CML: A Higher-order Concurrent Language

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

1 Introduction

Concurrent ML (CML) is a high-level, high-performance language for concurrent programming. It is an extension of Standard ML (SML)\cite{MTH90}, and is implemented on top of Standard ML of New Jersey (SML/NJ)\cite{AM87}. CML is a practical language and is being used to build real systems. It demonstrates that we need not sacrifice high-level notation in order to have good performance.

Although most research in the area of concurrent language design has been motivated by the desire to improve performance by exploiting multiprocessors, we believe that concurrency is a useful programming paradigm for certain application domains. For example, interactive systems often have a naturally concurrent structure\cite{CP85, RG86, Pik89, Haa90}. Another example is distributed systems: most systems for distributed programming provide multi-threading at the node level (e.g., Isis\cite{BCJ90} and Argus\cite{LCJS87}). Sequential programs in these application domains often must use complex and artificial control structures to schedule and interleave activities (e.g., event-loops in graphics libraries). They are, in effect, simulating concurrency. These application domains need a high-level concurrent language that provides both efficient sequential execution and efficient concurrent execution: CML satisfies this need.

1.1 An overview of CML

CML is based on the sequential language SML\cite{MTH90} and inherits the good features of SML: functions as first-class values, strong static typing, polymorphism, datatypes and pattern matching, lexical scoping, exception handling and a state-of-the-art module facility. A brief introduction to SML is given as an appendix; also see [Har86]. The sequential performance of CML benefits from the quality of the SML/NJ compiler. In addition CML has the following properties:

- CML provides a high-level model of concurrency with dynamic creation of threads and typed channels, and rendezvous communication. This distributed-memory model fits well with the mostly applicative style of SML.
- CML is a higher-order concurrent language. Just as SML supports functions as first-class values, CML supports synchronous operations as first-class values. These values, called events, provide the tools for building new synchronization abstractions. For example, we have found uses for widely varying abstractions, such as remote procedure call, multicast channels and buffered channels. This flexibility is important, as it allows the synchronization and communication abstractions to be tailored to the application. CML’s events are an extension of the mechanism described in [Rep88].
- CML provides integrated I/O support. Potentially blocking I/O operations, such as reading from an input stream, are full-fledged synchronous operations. Low-level support is also provided, from which distributed communication abstractions can be constructed.
- CML provides automatic reclamation of threads and channels, once they become in accessible. This permits speculative communication that is not possible in other threads packages.
- CML uses pre-emptive scheduling. To guarantee interactive responsiveness, a single thread cannot be allowed to monopolize the processor. Pre-emption insures that a context switch will occur at regular intervals, which allows “off-the-shelf” code to be incorporated in a concurrent thread without destroying interactive responsiveness.
- CML is efficient. Thread creation, thread switching and message passing are very efficient (benchmarks are given in section 5). Experience with CML has shown that it is a viable language for implementing usable interactive
systems; this is in direct contrast with Haahr’s experience
with concurrency in scheme [Haar90]. In fact, message passing
in CML is significantly faster than in Sun’s light-weight
process library.

- CML is portable. It is written in SML and runs on essen-
tially every system supported by SML/NJ (four different
architectures and many different operating systems).

1.2 Organization of this paper

The rest of the paper has four parts: section 2 describes and
motivates the design of CML (a complete description can be
found in [Rep90b]); section 3 describes the use of CML in two
application areas; sections 4 and 5 describe the implementa-
tion and its performance; and, lastly, section 6 compares CML
with related work. A brief introduction to SML is included
as an appendix.

2 Basic concurrency primitives

A CML program consists of a number of threads, which use
message passing on typed channels to communicate and syn-
chronize. The signature of the basic thread and channel oper-
ations is given in figure 1. The function spawn dynamically
creates a new thread to evaluate its argument. Channels are
also created dynamically, using the function channel

```
(* create a new thread *)
val spawn : (unit -> unit) -> thread_id

(* create a new channel *)
val channel : unit -> '1a chan

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

Figure 1: Basic concurrency primitives

creates a new thread to evaluate its argument. Channels are
also created dynamically, using the function channel

1The “1” in the type variable of channel’s result type is the strength of
the variable. This is a technical mechanism used to allow polymorphic use of
updatable objects without creating type loopholes.

It is possible to use polling to implement selective commu-
nication, but to do so is awkward and requires busy wait-
ing. Usually selective communication is provided as multi-
plexed I/O operation. This can be a multiplexed input opera-
tion, such as occam’s ALT construct [Bur88], or a general-
ized (or symmetric) select operation that multiplexes both in-
put and output communications, such as Pascal-m’s select
construct [AB86]. Implementing generalized select on a multi-
processor can be difficult [Bor86], but, as will be shown in sec-
tion 2.2, there are situations in which generalized selective
communication is necessary.

There is a fundamental con
flict between the desire for abstraction and the need for selective communication. For example,
consider a server thread that provides a service via a request-
reply, or remote procedure call (RPC) style, protocol. The server
side of this protocol would be something like:

```
fun serverLoop () = if serviceAvailable()
then let val request = accept reqCh in
  send (replyCh, doit request);
  serverLoop ()
else doSomethingElse()
end
```

where the function doit actually implements the service.
This protocol requires that clients obey the following two rules:

1. A client must send a request before trying to read a reply.
2. Following a request the client must read exactly one reply
   before issuing another request.

If all clients follow these rules, then we can guarantee that
each request is answered with the correct reply. A obvious
way to improve the reliability of programs that use this ser-
vice is to bundle the client-side protocol into a function that
hides the details, thus ensuring that the rules are followed.
The following code implements this abstraction:

```
fun clientCall x = (send(reqCh, x); accept replyCh)
```

While this insures that the protocol is observed, it hides too
much. If a client blocks on a call to clientCall (e.g., if the
server is not available), then it cannot respond to other com-
munications. The client would like to use selective commu-
nication in this situation, but cannot, because the synchronous
aspect of the protocol has been hidden by the function abstrac-
tion. The next section describes our solution to this dilemma.

