A Concurrent ML Tutorial

David Johnston[1]
Dept. Electronic Systems Engineering
University of Essex

October 4, 2002

[1] djjohn@essex.ac.uk

Contents

Chapter 1

Introduction

Concurrent Meta Language (CML) is a version of the functional language Meta Language (ML) with extensions to support concurrency. CML is an ideal way to design parallel applications combining the rapid prototyping capability of ML with a simple, powerful and well-defined model of parallelism that is derived in turn from the formal CSP (Communicating Sequential Processes) model.

The documentation available for CML is fairly succinct and though there is a little tutorial available on the Web, there is a overall lack of "hand-holding" material on how to actually use the facilities supplied or get a CML program up and running for the first time. The latter task proved extremely difficult in practice - and in lieu of documentation some demonstration software had to be reverse engineered.

The aim of this tutorial is to redress these deficiencies, which could prevent an excellent system from being picked up and more widely used. Some of the examples are taken directly from code fragments written by John Reppy (the developer of CML). However, all the applications in this tutorial are presented in a complete, self-contained and ready-to-run form.

CML's capabilities are even more powerful than those of `occam` - a language that was an earlier but still effective realisation of CSP. CML, unlike `occam`, supports the dynamic creation of processes and channels. CML also supports selection (or alternation) between a collection of both send and receive operations whereas occam can only select between receive operations.

The consequence of removing this restriction on selection lends a more robust and "communication eager" style to CML applications. A significant consequence is that while mistakes in `occam` programs will readily cause deadlock, mistakes in CML rarely do. Deadlock is after all notoriously difficult to identify and debug, and is probably the biggest challenge in parallel programming.

The CML code examples for this tutorial are not intended as realistic applications, but as the minimal testbed programs to show use of facilities in the language in an idiomatic manner. Some of the operations available, may be somewhat novel and difficult to understand for those accustomed to other parallel systems. For example, an understanding of the "wrap" and "guard" functions is critical to use effectively the "event calculus" model of CML, but the distinction between the two is rather subtle and confusing. Care has been taken to break in the more challenging concepts of the language. On the other hand, trivial facilities are grouped together in a miscellaneous section, where understanding or using these function calls is not at all difficult.

Chapter 2

Getting Up and Running

To bring in and then make available the facilities of CML either type in the following two lines at the interactive ML prompt or place them at the top of your ML source file.

```
CM.make();
open CML;
```

You will **also** require a file in your current directory with the name "sources.cm". This is like a makefile; you should locate a suitable "sources.cm" file that you can adapt for your use in the CML release. The CM.make() command is like issuing a conventional make.

Note that you **cannot** run a CML program from the top level command line, but you have to run it within a RunCML.doit context i.e. the "main" program is passed as the first argument to the RunCML.doit call. The second argument is the time-slice period that will be used in the pseudo-concurrency on a single processor. This defaults to 20mS if NONE is supplied as the argument. The RunCML.doit call only returns when none of the component processes can run any longer.

There is only one way to start a new process and that is via the "spawn" function, which returns an identifier (Thread IDentifier or TID) for that process. The following example starts a "mother" process which in turn initiates a "daughter" process. Both processes identify themselves by their TIDs. Signatures of introduced functions are:

```
val CM.make : unit -> unit
val RunCML.doit : (unit -> unit) * Time.time option -> OS.Process.status
val spawn : (unit -> unit) -> thread_id
val getTid : unit -> thread_id
val tidToString : thread_id -> string
```

Note that all new processes initiated must be a function with **no** input and **no** output i.e. they are of signature `unit->unit`. Let us imagine, in contrast, an arbitrary function `fn : 'a -> 'b` with non-null input and output. It makes sense to simultaneously supply a spawned function with some input. However, if we were to collect the output as part of the spawn, we would have essentially waited until the process is finished as the return value of a function is (in sensible code at least) the last thing computed. We could therefore sensibly spawn a function of type `fn : 'a -> unit` but this is not symmetric. Anyhow it will later be shown how to achieve this functionality with the basic spawn, and such a function called spawnc is indeed supplied as part of CML.

