Concurrent Programming with Events

—

The Concurrent ML Manual
(Version 0.9.8)

February 1, 1993

John H. Reppy

AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

Contents

License and disclaimer

This distribution is provided free of charge, under the following “X-windows style” copyright:

Portions of this software may also be under the **SML/NJ** copyright:

Preface

Concurrent ML (**CML**) is a system for concurrent programming in Standard ML (**SML**)[MTH90]. A **CML** program consists of a set of *threads* (or light-weight processes). A thread is the sequential evaluation of a **ML** expression. It does not have to be a terminating computation; in fact, infinitely looping threads are often useful. The evaluation of a thread may involve communication with other threads, which is done by sending a message on a channel. Message passing is synchronous and forms the basis of communication and synchronization in **CML**. This model is extended by *first-class synchronous operations*[Rep88], which provide a mechanism for building new synchronization and communication abstractions.

CML is implemented on top of Standard ML of New Jersey (**SML/NJ**). We use **SML/NJ**'s first-class continuations[DHM91] to implement threads and its asynchronous signal mechanism[Rep90] to implement preemptive thread scheduling. The implementation is derived from the one described in [Rep89].

In addition to this manual, the distribution also contains an overview paper, which is a revised version of [Rep91a], and a description of the formal semantics of the **CML** concurrency primitives, which is a revised version of [Rep91b]. The author's disseratation [Rep92] provides further details about **CML**.

We would like to hear from you. If you ftp a copy of the distribution, then please send us mail; we will use this information for sending bug fixes between releases. If you have any comments, suggestion or bug reports, then please send them to us at:

`sml-bugs@research.att.com`[1]

or surface mail to:

John H. Reppy
Rm. 2A-428
AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

How to read this manual

This manual attempts to satisfy many needs: tutorial, user's guide and reference manual, as well as release notes. For a tutorial introduction to **CML**, you should read Chapter 1. Source code for many of the examples is included in the distribution; Appendix C gives a mapping from examples to source files. Chapters 2–5

[1]The old address of `cml-bugs@cs.cornell.edu` should still work.

are a more complete and systematic presentation of **CML**. The information in Chapter 6 is important for the development and construction of systems based on **CML**. Some hints on debuggin **CML** programs and a description of the debugging support provided is given in Chapter 7. Installation and licensing information can be found in Chapter 8. Appendix A contains the complete signatures of the **CML** system; Appendix B gives the signatures of the library modules. Lastly, Appendix C gives list of the source files in the distribution corresponding to the examples.

Acknowledgements

The design and implementation of **CML** was done at Cornell University, and was was supported, in part, by the NSF and ONR under NSF grant CCR-85-14862, and by the NSF under grant CCR-89-18233. The graphs in Chapter 1 were drawn using `dag`[GNV88]. Some of the plumbing fixtures described in Section B.1 of the appendix were suggested in [Ram90]. Georges Lauri was very helpful in getting the bugs out of the first distribution. Cliff Krumvieda, Clément Pellerin and Thomas Yan have provided bug reports and useful feedback about the system and documentation. Cliff Krumvieda has also provided the design of an improved trace facility. And Tim Teitelbaum provided helpful comments on this manual.

Chapter 1

A CML tutorial

This chapter provides information on how to load and run the system, and a tutorial introduction to the concurrency features and common programming techniques of **CML**. We assume familiarity with **SML** (and **SML/NJ**), which provides the sequential core of **CML**. See [Pau91] or [Har86] for an introduction to **SML**.

1.1 Getting started

If **CML** has not been installed on your local system, you will need to load it.[1] To do so, change to the root directory of the **CML** installation; there you will find a file `load-cml`. Start up **SML/NJ**, and load this file:

```
% sml
Standard ML of New Jersey, Version 0.93, February 1, 1993
val it = () : unit
- use "load-cml";
[opening load-cml]
val loadCML = fn : unit -> unit
val loadAll = fn : unit -> unit
[closing load-cml]
val it = () : unit
-
```

Applying the function `loadCML` will load the various pieces of **CML**; the function `loadAll` will load **CML** plus the various library modules (see Appendix B). Loading the system defines a number of top-level structures; in this chapter we will be concerned just with the structures `CML` and `RunCML`. The full signatures of these structures and of the other top-level structures can be found in Appendix A. Note that loading the system does *not* open these structures. Because the `use` function does not yet support relative paths, you need to be in the **CML** root directory for `loadCML` to work. Once you have loaded the system, you can change to your own personal source directory using the function `System.Directory.cd`.

[1] See section 8.2 for information on installing a pre-loaded version of **CML**.

1.2 Basic concurrency primitives

Once you have **CML** loaded, the first thing you need to know is how to make an **ML** program multi-threaded; i.e., how to create new threads. This is easily done using the following function

```
val spawn : (unit -> unit) -> thread_id
```

Applying `spawn` to a function f causes a new thread to be created, which evaluates the application of f to `()`. The newly created thread is called the *child*, and its creator is the *parent*. The return value of `spawn` is a *thread id* that uniquely identifies the child thread. The child thread will run until the evaluation of the function application is complete, at which time it terminates. A thread can also terminate itself by calling the function

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

which never returns (hence the `'a` return type). Another cause of thread termination is an uncaught exception (see Section 7.4).

Since **CML** runs on a single processor in the address space of a single UNIX process, threads must share the CPU. **CML** uses preemptive scheduling, so we don't need to be concerned with insuring that threads are scheduled to run.[2]

Once we have multiple threads, we need a way for our threads to cooperate: i.e., a way to communicate and synchronize. There are a number of different language mechanisms for providing this. **CML** uses *first-class synchronous operations*[Rep88] for synchronization and communication. This mechanism is based on *synchronous* message passing via typed channels. Channels are dynamically created by the function

```
val channel : unit -> '1a chan
```

The result type is *weakly polymorphic* in order to avoid type loop-holes that can be introduced by polymorphic state.[3] Two operations are provided for channel communication:

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

When a thread executes a `send` or `accept` on a channel, we say that it is *offering* communication. The thread will block until some other thread offers a *matching* communication (i.e., the complementary operation on the same channel). When two threads offer matching communications, they exchange the message and both threads continue execution. This is also called *rendezvous* style communication, since two threads must rendezvous to exchange data.

Example 1.1

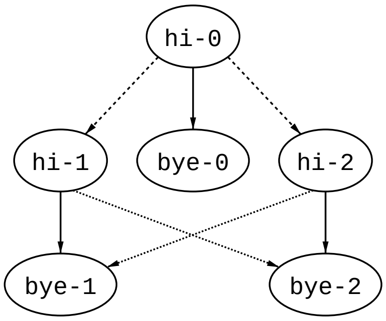
This simple example consists of a root thread that creates a channel `ch` and then spawns two children, which use `ch` to communicate once. All three threads print a message at their beginning and end. Note that the function `CIO.print` is used to print the messages; the structure `CIO` provides an implementation of I/O operations that can be safely used in a **CML** program (see section 5.1).

[2]This is in contrast with coroutine based implementations, where the issue of scheduling rears its ugly head.

[3]This is one of the places in which **SML/NJ** varies from the definition ([MTH90]). The *imperative* types of the definition are more restrictive then the weak types of **SML/NJ**.

```
fun simple_comm () = let
      val ch = channel()
      val pr = CIO.print
      in
        pr "hi-0\n";
        spawn (fn () => (pr "hi-1\n";  send(ch, 17);  pr "bye-1\n"));
        spawn (fn () => (pr "hi-2\n";  accept ch;     pr "bye-2\n"));
        pr "bye-0\n"
      end
```

An outside observer of this program's execution will see the messages "`hi-0`," "`hi-1`," ..., "`bye-2`," printed in some order. The possible orders are characterized by the following graph:



The solid edges are induced by the order of sequential execution, the dashed edges by `spawn`, and the dotted edges by the rendezvous. Even though this program does not involve non-deterministic choice (i.e., from selective communication), it exhibits indeterminacy because of the independence of the threads. Indeterminate behavior is both the bane and boon of concurrent programming. It limits our ability to predict program behavior, but provides flexibility in scheduling independent tasks.

1.3 Running a CML program

CML is currently designed to be used in "batch" mode: you enter a program in the single-threaded top-level environment, and then you run it. Threads should only be spawned inside other functions (never at top-level). To run a **CML** program you use the function

```
val doit : ((unit -> unit) * int option) -> unit
```

from the structure `RunCML`. The first argument is your *root* thread, which is a function that creates the other threads of your program. The second argument specifies the size of the scheduling time-slice in milliseconds. If `NONE` is supplied as the time-slice value, then preemptive scheduling is disabled, but, since various system services depend on preemption to work correctly, it is not recommended.

Example 1.2

For example, let us look at the steps required to run the program given in Example 1.1. Assume that we are in the root directory of the **CML** distribution and are running **CML**. Then we can load and run the example as follows:

```
- open CML;
open CML
- use "examples/ex-simple-comm.sml";
[opening examples/ex-simple-comm.sml]
val simple_comm = fn : unit -> unit
[closing examples/ex-simple-comm.sml]
val it = () : unit
- RunCML.doit(simple_comm, SOME 20);
hi-0
hi-1
hi-2
bye-0
bye-2
bye-1
val it = () : unit
-
```

In this example, we first open the `CML` structure to provide the right environment for compiling the example; then we load the example file and run it. The "`SOME 20`" argument to `doit` specifies a time-slice of 20 milli-seconds.

Termination in concurrent programs is a more involved process than in sequential programs. A **CML** program will terminate if it deadlocks, i.e., all of its threads are blocked (threads waiting for input or timeouts are not considered blocked). In the case of Example 1.2, all of the user threads terminate and the system threads are all blocked, so the whole system terminates. In general, it is necessary to terminate under software control, since it is often the case that a program has some threads that will always have the potential of running (e.g., monitoring an input stream). The function `RunCML.shutdown` forces a clean shutdown of the system. The are a number of issues related to the initialization and clean termination of **CML** programs that we have not discussed here. See Chapter 6 for details.

It is also possible to kill a **CML** program by hitting the break key (e.g. control-C), which should return control to the **SML** top loop.[4] If, for some reason, this does not terminate **CML**, you should use the UNIX commands `ps`(1) and `kill`(1) to kill the program.

