

3.1 Programming techniques

Events provide a tremendous amount of flexibility in defining new synchronization mechanisms, and there is no way that we could catalogue them all here. Instead, we give some small examples of the programming techniques that are available.

The `choose` operation provides a "*parallel or*" mechanism, but it is also possible to implement a "*parallel and*" operation. The following curried function returns an event that produces the folding of two events.

```
(* pAnd : (('a -> 'b) -> 'c) -> ('a event * 'b event) -> 'c event *)
  fun pAnd f (ev1, ev2) = choose [
        wrap (ev1, fn x => f (x, sync ev2)),
        wrap (ev2, fn y => f (sync ev1, y))]
```

The expression

```
  sync (pAnd (op + : (int * int) -> int) (receive c1, receive c2))
```

will read an integer from channel `c1` and one from `c2` (in some order) and return their sum.

Another important application of events is building *remote procedure call* (RPC) style interfaces. The following function builds the client and server sides of a RPC interface to a function.

```
(* mkRemote : ('1a -> '1b) -> (('1a -> '1b event) * unit event) *)
  fun mkRemote f = let
        val argCh = channel() and resCh = channel()
        fun clientF x = wrap(transmit(x, argCh), fn () => (accept resCh))
        val serverF = wrap(receive argCh, fn x => (send (f x, resCh)))
        in
          (clientF, serverF)
        end
```

This can be used to build a static interface to a server. For example, the following server produces system-wide unique identifiers.

```
val (getId : int event) = let
      val cnt = ref 0
      val (get, put) = mkRemote (fn () => ((!cnt) before (inc cnt)))
      fun loop () = (sync put; loop())
      in
        process loop; get ()
      end
```

We saw above that `anyevent` can be used to poll for input, but it can also be used to implement **Ada**'s *conditional entry-call* mechanism[DoD83]. If `entryFn` is an event value for a RPC, then the following is a conditional entry call of it.

```
choose [
  entryFn,
  wrap (anyevent, fn () => (raise ServerNotReady))]
```

If the server is not ready to handle `entryFn`, then the exception `ServerNotReady` is raised. This example illustrates that we can treat user-defined synchronous operations, such as `entryFn`, on an equal basis with the built-in operations.

It is sometimes useful for a server to manage dynamic lists of clients. For example, a window manager needs to monitor output requests for a variable number of windows. One way to implement this is by using a unique identifier for each window, and tagging all requests with this identifier. Another approach is to use a different channel for each client. The server process then can maintain a list of clients and their events. The following code manages a dynamic list of clients and deals with the situation in which a client dies unexpectedly.

```
datatype client = Client of (procid * unit event)

val (clients : client list ref) = ref nil

fun removeClient pid = let
      fun find ((c as Client(p, _)) :: rest) =
            if (p = pid) then rest else c :: (find rest)
      in
        clients := find (!clients)
      end

exception DeadProc of procid

fun mkEvent () = let
      fun f (Client(pid, ev)) = choose [
            ev, wrap (wait pid, fn () => (raise (DeadProc pid)))]
      in
        choose (map f (!clients))
```

```
        end

    fun serverLoop ev = serverLoop (
          (sync ev; ev) handle (DeadProc pid) => (removeClient pid; mkEvent()))
```

Each client is represented by it procid and an event value that is the amalgamation of the server-side events for the client. From this representation mkEvent builds an event that catches the death of the client; the server then removes dead clients from its client list.

3.2 Buffered multi-cast

As a final illustration of concurrent programming using events, we present the complete implementation of a new communication abstraction: *buffered multi-cast channels.* A multi-cast channel has a number of *output ports.* When a process sends a message on a multi-cast channel, it is replicated once for each output port. Output ports are buffered, so message sending is asynchronous. Messages appear at output ports in the same order. This abstraction has the following interface:

```
type 'a mchan
type 'a port

val mChannel : unit -> '1a mchan
val newPort : multicast : 'a mchan -> 'a port
val multicast : ('a * 'a mchan) -> unit
val readPort : 'a port -> 'a event
```

New multi-cast channels are created using mChannel and new ports using newPort. The multicast operation asynchronously broadcasts a message to the ports of a multi-cast channel and readPort returns an event value for receiving a message from a port. Figure 4 is the implementation of this interface.