A second argument that supports the `unit -> unit` signature of spawned processes is to encourage a style where all processes are running in separate data spaces and can only legitimately pass data via channels.

```
CM.make();
open CML;
```

```
fun daughter () =
let
        val daughter_tid = getTid ();
in
        print ( "DAUGHTER: my tid = " ^ (tidToString daughter_tid) ^ "\n" )
end;

fun mother () =
let
        val mother_tid   = getTid ();
        val daughter_tid = spawn daughter;
in
        print ( "MOTHER: my tid = " ^ (tidToString mother_tid  ) ^
               " DAUGHTER's tid = " ^ (tidToString daughter_tid) ^ "\n" )
end;


RunCML.doit ( mother,  SOME(Time.fromMilliseconds 10) );
```

It can be noticed how little code infrastructure is required to initiate a multiple process application. Output should appear similar to:

```
DAUGHTER: my tid = [000004]
MOTHER: my tid = [000003] DAUGHTER's tid = [000004]
```

Chapter 3

Communication

Communication occurs synchronously across channels. A channel is typed, and can only be used to transmit data of that type. Signatures of introduced functions are:

```
val channel : unit -> 'a chan
val send : 'a chan * 'a -> unit
val recv : 'a chan -> 'a
```

The application sends a message between process A and process B.

```
fun main () =
let
        val ch = channel() : string chan;
        fun A () = send( ch, "hello" );
        fun B () = print ( "received " ^ recv(ch) ^ "\n" );
        val _ = spawn A;
        val _ = spawn B;
in
        ()
end;

RunCML.doit ( main,  SOME(Time.fromMilliseconds 10) );
```

Note that the communication only works because the channel "ch" is in the shared data space of "main". What happens if A and B are not inside "main"? Well, one could send the channel to each process as a data item, but a channel would be necessary to do this! Catch 22! In fact we can make the channel a parameter to A and B, and recast the application as below.

```
fun A ch () = send( ch, "hello" );
fun B ch () = print ( "received " ^ recv(ch) ^ "\n" );

fun main () =
let
        val ch = channel() : string chan;
        val _ = spawn (A ch);
        val _ = spawn (B ch);
in
        ()
end;

RunCML.doit ( main,  SOME(Time.fromMilliseconds 10) );
```

This is far more sensible architecturally - notice we have now solved the problem of spawning a function which takes a parameter. We place a dummy null argument as the last so-called "curried" parameter in the function declarations.

Chapter 4

Events

The "send" and "recv" functions are actually not CML primitives, but are derived instead from typed channel event primitives with signatures:

```
val sendEvt : 'a chan * 'a -> unit event
val recvEvt : 'a chan -> 'a event
val sync    : 'a event -> 'a
```

An event indicates a potential send or receive. It is only when an event is synchronised upon, that a send or receive actually happens. The best explanation of the semantics is to look at the derivation of "send" and "recv":

```
val send = sync o sendEvt; (* o indicates functional composition *)
val recv = sync o recvEvt;
```

An enabled event, that allows execution to go beyond the synchronisation point, indicates that both ends of the communication are ready to start. Why do we have events? It is so we can respond to those communications which are ready to go, without locking up the system waiting for a particular communication to first start and then complete (if indeed no deadlock has occurred). In short, events allow one to respond dynamically to a number of possible eventualities which may occur in an unpredetermined order. This does much to decouple applications and avoid deadlock. The key function here is "choose" which takes a list of events, and returns one which is ready to go when synchronised upon. Note there is no point in having events without a "choose" and vice versa!

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

The code below attempts to do two different communication between two processes. Which send event is synced upon is determined by a "choose" and similarly for the receive events. One might suspect that since both ends of the communication are waiting for something to happen, nothing actually does because there are no sends or receives outside the context of a choose to definitely start something off.

In fact, the code is genuinely non-deterministic: sometimes one communication will happen, sometimes the other. The semantic can be confusing because although `choose [ event1, event2 ]` has a return type of event: this does not mean that at this stage the actual event has been chosen. The evaluation of the choose is **lazy** and only carried out when the event is synced upon!

In this application, the communication which actually takes place is only chosen after the synchronisation at both ends, and this is an arbitrary choice form amongst the set of potential communications which can take place at this stage.