1.4 Streams

One use of processes and channels is for stream (or data-flow) style programming[ASS85]. Streams can be viewed as infinite lists; for example

$$\textbf{Nat} = f(0) \text{ where } f(n) = n \cdot f(n+1)$$

defines the stream (or list) of natural numbers (where "·" is lazy cons). This definition can be implemented as follows:

```
fun makeNatStream () = let
      val ch = channel()
      fun f i = (send(ch, i); f(i+1))
      in
        spawn (fn () => f 0);
        ch
      end
```

[4] If the garbage collector is running, then termination is postponed until after it finishes.

where the function f is represented by a thread, and the stream is represented by an integer valued channel. This style of programming has been used to implement computations with power series in the language **newsqueak**[McI90].

Example 1.3 (Sieve of Eratosthenes)

As an example of this style of programming, we will implement the *Sieve of Eratosthenes* for computing prime numbers. To start with, we need the stream of natural numbers greater than one, which is provided by the following generalization of the natural number stream:

```
fun counter start = let
      val ch = channel()
      fun count i = (send(ch, i); count(i+1))
      in
        spawn (fn () => count start);
        ch
      end
```

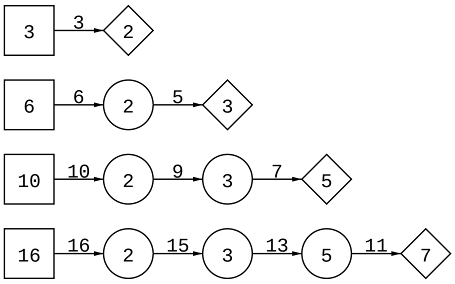
The function `sieve` produces a new stream of prime numbers, which is represented by a chain of threads connected by channels:

```
fun sieve () = let
      val primes = channel ()
      fun filter (p, inCh) = let
        val outCh = channel()
        fun loop () = let val i = accept inCh
              in
                if ((i mod p) <> 0) then send (outCh, i) else ();
                loop ()
              end
        in
          spawn loop;
          outCh
        end
      fun head ch = let val p = accept ch
            in
              send (primes, p);
              head (filter (p, ch))
            end
      in
        spawn (fn () => head (counter 2));
        primes
      end
```

Looking at the implementation of `sieve`, we see that the chain of threads starts with the `counter` thread and ends with the `head` thread. There is one `filter` thread in the chain for each prime number discovered so far; the `head` thread creates new filters as new primes are found. The function `primes` prints the first `n` prime numbers.

```
fun primes n = let
      val ch = sieve ()
      fun loop 0 = ()
        | loop i = (CIO.print(makestring(accept ch)^"\n"); loop(i-1))
      in
        loop n
      end
```

The following picture shows the state of the sieve at each iteration of evaluating `primes 4`:



The `counter` thread is represented as a square labeled by the current counter value. The filters are circles labeled by their prime number, and the `head` is represented by a diamond labeled with the next prime. The numbers labeling the arrows between the threads are the values that the threads on the left are offering the threads on the right.

Note that in this example there is no mechanism for capturing an intermediate value or state of the stream (the way one can do with a lazy infinite list).

1.5 First-class synchronous operations

Thread creation and message passing provide a base for concurrent programming, but we need more to write real applications. It is necessary for threads to be able to manage communication with multiple partners. One common way to provide this ability is the *select* operation, which allows a thread to offer multiple communications. The first communication that is matched is selected. If more than one communication is matched, then a non-deterministic choice is made. **CML** provides the select mechanism as part of the more general mechanism of *first-class synchronous operations*.

The basic idea is to treat synchronous operations, such as channel I/O, as values. These values may then be combined with other values to form new synchronization and communication abstractions. We start with a new type constructor

```
type 'a event
```

If a synchronous operation has the type $\tau \rightarrow \tau'$, then the event-valued version will have the type $\tau \rightarrow \tau'$ `event`. For example, the basic channel I/O operations are built by:

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

To synchronize on an event value, we have the function

```
val sync : 'a event -> 'a
```

We can draw an analogy with first-class function values in which the `event` type constructor plays the role of `->` and `sync` plays the role of function application. Returning to the channel I/O operations, we can redefine them in terms of events as:

```
val accept = sync o receive
val send   = sync o transmit
```

From these base event values we can build new values by *wrapping* them with post-synchronization actions. For example, we can define a string valued interface to a boolean valued channel, `ch`, by

```
wrap (
  receive ch,
  fn true => "yes" | false => "no")
```

Synchronizing on this value will read a value from `ch` and map it to either `"yes"` or `"no"`. As one might expect, the `wrap` function has the type

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

A wrapper function may also contain synchronous operations. For example, a very common paradigm in concurrent programs is a request/result exchange (or *remote procedure call* (RPC)) with a server thread. The client side of this exchange can be wrapped up into a single synchronous event:

```
wrap(transmit reqCh, fn () => accept resCh)
```

where `reqCh` and `resCh` are the request and result channels, respectively (section 1.8 discusses these kinds of protocols in greater detail).

The primitives that we have examined so far are sufficient for a small example of building new abstractions.

Example 1.4 (Futures)

A **Multi-lisp** *future* is the parallel evaluation of an expression[Hal85]. The result is acquired by *touching* the future; the touching thread may have to wait for the future computation to complete. Since touching a future is a synchronous operation, we will represent futures as events. The `future` operation has the type:

```
val future : ('a -> '2b) -> 'a -> '2b event
```

and `sync` is the touch operator. The implementation of `future` is straight-forward: we spawn a new thread to evaluate the application and create a channel for reporting the result.

```
fun future f x = let
      datatype 'a msg_t = RESULT of 'a | EXN of exn
      val resCh = channel()
      fun repeater x = (send(resCh, x); repeater x)
      in
        spawn (fn () => repeater(RESULT(f x) handle ex => EXN ex));
        wrap (
          receive resCh,
          fn (RESULT x) => x | (EXN ex) => raise ex)
      end
```

Since the evaluation of a future might result in a raised exception, the result channel (`resCh`) can carry either the result, or an exception. A future value may be touched several times, so we use the tail-recursive function `repeater` to repeatedly send the result (or exception) message on the result channel.[5] This might seem to pose a problem with resource reclamation, but if the future event value becomes garbage, then the garbage collector will also reclaim the channel and thread. In general, any process that communicates infinitely often will be garbage collected if it becomes disconnected from the rest of the system.[6] This is an important property, which we often exploit.

The event-value produced by the `future` abstraction may cause an exception to be raised when a thread synchronizes on it. We can hide this behavior by wrapping an exception handler around the event:

```
(* myFuture : ('a -> '2b) -> 'a -> '2b option event *)
  fun myFuture f x = wrapHandler (
        wrap (future f x, SOME),
        fn _ => NONE)
```

This new operation will return `NONE` in the situation where an exception was raised. The `wrapHandler` function has the type

```
  val wrapHandler : 'a event * (exn -> 'a) -> 'a event
```

If synchronizing on the first argument raises an exception, then the exception will be caught and passed to the second argument.

The future abstraction in Example 1.4 has the same status as the built-in abstractions, since it has an event-valued interface. The synchronous character of the abstraction has not been hidden (although the details of the implementation have been). For the primitives we have seen so far, we could have used function abstraction, instead of events, to produce the same results. But with selective communication, we need to have the synchronous character of our abstractions exposed – function abstraction hides too much.

CML provides selective communication via the `choose` combinator, which builds an event value that is the "*non-deterministic or*" of a homogeneously typed list of event values. As expected, `choose` has the type

```
val choose : 'a event list -> 'a event
```

As we saw in Example 1.1, there is non-determinism in the order in which parallel actions occur; with `choose` there is also non-determinism in which actions occur.

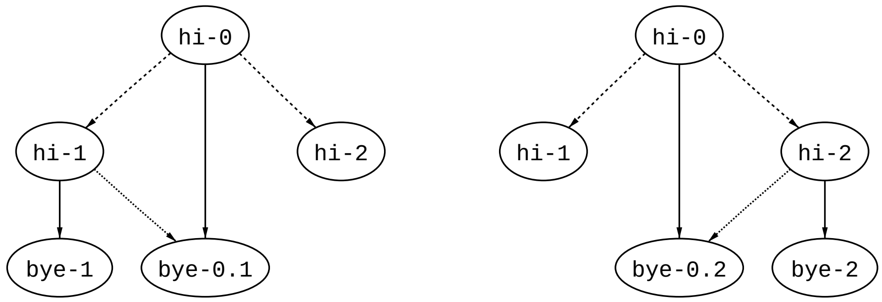
Example 1.5

To make this concrete, consider the following variation on Example 1.1:

[5] A better way to do this is to use a *write-once* condition variable (see Chapter 4).
[6] There are a couple of technical exceptions to this, but they tend not to arise in practice.

```
fun simple_comm2 () = let
      val ch1 = channel() and ch2 = channel()
      val pr = CIO.print
      in
        pr "hi-0\n";
        spawn (fn () => (pr "hi-1\n";  send(ch1, 17);  pr "bye-1\n"));
        spawn (fn () => (pr "hi-2\n";  send(ch2, 37);  pr "bye-2\n"));
        sync (choose [
          wrap (receive ch1, fn _ => pr "bye-0.1\n"),
          wrap (receive ch2, fn _ => pr "bye-0.2\n")
        ])
      end
```

The observational behavior of this program fits one of two graphs, depending on the non-deterministic choice made by the parent thread:



This example also illustrates a very common **CML** idiom: that of synchronizing on a choice of events. To support this form, **CML** provides the `select` operation for synchronizing on a list of event values. It is a derived form with the following definition:

```
val select = sync o choose
```

1.6 More synchronous operations

In addition to channel communication, the event framework supports other synchronous conditions in a natural way. A thread can synchronize on another threads termination using the function:

```
val threadWait : thread_id -> unit event
```

This mechanism is useful for when a server needs to handle the case of premature termination by one of its clients.

Example 1.6

The `threadWait` event can be used by servers that need to be notified of the termination of a client. For example, consider a *token server* with the following interface:

```
signature TOKEN_SERVER =
  sig
    structure CML : CONCUR_ML
    type ('a, 'b) token
    val newToken : ('a -> 'b) -> ('a, 'b) token
    exception NotTokenHolder
    val getOperation : ('a, 'b) token -> ('a -> 'b)
    val releaseToken : ('a, 'b) token -> unit
    val acquireToken : ('a, 'b) token -> unit CML.event
  end (* TOKEN_SERVER *)
```

This server implements a facility for controlling access to an operation. Associated with the operation is a *token*. While multiple threads may have access to the token value, only one thread actually *possesses* it at any given time, and only the possessor may do the operation. The function `newToken` produces a new token for controlling access to a given operation. Given a token, the function `getOperation` extracts the associated operation. If an attempt to use this operation is made by a thread other than the current token holder, then the exception `NotTokenHolder` will be raised. The functions `acquireToken` and `releaseToken` are used to change token holders. Note that a thread can synchronize on receiving possession of the token. Figure 1.1 contains the implementation of this server. The token server can be in one of two states: either the token is currently held by some thread (`heldLoop`) or the token is available (`availLoop`). If the current token holder terminates, it is necessary for the token server to reclaim the token. The `threadWait` event in `heldLoop` provides notification of this situation.