```
2.1 First-class synchronous operations

Following [Rep88] and [Rep89], we make synchronous operations into first-class values by introducing a new kind of value, called an event, which represents a potential synchronous operation (analogous to the way a function value describes a potential computation). These values provide an abstraction of a protocol implementation, without obscuring the synchronous aspect of a protocol. Figure 2 gives the signature of the basic event operations from the [Rep88] model. CML extends this model in several significant ways, which are discussed in section 2.4. The operator sync forces synchronization on an event value. The functions receive and transmit are used to build the base event values that describe channel communication. The event values returned by receive and transmit are called base event values. We can define accept and send using these and function composition:

```plaintext
val sync : 'a event -> 'a
val receive : 'a chan -> 'a event
val transmit : ('a chan * 'a) -> unit event
val choose : 'a event list -> 'a event
val wrap : ('a event * ('a -> 'b)) -> 'b event
```

Figure 2: Basic event operations

The functions choose and wrap are combinators for constructing new event values: choose provides a generalized selective communication mechanism and wrap combines an event with a post-synchronization action, called the wrapper. For example, if we have two integer valued channels, ch1 and ch2, then the following expression will block until a message is available on one of the channels:

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

When a message becomes available, then we say the receive event is enabled. If both base events are enabled at the same time, then one will be chosen non-deterministically. When a base event value is synchronized on, it produces a result, which to which its wrappers are applied. For example, if the above expression synchronizes on “receive ch1,” then the result will be multiplied by 2. CML supports generalized (or symmetric) selective communication; a choice event value may have both input and output operations in it. A formal operational semantics of CML has been developed and will be included in the author’s forthcoming Ph.D. thesis [Rep91b].

To understand this the higher-order nature of this mechanism, it is helpful to draw an analogy with first-class function values. Table 1 compares these two higher-order mechanisms.

<table>
<thead>
<tr>
<th>Property</th>
<th>Function values</th>
<th>Event values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type constructor</td>
<td>-&gt; event</td>
<td>-&gt; event</td>
</tr>
<tr>
<td>Introduction</td>
<td>x-abstraction</td>
<td>receive transit etc.</td>
</tr>
<tr>
<td>Elimination</td>
<td>application</td>
<td>sync</td>
</tr>
<tr>
<td>Combinators</td>
<td>composition</td>
<td>choose wrap etc.</td>
</tr>
<tr>
<td></td>
<td>map etc</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Relating first-class functions and events

The great benefit of this approach to concurrency is that it allows the programmer to create new first-class synchronization and communication abstractions. For example, we can define an event-valued function that implements the client-side of the RPC protocol given in the previous section:

```plaintext
fun clientCallEvt x = wrap {
  transmit(reqCh, x),
  fn () => accept replyCh
}
```

Application of the function transmit produces an event value that represents the sending of the request. If the server accepts the request, then we need to wait for a reply. To do this we wrap the request event with a function that will accept the reply. In the following section, we give a more substantial example.

2.2 An example

An example that illustrates a number of key points is an implementation of a buffered channel abstraction. Buffered channels provide a mechanism for asynchronous communication, which is similar to the actor mailbox [Agh86]. The source code for this abstraction is given in figure 3. The function buffer creates a new buffered channel, which consists of a buffer thread, an input channel and an output channel; the function bufferSend is an asynchronous send operation; and the function bufferReceive is an event-valued receive operation. The buffer is represented as a queue of messages, which is implemented as a pair of stacks (lists); the Appendix gives
abstype 'a buffer_chan = BC of {
  inch : 'a chan,
  outch : 'a chan
}
with
  fun buffer () = let
    val inCh = channel() and outCh = channel()
    fun loop ([], []) = loop([accept inCh], [])
    | loop (front as (x::r), rear) = sync {
        choose [
          wrap (receive inCh,
            fn y => loop(front, y::rear)),
          wrap (transmit(outCh, x),
            fn () => loop(r, rear))
        ]
    | loop ([], rear) = loop(rev rear, [])
  in
    spawn (fn () => loop([], []));
    BC{inch=inCh, outch=outCh}
  end

fun bufferSend (BC{inch, ...}, x) = send(inch, x)
fun bufferReceive (BC{outch, ...}) = receive outch
end (* abstype *)

Figure 3: Buffered channels

more details on this applicative implementation of queues.

This example illustrates several key points:

- Buffered channels are a new communication abstraction, which have first-class citizenship. A thread can use the bufferReceive function in any context that it could use the built-in function receive, such as selective communication.

- The buffer loop uses both input and output operations in its selective communication. This is an example of the necessity of generalized selective communication. If we have only a multiplexed input construct (e.g., occam's ALT), then we must use a request/reply protocol to implement the server side of the bufferReceive operation (see pp. 37-41 of [Bur88], for example). But if a request/reply protocol is used, then the bufferReceive operation cannot be used in a selective communication by the client.