`occam` cannot do a "choose" (called an `ALT` in `occam`) on output channels only but not on input channels. So sends in `occam` are deterministic but receives, within an ALT construct, need

not be. This occam semantic is in some sense clearer: any receive corresponding to a committed send can take place. CML restores the symmetry because a "choose" can occur on both send and receive events and indeed on any combination of send and receive events. However, the types of the events selected between must be the same, so there has to be some syntactic jiggery pokery in the general case. This can be tricky as send events are null or unit events and receive events have the same type as the channel being used. However, this discrepancy can largely be programmed around!

This restriction on "choose" arises because ML is a statically typed language - though it should be remembered that types themselves are often expressed in terms of type variables rather than being literal types so the restriction is not as severe as it may otherwise appear. ML does lose some power due to the lack of dynamic typing but it is a practical and equitable compromise for both security and runtime performance. At the end of the day the strong typing does bring enormous benefits, and meeting the requirement for strong typing in fact determines the form and architecture of the CML. The nature of CML can be regarded as not arbitrary and almost derivable from the requirements of the CSP model meeting ML.

```
fun A ch1 ch2 () =
        sync
        (
                choose
                [
                        sendEvt( ch1, "hello 1" ),
                        sendEvt( ch2, "hello 2" )
                ]
        );

fun B ch1 ch2 () =
let
        val message = sync ( choose [ recvEvt(ch1) , recvEvt(ch2) ] );
in
        print ( "received " ^ message ^ "\n" )
end;

fun main () =
let
        val ch1 = channel() ;
        val ch2 = channel() ;
        val _ = spawn (A ch1 ch2);
        val _ = spawn (B ch1 ch2);
in
        ()
end;

RunCML.doit ( main,  SOME(Time.fromMilliseconds 10) );
val select = sync o choose;
```

In fact, there is a select function defined as:

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

to reflect the common pattern of usage as above - but "select" is derived rather than atomic - and should not make one think that the CML model is more complicated than it actually is!

Chapter 5

Polling

If events are somewhat difficult to understand, what about simply polling a channel to see if it is ready? In fact, CML does provide a couple of routines to do this - these will commit the communication action if it is possible at that instant.

```
val sendPoll : 'a chan * 'a -> bool (* returns true if send succeeded *)
val recvPoll : 'a chan -> 'a option (* returns SOME(<received value>) on success
                                       NONE                       on failure *)
```

Here are two possible implementations (see later for explanations of functions used):

```
fun sendPoll (ch,a) =
sync (choose [
                    wrap( sendEvt(ch,a),                          fn e => true),
                    wrap( timeOutEvt (Time.fromMicroseconds 1) , fn e => false)
            ]);

fun recvPoll ch =
sync (choose [
                    wrap( recvEvt(ch),                            fn a => SOME(a)),
                    wrap( timeOutEvt (Time.fromMicroseconds 1) , fn a => NONE)
            ]);
```

The code below attempts to set up two communications between two processes in an indeterministic order as before. When a successful communication occurs on a particular channel, the list of channel arguments has that channel filtered out. When the channel list is empty, all communications have been completed successfully.

```
(* delays for a second - see later for explanation *)
fun delay () = sync ( timeOutEvt( Time.fromMilliseconds(1000) ) );

(* l is list of channels on which a communication has yet to take place *)
fun A [] () = print "A: completed\n"
 |  A  l () =
let
        fun f ch = not ( sendPoll(ch, "hello") );
        val l2 =  List.filter f l ;
                        (* keep channels where a send has not happened *)
in
        (print "A\n"; delay (); A l2 () ) (* recursive call *)
end;
```

```
fun B [] () = print "B: completed\n"
 |  B  l () =
let
        fun success (NONE: 'a option) = false
         |  success (SOME(_)) = true;
        val l2 = List.filter ( not o success o recvPoll ) l;
                        (* keep channels where a recv has not happened *)
in
        ( print "B\n"; delay (); B l2 () ) (* recursive call *)
end;

fun main () =
let
        val ch1 = channel() : string chan ;
        val ch2 = channel() : string chan ;
        val _ = spawn (A [ch1, ch2]);
        val _ = spawn (B [ch1, ch2]);
in
        ()
end;

RunCML.doit ( main,  SOME(Time.fromMilliseconds 10) );
```

In tests, the above code **never** terminates and a successful communication **never** takes place! Polling does not permit the intense decoupling of events, and in order for the application to run to completion, one of the processes has to be replaced by one which does definite communications i.e. use either A or B below.

```
(* map applies function to each list element *)
fun A l () = ( map ( fn ch => send(ch, "hello") ) l; () );
fun B l () = ( map recv l; () );
```

Why is such an approach necessary? Well the polling functions are momentary so a successful communication with "sendPoll" and "recvPoll" relies on the coincidence of both ends polling at the same time. Now, it is not proposed that this will never work and the semantics of CML on this point are not obvious and certainly confused by pseudo-parallel operation. However, it is clearly something that should not be relied upon!