Another kind of synchronous operation is synchronizing on the clock. **CML** supports this with the function

```
datatype time = TIME of sec : int, usec : int
val timeout : time -> unit event
```

which produces a value for synchronizing on a real-time delay. This could be used by the above token server, for example, to reclaim the token after a fixed period during which the token is unused by the possessor.

CML also supports I/O operations as synchronous operations. For example, the event valued function `CIO.inputLineEvt` allows a thread to synchronize on the availability of a line of input from a stream. Chapter 5 describes **CML**'s I/O support in greater detail.

1.7 Building new synchronous abstractions

The key motivation for events is to provide a mechanism for the user to build new synchronous abstractions. In this section we describe the implementation of a couple of non-trivial abstractions, which have found use in real applications.

Example 1.7 (Buffered channels)

Some concurrent languages, such as *actor* languages[Agh86], use asynchronous message passing for process communication. One way to support this is by implementing *buffered channels*. We need operations to create new channels and to send and receive messages. Since the channels are buffered, the send operation is asynchronous. The receive operation is synchronous, so we provide an event-valued interface to it. The following signature defines the interface:

```
structure TokenServer : TOKEN_SERVER =
  struct
    structure CML = CML
    open CML

    datatype ('a, 'b) token = TOKEN of {
        operation : 'a -> 'b,              (* the protected operation *)
        acquire_ch : thread_id chan,       (* the channel for requesting the token *)
        check : unit -> unit,              (* check for token possession *)
        release : unit -> unit              (* release the token *)
      }

    exception NotTokenHolder
    fun newToken operFn = let
          val acqCh = channel() and relCh = channel() and holdCh = channel()
          fun server () = let
                val acquireEvt = receive acqCh
                val releaseEvt = receive relCh
                val myId = getTid()
                fun heldLoop curHolder = select [
                        wrap (choose [releaseEvt, threadWait curHolder],
                          fn () => availLoop ()),
                        wrap (transmit(holdCh, curHolder),
                          fn () => heldLoop curHolder)
                      ]
                and availLoop () = select [
                        wrap (acquireEvt, fn id => heldLoop id),
                        wrap (transmit(holdCh, myId), fn () => availLoop())
                      ]
                in
                  availLoop ()
                end
          fun checkFn () = if sameThread(getTid(), accept holdCh)
                  then () else raise NotTokenHolder
          in
            spawn server;
            TOKEN{
                operation = fn x => (checkFn(); operFn x),
                acquire_ch = acqCh,
                check = checkFn,
                release = fn () => send(relCh, ())
              }
          end

    fun getOperation (TOKEN{check, operation, ...}) = (check(); operation)
    fun releaseToken (TOKEN{check, release, ...}) = (check(); release())
    fun acquireToken (TOKEN{acquire_ch, ...}) = transmit(acquire_ch, getTid())

  end (* TokenServer *)
```

Figure 1.1: Token server implementation

```
signature BUFFER_CHAN =
  sig
    structure CML : CONCUR_ML
    type 'a buffer_chan
    val buffer : unit -> '1a buffer_chan
    val bufferSend : ('a buffer_chan * 'a) -> unit
    val bufferAccept : 'a buffer_chan -> 'a
    val bufferReceive : 'a buffer_chan -> 'a CML.event
  end (* BUFFER_CHAN *)
```

To implement the buffer queue, we use an infinitely looping thread. This thread maintains the queue as a pair of stacks (implemented as lists). When the buffer is empty, then the buffer thread only accepts input; when there is stuff in the buffer, then it also offers the front of the queue as output. The following functor implements this scheme:

```
functor BufferChan (CML : CONCUR_ML) : BUFFER_CHAN =
  struct
    structure CML = CML

    open CML

    datatype 'a buffer_chan = BC of {inch : 'a chan, outch : 'a chan}

    fun buffer () = let
          val inCh = channel() and outCh = channel()
          fun loop ([], []) = loop([accept inCh], [])
            | loop (front as (x::r), rear) = select [
                  wrap (receive inCh, fn y => loop(front, y::rear)),
                  wrap (transmit(outCh, x), fn () => loop(r, rear))
                ]
            | loop ([], rear) = loop(List.rev rear, [])
          in
            spawn (fn () => loop([], []));
            BC{inch=inCh, outch=outCh}
          end

    fun bufferSend (BC{inch, ...}, x) = send(inch, x)
    fun bufferAccept (BC{outch, ...}) = accept outch
    fun bufferReceive (BC{outch, ...}) = receive outch

  end (* functor BufferChan *)
```

Asynchronous communication can also be implemented by spawning a new thread to send the message:

```
fun asyncSend (ch, msg) = (spawn (fn () => send (ch, msg)); ())
```

This essentially uses the new thread as a cell in the buffer queue. This technique is most often used for "single-shot" communications, since the order of messages may not be preserved. When using either of these techniques, it is important to avoid situations in which messages are produced much more rapidly than they are consumed. If such a discrepancy occurs for a sustained period, substantial memory resources will be consumed by the buffers, which will degrade performance.

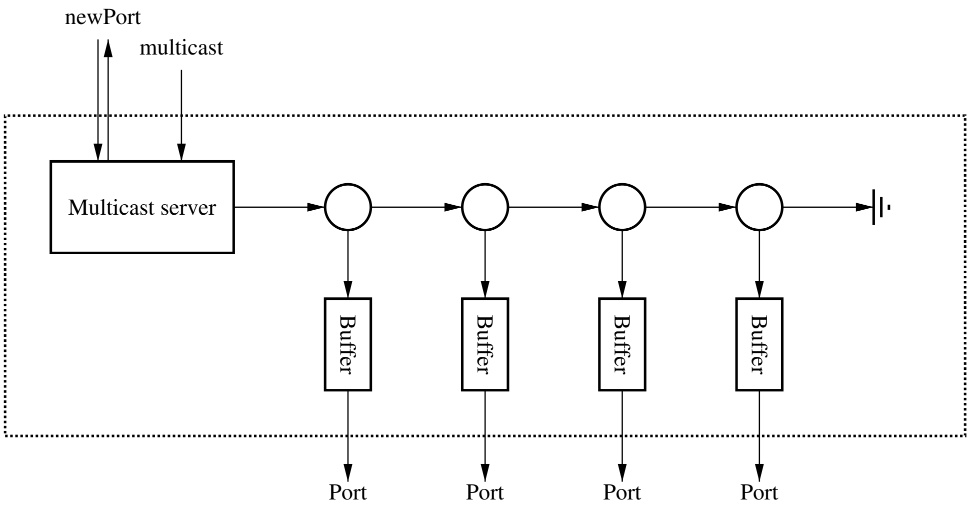
Example 1.8 (Multicast)

Another useful example is the implementation of *buffered multicast channels*, which build on the buffered channels by providing *fan-out*. A multicast channel has a number of *output ports*. When a thread sends a message on a multicast channel, it is replicated once for each output port. In addition to the standard channel operations (create, send and accept), we need an operation to create new ports. The following signature gives the multicast interface:

```
signature MULTICAST =
  sig
    structure CML : CONCUR_ML
    type 'a mchan
    val mChannel : unit -> '1a mchan
    val newPort : 'a mchan -> 'a CML.event
    val multicast : ('a mchan * 'a) -> unit
  end (* MULTICAST *)
```

New multicast channels are created using `mChannel` and new ports using `newPort`. The `multicast` operation asynchronously broadcasts a message to the ports of a multicast channel. A port is represented by an event value; synchronizing on a port event will return the next multicast message.

A multicast channel consists of a server thread, which initiates the broadcast and creates new ports and a chain of ports. Each port consists of a buffer and a "*tee*" thread that inserts the incoming message in the buffer and propagates it to the next port. The port buffer is implemented using the library version of the buffered channels from Example 1.7 (see appendix section B.2). The following picture gives a schematic view of a multicast channel with four ports:



The source code for the `Multicast` structure is given in figure 1.2. All of the interesting stuff is in the function `mChannel`, which spawns a new server thread. A multicast channel value is a request/reply channel pair, which provides an interface to the server thread. A request is either a message to be broadcast, or a request for a new port. The interface between the server thread and the first port in the chain and the interface between a tee thread and the next port is an output function. The output function at the end of the chain is a sink.

1.8 Client-server protocols

The client-server paradigm is a very important model for structuring **CML** programs. Servers provide a mechanism for mediating access to shared state (the token server in Example 1.6 is one example of this). In general, clients communicate with the server using a request-reply protocol (also called a a *remote procedure call* (RPC), although the server and client are in the same name space). One way to implement such a protocol was presented on page 9, where `transmit` was used to send the request and a wrapper was used to wait for the reply. This approach uses the server's willingness to accept a request as the event to synchronize on, but, for some services, we need to synchronize on the reply. Thus we need a way to include communications to be done prior to the synchronization point of an event value (the dual of `wrap`, which includes communications after the synchronization point).

The `guard` combinator provides this mechanism. It is essentially a "delay" operator, which wraps a event-valued function (or *thunk*) as an event value. The `sync` operator plays the role of force, evaluating the guard prior to synchronizing on the resulting event value. It has the type:

```
val guard : (unit -> 'a event) -> 'a event
```

The `guard` operation allows the abstract implementation of a protocol in which the client first sends a request to the server.