- The buffer thread is a good example of a common CML programming idiom: using threads to encapsulate state. This style has the additional benefit of hiding the state of the system in the concurrency operations, which makes the sequential code cleaner. These threads serve the same role that monitors do in some shared-memory concurrent languages.

- Once created, the buffer thread never terminates, which may seem to pose a problem for resource recovery. If a buffered channel ever becomes unreachable from all non-blocked threads, then the buffer thread and channels will be reclaimed by the garbage collector. Because the channels and suspended thread state are normal SML/NJ heap objects, we get thread reclamation for free. In general, threads that communicate infinitely will block and be garbage collected if they become disconnected from the active part of the system. There are certain pathological live-lock situations, in which this fails, but these do not arise in practice.

A more complete version of this abstraction is included in the CML distribution.

### 2.3 Other synchronous operations

One of the nice aspects of events is that other primitive synchronous operations are easily accommodated in the framework. There are three examples of this in CML: synchronization on thread termination (sometimes called process join), low-level I/O support and timeouts. The function

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

produces an event for synchronizing on the termination of another thread. This is often used by servers that need to release resources allocated to client threads. Support for low-level I/O is provided by the functions:

```
val syncOnInput : int -> unit event
val syncOnOutput : int -> unit event
val syncOnExcept : int -> unit event
```

which allow threads to synchronize on the status of file descriptions. These operations are used in CML to implement a multi-threaded I/O stream library. There are two functions for synchronizing with the clock:

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

The function `waitUntil` returns an event that will synchronize at the given time, the function `timeout` delays the specified time from the time `sync` is applied.

CML also provides a polling mechanism

```
val poll : 'a event -> 'a option event
```

where the SML type constructor `option` is

```
datatype 'a option = NONE | SOME of 'a
```

which builds an event for polling its argument. The event value `poll(ev)` is always enabled. If it is chosen when involved in a synchronization, then it will return `(SOME ev)`, if `ev` is enabled and would return `v`; otherwise, it will return `NONE`.

### 2.4 Extending the mechanism

The concurrency mechanism described thus far is essentially that presented in [Rep88] and [Rep89]. CML extends this model in several important ways, in this section we motivate and describe these extensions.
Consider a protocol consisting of a sequence of client-server communications: \( c_1; c_2; \ldots; c_n \). When this protocol is packaged up in an event value, one of the \( c_i \) is designated as the commit point; the communication on which this event is chosen in a selective communication. In the mechanism of [Rep88], the only possible commit point is \( c_1 \). The \texttt{wrap} construct allows on to tack on \( c_2; \ldots; c_n \) after \( c_1 \) is chosen, but there is no way to make any of the other \( c_i \) the commit point. This asymmetry is a serious limitation to the original mechanism. To illustrate this problem, consider a server that implements an input stream abstraction. Since this abstraction should be smoothly integrated into the concurrent model, we make the input operations event-valued; for example, the function

\[
\text{val input : instream -> string event}
\]

would be used to read a single character. Other operations, such as \texttt{input_line} would also be provided. Let us assume that the implementation of these operations uses a request-reply protocol; thus, a successful \texttt{input} operation involves the communication sequence

\[
\text{send(chreq, REQ_INPUT); accept(chrep)}
\]

Packaging this up as an event (as we did in section 2.1), will make the \texttt{send} communication be the commit point, but this is not the right semantics. To see the problem, consider the case where a client thread wants to synchronize on the choice of reading a character and a five second timeout; e.g.:

\[
\text{sync (choose [}
\text{wrap (timeout (TIME{sec=5, usec=0}),}
\text{fn () => raise Timeout),}
\text{input instream}]
\)

The server will probably accept the request within the five second limit, even though the wait for input might be indefinite. The right semantics for the \texttt{input} operation requires making the \texttt{accept} be the commit point. The \texttt{guard} combinator provides a mechanism for doing this.

The \texttt{guard} combinator is the dual of \texttt{wrap}; it bundles code to be executed before the commit point; this code can include communications. It has the type

\[
\text{val guard : (unit -> 'a event) -> 'a event}
\]

A guard is essentially a suspension that is forced when \texttt{sync} is applied to it. As a simple example of the use of \texttt{guard}, the \texttt{timeout} function, described above, is actually implemented use \texttt{waitUntil} and a guard:

\[
\text{fun timeout t = guard (}
\text{fn () => waitUntil (add_time (t, currentTime()))}
\)

where \texttt{currentTime} returns the current wall-clock time.

Returning to our RPC example from above, we can now build an abstract RPC operation with the reply as the commit point. The two different versions are:

\[
\text{fun clientCallEvt1 x = wrap (}
\text{transmit(reqCh, x),}
\text{fn () => accept replyCh)}
\]

\[
\text{fun clientCallEvt2 x = guard (fn () => (}
\text{send(reqCh, x);}
\text{receive replyCh)}
\]

where \texttt{clientCallEvt1} commits on the server’s acceptance of the request and \texttt{clientCallEvt2} commits on the server’s reply to the request. Using guards to generate requests like this raises a couple of other problems. First of all, if the server cannot guarantee that requests will be accepted promptly, then evaluating the guard may cause delays. The solution to this is to spawn a new thread to issue the request:

\[
\text{fun clientCallEvt3 x = guard (fn () => (}
\text{spawn(fn () => send(reqCh, (replyCh, x)));}
\text{receive replyCh)}
\]

Another alternative is for the server to be a clearing-house for requests; spawning a new thread to handle each new request.