The traditional downside of polling is the application wasting time doing nothing either "busy-waiting" or "spin-locking" (choose your terminology). In fact, a delay was inserted in the above code to avoid the screen being filled with output from processes spin-locking very quickly. Of course, the smaller the delay the greater the probability of a coincident sendPoll and recvPoll, but in practice this never happened even when the delay was removed entirely.

In short, there appear to be no advantages to polling over the use of events. Indeed polling is considerably less powerful and introduces undesirable side-effects. The best advice is **not** to use the polling functions. The only justification for true polling is as an efficiency hack. For example, if a hardware bit is expected to be set very soon, checking for it twice could well be quicker than setting up event structures to deal with it.

Chapter 6

Special Events

Other events are possible apart from communication events:

```
val never : 'a event
val alwaysEvt : 'a -> 'a event
val timeOutEvt : Time.time -> unit event
val atTimeEvt : Time.time -> unit event
val joinEvt : thread_id -> unit event
val exit : unit -> 'a
```

6.1 never

An event value that is never enabled for synchronisation. Equivalent to:

```
choose []
```

6.2 alwaysEvt a

An event value that is always enabled for synchronisation and returns the value a on synchronisation. One possible implementation is:

```
fun alwaysEvt a =
let
        val ch = channel();
in
        spawn( fn () => send( ch, a) ); recvEvt ch
end;
```

6.3 atTimeEvt t

Creates an event that is enabled at a given absolute time t (where t has type "time").

6.4 timeOutEvt t

Will provide an event that is enabled after time interval t following synchronisation (not after interval t following event creation). A delay of one second will be **always** be given by `sync (timeOutEvt (Time.fromSeconds 1))`. This is useful for providing a time-out event. Compare this with absolute event:

```
atTimeEvt (  (Time.+)( Time.now(), Time.fromSeconds 1) )
```

which has a second before its use-by-date! Thereafter it will cause no delay! Later it will be shown how to derive `timeOutEvt` from `atTimeEvt`.

6.5 `joinEvt tid`

This will create an event that is enabled on termination of the associated thread. Basically it is a way of waiting for the thread to finish.

6.6 `exit ()`

This will explicitly terminate the thread from which it is called. It is not an event, but may cause the enabling of a joinEvt event. You can see the return type of exit is arbitrary i.e. `a'`. This is because exit does not return and so can blend in (type-wise) wherever it is put!

Chapter 7

Event Functions

The following functions are introduced:

```
val wrap : 'a event * ('a -> 'b) -> 'b event
val guard : (unit -> 'a event) -> 'a event
val withNack: (unit event -> 'a event) -> 'a event
val wrapHandler : 'a event * (exn -> 'a) -> 'a event
```

7.1 wrap

"wrap" is a way of changing the type of an event. The following code shows how to choose between different receive event types. A "wrap" is used to turn each different receive event type into a common receive event type. Note all send events are null events and so can be dealt with in the same way.

```
fun A ch1 ch2 () =
sync
(
        choose
        [
                timeOutEvt (Time.fromMicroseconds 1), (* 0 uS => no comms occurs *)
                sendEvt( ch2, 1234      ) ,
                sendEvt( ch1, "hello 1" )
        ]
);

datatype universal = INT of int | STRING of string;

fun universal_to_string    (INT(i)) = Int.toString i
 |  universal_to_string (STRING(s)) = s;

fun B ch1 ch2 () =
let
        val message = sync ( choose [
                                        wrap( recvEvt(ch1) , fn s => STRING(s)  ),
                                        wrap( recvEvt(ch2) , fn i => INT(i)     )
                                ] );
in
        print ( "received " ^ universal_to_string(message) ^ "\n" )
end;
```

```
fun main () =
let
        val ch1 = channel() ;
        val ch2 = channel() ;
        val _ = spawn (A ch1 ch2);
        val _ = spawn (B ch1 ch2);
in
        ()
end;

RunCML.doit ( main,  SOME(Time.fromMilliseconds 10) );
```

Note we have placed a timeout event in the send choose for further illustration. However, there are no communication problems in this application so the time event only triggers in this instance if a zero timeout is given.