Example 1.9 (Clock server)

As an example, assume that we want to build a clock server to implement an event constructor

```
val waitUntil : time -> unit event
```

which returns an event for synchronizing on the specified time (in fact, `waitUntil` is provided by **CML** as a built-in operation; see Chapter 3). The basic client-server protocol is for the client to send a request to the clock server, which includes the wake-up time and a fresh reply channel; when the wake-up time arrives, the server sends a message on the reply channel. The reply channel serves as a unique identifier for the client's request. The client-side code is:

```
fun waitUntil t = guard (fn () => let
      val replyCh = channel()
      in
        spawn (fn () => send (timerReqCh, (t, replyCh)));
        receive replyCh
      end
```

where `timerReqCh` is a global channel for communicating with the timer server.[7] The server is built using a couple of functions provided by the **SML/NJ** structure `System.Timer`: `earlier`, for testing the order of two times and `sub_time`, for computing the time difference between two times. We also assume the existence of a function `gettimeofday`, for getting the current time of day (this mechanism exists in **SML/NJ**, but it is not easily accessed). The clock server is implemented as:

[7] Using a top-level channel like `timerReqCh` introduces some initialization requirements; see Chapter 6 for details.

```
fun server [] = server [accept timerReqCh]
  | server (waitingList as ((nextTim, _)::rest)) = let
      fun insert (x as (t, _), []) = [x]
        | insert (x as (t, _), (y as (t', _))::r) =
            if earlier(t, t') then x::y::r else y::(insert(x, r))
      fun wakeup () = let
            val now = gettimeofday()
            fun wake [] = []
              | wake (waitingList as ((t, replyCh)::r)) =
                  if earlier (now, t)
                    then waitingList
                    else (spawn (fn () => send (replyCh, ())); wake r)
            in
              wake waitingList
            end
      in
        select [
            wrap (timeout (sub_time (nextTim, gettimeofday())),
              fn () => server (wakeup ())),
            wrap (receive timerReqCh,
              fn req => server (insert (req, waitingList)))
          ]
      end
```

The server maintains an ordered list of pending wake-up requests. The function `insert` adds a request to the list. When it has pending requests in its queue, it computes the delay to the next wake-up and uses `timeout` to wait for it. The function `wakeup` removes those elements of the list that are ready for wake-up messages. For each wake-up message a new thread is spawned to send it; this avoids problems in the case that the client uses `waitUntil` as part of a choice and a different event is selected.

The clock server example has the property that the server is *idempotent*; i.e., that the handling of a given request is independent of other requests. Thus, using a new thread to send a reply is sufficient to protect the server against the situation in which the client selects another event in a selection. For some services, however, this is not sufficient; the server needs to know whether to *commit* or *abort* the transaction. The following combinator provides such a mechanism:

```
val wrapAbort : ('a event * (unit -> unit)) -> 'a event
```

It associates an abort action with an event. If the resulting event is involved in a selective synchronization and another event is chosen, then a thread is spawned to evaluate the abort action.

Example 1.10 (Input line event)

For example, consider the implementation of a buffered input stream such as is provided by the `CIO` structure. The abstraction should provide event valued operations for reading both single characters and complete lines of input:

```
val inputCharEvt : instream -> string event
val inputLineEvt : instream -> string event
```

The basic protocol is that the client sends a request to the server, which then requests input from the operating system. Once input is available, the server sends it to the client. The following code implements the client side of the `inputLineEvt` function.[8]

```
datatype input_req = INPUTLN of (string chan * unit chan) | ···

fun inputLineEvt (INSTRMreq_ch, ...) = guard (fn () => let
      val replyCh = channel () and abortCh = channel ()
      in
        spawn (fn () => send (req_ch, INPUTLN(replyCh, abortCh)));
        wrapAbort (receive replyCh, fn () => send (abortCh, ()))
      end)
```

The client's request consists of the desired operation (`INPUTLN`), a reply channel and an abort channel. Note that the request is sent asynchronously; this is done to avoid blocking the client in the case that the server is busy. When the server has the input necessary to satisfy the request, it synchronizes on the choice of receiving the abort notification or having its reply accepted. If the reply is accepted, then the server commits the transaction (i.e., discards the input). On the other hand, if an abort message is received, then the server aborts the transaction and buffers the input for the next request.

[8]The server side code is too involved to present here; the interested reader is referred to the source code of the `CIO` structure.

```
functor Multicast (BC : BUFFER_CHAN) : MULTICAST =
  struct
    structure CML = BC.CML

    open CML

    datatype 'a mchan = MChan of ('a request chan * 'a event chan)
         and 'a request = Message of 'a | NewPort

    fun mChannel () = let
          val reqCh = channel() and respCh = channel()
          fun mkPort outFn = let
                val buf = BC.buffer()
                val inCh = channel()
                fun tee () = let val m = accept inCh
                      in
                        BC.bufferSend(buf, m);
                        outFn m;
                        tee()
                      end
                in
                  spawn tee;
                  (fn m => send(inCh, m), BC.bufferReceive buf)
                end
          fun server outFn = let
                fun handleReq NewPort = let val (outFn', port) = mkPort outFn
                      in
                        send (respCh, port);
                        outFn'
                      end
                  | handleReq (Message m) = (outFn m; outFn)
                in
                  server (sync (wrap (receive reqCh, handleReq)))
                end
          in
            spawn (fn () => server (fn _ => ()));
            MChan(reqCh, respCh)
          end

    fun newPort (MChan(reqCh, respCh)) = (send (reqCh, NewPort); accept respCh)

    fun multicast (MChan(ch, _), m) = send (ch, Message m)

  end (* Multicast *)
```

Figure 1.2: Multicast implementation

Chapter 2

Basic concurrency primitives

CML is based on a distributed-memory model[1] with synchronous message passing on typed channels. These basic concurrency operations are defined in the structure `CML` and are described in this chapter. The next chapter describes the rest of the `CML` structure, which extends the basic concurrency model with first-class synchronous operations.

2.1 Threads

A **CML** program consists of a collection of threads running (logically at least) in parallel. Scheduling of threads is preemptive, so threads should not share mutable data. The operations on threads are given in figure 2.1. Threads are dynamically created by the `spawn` function, which returns the unique thread identifier

```
type thread_id
val spawn : (unit -> unit) -> thread_id
val exit : unit -> 'a
val getTid : unit -> thread_id
val sameThread : (thread_id * thread_id) -> bool
val tidLessThan : (thread_id * thread_id) -> bool
val tidToString : thread_id -> string
val yield : unit -> unit
```

Figure 2.1: Thread operations

(`thread_id`) of the new thread. Thread ids can be tested for equality using the `sameThread` function (`thread_id` is not an equality type). Certain multi-thread protocols (e.g., [Bor86]) require some method to avoid cyclic dependencies that can lead to deadlock. For this reason, an abstract ordering is defined on `thread_id` values; the function `tidLessThan` tests this order. A string representation of a thread id can be created using `tidToString`. A thread can choose to relinquish the CPU by calling the `yield` function, which forces a context switch, but, since scheduling is preemptive, this should rarely be needed.[2] A thread id can also be used to implement a *process join* using the `threadWait` function described in the next chapter.

[1]Note, however, that the implementation is a shared-memory model.

[2]The `yield` function may go away in a future release.

2.2 Channels

Synchronous communication of typed channels is the basis of **CML**'s communication and synchronization mechanism. Figure 2.2 gives the basic operations on channels. New channels are dynamically created using

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

Figure 2.2: Channel operations

the `channel` function, which is a weakly polymorphic function (like the `ref` constructor). Two channels can be tested to see if they are the same channel using the predicate `sameChannel`. A thread can read a message from a channel using the `accept` function, and send one using `send`. Message passing is synchronous, so when a thread sends a message on a channel, it must wait until another thread is ready to accept a message from that channel.

Chapter 3

Events

CML extends the basic synchronous message passing of the previous chapter by making synchronous operations first-class values[Rep88]. These values, called *events*, are representations of synchronous operations (much the same way that function values represent computations). There are various functions for creating the basic event values, as well as combinators for producing more complex values. Figure 3.1 gives the signature of the event operations defined in the `CML` structure. The rest of this chapter will discuss these operations.

```
val sync   : 'a event -> 'a
val select : 'a event list -> 'a
val poll   : 'a event -> 'a option

val choose      : 'a event list -> 'a event
val wrap        : ('a event * ('a -> 'b)) -> 'b event
val wrapHandler : ('a event * (exn -> 'a)) -> 'a event
val guard       : (unit -> 'a event) -> 'a event
val wrapAbort   : ('a event * (unit -> unit)) -> 'a event

val always     : 'a -> 'a event
val transmit   : ('a chan * 'a) -> unit event
val receive    : 'a chan -> 'a event
val threadWait : thread_id -> unit event
val timeout    : time -> unit event
val waitUntil  : time -> unit event
```

Figure 3.1: Event operations

3.1 Simple events

An event value describes a potential synchronous operation. A thread can synchronize on an event value by applying the `sync` operation. The message passing model of the previous section provides the core of the event mechanism. The functions `receive` and `transmit` are used to build event values that describe channel I/O operations. These can be used, for example, to define the operations `accept` and `send`:

```
val accept = sync o receive
val send   = sync o transmit
```

The event values built by `receive` and `transmit` are called *base event* values. Another base event constructor is `always`, which produces an event that is always immediately available for synchronization and produces the argument fed to `always`. The base event constructor `waitUntil` provides a mechanism for synchronizing on the termination of another thread.

3.2 Wrappers

One method of building new event values from base events is to wrap a post-synchronization action around the event. For example, we can turn a channel of integers into a channel, `ch`, of squares by

```
wrap (receive ch, fn (x : int) => x*x)
```

When synchronization occurs on this event, the value read from `ch` will be fed to the function and its square will be returned as the synchronization result.

It is also possible to wrap an exception handler around an event. Building on our previous example, consider the situation in which "`x*x`" causes an integer overflow. The following event will return zero in that case:

```
wrapHandler (
  wrap (receive ch, fn (x : int) => x*x),
  fn Overflow => 0)
```

The function that is used as an exception handler should cover all of the possible exceptions; otherwise a `Match` exception can occur when applying the handler.

3.3 Selective communication

Selective communication is necessary for threads to manage deadlock-free communication with multiple partners. **CML** supports selective communication in an extremely general way via the `choose` operator. This constructs the nondeterministic choice of a list of events. For example, a thread can monitor input from two channels (of the same type) as follows:

```
choose [
    receive ch1,
    receive ch2
  ]
```

Since the thread is probably interested in which channel a message is from, wrappers can be added to tag the values:

```
choose [
    wrap (receive ch1, fn x => (1, x)),
    wrap (receive ch2, fn x => (2, x))
  ]
```

We can mix both input and output operations in the same `choose` expression. This raises the question of what happens if a thread attempts both input and output on the same channel at the same time? For example:

```
choose [
    wrap (transmit(ch, 1), fn () => false),
    wrap (receive ch, fn _ => true)
  ]
```

In **CML** synchronizing on this event value will block the thread until another thread either sends or accepts a message on `ch`. The thread does not communicate with itself. If a thread synchronizes on the empty choice (i.e., `choose[]`), then it will block forever (actually it will be garbage collected).