The representation of a mchan value consists of a request channel and a response channel for communicating with a dedicated server process. The functions multicast and newPort are implemented as requests to the server process; in the case of newPort, the server responds with a new port.

For each associated output port, there is a port process. When the server receives a message request, it sends it to a port process. The port process adds the message to its buffer and sends the message to another port process. In this way, the message is propagated to all of the ports. The server and port processes both represent the next port to propagate the message to by the function value outFn. Initially, when there are no ports, outFn is a no-op. The function mkPort, which creates new ports, takes the server's current outFn as

```
datatype 'a request = Message of 'a | NewPort

datatype 'a mchan = MChan of ('a request chan * 'a port chan)
     and 'a port = Port of 'a event

(* mChannel : unit -> 'a mchan *)
 fun mChannel () = let
       val reqCh = channel() and respCh = channel()
       fun mkPort outFn = let
             val inCh = channel() and msgCh = channel()
             fun portLoop buffer = portLoop (
                   sync (choose [
                     (case buffer
                      of nil => noevent
                       | (x :: rest) => wrap (transmit(x, msgCh), fn () => rest)),
                     wrap (receive inCh, fn m => (outFn m; buffer @ [m]))]))
             in
               process (fn () => portLoop nil);
               ((fn m => send (m, inCh)), Port(receive msgCh))
             end
       fun serverLoop outFn = let
             fun handleReq (NewPort) = let
                   val (outFn', port) = mkPort outFn
                   in
                     send (port, respCh);
                     outFn'
                   end
               | handleReq (Message m) = (outFn m; outFn)
             in
               serverLoop (sync (wrap (receive reqCh, handleReq)))
             end
       in
         process (fn () => serverLoop (fn _ => ()));
         MChan(reqCh, respCh)
       end

(* newPort : 'a mchan -> 'a port *)
 fun newPort (MChan(reqCh, respCh)) = (send (NewPort, reqCh); accept respCh)

(* multicast : ('a * 'a mchan) -> unit *)
 fun multicast (m, MChan(reqCh, _)) = send (Message m, reqCh)

(* readPort : 'a port -> 'a event *)
 fun readPort (Port ev) = ev
```

Figure 4: Multi-cast channels

an argument and returns a new outFn that talks to the newly created port process. We can view the port processes as forming a linked list with the outFn values playing the roles of links and the initial no-op outFn as the null link.

In addition to the outFn, each port process has a channel for receiving messages, a buffer for messages and an output channel. When there are messages in the buffer, the port process synchronizes on the choice of sending the first buffered message and receiving another message from the server. When the buffer is empty, the port process waits for the next message from the server.

4 Implementation

Concurrent **SML** is implemented in **SML/NJ**. The implementation consists of two files: "events.sig," which contains the interface signature, and "events.sml," which contains the actual implementation. The source text of these files are included as appendices.

The implementation relies heavily on first-class continuations, so we first give a brief introduction to **SML/NJ**'s continuation mechanism. Then we describe the implementation of processes, channels and events using continuations.

4.1 Continuations

The continuation of an expression is a function that executes the rest of the program, when given the result of the expression as an argument. For example, in the program

```
if (a < b) then f() else g()
```

the continuation of "(a < b)" can be described as "if the value is true, then call f, otherwise call g".

In **SML**, continuations are a parameterized abstract type with two operations[DM]:

```
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
val throw  : 'a cont -> ('a -> 'b)
```

A τ cont is the type of a function representing the rest of the program with a formal parameter of type τ. Continuations are created using callcc (call with current continuation) and are applied using throw. A simple example is the expression

```
callcc (fn (k : int cont) => (throw k 5; 6)) + 7
```

The variable `k` is bound to the `int cont` that adds 7 to its argument; the `throw` applies `k` to 5, giving 12.

4.2 Implementing processes

Using first-class continuations, it is possible to implement light-weight processes (or *threads*) directly in a high-level language[Wand80,HFW84]. In a conventional implementation of process abstraction, a *process state vector* is maintained for every suspended process; the state vector contains the necessary information to restart the process. A continuation is an abstraction of exactly this information; thus we can represent a suspended process by its resumption continuation.

Our implementation of processes and process management is similar to that of [Wand80] and [HFW84], so we skip most of the low-level details here.