7.2 guard

A guard is for pre-synchronisation actions. A guard operation propagates an event. However, the function which does the propagation is not actioned by the call to guard. Instead this function is called when a synchronisation is performed on the output event. The null argument is present to prevent the immediate evaluation of the function.

Paradoxically the output of guard is available **before** the code which produces this output is executed! It should be realised that the output is only an event, which only becomes meaningful when synchronised. The term "pre-synchronisation" is applicable because the function is executed and only then produces an **enabled** output event.

The first example in the use of the guard shows the implementation of a relative timeout in terms of an absolute temporal event.

```
fun timeout t = guard ( fn () => CML.atTimeEvt (Time.+ (t, Time.now())) );

fun main () =
let
        val e = timeout (Time.fromSeconds 1);;
        val _ = sync e;
in
        ()
end;

RunCML.doit ( main,  SOME(Time.fromMilliseconds 10) );
```

The function which evaluates the absolute time at which to enable the output event is only executed when the sync is called
bfnot when the guard is called. The results is that the sync always causes a delay of exactly one second. The effect is of a relative delay.

The second example uses this prior evaluation to ensure that two channels are used more or less evenly (to within four messages) in an indeterminate system.

```
fun sendf s ch () = ( send(ch, s); sendf s ch () );

fun receivef an bn a b () =
let
      val message =
```

```
      sync
      (
            choose
            [
                  guard( fn () => if (an >= bn + 4) then never else (recvEvt a) ),
                  guard( fn () => if (bn >= an + 4) then never else (recvEvt b) )
            ]
(*** this code would work identically
            choose
            [
                  if (an >= bn + 4) then never else (recvEvt a),
                  if (bn >= an + 4) then never else (recvEvt b)
            ]
***)
      );
      val an = if ( message = "A" ) then (an + 1) else an;
      val bn = if ( message = "B" ) then (bn + 1) else bn;

      val _ = print( "got " ^ message ^
                     " an is " ^ Int.toString(an) ^
                     " bn is " ^ Int.toString(bn) ^
                     " disparity is " ^ Int.toString(abs(an - bn)) ^"\n");
in
        receivef an bn a b ()
end;

fun main () =
let
        val a = channel();
        val b = channel();
in
        spawn( sendf "A" a );
        spawn( sendf "B" b );
        spawn( receivef  0 0 a b);
        ()
end;
```

In fact, this example is not great as the guard can be removed, and the application will still behave the same. However, it does show how the guard function fits in with the related use of guarded channels in `occam`, where the choice of channels examined in an alternation can be determined by boolean conditionals. A guard is much more general operation in CML.

The third example is taken from John Reppy, and gives a number of different client/server implementations using CML:

```
val CAPITALISE = implode o (map Char.toUpper) o explode; (* capitalise a string *)

fun server reqCh replyCh () = send(replyCh, CAPITALISE( recv reqCh) );
(* capitalisation server *)

fun client reqCh replyCh () =
let
        fun clientCallEvt1 x = wrap( sendEvt(reqCh, x), fn () => recv replyCh);
        fun clientCallEvt2 x = guard ( fn () => ( send(reqCh,x); recvEvt replyCh ));
        fun clientCallEvt3 x = guard
```

```
                                ( fn () =>
                                        (
                                            spawn ( fn () => send(reqCh,x) );
                                            recvEvt replyCh
                                        )
                                );
        val clientCallEvt = clientCallEvt3;
                            (* also try clientCallEvt1 and clientCallEvt2 *)
        val e = clientCallEvt "Hello World";
        val reply = sync e;
in
        print ( reply ^ "\n");
end;

fun main () =
let
        val reqCh = channel();
        val replyCh = channel();
in
        spawn( server reqCh replyCh );
        spawn( client reqCh replyCh );
        ()
end;
RunCML.doit ( main,  SOME(Time.fromMilliseconds 10) );
```