The other problem is more serious: what if this RPC event is used in a selective communication and some other event is chosen? How does the server avoid blocking forever on sending a reply? For idempotent services, this can be handled by having the client create a dedicated channel for the reply and having the server spawn a new thread to send the reply. The client side of this protocol is

\[
\text{fun clientCallEvt4 x = guard (fn () => let}
\text{val replyCh = channel() in}
\text{spawn(fn () => send(reqCh, (replyCh, x)));}
\text{receive replyCh end)}
\]

When the server sends the reply it evaluates

\[
\text{spawn (fn () => send(replyCh, reply))}
\]

If the client has already chosen a different event, then this thread blocks and will be garbage collected.

For services that are not idempotent, this scheme is not sufficient; the server needs a way to abort the transaction. The function

\[
\text{val wrapAbort : ('a event * (unit -> unit)) -> unit}
\]
is provided for this purpose. The semantics are that if the wrapped event is not chosen in a \texttt{sync} operation, then a new thread is spawned to evaluate the second argument, called the \texttt{abort} function. This is the complement of \texttt{wrap} in the sense that if you view every base-event in a choice as having both a wrapper and an abort function, then, when \texttt{sync} is applied, the wrapper of the chosen event is called and the abort functions of the other base events are spawned off.

The client code for the RPC using \texttt{abort} must allocate two channels: one for the reply and one for the abort message:

\begin{verbatim}
datatype abort_msg = ABORT

fun clientCallEvt5 x = guard (fn () => let
  val replyCh = channel()
  val abortCh = channel()
  fun abortFn () = send (abortCh, ABORT)
  in
    spawn(fn () =>
      send (reqCh, (replyCh, abortCh, x)));
    receive replyCh
  end)
end
\end{verbatim}

When the server is ready to reply (i.e., commit the transaction), it synchronizes on the value

\begin{verbatim}
choose{
  wrap (receive abortCh, 
    fn ABORT => abort the transaction),
  wrap (transmit (replyCh, reply), 
    fn () => commit the transaction)
}
\end{verbatim}

This mechanism is used to implement the concurrent stream I/O library in CML.

Another extension to [Rep88] is the function:

\begin{verbatim}
val wrapHandler = ('a event * (exn -> 'a)) -> 'a event
\end{verbatim}

wraps an exception handler around an event\(^2\). For example, \texttt{syncOnInput} will raise an exception if the file specified by the descriptor \texttt{fd} has been closed. Using \texttt{wrapHandler}, we can define a better behaved version of \texttt{syncOnInput}:

\begin{verbatim}
fun waitForInput fd = wrapHandler (
  wrap (syncOnInput fd, fn () => true),
  fn _ => false)
\end{verbatim}

Upon synchronization, this will return \texttt{true} if input is available, and \texttt{false} if the file is closed.

2.5 Stream I/O

CML provides a concurrent version of the SML stream I/O primitives. Input operations in this version are event valued, which allows them to be used in selective communication. For example, a program might want to give a user 60 seconds to supply a password. This can be programmed as:

\begin{verbatim}
fun getpasswd () = sync (choose |
  wrap (timeoutsec=60, usec=0, 
    fn () => NONE),
  wrap (input_line std_in, SOME) |
)
\end{verbatim}

This will return \texttt{NONE}, if the user fails to respond within 60 seconds, otherwise it wraps \texttt{SOME} around the user’s response\(^3\).

Streams are implemented as threads which handle buffering. The input operations are actually request/reply/abort protocols, similar to the one discussed above.

3 Applications

CML is more than an exercise in language design; it is intended to be a useful tool for building large systems. In this section we describe two applications of CML, and how they use the features of CML.

3.1 Interactive systems

Providing a better foundation for programming interactive systems, such as programming environments, was the original motivation for this line of research[RG86]. Because of their naturally concurrent structure, interactive systems are one of the most important application areas for CML. Concurrency arises in several ways in interactive systems:

User interaction. Handling user input is the most complex aspect of an interactive program. Most interactive systems use an \texttt{event-loop} and \texttt{call-back} functions. The event-loop receives input events (e.g., mouse clicks) and passes them to the appropriate \texttt{event-handler}. This structure is a poor-man’s concurrency: the event-handlers are coroutines and the event-loop is the scheduler.

Multiple services. For example, consider a document preparation system that provides both editing and formatting. These two services are independent and can be naturally organized as two separate threads. Multiple views are implemented by replicating the threads.

Interleaving computation. A user of a document preparation system may want to edit one part of a document while another part is being formatted. Since formatting may take a significant amount of time, providing a responsive interface requires interleaving formatting and editing. If the editor and formatter are separate threads, then interleaving comes for free.

\(^2\)\texttt{exn} is the type of exception values.

\(^3\)Since \texttt{SOME} is a constructor function (i.e., \texttt{SOME : 'a -> 'a option}), it can be used as a function argument.
Output driven applications. Most windowing toolkits, for example Xlib\textsuperscript{[Nye90]}, provide an input oriented model, in which the application code is occasionally called in response to some external event. But many applications are output oriented. Consider, for example, a computationally intensive simulation with a graphical display of the current state of the simulation. This application must monitor window events, such as refresh and resize notifications, so that it can redraw itself when necessary. In a sequential implementation, the handling of these events must be postponed until the simulation is ready to update the displayed information. By separating the display code and simulation code into separate threads, the handling of asynchronous redrawing is easy.