For type-checking convenience, we represent process resumptions as

```
type proc = (unit -> unit)
```

Each process also has an associated `procid`, which is a record of process specific information. There is a global variable, `currentProc`, that references the `procid` of the current thread. Two functions

```
val enqueue : (procid * proc) -> unit
val dequeue : unit -> (procid * proc)
```

are used to manage the process ready queue. The next process is dispatched by

```
val dispatch : unit -> 'a
```

The return type of `dispatch` is unconstrained, since it never returns (as are the return types of `raise` and `throw`).

Process creation is worth looking at in more detail.

```
(* create a new process *)
  fun process f = callcc (fn parent_k => let
        val pid = newPid()
        fun child () = (
              (f ()) handle exn => (
                print (pid2string pid); print " uncaught exception ";
                print (System.exn_name exn); print "\n");
              notify pid;
              dispatch())
```

```
fun parent () = (throw parent_k pid)
in
  enqueue (pid, child);
  enqueueCurProc parent;
  dispatch ()
end)
```

The body of the child process is wrapped in an exception handler. This will catch any exceptions missed by the process body, print a message and terminate the thread cleanly. The call to `notify` is used to support the `wait` event and is discussed below.

This implementation of processes is very light-weight. For example, running the sieve program of section 2 to find the 1000th prime number requires a total of about 350,000 bytes of live data, including the or about 350 bytes per process (and this includes the cost of 1000 channels). This is in direct contrast with more conventional thread packages, which use several hundred bytes for the process state vector and thousands of bytes for the process stack. The low space cost of threads is principally a result of the fact that **SML/NJ** does not use a run-time stack[AJ89].

4.3 Implementing channels

Figure 5 gives the data-types used to represent channels. A channel consists of an input

```
datatype 'a chanq = CHANQ of {
    front   : (bool ref * 'a) list ref,
    rear    : (bool ref * 'a) list ref
  }

datatype chan_state = IDLE | INPUT_WAIT | OUTPUT_WAIT

datatype 'a chan = CHAN of {
    state   : chan_state ref,
    inq     : (procid * 'a cont) chanq,
    outq    : (procid * 'a * unit cont) chanq
  }
```

Figure 5: Channel data structures

queue and an output queue; it can be in one of three states: `IDLE`, when both queues are empty; `INPUT_WAIT`, when there are waiting processes in the input queue; and `OUTPUT_WAIT`, when there are waiting process/message pairs in the output queue. At anytime, at most one of the queues will be non-empty. The "`bool ref`" slots are used to *unlog* pending I/O operations (described below). The functions

```
val insert : ('a chanq * bool ref * 'a) -> unit
val remove : 'a chanq -> 'a
```

are used to manage the channel waiting queues. Although `send` and `accept` can be implemented using events, they are actually implemented directly for efficiency reasons.

4.4 Implementing events

To motivate the implementation of events, we examine the implementation of the `P` operation on binary semaphores, which is perhaps the simplest synchronous operation. In our setting, it has the implementation (we separate out the body of the operation for pedagogical reasons)

```
datatype semaphore = SEMAPHORE of {
    flg : bool ref,
    waitq : unit cont list ref
  }

(* P : semaphore -> unit *)
fun P (SEMAPHORE{flg, waitq}) = let
      fun Pbody (resumek) = if (!flg)
            then (flg := false)
            else (waitq := !waitq @ [resumek]; dispatch())
      in
        callcc Pbody
      end
```

Notice that the resumption continuation of the calling process is a free variable in the body of the operation. This observation, which holds for all synchronous operations, is the key to the implementation of events. It suggests that an "`'a event`" value can be represented by an "`('a cont -> 'a)`" value. With this representation, an event-style implementation of `P` is

```
(* P : semaphore -> unit event *)
fun P (SEMAPHORE{flg, waitq}) = let
      fun Pbody (resumek) = if (!flg)
            then (flg := false)
            else (waitq := !waitq @ [resumek]; dispatch())
      in
        Pbody
      end
```

It follows that `sync` is implemented directly by `callcc` and `wrap` is implemented by

```
fun wrap (evt, f) = fn k => (throw k (f (callcc evt)))
```

Unfortunately, this simple representation of events is unable to support the choose operation. When synchronizing on an event composed of several base events, there are three steps that we must take:

- *Polling*: first we must poll the base events to see if any of them are immediately satisfiable.
- *Logging*: if there are no immediately satisfiable events, then we must add the process to the waiting queues of the base events.
- *Unlogging*: when one of the base events is satisfied, we must remove the process from the other base events' waiting queues.