The application shows how a client/server system, can have various commit points. `clientCallEvt1` suspends the client until the server has responded. `clientCallEvt2` returns an event, rather than auctioning any communication. This event then can be used in a choose, giving greater flexibility rather than waiting around for something to happen i.e. `clientCallEvt2` could be used with a timeout. `clientCallEvt3` does the same but spawns a sending process, so other things can be done by the client process in the meanwhile. Note that the server can still only deal with one request at a time. A further extension is to spawn a specific evaluation process in the server and dedicate a specific reply channel to each client.

```
val CAPITALISE = implode o (map Char.toUpper) o explode; (* capitalise a string *)
fun server reqCh  () =
let
        val ( replyCh, x) = recv reqCh;
in
        spawn ( fn () => send(replyCh, CAPITALISE(x) ) ); server reqCh ()
end;

fun client reqCh () =
let
        fun clientCallEvt4 x =  guard
                                (
                                        fn () =>
                                        let
                                                val replyCh = channel();
                                        in
                                                spawn( fn () => send(reqCh, (replyCh,x) ) );
                                                recvEvt replyCh
                                        end
                                );
```

```
        val e = clientCallEvt4 "hello world with clientCallEvt4";
        val reply = sync e;
in
        print ( reply ^ "\n");
end;

fun main () =
let
        val reqCh = channel();
in
        spawn( server reqCh  );
        spawn( client reqCh  );
        ()
end;

RunCML.doit ( main,  SOME(Time.fromMilliseconds 10) );
```

CML allows parallel architectures such as the client/server one above to be experimented with at little cost. Four generations of increasingly superior client/server solutions are given above.

7.3 withNack

"withNack" is again a way of executing code while transmitting an event, similar to guard i.e. pre-synchronisation. The supplied function will be executed on synchronisation as with guard, but this time the argument is a "negative acknowledgement" unit event (rather than a unit). If the return value is **not** chosen in a synchronisation, then the "negative acknowledgement" event is enabled. The code below picks up, for each of the two communications, the occasion on which each channel is not used. The mechanism is clearly powerful but its application use is not immediately apparent. It is suggested as a way of informing servers than a client has aborted a transaction.

```
fun A ch1 ch2 () = (send(ch1, "hello from channel 1");
                    send(ch2, "hello from channel 2"));

fun B ch1 ch2() =
let
        fun check_event evt name () =
        let
               val _ = sync evt;        (* stalls here if event not enabled *)
        in
               print ( name ^ " not chosen \n")
        end;

        (* "return" value is event for "choose" *)
        fun g ch name nack  =   ( spawn ( check_event nack name ); recvEvt ch );

        val e = choose [ withNack (g ch1 "channel 1" ),
                         withNack (g ch2 "channel 2")] ;
in
        print ( "RECEIVED: " ^ sync(e) ^ "\n" );
        print ( "RECEIVED: " ^ sync(e) ^ "\n" )
end;

fun main () =
```

```
let
        val ch1 = channel() : string chan;
        val ch2 = channel() : string chan;
in
        spawn (A ch1 ch2);
        spawn (B ch1 ch2);
        ( )
end;

RunCML.doit ( main,  SOME(Time.fromMilliseconds 10) );
```

7.4 wrapHandler

"wrapHandler", similarly to "guard", "wrap" and "withNack" passes an event through. If there is an exception in the post-synchronisation code the exception will be passed to the supplied exception handling function. In the application below the string announcing the exception will be received. If the exception is removed the application will receive the capitalised input string instead.

```
val CAPITALISE = implode o (map Char.toUpper) o explode; (* capitalise a string *)

fun exception_handler e   = " ########### An Exception Has Occurred ############";

fun cause_an_exception () =  ( 5 div 0 ; () );

fun caps s =  (  cause_an_exception ();  CAPITALISE s );
(* capitalises a string and causes an exception *)

fun A ch () = send(ch, "hello from channel");

fun B ch () =
let
        val re = wrapHandler(  wrap ( recvEvt ch, caps ), exception_handler);
        val s  = sync re;
in
        print ("RECEIVED: " ^ s ^ "\n")
end;

fun main () =
let
        val CH = channel() : string chan;
in
        spawn (A CH); spawn (B CH); ( )
end;

RunCML.doit ( main,  SOME(Time.fromMilliseconds 10) );
```