The root cause of these forms of concurrency is computer-human interaction: humans are asynchronous and slow.

CML has been used to build a multi-threaded interface to the X protocol\textsuperscript{[SG86]}, called eXene. This system provides a similar level of function as Xlib\textsuperscript{[Nye90]}, but with a substantially different, and we think better, model of user interaction. Windows in eXene have an environment, consisting of three streams of input from the window’s parent (mouse, keyboard and control), and one output stream for requesting services from the window’s parent. For each child of the window, there will be corresponding output streams and an input stream. The input streams are represented by event values and the output streams by event valued functions. A window is responsible for routing messages to its children, but this can almost always be done using a generic router function provided by eXene. Typically, each window has a separate thread for each input stream as well as a thread, or two, for managing state and coordinating the other threads. By breaking the code up this way, each individual thread is quite simple. This model is similar to those of [Pik89] and [Haa90].

This structure allows us to use delegation techniques to define new behavior from existing implementations. Delegation is an object-oriented technique (so we know it must be good), that originated in concurrent actor systems\textsuperscript{[Lie86]}. As an example, consider the case of adding a menu to an existing text window. We can do this in a general way by defining a wrapper that takes a window’s environment and returns a new, wrapped, environment. The wrapped environment has a thread monitoring the mouse stream of the old environment. Normally, this thread just passes messages along to the wrapped window, but when a mouse “button down” message comes along, the thread pops up the menu and consumes mouse messages until an item is chosen. Emden Gansner, of AT&T Bell Laboratories, has developed a “widget” toolkit on top of eXene, which uses these delegation techniques heavily.

The implementation of eXene which is currently about 8,500 lines of CML code, uses threads heavily. At the lowest level, threads are used to buffer communication with the X server. There are threads to manage shared state, such as graphics contexts, fonts and keycode translation tables. Because the internal threads are fairly specialized and tightly integrated, there is not much use of events as an abstraction mechanism.

The use of events as an abstraction mechanism is common at the application programmer’s level. In addition to the event-valued interface of the window environments, there are higher-level objects that have abstract synchronous interfaces. One example is a virtual terminal window (vtty). This provides a synchronous stream interface to its clients, which is compatible with the signature of CML’s concurrent I/O library. If the client-code is implemented as a functor\textsuperscript{[Mac84]} (parameterized module), then it can be used with either the concurrent I/O library or the vtty abstraction.

The vtty abstraction is a good example of where user-defined abstract synchronous operations are necessary for program modularity. At any time, the vtty thread must be ready to receive input from the user and output from the application; thus it needs selective communication. The underlying window toolkit (eXene) provides an abstract interface to the input stream, but, since it is event-valued, it can be used in the selective communication.

Another example of the use of new communication abstractions is a buffered multicast channel (a simple version is described in [Rep90b]). This abstraction has proven quite useful in supporting multiple views of an object. When the viewed object is updated, the thread managing its state sends a notification on the multicast channel. The multicast channel basically serves the role of a call-back function, while freeing the viewed object from the details on managing multiple views. All of the details of creating/destroying views and distributing messages are taken care of by the multicast abstraction.

3.2 Distributed ML

Another project involving CML is the development of a distributed programming toolkit for ML, which is being done in collaboration with Bard Bloom, Robert Cooper, Chet Murthy and Tim Teitelbaum at Cornell University. The low-level I/O support of CML is sufficient to build a structured synchronous interface to network communication (as was done
in our X-windows application). Higher-level linguistic support for distributed programming, such as the promise mechanism of [LS88], can be built using events to define the new abstractions. Some of these ideas have been prototyped by Chet Murthy at Cornell University as part of a re-implementation of the Nuprl interactive proof system. Nuprl can require vast amounts of system resources when used to develop large proofs, but it also provides ample opportunity for coarse-grain parallelism. The system is structured as a poll of proof servers distributed across a local area network of workstations. A user’s session manager will farm out pieces of the proof to idle and available servers; it uses an object similar to a promise as a place-holder for the outstanding work.

4 Implementation

CML is written entirely in SML, using a couple of non-standard extensions provided by SML/NJ: first-class continuations [DHM91] and asynchronous signals [Rep90a]. We added one minor primitive operation to the compiler (a ten line change in a 30,000 line compiler), which was necessary to guarantee that sync preserve tail-recursion. Threads are implemented à la [Wan80], using first-class continuations, and the SML/NJ asynchronous signal facility is used to implement pre-emptive scheduling.

Unlike other continuation-passing style compilers, such as [Ste78] and [KKR+86], the code generated by the SML/NJ compiler does not use a run-time stack. This means that callcc and throw are constant-time operations. While this is possible using a stack [HDB90], heap-based implementations are better suited for implementing light-weight threads (Haahr’s experience bears this out [Has90]).

Event values have a natural implementation in terms of first-class continuations. Without the choose operator, an event value could be represented as

```
type 'a event = 'a cont -> 'a
```

with sync being directly implemented by callcc. This representation captures the intuition that an event is just a synchronous operation with its synchronization point continuation as a free variable. The choose operator requires polling, since we need to see which (if any) base events are immediately available for synchronization. Thus, the implementation of an event value is a list of base events, with each base event represented by a polling function, a function to call for immediate synchronization and a function for blocking. The implementation of a precursor to CML is described in detail in [Rep89].

The support of poll requires grouping base events that are being polled as a group. Note that poll has the property that the event

```
poll(poll e)
```

is equivalent to

```
wrap(poll e, SOME)
```

We use this property to collapse nested applications of poll.