An event is represented by a list of base event descriptors. A base event descriptor is a pair of functions: a polling function and the event function (see figure 6). The polling

```
datatype evt_sts = EVT_ANY | EVT_READY | EVT_BLOCK

type 'a base_evt = (unit -> evt_sts) * (bool ref * 'a cont -> unit)

datatype 'a event = EVT of 'a base_evt list
```

Figure 6: Event data structures

function is used to determine if a base event is immediately satisfiable; it returns `EVT_ANY` for `anyevent`, `EVT_READY` for other immediately satisfiable events, and `EVT_BLOCK` for base events that are not immediately satisfiable. The event function implements the synchronous operation, logging, and wrapper. To deal with unlogging, we use a boolean flag shared among all of the base events of an event; this flag will be set to true when one of the base events is satisfied[4]. These *dirty* items are ignored by the polling functions. This representation of events is essentially the *canonical event form* described in [Reppy88].

The implementation of `sync` (figure 7) uses the function `extract` to poll the list of events. If this is non-empty, `sync` then uses the function `select` to select one of the immediately satisfiable events. The function `random` (not shown) generates the selection index; we currently use a psuedo round-robin scheme. If there are no immediately satisfiable events, then we log the synchronization point continuation with the base event waiting queues and dispatch the next process.

Figure 8 contains the implementation of `wrap`. This is basically our previous implementation (with a slight modification to handle the extra argument) distributed over the list of base events. The continuation `return_k` is necessary to avoid applying the wrapper

[4] This trick is due to Norman Ramsey at Princeton University.

```
datatype 'a ready_evts
  = NO_EVTS                              (* no ready events *)
  | ANY_EVTS of (int * 'a base_evt list)  (* list of ready anyevents *)
  | RDY_EVTS of (int * 'a base_evt list)  (* list of ready events *)

val extract : 'a base_evt list -> 'a ready_evts

(* sync : 'a event -> 'a *)
  fun sync (EVT el) = callcc (fn sync_k => (
        case el
         of nil => ()
          | [(pollFn, evtFn)] => (case pollFn()
                of EVT_BLOCK => evtFn (ref false, sync_k)
                 | _ => evtFn (ref true, sync_k))
          | el => let
              fun select (n, l) = let
                    val (_, evtFn) = nth (l, random n)
                    in
                      evtFn (ref true, sync_k)
                    end
              in
                case extract el
                 of NO_EVTS => let
                      val evtflg = ref false
                      fun log (_, evtFn) = evtFn(evtflg, sync_k)
                      in
                          app log el
                      end
                  | ANY_EVTS anyevts => select anyevts
                  | RDY_EVTS evts => select evts
              end
        (* end case *);
        dispatch()))
```

Figure 7: Implementing sync

function when logging the event. The choose operation is implemented by flattening the list of event lists.

Base events are represented by singleton lists. For example, the implementation of transmit is given in figure 9. The polling function first cleans the head of the input queue, removing any dirty items. Then it tests to see if any process is waiting for input. The event function must deal with two cases: either the input queue is empty and the current process must be added to the output waiting queue, or there is a process waiting for input and both processes can proceed. The implementation of receive is symmetric.

```
(* wrap : ('a event * ('a -> 'b)) -> 'b event *)
  fun wrap (EVT el, f) = let
        fun wrap' (nil, evts) = evts
          | wrap' ((tstFn, evtFn) :: rest, evts) = let
              fun evtFn' (flg, k) = callcc (fn return_k => (
                    throw k (f (callcc (fn wrapper_k => (
                            evtFn (flg, wrapper_k);
                            throw return_k ()))))))
              in
                wrap' (rest, (tstFn, evtFn') :: evts)
              end
        in
          EVT(wrap' (el, nil))
        end
```

Figure 8: Implementing `wrap`

The implementation of the base events `anyevent`, `rdyevent` and `noevent` are trivial: `anyevent` and `rdyevent` have polling functions that return `EVT_ANY` and `EVT_READY`, respectively, and trivial event functions; `noevent` is represented by the empty base event list.