The operation `select` provides a useful short-hand for a common **CML** idiom. It is defined as

```
val select = sync o choose
```

3.4 Advanced event programming

The `guard` and `wrapAbort` combinators are used for implementing complex client-server protocols. The `guard` function acts as a delay operator when applied to an event valued function; when `sync` is applied to the guarded event, the delayed function is evaluated, and its result is used for synchronization. As a simple example of `guard`, a conditional event constructor can be implemented:

```
fun condEvt (pred, evt) = guard (fn () => if pred() then evt else choose[]);
```

This takes a boolean valued predicate and an event value; when `sync` is applied to the guarded event, the predicate is evaluated and, if true, then `evt` is returned, otherwise the null event is returned. A more typical use of `guard` is to initiate a transaction with a server while returning an event to synchronize on the transaction's completion; section 1.8 gives several examples of this.

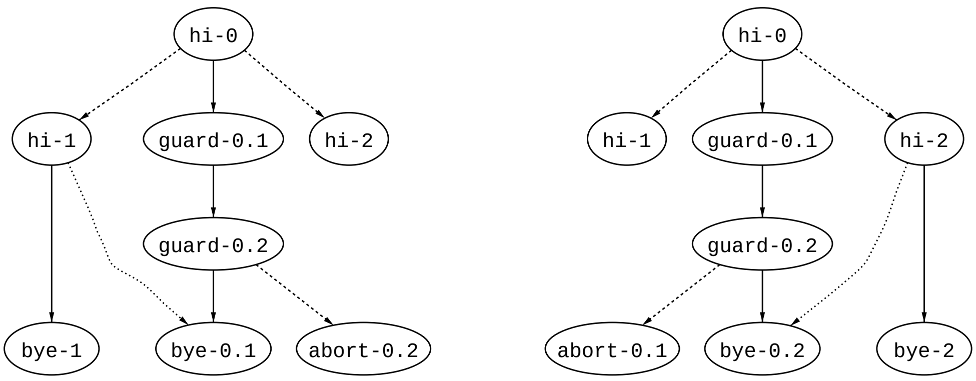
Sometimes, when `guard` is being used to initiate a transaction, it is necessary for the server to be notified if the transaction will not complete (i.e., because a different event in a select communication was selected). The `wrapAbort` combinator provides this mechanism. It associates a *abort* function with an event value. When the event value is involved in a selective communication, if a different event is chosen, then a thread is spawned to evaluate the abort function.

Example 3.1

We can illustrate the semantics of `guard` and `wrapAbort` with the following variation of Example 1.5.

```
fun simple_comm3 () = let
      val ch1 = channel() and ch2 = channel()
      val pr = CIO.print
      in
        pr "hi-0\n";
        spawn (fn () => (pr "hi-1\n";  send(ch1, 17);  pr "bye-1\n"));
        spawn (fn () => (pr "hi-2\n";  send(ch2, 37);  pr "bye-2\n"));
        select [
          guard (fn () => (
            pr "guard-0.1\n";
            wrapAbort (wrap (receive ch1, fn _ => pr "bye-0.1\n"),
              fn () => pr "abort-0.1\n"))),
          guard (fn () => (
            pr "guard-0.2\n";
            wrapAbort (wrap (receive ch2, fn _ => pr "bye-0.2\n"),
              fn () => pr "abort-0.2\n")))
        ]
      end
```

The observational behavior of this program fits one of two graphs, depending on the non-deterministic choice made by the parent thread:



These graphs illustrate the complementary relationship between the abort actions and wrapper functions. Note that the guards are always evaluated in left-to-right order, although depending on this is bad style.

3.5 Polling and timeouts

While selective communication provides a great deal of flexibility in scheduling thread communication, it is sometimes useful to use polling. **CML** provides two mechanisms for polling event values.

The operation `poll` is a non-blocking form of `sync`; it returns `NONE` instead of blocking.[1] For example, the expression

```
poll (receive ch)
```

will return `SOME` *msg*, if a message *msg* is available on the channel, otherwise it returns `NONE`. The expression

```
poll (transmit (ch, msg))
```

[1]In version 0.9.3 and earlier, `poll` was an event value constructor.

is a *conditional* send, which returns `SOME()` if successful. It is similar to **Ada**'s conditional entry call.

Another way to implement polling behavior is by using timeouts. This is most often useful when dealing with the external world. The expression `timeout` t returns an event that will be available for synchronization approximately t time units after `sync` is called. The argument to `timeout` is a value of the type

```
datatype time = TIME of {sec : int, usec : int}
```

Note that timeouts are measured in real-time (not cpu-time).[2] There is also a mechanism for synchronizing on an absolute time. The function `waitUntil` returns an event value that will be available for synchronization at the specified time. The `timeout` function can be implemented as

```
fun timeout t = guard (fn () => waitUntil (addTime (t, currentTime())))
```

where `currentTime` returns the current time of day.

It is worth comparing the semantics of `poll` and `timeout` because there are some subtle differences. Consider, for example the function

```
fun poll' evt = select [
        wrap (timeout(TIME{sec=0, usec=0}), fn () => NONE),
        wrap (evt, SOME)
      ]
```

one might consider it equivalent to `poll`, but there is a difference when the argument event is immediately available for synchronization. In the case of `poll`, the answer will always be `(SOME` v`)` (assuming the result of `evt` is v), while in the case of `poll'`, the answer is the non-deterministic choice of `NONE` and `(SOME` v`)`. While the `poll` operation guarantees that an immediately available synchronization result will be returned, it may be less useful than non-zero timeouts in practice. When the event value involves guards (e.g., a request/reply protocol), then a poll result of `NONE` is not be very informative.

[2]Because of scheduling uncertainties and garbage collection, timeouts may delay for more than the requested amount of time.

Chapter 4

Condition variables

Although **CML** is currently only implemented on single processor machines, we hope to have a shared-memory multi-processor implementation available in the future, which raises the question of the effectiveness of **CML** as a parallel programming language. Condition variables are an experimental feature for supporting parallel programming. Figure 4.1 gives the signature of the condition variable operations. A condition

```
type 'a cond_var

val condVar : unit -> '1a cond_var

val writeVar : ('a cond_var * 'a) -> unit
exception WriteTwice

val readVar : 'a cond_var -> 'a
val readVarEvt : 'a cond_var -> 'a event
```

Figure 4.1: Condition variable operations

variable is essentially a "write-once" shared variable, which one can synchronize on. The semantics of condition variables are defined in terms of an implementation using channels (see Figure 4.2). The actual implementation is more efficient.

Example 4.1 (Futures)

As an example of how condition variables can be used to support parallel programming, we can implement futures with them (also see Example 1.4).

```
fun future f x = let
      datatype 'a msg_t = RESULT of 'a | EXN of exn
      val result = condVar()
      in
        spawn (fn () => writeVar (result, RESULT(f x) handle ex => EXN ex));
        wrap (
          readVarEvt result,
          fn (RESULT x) => x | (EXN ex) => raise ex)
      end
```

```
abstype 'a cond_var = CV of {req_ch : 'a chan, reply_ch : 'a chan}
with
  fun condVar () = let
        val reqCh = channel() and replyCh = channel()
        fun condvar () = let
              val v = accept reqCh
              fun loop () = (send(replyCh, v);  loop())
              in
                loop ()
              end
        in
          spawn condvar;
          CV{req_ch = reqCh, reply_ch = replyCh}
        end

  exception WriteTwice
  fun writeVar (CV{req_ch, reply_ch}, v) = select [
          transmit(req_ch, v),
          wrap (receive reply_ch, fn _ => raise WriteTwice)
        ]

  fun readVar (CV{reply_ch, ...}) = accept reply_ch
  fun readVarEvt (CV{reply_ch, ...}) = receive reply_ch
end
```

Figure 4.2: Condition variable implementation

The major advantage of this implementation of futures over the one given in Chapter 1, is that once the answer is available, no message-passing or context switches are involved in reading the result.

Condition variables can also be used wherever "single-shot" communication is required. For example, a common style of implementing request-reply (RPC-style) protocols involves allocating a fresh reply channel for each request (e.g., Example 1.9). Condition variables provide a cheaper reply mechanism in these cases.[1] Note, however, that condition variables are asynchronous on their output. Thus, in the case where the server needs to know if the reply has been accepted (e.g., see Example 1.10) they are unsuitable.

[1]In fact, experimental evidence suggests that allocating a fresh condition variable for each reply is substantially faster than using a dedicated reply channel.

Chapter 5

Multi-threaded I/O

I/O poses two problems for concurrent programs: first, concurrency control is required on access to the I/O state and, second, I/O operations, which are potentially blocking, need to be supported by the synchronization primitives. **CML** supports I/O both at the stream level and at the file descriptor level. The stream I/O library includes most of the **SML/NJ** stream I/O operations as well as event valued versions of the input operations.

5.1 Stream I/O

The structure `CIO` is a first-pass at support for multi-threaded streams (*à la* **SML/NJ**'s in and out streams). It includes a large subset of the `IO` structure provided by **SML/NJ**, as well as event valued versions of the input stream functions:

```
val lookaheadEvt : instream -> string
val inputEvt     : instream * int -> string event
val inputcEvt    : instream -> int -> string event
val inputLineEvt : instream -> string event
```

The event values produced by these functions have the same semantics as the corresponding **SML/NJ** input operations. Output operations are assumed to be nonblocking, and thus event valued versions of them are not provided. Appendix A gives the complete signature of the `CIO` structure.

Example 5.1

Consider a game program in which the player is given a limited time to answer a question. The following function could be used to implement a question/response interaction with the player:

```
fun getAnswer (question, t) = let open CML CIO
      in
        print question; flush_out std_out;
        select [
            wrap (inputLineEvt std_in, SOME),
            wrap (timeout t, fn () => NONE)
          ]
      end
```

This function prints the question (flushing `std_out` in case there is no terminating newline) and waits for an answer. If the user does not respond before the time specified by `t` has elapsed, then `NONE` is returned.

There is a strong similarity between channel I/O and stream I/O. The `CIO` structure provides a mechanism for creating streams with a channel style interface at the other end:

```
exception ClosedStream
val openChanIn  : unit -> ((string -> unit) * instream)
val openChanOut : unit -> (string event * outstream)
```

These function return a pair representing the two sides of a stream; one side with a channel-style interface and the other with a stream interface. This mechanism is useful for hooking existing stream based code (i.e., sequential code) into a concurrent framework. Note that the instreams are buffered and thus the send-style operation at the other end is asynchronous.