The implementation of guard is straight forward: a guard event value is represented by the guard function, when sync is applied to a guard event, the guard function is evaluated. This forcing is recursive, since a guard function might return a guard event. When a combinator, such as wrap, is applied to a guard event, the suspension is just pulled up one level. For example, the guard case of the wrap implementation is

```
fun wrap (GUARD g, f) = GUARD (fn() => wrap(g(), f))
```

where GUARD is the internal representation’s constructor for guard events. The application of choose to a guard event is a little more tricky, since multiple guards may be involved.

The most difficult CML feature to support is wrapAbort. If wrapAbort is applied to a choice of several events, then, at synchronization time, we must invoke the abort function if, and only if, none of the events was selected. We handle this by supplying an abort function for each element of the choice, with one designated as the leader. When invoked, the other abort threads send the leader a message; if the leader collects messages from all of the other threads, then it evaluates the abort function. The abort function will be evaluated if, and only if, all of the abort threads are invoked.

The implementation consists of about 1,380 lines of commented code. This breaks down into: 810 lines to implement threads, channels, events, low-level I/O support and pre-emptive scheduling; 430 lines to implement the stream I/O library; and 140 lines to handle initialization and termination. In addition, there is a small library of useful abstractions, such as buffered channels.
5 Performance

Providing a better notation for programming is not enough; it must be efficient enough that users will be willing to write major systems in it. To a great extent, SML/NJ has met this goal as an implementation of a high-level language [AJ89]. Our benchmark results show that CML maintains this standard.

5.1 The benchmarks

We have conducted a series of benchmarks on three different machines (see table 2). Each benchmarked operation was performed 100,000 times; the loops were unrolled ten times to reduce loop overhead. For each benchmark, we give the combined user and system CPU time and the time spent garbage collecting. All times are in micro-seconds. The benchmarks were run using version 0.67 of SML/NJ and version 0.9.3 of CML.

The first set of benchmarks (see table 3) measures the cost of some basic operations:

- **Null function call** This measures the time required to call the null function; this operation is about 12 instructions on the SPARC. Because of the compiler technology used by SML/NJ, the cost of this operation depends heavily on the calling context; if there are many registers to be saved, then the cost will be higher.

- **Thread switch** This measures the cost of an explicit context switch.

- **Thread spawn/exit** This measures the time it takes to spawn and run a null thread, which includes the cost of two context switches (the spawn operation switches control to the newly spawned thread and a terminating thread must dispatch a new thread).

- **Rendezvous** This measures the cost of a send/accept rendezvous between two threads.

- **Event rendezvous** This is an implementation of the rendezvous benchmark using transmit/receive and sync.

The second set (see table 4) measures the cost of several different implementations of a simple service. The simple service is essentially a memory cell; a transaction sets a new value and returns the old value. The implementations are:

- **Function call** This is the sequential implementation, which uses an own variable to keep track of the state between calls.

- **RPC** This uses a request/reply exchange implemented using send and accept.

- **Event RPC** This implements the request/reply exchange as an event value.

5.2 Garbage collection overhead

The high garbage collection overhead in these benchmarks is mostly a result of the way the current SML/NJ collector, which is a simple generational collector, keeps track of intergenerational references [App89]. Each time a mutable object is updated, a record of that update is added to the store-list. This store list is examined for potential roots at the beginning of each garbage collection. The implementation of CML uses a small number of very frequently updated objects: the thread ready queue, current thread pointer and channel waiting queues. This “hot-spot” behavior is the worst-case scenario for SML/NJ’s collector, destroying the \( O(\text{LIVE}) \) normally expected from copying collection. The collector also suffers from the problem of poor “real-time” responsiveness.

We have designed a new, multi-generational, collector for SML/NJ [Rep91a], which uses the page-protection techniques of [Sha87] and [AEL88] to implement the write barrier. This new collector improves the performance of CML in two ways: the hot-spot update behavior only incurs a constant cost for garbage collection and eliminating the store-list reduces the frequency of garbage collection and the cost of update operations. An instruction count analysis predicts a 25% reduction in the cost of a CML context switch. Unfortunately, at the time of this writing (March 1991), the new collector is not stable enough to run the CML benchmarks.

5.3 Analysis

We have noticed a discrepancy between the measured times and instruction counts on the SPARC processors. For example, a CML thread switch takes about 100 instructions, which suggests an average of \( \approx 5 \) cycles per instruction (CPI) on the SPARCstation 1. The SPARCstation 1 has a write-through cache, which we suspect is causing this high CPI figure. For the DECstation, the number is around 2 CPI, which seems more reasonable.

The measurements show that the penalty for using abstract interfaces (i.e., hiding channel communication in event values) is acceptable. Table 5 gives the ratio between the non-GC time of the event version and the non-event version of the two communication protocols we benchmarked.