The `wait` function allows synchronization on process termination. This is implemented by a list of waiting processes in the `procid` object. The `wait` event function adds the calling process to the waiting list. When a process dies, it calls `notify`, which dispatches the waiting processes. As with channels, we use a dirty flag to mark obsolete entries on the waiting list.

5 Summary

We have presented the design and implementation of a coroutine package in **SML/NJ**. Our package provides very light-weight processes as well as a flexible mechanism for implementing user-defined synchronization abstractions. We have shown how to use this mechanism to implement a number of synchronous operations. The implementation is included in the appendices.

```
(* transmit : ('a * 'a chan) -> unit event *)
  fun transmit (msg, chan as CHAN{state, inq, outq}) = let
        fun pollFn () = (case (!state)
               of INPUT_WAIT =>
                    if (clean inq) then (state := IDLE; EVT_BLOCK) else EVT_READY
                | _ => EVT_BLOCK)
        fun evtFn (flg, kont) = (case (!state)
               of INPUT_WAIT => let
                    val _ = if (not (!flg)) then (flg := true; raise Sync) else ()
                    val (rpid, rkont) = remove inq
                    in
                      enqueue (rpid, fn () => (throw rkont msg));
                      enqueueCurProc (throw kont)
                    end
                | _ => (
                    insert (outq, flg, (getpid(), msg, kont));
                    state := OUTPUT_WAIT))
        in
          EVT[(pollFn, evtFn)]
        end
```

Figure 9: The transmit function

References

[AJ89] Appel, A.W. and T. Jim. "Continuation-passing, closure-passing style," *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, January 1989, pp. 293-302.

[AM87] Appel, A.W. and D.B. MacQueen. "A standard ml compiler," *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 274, Springer-Verlag, September 1987, pp.301-324.

[Cardelli86] Cardelli, L. "Amber," *Combinators and Functional Programming Languages*, Lecture Notes in Computer Science 242, Springer-Verlag, 1986, pp.21-47.

[DM] Duba, B. and D. MacQueen. "Type-checking first-class continuations," *in preparation.*

[DoD83] United States Department of Defense. *Ada reference manual*, 1983.

[HFW84] Haynes, C.T., D.P. Friedman and M. Wand. "Continuations and coroutines," *Conference record of the 1984 ACM Conference on Lisp and Functional Programming*, 1984, pp. 293-298.

[HMM86] Harper, R., D. MacQueen and R. Milner. "Standard ml," *ECS-LFCS-86-2*, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1986.

[HMT88] Harper, R., R. Milner and M. Tofte. "The definition of standard ml (version 2)," *ECS-LFCS-88-62*, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1988.

[Hoare85] Hoare, C.A.R. *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[INMOS83] INMOS Limited. *Occam Programming Manual*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Reppy88] Reppy, J.H. "Synchronous operations as first-class values," *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988, pp. 250-259.

[RG86] Reppy, J.H. and E.R. Gansner. "A foundation for programming environments," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, December 1986, pp. 218-227.

[Wand80] Wand, M. "Continuation-based multiprocessing," *Conference Record of the 1980 Lisp Conference*, August 1980, pp. 19-28.

A events.sig

```
signature EVENTS =
sig

  (** processes **)
    type procid
    type proc

    val process : (unit -> unit) -> procid

    val yield : unit -> unit

    val getpid : unit -> procid

    val pid2string : procid -> string

  (** channels **)
    type 'a chan

    val channel : unit -> '1a chan

    val send : ('a * 'a chan) -> unit
    val accept : 'a chan -> 'a

  (** events **)
    type 'a event

    exception Sync

    val sync : 'a event -> 'a
    val wrap : ('a event * ('a -> 'b)) -> 'b event
    val choose : 'a event list -> 'a event

    val anyevent : unit event
    val rdyevent : unit event
    val noevent : 'a event

    val transmit : ('a * 'a chan) -> unit event
    val receive : 'a chan -> 'a event

    val wait : procid -> unit event

  (* reset the system *)
    val reset : unit -> unit

end (* signature EVENTS *)
```