5.2 Low-level I/O

The `CML` structure includes the functions

```
exception InvalidFileDesc of int
val syncOnInput  : int -> unit
val syncOnOutput : int -> unit
val syncOnExcept : int -> unit
```

which provide an event-valued interface to the UNIX `select`(2) system call. If an attempt is made to synchronize on a closed file descriptor, then the exception `InvalidFileDesc` will be raised. These are used by the `CIO` structure to avoid blocking the system while waiting for I/O, and can also be used to implement I/O on sockets.

Chapter 6

Initialization and termination

The structure `RunCML` provides a collection of facilities to support the initialization and clean termination of **CML** programs. This chapter describes how to use these facilities and when they are necessary.

6.1 Servers and top-level channels

CML provides a mechanism for the automatic initialization and termination of services. This is used, for example, by the `CIO` structure to spawn the threads for the standard streams on initialization and to flush output buffers on termination. The function

```
val logServer : (string * (unit -> unit) * (unit -> unit)) -> unit
```

is used to register initialization and termination functions for a service. This function is usually called at the top-level of the structure that implements the service. The first argument is a string for uniquely identifying the service;[1] the second argument is the initialization function and the third is the termination function. The initialization functions are called in the order that they were registered, and termination functions in the reverse order. The first service to be initialized (and last to be terminated) is the `CIO` service, thus other services may safely use I/O in their initialization and termination. Note, however, that if a termination protocol is implemented using multiple threads, then the termination function should not return until all of the threads have terminated (or timed out); otherwise, a race condition exists. A server can be unregistered by the function:

```
val unlogServer : string -> unit
```

For a complete application, this mechanism provides a convenient way to handle initialization and termination. But, it is most useful in the development cycle, where a program is going to be run, modified and then run again. The auto-initialization mechanism helps insure that each run will start with a clean slate.

Services usually employ one or more channels bound at top-level for communication with clients. These can cause problems in the development cycle, since the channels may have blocked threads in their queues. If the channel is not cleaned between runs, then the threads will be carried over. To avoid this problem, there is a mechanism for registering top-level channels:

[1] If the same string is used twice, then the second call will replace the first.

```
val logChannel : (string * 'a chan) -> unit
val unlogChannel : string -> unit
```

Registered channels will have their queues cleared at initialization.

A simple example will illustrate the use of this mechanism:

Example 6.1 (Unique Ids)

Consider a service for generating system-wide unique identifiers with the signature:

```
signature UNIQUE_ID =
  sig
    eqtype id
    val nextId : unit -> id
  end (* UNIQUE_ID *)


structure UniqueId : UNIQUE_ID =
  struct
    datatype id = ID of int

    val idCh : id CML.chan = CML.channel ()

    fun server i = (CML.send (idCh, ID i);  server (i+1))

    fun nextId () = CML.accept idCh

    val _ = RunCML.logChannel ("UniqueId.idCh", idCh)

    val _ = RunCML.logServer ("UniqueId",
              fn () => (CML.spawn (fn () => server 0); ()),
              fn () => ())

  end (* structure UniqueId *)
```

The initialization function spawns the server thread. The termination function does not do anything, although we could have implemented a termination protocol. Upon termination, the server will most likely be blocked on `idCh`; but since it is registered, this will not pose any problems.

6.2 Starting and stopping a CML program

Before a **CML** program can run there are a number of things that must be done, such as enabling preemptive scheduling and initializing the top-level channels and registered services. This is all handled by the function

```
val doit : ((unit -> unit) * int option) -> unit
```

in the structure `RunCML`. The first argument is the root thread and the second is the time quantum (measured in milliseconds) for thread preemption. Specifying the value `NONE` for the time quantum will disable preemptive scheduling, but this can cause a program to block indefinitely if it uses timeouts. Preemption is implemented using the interval timer provided by the operating system and the **SML/NJ** signals facility[Rep90]. The choice of a good time quantum is application dependent: for programs with "real-time" responsiveness requirements, small values (between 10 and 40 ms.) are better; for other applications, larger values will reduce scheduling overhead.[2] Of course, different hardware/OS platforms support different timer granularities, but most provide 50 Hz. or finer.

Once you have built an application using **CML**, you may want to build an executable image that can be run from shell command line. The function

```
val exportFn : (string * ((string list * string list) -> unit) * int option)
      -> unit
```

provides this service. It is essentially the same as the function `IO.exportFn` provided by **SML/NJ**, except that it has an extra third argument for specifying the timer quantum and it binds in the code for service initialization and termination.[3]

There is also a mechanism for forcing termination of **CML** programs. The function

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

in the structure `RunCML` terminates the system. This includes terminating the registered services. Termination is robust: if a service termination function does not complete within five seconds, then a timeout message will be printed to the standard error stream. You may also terminate a run by typing your interrupt character (e.g. control-C), which will asynchronously force a shutdown.

[2] For most machines, a 20 ms. scheduling quantum produces an overhead of less than 3%[Rep90].

[3] As with **`IO.exportFn`**, you probably want to start with a version of **SML** that has the compiler in the heap. See the **`makeml`**(1) man page in the **SML/NJ** distribution for details.

Chapter 7

Debugging CML programs

Debugging concurrent programs can be quite difficult, because of their nondeterministic behavior. **CML** currently has fairly limited support for debugging; the `TraceCML` module provides a mechanism for controlling diagnostic output, a mechanism for monitoring threads for unexpected termination, and a mechanism for reporting uncaught exceptions in threads.

7.1 Debugging hints

Although there is little support for debugging in **CML**, there are a few techniques that can make debugging easier. The most important thing to do is to debug your sequential code in **SML** before using it in a **CML** program; you can use the **SML/NJ** debugger to do this[Tol90]. Even with correct sequential code, your program may not work. In this situation, there are two techniques you can use: adding print statements and turning off preemptive scheduling (see Chapter 6). Try to isolate the problem to one or two threads, and then monitor their state with print statements. Unfortunately, this technique will often not work with race conditions, since the print statements change the program timing. The only real solution to race conditions is to design your communication structure carefully. Turning off preemption increases the repeatability of your program's behavior, but programs that use stream I/O and timeouts will not function correctly without preemption.

7.2 Trace modules

For a large **CML** program, it probably makes sense to systematically include diagnostic output throughout the system. The `CMLTrace` structure supports this process by providing a mechanism, called *trace modules*, for selectively enabling diagnostic output for different parts of the system. The basic idea is that one defines a heirarchy of trace modules, which provide valves for turining debugging output on and off. Figure 7.1 gives the operations on trace modules. The value `traceRoot` is the root of the trace module heirarchy, and has the name "`/`." A new trace module may be created as the child of an existing module by using the `traceModule` function; the name of a trace module is returned by `nameOf` and a name can be mapped to a trace module by `moduleOf` (this raises the exception `NoSuchModule` if no module of the given name exists). The name of a trace module in a heirarchy is similar to a UNIX-style pathname; i.e., the individual module names are separated by "`/`". The following transcript illustrates this:

```
type trace_module

val traceRoot : trace_module

val traceModule : (trace_module * string) -> trace_module
val nameOf : trace_module -> string
val moduleOf : string -> trace_module

val traceOn : trace_module -> unit
val traceOff : trace_module -> unit
val traceOnly : trace_module -> unit
val amTracing : trace_module -> bool

val status : trace_module -> (trace_module * bool) list

val trace : (trace_module * (unit -> string list)) -> unit

datatype trace_to
  = TraceToOut
  | TraceToErr
  | TraceToNull
  | TraceToFile of string
  | TraceToStream of CIO.outstream

val setTraceFile : trace_to -> unit
```

Figure 7.1: Trace module operations

```
- val foo = TraceCML.traceModule (TraceCML.traceRoot, "foo");
val foo : TraceCML.trace_module
- TraceCML.nameOf foo;
val it = "/foo/" : string
- val bar = TraceCML.traceModule (foo, "bar");
val bar : TraceCML.trace_module
- TraceCML.nameOf bar;
val it = "/foo/bar/" : string
```

Each trace module is a point of control for diagnostic printing. The function `trace` is used to conditionally print according to the state of a given module; the second argument to `trace` is a function that is evaluated if the module given as the first argument is enabled. The results of evaluating the second argument are concatenated and printed to the current trace file (see below).

The functions `traceOn` and `traceOff` are used to change the status of a trace module and all of its descendents. For example, applying `traceOn` to `foo` (as defined above), will enable `foo`, `bar` and `baz`. A subsequent application of `traceOff` to `bar` will leave just `foo` and `baz` enabled. The function `traceOnly` is used to turn on a module without enabling its descendents, and the function `amTracing` returns the current status of a module. The function `status` returns a list of a module and all of its descendents and their associated state; the list is in pre-order.

The function `setTraceFile` is used to set the destination of the trace output. The value `TraceToOut` specifies that output should be directed to `CIO.std_out` (the default); the value `TraceToErr` specifies `CIO.std_err`; while th value `TraceToNull` causes output to be discarded. An arbitrary `CIO` output stream may be specified using either `TraceToFile` or `TraceToStream`. In the former case, the specified file is opened for writing; if the attempt to open fails, then `CIO.std_out` is used.

The trace module operations are designed to be called either from the **SML/NJ** top-level loop, or while executing a **CML** program (although `trace` is a no-op from the top-level).

7.3 Thread watching

Another common bug in **CML** programs is when a thread unexpectedly dies. The `TraceCML` module addresses this problem by providing a mechanism for watching specific threads for unexpected termination. The interface is:

```
val watcher : trace_module
val watch : (string * CML.thread_id) -> unit
val unwatch : CML.thread_id -> unit
```

where the trace module `watcher` controls printing of messages (this is enabled by default); the function `watch` causes the specified thread to be watched; and the function `unwatch` disables the watching of a thread. The following transcript illustrates the use of the facility:

```
- fun wspawn (name, f) = let
=     val tid = CML.spawn f
=     in
=        TraceCML.watch (name, tid);
=       tid
=      end;
val wspawn = fn : string * (unit -> unit) -> CML.thread_id
- fun test () = (wspawn ("dummy", fn () => ()); ());
val test = fn : unit -> unit
- RunCML.doit (test, SOME 100);
WARNING!  Watched thread dummy[9] has died.
val it = () : unit
```

7.4 Uncaught exceptions

One of the the most common errors in **CML** (and **SML**) programs is an uncaught exception. This is particularly nasty in **CML**, since if a thread raises an uncaught exception, then it will terminate without a trace (unless it is being watched). The `TraceCML` structure provides support for reporting uncaught exceptions.