These numbers demonstrate that we can build real system software using our high-level notation, without paying an unacceptable performance penalty. With specialized com-
### Table 2: Benchmark machines

<table>
<thead>
<tr>
<th>Machine</th>
<th>Processor</th>
<th>Memory</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARCstation 1</td>
<td>20MHz SPARC</td>
<td>28Mb</td>
<td>SunOS 4.1.1</td>
</tr>
<tr>
<td>Sun 4/490</td>
<td>33MHz SPARC</td>
<td>32Mb</td>
<td>SunOS 4.0.3</td>
</tr>
<tr>
<td>DECstation 3000/200</td>
<td>25MHz R3000</td>
<td>16Mb</td>
<td>ULTRIX 4.1</td>
</tr>
</tbody>
</table>

### Table 3: Basic concurrency operation benchmarks

<table>
<thead>
<tr>
<th>Operation</th>
<th>SPARC 1</th>
<th>Sun 4/490</th>
<th>DEC 5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>null function call</td>
<td>1.9 μS</td>
<td>1.0 μS</td>
<td>0.7 μS</td>
</tr>
<tr>
<td>thread switch</td>
<td>25</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>thread spawn/exit</td>
<td>85</td>
<td>38</td>
<td>22</td>
</tr>
<tr>
<td>rendezvous</td>
<td>94</td>
<td>45</td>
<td>27</td>
</tr>
<tr>
<td>event rendezvous</td>
<td>163</td>
<td>77</td>
<td>44</td>
</tr>
</tbody>
</table>

### Table 5: Cost of abstraction

<table>
<thead>
<tr>
<th>Machine</th>
<th>Event cost : non-event cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARCstation 1</td>
<td>Rendezvous : RPC : 1.73 : 1</td>
</tr>
<tr>
<td>Sun 4/490</td>
<td>Rendezvous : RPC : 1.71 : 1</td>
</tr>
<tr>
<td>DEC 5000/200</td>
<td>Rendezvous : RPC : 1.62 : 1</td>
</tr>
</tbody>
</table>

### Table 6: μSystem benchmarks

<table>
<thead>
<tr>
<th>Operation</th>
<th>SPARC 1</th>
<th>Sun 4/490</th>
<th>DEC 5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>null function call</td>
<td>0.3 μS</td>
<td>0.2 μS</td>
<td>0.4 μS</td>
</tr>
<tr>
<td>task switch</td>
<td>131</td>
<td>54.3</td>
<td>9.6</td>
</tr>
<tr>
<td>task create</td>
<td>371</td>
<td>256</td>
<td>40.0</td>
</tr>
<tr>
<td>send/receive</td>
<td>292</td>
<td>119</td>
<td>27.7</td>
</tr>
<tr>
<td>send/receive/reply</td>
<td>308</td>
<td>123</td>
<td>28.4</td>
</tr>
</tbody>
</table>

### 5.4 Comparison with the μSystem

Lastly, to put our measurements into perspective, we implemented a similar (allowing for linguistic differences) set of benchmarks in the μSystem, which is a C light-weight process library[890]. We ran these benchmarks on the same machines; the results are reported in table 6. The times are given in micro-seconds and represent the sum of the user and system CPU times; obviously there is no garbage collection overhead. Even though the μSystem provides a lower-level concurrency model (no selective communication, for example), and is implemented in a lower-level language, CML provides as good or better performance. This shows that we can have the advantages of the higher level language without sacrificing performance; a rare situation indeed.

### 6 Related work

There are many approaches to concurrent language design (see [AS83] for an overview); our approach is an offspring of the CSP-school of concurrent language design. CML began as a reimplementation of the concurrency primitives of PML[Rep88] in SML/NJ, but has evolved into a significantly more powerful language. PML in turn was heavily influenced by amber[Car86]. There have been other attempts at adding concurrency to various versions of ML. Most of these have been based on message passing ([Hol83], [Mat89], and [Ram90] for example), but there is at least one shared memory approach[Cm90]. As we have shown in this paper, message passing fits very nicely into SML. It allows an applicative style of programming to be used most of the time; the state modifying operations are hidden in the thread and channel abstractions. CML extends the message passing paradigm by making synchronous operations first-class, which provides a mechanism for building user-defined synchronization abstractions. While this idea was first proposed in [Rep88], we have made several significant improvements:

- **CML** improves the event type in several ways. The poll operation provides a cleaner semantics than the PML polling mechanism, and the wrapHandler operation provides needed support for writing wrappers. The guard and wrapAbort operations are important new features that allow substantially more sophisticated protocols to be implemented as events.
- Because CML is implemented in SML, objects such as threads, channels and the scheduling queue are heap allocated. This has two advantages: threads cannot overflow fixed-size stacks (a common problem in many thread packages), and the memory resources used by threads and channels are reclaimed by the garbage collector. The latter is heavily exploited in CML programs (see section 2.2).
- **CML** threads are significantly lighter-weight than those of most threads packages. Because the memory overhead of CML threads is very small (tens of bytes compared to...
Table 4: RPC protocol benchmarks

<table>
<thead>
<tr>
<th>Operation</th>
<th>SPARC 1 (μs)</th>
<th>GC (μs)</th>
<th>Sun 4/490 (μs)</th>
<th>GC (μs)</th>
<th>DEC 5000 (μs)</th>
<th>GC (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>function call</td>
<td>4.5</td>
<td>0</td>
<td>2.3</td>
<td>0</td>
<td>1.5</td>
<td>0</td>
</tr>
<tr>
<td>RPC</td>
<td>197</td>
<td>35</td>
<td>95</td>
<td>20</td>
<td>54</td>
<td>32</td>
</tr>
<tr>
<td>event RPC</td>
<td>252</td>
<td>35</td>
<td>117</td>
<td>20</td>
<td>68</td>
<td>33</td>
</tr>
</tbody>
</table>

kilo-bytes for PML), a more profligate use of threads is possible.

- CML provides integrated support for both low-level and stream-based I/O.