B events.sml

The following is the complete source code for the events package.

```
structure Events  : EVENTS =
struct

    fun reverse (nil, rl) = rl
      | reverse (x :: rest, rl) = reverse(rest, x :: rl)

  (** Processes **)

    datatype procid = PROCID of {
        pid     : int,
        isDead  : bool ref,
        waiters : (procid * bool ref * unit cont) list ref
      }

  (* return a string representation of a process id *)
    fun pid2string (PROCID{pid, ...}) = implode ["[", makestring pid, "]"]

    type proc = (unit -> unit)

  (* the process ready queue *)
    local
    (* the queue of ready processes waiting to be run *)
      val (rdyQFront : (procid * proc) list ref) = ref nil
      val (rdyQRear : (procid * proc) list ref) = ref nil
    in

    exception AllDead  (* raised when the process queue is empty *)

  (* remove a (procid * proc) pair from the ready queue. *)
    fun dequeue () = (case (!rdyQFront)
           of (p :: rest) => (rdyQFront := rest; p)
            | nil => (case reverse(!rdyQRear, nil)
               of (p :: rest) => (rdyQFront := rest; rdyQRear := nil; p)
                | nil => raise AllDead))

  (* add a (procid * proc) pair to the ready queue. *)
    fun enqueue p = (case (!rdyQFront)
           of nil => (rdyQFront := reverse(!rdyQRear, [p]); rdyQRear := nil)
            | _ => rdyQRear := p :: !rdyQRear)

  (* reset the queue *)
    fun resetProcQ () = (rdyQFront := nil; rdyQRear := nil)

    end (* local *)

    local
    (* generate new process ids *)
```

```
    val nextPid = ref 0
    fun newPid () = let val id = !nextPid
          in
            nextPid := id + 1;
            PROCID{pid = id, isDead = ref false, waiters = ref nil}
          end

  (* the current process *)
    val currentProc = ref (newPid())

  (* notify any waiting processes of the death of a process *)
    fun notify (PROCID{isDead, waiters, ...}) = let
          fun notify' (pid, flg, kont) = if (!flg)
                then (flg := true; enqueue (pid, throw kont))
                else ()
          in
            isDead := true;
            app notify' (!waiters)
          end

  in

(* add the current process to the ready queue *)
  fun enqueueCurProc resume = (enqueue(!currentProc, resume))

(* dispatch the next process on the ready queue *)
  fun dispatch () = let
        val (nextpid, nextProc) = dequeue ()
        in
          currentProc := nextpid;
          nextProc ();
          dispatch ()
        end

(* create a new process *)
  fun process f = callcc (fn parent_k => let
        val pid = newPid()
        fun child () = (
              (f ()) handle exn => (
                print (pid2string pid); print " uncaught exception ";
                print (System.exn_name exn); print "\n");
              notify pid;
              dispatch())
        fun parent () = (throw parent_k pid)
        in
          enqueue (pid, child);
          enqueueCurProc parent;
          dispatch ()
        end)

(* yield control to the next process *)
  fun yield () = callcc (fn k => (enqueueCurProc (throw k); dispatch()))
```

```
  fun getpid () = (!currentProc)

(* reset the system, clearing the ready queue *)
  fun reset () = (nextPid := 0; resetProcQ())

  end (* local *)


(** Channels **)

  datatype 'a chanq = CHANQ of {
      front   : (bool ref * 'a) list ref,
      rear    : (bool ref * 'a) list ref
    }

  datatype chan_state = IDLE | INPUT_WAIT | OUTPUT_WAIT

  datatype 'a chan = CHAN of {
      state   : chan_state ref,
      inq     : (procid * 'a cont) chanq,
      outq    : (procid * 'a * unit cont) chanq
    }

(** Channel queue routines **)

  fun newq () = CHANQ{front = ref nil, rear = ref nil}

(* insert an item into a channel queue *)
  fun insert (CHANQ{front, rear}, flg, item) = (case (!front)
         of nil => (front := reverse(!rear, [(flg, item)]); rear := nil)
          | _ => rear := (flg, item) :: !rear)

(* remove an item from a channel queue and set its dirty flag *)
  fun remove (CHANQ{front, rear}) = (case (!front)