Chapter 8

Remote Procedure Call

Let us consider the spawning of a function of arbitrary type `fn : 'a -> 'b`. This is in essentially a Remote Procedure Call (RPC), and can be implemented readily as below. This is similar to the operation of the server shown earlier, except that an RPC executes mobile dynamic code rather than static code.

```
fun RPC f arg  =
let
        val ch = channel();
        fun f' arg ch f () = send( ch, f(arg) ); (* function at end of channel *)
        val _ = spawn ( f' arg ch f );           (* launch it *)
in
        recv ch                                  (* receive result *)
end;

fun main () = print ( "sqrt(5.0) = " ^ Real.toString(RPC Math.sqrt 5.0) ^ "\n");

RunCML.doit ( main,  SOME(Time.fromMilliseconds 10) );    (* invokes the RPC *)
```

To avoid the hold-up of waiting for the RPC to return, a more sensible return type is an event.

```
fun RPC f arg  =
let
        val ch = channel();
        fun f' arg ch f () = send( ch, f(arg) ); (* function at end of channel *)
        val _ = spawn ( f' arg ch f );           (* launch it *)
in
        recvEvt ch                               (* result event *)
end;

fun main () =
let
        val e = wrap(  RPC Math.sqrt 5.0, fn r => "sqrt(5.0) = " ^ Real.toString(r) ^ "\n");
in
        print (sync(e))
end;

RunCML.doit( main ,  SOME(Time.fromMilliseconds 10) );    (* invokes the RPC *)
```

Chapter 9

Exploration of the semantics of wrap and guard

To understand the semantics of "wrap" and "guard" (and in particular execution order) we can produce versions of these functions which print at the beginning and end.

```
fun guard_p string f =
let
        val _ = print ("PRE" ^ string);
        val res =  guard f;
        val _ = print ("POST" ^ string);
in
        res
end;

fun wrap_p string evt =
let
        val _ = print ("PRE" ^ string);
        val res = wrap(evt,  fn x => x );
        val _ = print ("POST" ^ string);
in
        res
end;
```

Tests revealed that corresponding "PRE" and "POST" print-outs always appear consecutively in any test run i.e. a single print statement is sufficient to establish execution order:

```
fun guard_p string f = (print string; guard f);
fun wrap_p string evt = (print string; wrap(evt,  fn x => x ) );
```

Particular named wrap and guard operations were then created:

```
fun wrapA evt = wrap_p "wrapA\n" evt;
fun wrapB evt = wrap_p "wrapB\n" evt;
fun guardA f = guard_p "guardA\n" f;
fun guardB f = guard_p "guardB\n" f;
```

Then nested expressions were created around a basic event:

```
fun mk_event () = (print "make event\n"; alwaysEvt 5);
```

The following two tables show the execution order of two nested expressions as established by the order of the individual print statements placed within the component functions.

9.1 First Expression

```
sync (wrapB( (guardB (fn () => wrapA( guardA ( mk_event ))))));
       2       1                _       _        _            => order before sync
       2       1                4       3        5            => order  after sync
```

Before the outer sync the innermost code is **not** executed. This is quite counterintuitive that actions on the outside affect things on the inside! The order of execution is traditionally inside out: simply from the way nested expressions are evaluated. guardB holds back the inner code but returns an event, whose type is changed by wrapB - giving the local inner to outer ordering. The sync then gets to work outside in! When the inner code is executed guardA holds back the mk_event, but returns an argument for wrapA to work with. The sync then propagates to execute the mk_event.

9.2 Second Expression

```
sync ( guardB( fn () => wrapB( guardA (fn () => wrapA ( mk_event ()))))));
        1                _       _                _       _   => order before sync
        1                3        2               5       4   => order  after sync
```

guardB stalls the inner code which gets called on the sync. guardA stalls the most inner code and propagates its event result upwards to wrapB. The sync propagates inwards and wrapA works on the event value created by mk_event.

9.3 Conclusion

The guard is responsible for breaking the conventional execution order. The altered execution order may be defined as work from the outermost guard outwards and then move on to the inner code. The same will be true for other pre-synchronisation operators. Note that the dataflow is **still** inside out. The event flow goes with the execution order but an event is a type not data, until it is synced. Then the data flows in the conventional manner but behind the scenes!