If a thread generates an uncaught exception, then a message is sent to the *uncaught-exception server*, which prints a message to the standard error stream. For example:

```
- fun blah () = (hd []; ());
val blah = fn : unit -> unit
- RunCML.doit(blah, NONE);
CML: uncaught exception Hd in thread [8]
val it = () : unit
```

It is possible to set the function called by the trace server using the function

```
val setUncaughtFn : ((CML.thread_id * exn) -> unit) -> unit
```

Continuing our example:

```
- fun complain _ = (CIO.print "goodbye\n"; RunCML.shutdown());
val complain = fn : 'a -> 'b
- TraceCML.setUncaughtFn complain;
- RunCML.doit (blah, SOME 100);
goodbye
val it = () : unit
```

This can be used to report the arguments of given exceptions, as well as their names.

In a large system, different sub-systems may have their own special exceptions that should be reported if uncaught. To support such systems, the `TraceML` structure allows additional handlers to be layered over the "catch-all" handler defined by `setUncaughtFn`. The function

```
val setHandleFn : ((CML.thread_id * exn) -> bool) -> unit
```

adds an uncaught-exception handler. When a thread has an uncaught exception, the various uncaught-exception handlers are applied to the thread ID and exception, until one of the handlers return true. If none of the handlers returns true, then the catch-all handler is applied. For example:

```
- exception Error of string;
exception Error of string
- fun handleError (_, Error s) = (CIO.print("ERROR: " ^ s ^ "\n"); true)
=   | handleError _ = false;
val handleError = fn : 'a * exn -> bool
- TraceCML.setHandleFn handleError;
val it = () : unit
- fun foo () = (raise Error "foo");
val foo = fn : unit -> 'a
- RunCML.doit (foo, SOME 100);
ERROR: foo
val it = () : unit
```

The default catch-all handler can be restored, and the other handlers removed by calling the function:

```
val resetUncaughtFn : unit -> unit
```

Chapter 8

Administrative details

8.1 How to get the release

CML is distributed via anonymous FTP from three sites, as a compressed tar file named `CML-0.9.8.tar.Z`. The sites and locations are:

FTP Site	Path
`ftp.cs.cornell.edu`	`/pub/CML-0.9.8.tar.Z`
`research.att.com`	`/dist/ml/CML-0.9.8.tar.Z`
`princeton.edu`	`/pub/ml/CML-0.9.8.tar.Z`

In addition to the **CML** distribution, you will also need the **SML/NJ** distribution, which is available from the latter two FTP sites. Version 0.9.8 of **CML** works with version 0.75 of SML/NJ, but we recommend a more recent release (0.93 is the most recent version as of this writing). The following is a sample ftp dialog:

```
% ftp ftp.cs.cornell.edu
Connected to ftp.cs.cornell.edu.
...
Name: anonymous
331 Guest login ok, send ident as password.
Password: your-name@your-machine
230 Guest login ok, access restrictions apply.
ftp> cd pub
250 CWD command successful.
ftp> binary
200 Type set to I.
ftp> get CML-0.9.8.tar.Z
ftp> quit
221 Goodbye.
%
```

Once you have the compressed tar file, you can extract the distribution by the command

```
% zcat CML-0.9.8.tar.Z | tar xof -
```

This will create a directory named `cml-0.9.8`, which is the root of the distribution.

We would like to keep track of the use of **CML**, so if you ftp a copy, please send electronic mail to `sml-bugs@research.att.com`. We will use this information to notify people of bug fixes and other minor changes between releases.

8.2 Installing CML

If you plan to use **CML** on a regular basis (or if you have disk space to waste), you may want to install a pre-loaded version of the system. This can be done using the `exportML` function of **SML/NJ**, but to make things easier, an installation script (`install-cml`) is provided. To install **CML**, first change to the root directory of the distribution and then run the `install-cml` command. This command recognizes the following options:

-o *name* Use *name* for the name of the exported object (default `cml`).

-sml *name* Use *name* for the name of the **SML/NJ** executable (default `sml`).

-all Load the library modules (described in Appendix B) in addition to the core modules of **CML**.

8.3 Release history

The following list describes the major aspects of the **CML** release history. The distribution includes a file named `CHANGES`, which contains a more detailed description of the changes between versions.

Version 0.9.8 (February 1993). This public release is the first that is being included in the **SML/NJ** distribution. It fixes a number of bugs, provides improved performance, and a new, more powerful, trace facility (thanks to C. Krumvieda).

Version 0.9.7 (July 1992). This internal release fixed some bugs and included a new scheduling mechanism that improves interactive responsiveness in systems like **eXene**.

Version 0.9.6 (October 1991). This public release tracks changes to the pervasive environment in **SML/NJ** version 0.74. It also fixes a couple of bugs and gains a performance boost by adapting the *var pointer* to hold the current thread ID (the var pointer is a dedicated register provided by **SML/NJ** to support multiprocessors).

Version 0.9.5 (July 1991). This public release corrects a typing problem that was exposed by compiling **CML** under **SML/NJ** version 0.70.

Version 0.9.4 (June 1991). This is the second major public release of CML. In addition to fixing a number of bugs, it provides a somewhat different interface than the earlier releases.

Version 0.9.3 (March 1991). This internal release was used for the benchmarks presented in [Rep91a]. It was the first implementation of the the `guard` and `abort` operations.

Version 0.9.2 (January 1991). This release fixes a serious space-leak bug. The actual fix required changes to **SML/NJ**, so this version requires **SML/NJ** version 0.68, or later.

Version 0.9 (November 1990). This is the first public release of **CML**. It is based on **SML/NJ** version 0.66, and contains the basic concurrency primitives, support for first-class synchronous operations, multithreaded I/O, preemptive scheduling and rudimentary debugging support.

8.4 Bug reports

We have tested **CML** on the following types of machines: Sun-3, Sun-4, DECStation, SGI Indigo (both R3000 and R4000[1]), IBM RS/6000 and NeXT machines. There are no known bugs in 0.9.8. Bug reports should be mailed to `sml-bugs@research.att.com`; please using the following format:

`Submitter:`	*Your name and email address.*
`Date:`	*The date of the report.*
`System(s) and Version:`	`CML`
`SML/NJ Version:`	*The version of SML/NJ you are using.*
`Machine:`	*The architecture and operating system you are using.*
`Severity:`	*How severe is the bug (major, minor)?*
`Problem:`	*A short description of the problem.*
`Transcript:`	*A short example that exhibits the bug, and the output transcript*
`Comments:`	*Any aditional comments.*
`Fix:`	*Extra points for filling in this field!!*

A template for this format is given in the file `doc/cml-bug-form`.

[1] There is a known hardware bug in the R4000 chip that can cause **SML** (and thus **CML**) to crash.

Bibliography

[Agh86] Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, Mass., 1986.

[ASS85] Abelson, H., G. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Mass., 1985.

[Bor86] Bornat, R. A protocol for generalized occam. *Software – Practice and Experience*, **16**(9), September 1986, pp. 783–799.

[DHM91] Duba, B., R. Harper, and D. MacQueen. Type-checking first-class continuations. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, January 1991, pp. 163–173.

[GNV88] Gansner, E. R., S. C. North, and K. P. Vo. DAG – a program that draws directed graphs. *Software – Practice and Experience*, **18**(11), November 1988, pp. 1047–1062.

[Hal85] Halstead, Jr., R. H. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, **7**(4), October 1985, pp. 501–538.

[Har86] Harper, R. Introduction to Standard ML. *Technical Report ECS-LFCS-86-14*, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, August 1986.

[McI90] McIlroy, M. D. Squinting at power series. *Software – Practice and Experience*, **20**(7), July 1990, pp. 661–683.

[MTH90] Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass, 1990.

[Pau91] Paulson, L. C. *ML for the Working Programmer*. Cambridge University Press, New York, N.Y., 1991.

[Ram90] Ramsey, N. Concurrent programming in ML. *Technical Report CS-TR-262-90*, Department of Computer Science, Princeton University, April 1990.

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

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

[Rep90] Reppy, J. H. Asynchronous signals in Standard ML. *Technical Report TR 90-1144*, Department of Computer Science, Cornell University, August 1990.

[Rep91a] Reppy, J. H. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991, pp. 293–305.

[Rep91b] Reppy, J. H. An operational semantics of first-class synchronous operations. *Technical Report TR 91-1232*, Department of Computer Science, Cornell University, August 1991.

[Rep92] Reppy, J. H. *Higher-order concurrency*. Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, NY, January 1992. Available as Technical Report TR 92-1285.

[Tol90] Tolmach, A. P. *Debugging in Standard ML of New Jersey*. Department of Computer Science, Princeton University, Princeton, N.J., 1990.

Appendix A

The top-level environment

The top-level **CML** environment consists of four structures; this appendix lists these with their interface signatures. These structures are implemented as functors, the source code of which can be found in the directory `cml-0.9.8/src`. The following table lists the top-level structures, with the source file and a short description of each structure:

Structure	Source file	Description
`CML`	`concur.sml`	The core structure of **CML**; it defines threads, channels and events.
`RunCML`	`run.sml`	Provides support for initialization and termination of **CML** programs.
`CIO`	`cio.sml`	A multi-threaded input/output facility.
`TraceCML`	`trace-cml.sml`	Support for diagnostic output and error catching

A.1 `CML`

The structure `CML` provides the core functionality of the **CML** system. See Chapters 2 and 3 for a discussion of its facilities. This structure has the following signature:

```
signature CONCUR_ML =
  sig

    val version : {major : int, minor : int, rev : int, date : string}
    val versionName : string

  (** events **)
    type 'a event

    val sync   : 'a event -> 'a
    val select : 'a event list -> 'a
    val poll   : 'a event -> 'a option

    val choose : 'a event list -> 'a event
```

```
  val guard : (unit -> 'a event) -> 'a event

  val wrap        : ('a event * ('a -> 'b)) -> 'b event
  val wrapHandler : ('a event * (exn -> 'a)) -> 'a event
  val wrapAbort   : ('a event * (unit -> unit)) -> 'a event

  val always : 'a -> 'a event
  val ALWAYS : unit event    (* for backward compatibility *)

(** threads **)
  type thread_id

  val spawn : (unit -> unit) -> thread_id

  val yield : unit -> unit
  val exit : unit -> 'a

  val getTid : unit -> thread_id
  val sameThread : (thread_id * thread_id) -> bool
  val tidLessThan : (thread_id * thread_id) -> bool
  val tidToString : thread_id -> string

  val threadWait : thread_id -> unit event

(** condition variables **)
  type 'a cond_var

  val condVar : unit -> '1a cond_var

  val writeVar : ('a cond_var * 'a) -> unit
  exception WriteTwice

  val readVar : 'a cond_var -> 'a
  val readVarEvt : 'a cond_var -> 'a event

(** channels **)
  type 'a chan

  val channel : unit -> '1a chan

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

  val sameChannel : ('a chan * 'a chan) -> bool

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

(** real-time synchronization **)
  datatype time = TIME of {sec : int, usec : int}  (* from System.Timer *)
    sharing type time = System.Timer.time
  val waitUntil : time -> unit event
  val timeout   : time -> unit event

(* low-level I/O support (not for general use) *)
```

```
    exception InvalidFileDesc of int
    val syncOnInput  : int -> unit event
    val syncOnOutput : int -> unit event
    val syncOnExcept : int -> unit event

  end (* signature CONCUR_ML *)
```