         of nil => let val ((flg, x) :: rest) = reverse(!rear, nil)
              in
                front := rest; rear := nil; flg := true; x
              end
          | ((flg, x) :: rest) => (front := rest; flg := true; x))

(* Clean a channel of satisfied transactions. We do this incrementally to
 * give a amortized constant cost. Basically we guarantee that the front
 * of the queue will be unsatisfied. Return true if the resulting queue
 * is empty.
 *)
  fun clean (CHANQ{front, rear}) = let
        fun clean' nil = nil
          | clean' (l as ((flg, _) :: rest)) = if !flg then clean' rest else l
        in
          case (clean' (!front))
           of nil => (case clean'(reverse(!rear, nil))
```

```
                of nil => (front := nil; rear := nil; true)
                 | l => (front := l; rear := nil; false))
            | l => (front := l; false)
        end


(* channel : unit -> '1a chan *)
  fun channel () = CHAN{state = ref IDLE, inq = newq(), outq = newq()}

(* send : ('a * 'a chan) -> unit *)
  fun send (msg, CHAN{state, inq, outq}) = callcc (fn send_k => (
        if ((!state = INPUT_WAIT) andalso (clean inq)) then state := IDLE else ();
        case (!state)
         of INPUT_WAIT => let
              val (rpid, rkont) = remove inq
              in
                enqueue (rpid, fn () => (throw rkont msg));
                enqueueCurProc (throw send_k)
              end
          | _ => (
              insert(outq, ref false, (getpid(), msg, send_k));
              state := OUTPUT_WAIT)
        (* end case *);
        dispatch()))

(* accept : 'a chan -> 'a *)
  fun accept (CHAN{state, inq, outq}) = callcc (fn accept_k => (
        if ((!state = OUTPUT_WAIT) andalso (clean outq)) then state := IDLE else ();
        case (!state)
         of OUTPUT_WAIT => let
              val (spid, msg, skont) = remove outq
              in
                enqueue (spid, throw skont);
                enqueueCurProc (fn () => (throw accept_k msg))
              end
          | _ => (
              insert(inq, ref false, (getpid(), accept_k));
              state := INPUT_WAIT)
        (* end case *);
        dispatch()))


(** Events **)

  exception Sync

  datatype evt_sts = EVT_ANY | EVT_READY | EVT_BLOCK

  type 'a base_evt = (unit -> evt_sts) * (bool ref * 'a cont -> unit)

  datatype 'a event = EVT of 'a base_evt list
```

```
local
  datatype 'a ready_evts
    = NO_EVTS                              (* no ready events *)
    | ANY_EVTS of (int * 'a base_evt list)  (* list of ready anyevents *)
    | RDY_EVTS of (int * 'a base_evt list)  (* list of ready events *)

(* Extract a list of ready events by polling a list of base events.  Priority
 * is given to events other than anevent. *)
  fun extract nil = NO_EVTS
    | extract ((evt as (pollFn, _)) :: rest) = (case (pollFn ())
         of EVT_ANY => extract_any (1, rest, [evt])
          | EVT_READY => extract_rdy (1, rest, [evt])
          | EVT_BLOCK => extract rest)
  and extract_rdy (n, nil, rdy_evts) = RDY_EVTS (n, rdy_evts)
    | extract_rdy (n, (evt as (pollFn, _)) :: rest, rdy_evts) = (case (pollFn ())
         of EVT_READY => extract_rdy (n+1, rest, evt :: rdy_evts)
          | _ => extract_rdy (n, rest, rdy_evts))
  and extract_any (n, nil, any_evts) = ANY_EVTS (n, any_evts)
    | extract_any (n, (evt as (pollFn, _)) :: rest, any_evts) = (case (pollFn ())
         of EVT_ANY => extract_any (n+1, rest, evt :: any_evts)
          | EVT_BLOCK => extract_any (n, rest, any_evts)
          | EVT_READY => extract_rdy (1, rest, [evt]))

(* Generate index numbers for "non-deterministic" selection.  We use a
 * round-robin style policy. *)
  val cnt = ref 0
  fun random i = ((!cnt mod i) before (inc cnt))

in

(* sync : 'a event -> 'a *)
  fun sync (EVT el) = callcc (fn sync_k => (
        case el
         of nil => ()
          | [(pollFn, evtFn)] => (case pollFn()
               of EVT_BLOCK => evtFn (ref false, sync_k)
                | _ => evtFn (ref true, sync_k))
          | el => let