Chapter 10

Miscellaneous CML API Calls

```
val version : {date:string, system:string, version_id:int list}
val banner : string
```

These give version identification information.

```
datatype order = LESS | EQUAL | GREATER; (* from standard ML basis
                                            - here for reference only *)
val sameTid : thread_id * thread_id -> bool
val compareTid : thread_id * thread_id -> order
```

These functions give information on the total ordering of Thread IDs.

```
val hashTid : thread_id -> word
```

returns a hashing of the thread ID.

```
val spawnc : ('a -> unit) -> 'a -> thread_id
```

A version of spawn that takes an input argument. Here is an implementation of spawnc in terms of spawn:

```
fun spawnc (f:'a -> unit ) (a:'a) =
let
        fun g a () = f a;
in
        spawn ( g a )
end;
```

```
val yield : unit -> unit
```

This function can be used to implement an explicit context switch. Since CML is preemptively scheduled, it should never be necessary for user programs to call this function.

```
val sameChannel : 'a chan * 'a chan -> bool
```

returns true, if the two channels are the same channel.

Chapter 11

CML Summarised

The entire CML API is presented below. Each component is categorised

```
val version : {date:string, system:string, version_id:int list}   MISC TRIVIAL
val banner : string                                               MISC TRIVIAL
type 'a event = 'a ?.RepTypes.event                               support
datatype thread_id = ...                                          support
val getTid : unit -> thread_id                                    support
val sameTid : thread_id * thread_id -> bool                       MISC TRIVIAL
val compareTid : thread_id * thread_id -> order                   MISC TRIVIAL
val hashTid : thread_id -> word                                   MISC TRIVIAL
val tidToString : thread_id -> string                             support
val spawnc : ('a -> unit) -> 'a -> thread_id                      DERIVED
val spawn : (unit -> unit) -> thread_id                           PRIMITIVE
val exit : unit -> 'a                                             PRIMITIVE
val joinEvt : thread_id -> unit event                             PRIMITIVE
val yield : unit -> unit                                          MISC TRIVIAL
datatype 'a chan = ...                                            support
val channel : unit -> 'a chan                                     support
val sameChannel : 'a chan * 'a chan -> bool                       MISC TRIVIAL
val send : 'a chan * 'a -> unit                                   DERIVED
val recv : 'a chan -> 'a                                          DERIVED
val sendEvt : 'a chan * 'a -> unit event                          PRIMITIVE
val recvEvt : 'a chan -> 'a event                                 PRIMITIVE
val sendPoll : 'a chan * 'a -> bool                               DERIVED
val recvPoll : 'a chan -> 'a option                               DERIVED
val never : 'a event                                              DERIVED
val alwaysEvt : 'a -> 'a event                                    DERIVED
val wrap : 'a event * ('a -> 'b) -> 'b event                      PRIMITIVE
val wrapHandler : 'a event * (exn -> 'a) -> 'a event              PRIMITIVE
val guard : (unit -> 'a event) -> 'a event                        PRIMITIVE
val withNack : (unit event -> 'a event) -> 'a event               PRIMITIVE
val choose : 'a event list -> 'a event                            PRIMITIVE
val sync : 'a event -> 'a                                         PRIMITIVE
```

MISC TRIVIAL	in the miscellaneous trivial section
support	critical but supporting role
DERIVED	derivable from other operations
PRIMITIVE	not yet established as derivable!

Process Control	Events	Event Operators
spawn	sendEvt	wrap
exit	recvEvt	wrapHandler
joinEvt	atTimeEvt	guard
		withNack
		choose
		sync

```
val select : 'a event list -> 'a                                    DERIVED
val timeOutEvt : Time.time -> unit event                            DERIVED
val atTimeEvt : Time.time -> unit event                             PRIMITIVE
```

Of course, it is possible to choose different sets of operations as primitive. However, those designated as primitive above are probably the most semantically clean. The primitives are summarised in their respective categories above. These are satisfyingly low in number: 3 for process control, 3 to create primitive events and 6 event operators. wrapHandler and withNack deal with negative occurrences (i.e. exceptions or branches not taken) so it is more difficult to envision them as building blocks. However, all the other components form a clean conceptual set.