Our implementation techniques are not particularly novel. The use of timers to implement pre-emption is an old technique used by most threads packages, and the use of first-class continuations to implement concurrency goes back, at least, to [Wan80]. A number of papers have been published on the use of first-class continuations to implement concurrency in scheme (e.g., [Wan80], [HFW84] and [DH89]), but we break new ground in a couple of ways: we describe the implementation of first-class synchronous operations using first-class continuations; we describe a complete language for concurrent programming and its use; and we provide performance figures. The one published application of continuation-based concurrency in scheme (which we know of) claims that most scheme implementations do not implement continuations efficiently enough to support this use of concurrency [Haa90] (the techniques of [HDB90] may address this problem). Our performance numbers and experience “in-the-field” suggest that the opposite is true for CML.

Using concurrency to implement interactive systems has been proposed and implemented by several people. In [RG86] we made the argument that concurrency is vital for the construction of interactive programming environments. [Pik89] and [Haa90] describe experimental window systems built out of threads and channels, but neither of these were fast enough for real use.

7 Conclusions

We have described a higher-order concurrent language, CML, and its use in real-world applications. Our experience with CML “in-the-field” and our measurements of the performance of the implementation show that CML is a practical tool for building real systems. We feel that CML is unique in that it combines a flexible high-level notation with good performance.

CML is a stable system, and is freely available for distribution. The latest version of both CML and its manual are available via anonymous ftp from cs.cornell.edu; for more information send electronic mail to:

    cml-bugs@cs.cornell.edu

Acknowledgements

The work on eXene has benefited greatly from the contributions of Emden Gansner; in the form of both source code and suggestions. Tim Teitelbaum provided useful comments on various drafts of this paper.

References


Appendix – A brief introduction to Standard ML

This appendix gives a brief introduction to the major aspects of Standard ML. SML is formally defined in [MTH90]; for a tutorial introduction see [Har86]. It has the following important characteristics:

- **SML** is a higher-order language: functions are first-class values.
- **SML** is strongly typed: every expression has a type, which is inferred by the compiler. There are no run-time type errors.
- **SML** has a polymorphic type-system: the type of an expression inferred by the compiler is the most general possible type for the expression.
- **SML** is lexically scoped.
- SML has a pattern matching facility for decomposing structured values. This facility is used in *equational* definitions of functions and in a generalized *case-expression*.
- SML has a powerful datatype mechanism: datatype declarations introduce *constructors* that may be used to build values in expressions and to decompose values in patterns.
- SML has a type-safe exception mechanism.
- SML has a "state-of-the-art" module facility.

A few examples illustrate most of the features used in this paper.

SML has a predefined list type constructor, which is defined by the declaration:

```plaintext
datatype 'a list = nil | :: of ('a * 'a list)
```

The first line says that list is a type constructor with two constructors: nil, which is the polymorphic empty-list, and "::" (pronounced *cons*). Constructors are used in expressions to build values, and used in patterns to destruct values. Note that type constructors are postfix operators (the "*: a" is a type variable), with the exception of two built-in constructors: "->" for function types and "*" for tuple types. The second line declares "::" to be an infix operator.

Because lists are an important type, SML provides syntactic sugar for list patterns and expressions. The syntax

```plaintext
[l1, ..., ln]
```

is syntactic sugar for

```plaintext
l1::l2::...::ln::nil
```

The list reverse function uses this syntax for the empty list in its patterns:

```plaintext
fun rev l = let
  fun rev' ([], l) = l
  | rev' (x:r, l) = rev' (r, x::l)
  in
  rev' (l, [])
end
```

The tail-recursive function `rev'` is a function from a pair of lists to a list. It is defined equationally in two clauses. The first clause says to return the second argument if the first is the empty list; the second clause is a tail-recursive call which conses the head of the first argument to the second. The function `rev` is polymorphic; it will work on lists of any type. Tail-recursion is the standard way to code loops in SML.

A more complicated example is the following functional implementation of polymorphic FIFO queues:

```plaintext
abstype 'a queue = Q of
  {front : 'a list, rear : 'a list}
with
  fun queue () = Q{front = [], rear = []}
  fun insert (x, Q{front, rear}) = Q{front = x::front, rear = rear}
  exception EmptyQ
  fun remove (Q{front = [], rear}) = raise EmptyQ
  | remove (Q{front = x::r, rear}) = (x, Q{front = r, rear = rear})
  fun head q = let val (x, _) = remove q in x end
end (* abstype *)
```

The syntax

```plaintext
{l1 = e1, ..., ln = en}
```

defines a labeled record. This syntax is also used in patterns to destruct record values. It is possible to abbreviate a labeled record pattern in two ways, as illustrated by the following example. The abbreviated pattern:

```plaintext
(Qfront, ...)
```

is equivalent to the pattern

```plaintext
(Qfront = front, rear = rear)
```

This `abstype` declaration defines a new type constructor, queue, together with a collection of operations. A queue value is represented by the constructor Q applied to a record of two fields: front and rear, which are both lists. This declaration introduces the following bindings into the environment:

```plaintext
type 'a queue
val queue : unit -> 'a queue
val insert : 'a * 'a queue -> 'a queue
exception EmptyQ
val remove : 'a queue -> 'a * 'a queue
val head : 'a queue -> 'a
```

Because this is an `abstype` declaration, the representation of a queue is not visible outside the declaration. The type `unit` is the type with one member: the empty tuple ".().."

SML/NJ provides first-class continuations as an experimental extension[DHM91]. The interface is:

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

These are used in the implementation of CML to implement threads.