              fun select (n, l) = let
                    val (_, evtFn) = nth (l, random n)
                    in
                      evtFn (ref true, sync_k)
                    end
              in
                case extract el
                 of NO_EVTS => let
                      val evtflg = ref false
                      fun log (_, evtFn) = evtFn(evtflg, sync_k)
                      in
                          app log el
                      end
                  | ANY_EVTS anyevts => select anyevts
```

```
              | RDY_EVTS evts => select evts
            end
        (* end case *);
        dispatch()))

(* wrap : ('a event * ('a -> 'b)) -> 'b event *)
  fun wrap (EVT el, f) = let
        fun wrap' (nil, evts) = evts
          | wrap' ((pollFn, evtFn) :: rest, evts) = let
              fun evtFn' (flg, k) = callcc (fn return_k => (
                    throw k (f (callcc (fn wrapper_k => (
                                  evtFn (flg, wrapper_k);
                                  throw return_k ())))))))
              in
                wrap' (rest, (pollFn, evtFn') :: evts)
              end
        in
          EVT(wrap' (el, nil))
        end

  end (* local *)

(* choose : 'a event list -> 'a event *)
  fun choose evts = let
          fun choose' (nil, nil, el) = el
            | choose' ((EVT evt) :: rest, nil, el) = choose' (rest, evt, el)
            | choose' (evts, e :: rest, el) = choose' (evts, rest, e :: el)
          in
              EVT (choose' (evts, nil, nil))
          end


(** Base events **)

(* anyevent : unit event *)
  val anyevent = EVT[
                ((fn () => EVT_ANY),
                (fn (_, k) => enqueueCurProc (throw k)))]

(* rdyevent : unit event *)
  val rdyevent = EVT[
                ((fn () => EVT_READY),
                (fn (_, k) => enqueueCurProc (throw k)))]

(* noevent : 'a event *)
  val noevent = EVT[]

(* transmit : ('a * 'a chan) -> unit event *)
  fun transmit (msg, chan as CHAN{state, inq, outq}) = let
        fun pollFn () = (case (!state)
               of INPUT_WAIT =>
                    if (clean inq) then (state := IDLE; EVT_BLOCK) else EVT_READY
```

```
                | _ => EVT_BLOCK)
        fun evtFn (flg, kont) = (case (!state)
                of INPUT_WAIT => let
                    val _ = if (not (!flg)) then (flg := true; raise Sync) else ()
                    val (rpid, rkont) = remove inq
                    in
                      enqueue (rpid, fn () => (throw rkont msg));
                      enqueueCurProc (throw kont)
                    end
                | _ => (
                    insert (outq, flg, (getpid(), msg, kont));
                    state := OUTPUT_WAIT))
        in
          EVT[(pollFn, evtFn)]
        end

  (* receive : 'a chan -> 'a event *)
    fun receive (chan as CHAN{state, inq, outq}) = let
        fun pollFn () = (case (!state)
                of OUTPUT_WAIT =>
                    if (clean outq) then (state := IDLE; EVT_BLOCK) else EVT_READY
                 | _ => EVT_BLOCK)
        fun evtFn (flg, kont) = (case (!state)
                of OUTPUT_WAIT => let
                    val _ = if (not (!flg)) then (flg := true; raise Sync) else ()
                    val (spid, msg, skont) = remove outq
                    in
                      enqueue (spid, throw skont);
                      enqueueCurProc (fn () => (throw kont msg))
                    end
                 | _ => (
                    insert (inq, flg, (getpid(), kont));
                    state := INPUT_WAIT))
        in
          EVT[(pollFn, evtFn)]
        end


  (* wait : procid -> unit event *)
    fun wait (PROCID{isDead, waiters, ...}) = let
        fun pollFn () = if (!isDead) then EVT_READY else EVT_BLOCK
        fun evtFn (flg, kont) = (
              if (!isDead)
                then enqueueCurProc (throw kont)
                else waiters := (getpid(), flg, kont) :: !waiters)
        in
          EVT [(pollFn, evtFn)]
        end

end (* structure Events *)
```