A.2 RunCml

The structure RunCML provides the bookkeeping code for starting and terminating **CML** programs (see Chapter 6). It has the following signature:

```
signature RUN_CML =
  sig
    structure CML : CONCUR_ML

  (* log/unlog channels and servers for initialization and termination *)
    exception Unlog
    val logChannel : (string * 'a CML.chan) -> unit
    val unlogChannel : string -> unit
    val logServer : (string * (unit -> unit) * (unit -> unit)) -> unit
    val unlogServer : string -> unit
    val unlogAll : unit -> unit

  (* run the system *)
    val doit : ((unit -> unit) * int option) -> unit
    exception Running

  (* export a CML program *)
    val exportFn : (string * ((string list * string list) -> unit) * int option)
          -> unit

  (* shutdown a run *)
    val shutdown : unit -> 'a
    exception NotRunning

  end (* RUN_CML *)
```

A.3 CIO

The CIO structure implements a concurrent version of most of the **SML/NJ** IO operations (see Chapter 5). It has the following signature:

```
signature CONCUR_IO =
  sig
    structure CML : CONCUR_ML

    exception Io of string

    type instream
```

```
    type outstream

    val std_in : instream
    val std_out : outstream
    val std_err : outstream

    val open_in : string -> instream
    val open_string : string -> instream
    val open_out : string -> outstream
    val open_append : string -> outstream
    val execute : (string * string list) -> (instream * outstream)
    val execute_in_env : (string * string list * string list)
          -> (instream * outstream)

    exception ClosedStream
    val openChanIn  : unit -> ((string -> unit) * instream)
    val openChanOut : unit -> (string CML.event * outstream)

    val close_in : instream -> unit
    val close_out : outstream -> unit

    val can_input : instream -> int
    val lookahead : instream -> string
    val input : instream * int -> string
    val inputc : instream -> int -> string
    val input_line : instream -> string
    val end_of_stream : instream -> bool

    val lookaheadEvt : instream -> string CML.event
    val inputEvt : instream * int -> string CML.event
    val inputcEvt : instream -> int -> string CML.event
    val inputLineEvt : instream -> string CML.event

    val output : outstream * string -> unit
    val outputc : outstream -> string -> unit
    val flush_out : outstream -> unit

    val print : string -> unit

  end (* CONCUR_IO *)
```

A.4 TraceCML

The `TraceCML` structure provides support for debuggin **CML** program (see Chapter 7). It has the following signature:

```
signature TRACE_CML =
  sig

    structure CML : CONCUR_ML
    structure CIO : CONCUR_IO

  (** Trace modules **)
```

```
  type trace_module

  val traceRoot : trace_module

  exception NoSuchModule

  val traceModule : (trace_module * string) -> trace_module
  val nameOf : trace_module -> string
  val moduleOf : string -> trace_module

  val traceOn : trace_module -> unit
  val traceOff : trace_module -> unit
  val traceOnly : trace_module -> unit
  val amTracing : trace_module -> bool

  val status : trace_module -> (trace_module * bool) list

  val trace : (trace_module * (unit -> string list)) -> unit

  datatype trace_to
    = TraceToOut
    | TraceToErr
    | TraceToNull
    | TraceToFile of string
    | TraceToStream of CIO.outstream

  val setTraceFile : trace_to -> unit

(** Thread watching **)

  val watcher : trace_module
  val watch : (string * CML.thread_id) -> unit
  val unwatch : CML.thread_id -> unit

(** Uncaught exception handling **)

  val setUncaughtFn : ((CML.thread_id * exn) -> unit) -> unit
  val setHandleFn : ((CML.thread_id * exn) -> bool) -> unit
  val resetUncaughtFn : unit -> unit

end; (* TRACE_CML *)
```

Appendix B

The CML Library

This release of **CML** includes a small library of modules implementing some common concurrent idioms; it will be expanded in future releases of **CML**. This appendix describes the interfaces of these library modules (which include the buffered channels used in Example 1.8). They can be found in the directory `cml-0.9.8/library`; the following table lists them with their source file and a short description:

Structure	Source file	Description
`Plumbing`	`plumbing.sml`	Fixtures for connecting threads.
`BufferChan`	`buffer.sml`	Buffered channels.
`Future`	`future.sml`	**Multi-lisp** style futures.
`Cobegin`	`cobegin.sml`	Cobegin/end with barrier synchronization.
`ConcurCallCC`	`callcc.sml`	A "safe" version of `callcc` and `throw`.

If your version of **CML** is installed with the library (see section 8.2), then these modules will be defined in the top-level environment, otherwise you must load them individually. These structures are implemented as functors, so to load one requires both reading in the source file and applying the resulting functor. The rest of this appendix gives the signature and a description of each library module.

B.1 Plumbing

The structure `Plumbing` contains various functions for connecting networks of processes together. It is defined by the functor

```
functor Plumbing (CML : CONCUR_ML) : PLUMBING
```

and has the following signature:

```
signature PLUMBING =
  sig
    structure CML : CONCUR_ML
    val sink : 'a CML.event -> unit
    val source : '1a -> '1a CML.event
    val iterate : ('1a * ('1a -> '1a)) -> '1a CML.event
    val connect : ('a CML.event * ('a -> unit CML.event)) -> unit
    val filter : ('a CML.event * ('a -> 'b) * ('b -> unit CML.event)) -> unit
  end (* PLUMBING *)
```

The functions `source` and `sink` provide end-points for a network of threads (Note that `source` is really just `always`). We can connect them to form a (useless) network:

```
sink (source 1)
```

In this example `source` will produce an infinite stream of `1`s, which will be consumed by `sink`. The function `iterate` also produces an infinite stream of values, but each value is computed from the previous. For example, the stream of numbers that we used in Example 1.3 could have been provided by:

```
iterate (2, fn x => (x + 1))
```

The functions `connect` and `filter` provide connections between threads. For example, the following code will print the numbers from `1` to `n`:

```
fun printN n = let
  open Plumbing
  val ch = CML.channel()
  in
    filter (
      iterate (1, fn x => (if (x >= n) then CML.exit() else x+1)),
      fn x => CIO.print(makestring x ^ "\n"),
      fn x => CML.transmit(ch, x));
    sink(CML.receive ch)
  end
```

This example uses `iterate` to generate the numbers; `filter` to print them; and `sink` to provide the demand to keep the flow going.

B.2 Buffered channels

Buffered channels provide a means of asynchronous communication. This facility is provided by the functor

```
functor BufferChan (CML : CONCUR_ML) : BUFFER_CHAN
```

which has the following signature:

```
signature BUFFER_CHAN =
  sig
    structure CML : CONCUR_ML
    type 'a buffer_chan
    val buffer : unit -> '1a buffer_chan
    val bufferIn : '1a CML.chan -> '1a buffer_chan
    val bufferOut : '1a CML.chan -> '1a buffer_chan
    val bufferSend : ('a buffer_chan * 'a) -> unit
    val bufferAccept : 'a buffer_chan -> 'a
    val bufferReceive : 'a buffer_chan -> 'a CML.event
  end (* BUFFER_CHAN *)
```

There are three different ways to create a buffered channel. If you already have a channel to connect to the input or output of the buffer, then the functions `bufferIn` or `bufferOut` can be used; otherwise the function `buffer` should be used (as in Example 1.8). Buffered channels provide an asynchronous output operation, `bufferSend`, and two forms of synchronous input operation, `bufferAccept` and `bufferReceive`. Note that in the case where you attach an existing channel to a buffer (using `bufferIn` or `bufferOut`), the associated channel operations may be substituted for the buffer operations.

B.3 Futures

Futures are a construct provided by **Multi-lisp** for introducing parallel evaluation of an expression. We provide them, more as a demonstration of building new abstractions, than because we think they are useful. The functor

```
functor Future (CML : CONCUR_ML) : FUTURE
```

implements this mechanism. It has the signature

```
signature FUTURE =
  sig
    structure CML : CONCUR_ML
    val future : ('a -> '2b) -> 'a -> '2b CML.event
  end (* FUTURE *)
```

B.4 Cobegin

The structure `Cobegin` provides a generalization of the `spawn` operation. It allows a list of threads to be spawned and returns an event for synchronizing on the termination of all of the threads (this is called *barrier synchronization*). The structure is implemented by the functor

```
functor Cobegin (CML : CONCUR_ML) : COBEGIN
```

which has the signature

```
signature COBEGIN = sig
    structure CML : CONCUR_ML
    val cobegin : (unit -> unit) list -> unit CML.event
  end (* COBEGIN *)
```

B.5 Safe callcc

This is the interface of a safe implementation of the first-class continuation primitives. A thread may only throw to one of its own continuations; an attempt to throw to another thread's continuation will result in the exception `BadCont` being raised.

```
signature CONCUR_CALLCC =
  sig
    exception BadCont
    type 'a cont
    val callcc : ('1a cont -> '1a) -> '1a
    val throw : 'a cont -> 'a -> 'b
  end
```

Appendix C

Source files for examples

The source code for the examples of this document are available on-line in the distribution in the directory "`cml-0.9.8/examples`." The following table maps the example numbers to file names:

Example	**Source File**
Example 1.1	`ex-simple-comm.sml`

Example 1.3	`ex-counter.sml`
	`ex-sieve.sml`
	`ex-primes.sml`

Example 1.4	`ex-future.sml`
Example 1.5	`ex-simple-comm2.sml`
Example 1.6	`ex-token-sig.sml` `ex-token.sml`
Example 1.7	`ex-buffer-sig.sml` `ex-buffer.sml`
Example 1.8	`ex-multicast-sig.sml` `ex-multicast.sml`
Example 3.1	`ex-simple-comm3.sml`
Example 4.1	`ex-future-cv.sml`
Example 5.1	`ex-get-answer.sml`
Example 6.1	`ex-unique-sig.sml` `ex-unique.sml`