# Prototyping Application Models in Concurrent ML

David Johnston, Martin Fleury, and Andy Downton

Dept. Electronic Systems Engineering, University of Essex, Wivenhoe Park, Colchester, Essex, CO4 3SQ, UK
{djjohn, fleum, acd}@essex.ac.uk
http://www.essex.ac.uk/ese/research/mma_lab/index.htm

**Abstract.** We report on the design of parallel applications using harnesses, which are abstractions of concurrent paradigms. A split/combine harness is prototyped using the functional language Concurrent ML (CML). The event model simplifies harness development by unifying the structure of serial and parallel algorithms. The portability of the harness between different hardware (both serial and parallel) is planned through the compact and elegant CML programming model.

## 1 Introduction

Concurrent Meta-Language (CML) is a version of the well-established functional language ML with extensions for concurrency by Reppy [16]. Concurrent processes cooperate through CSP [10] channels *i.e.* unbuffered communication. The instantiation of CSP formalism within CML is more powerful than that of the classical version of the parallel language occam2 [6], specifically:

1. dynamic recursion, process and channel creation
   (*c.f.* no recursion, static processes and static channels in occam2)
2. selection between channels based on input and output events
   (*c.f.* just input selection in occam2 for implementation reasons)
3. functions can be sent over channels
   (side effect of ML treating code and data the same)
4. events and event functions
   (*c.f.* only synchronised communication in occam2)

Modern occam [1] has absorbed some of these capabilities *e.g.* recursion and the ability to send processes over channels. It is worth also comparing CML with the occam2-like Handel-C [17], a silicon compiler language targeted at FPGAs, where 2 and 3 are also present. Modern occam and CML are general purpose in nature rather than static in the tradition of embedded languages. Parallel designs in both languages could map onto the fine-grained parallelism of Handel-C or onto systems composed of distributed processor nodes.

This paper demonstrates, through example, how the features of CML allow the rapid prototyping of software infrastructures (or harnesses) that support

generic application-level parallelism. Such harnesses have been available within occam2 for some time, but following through the consequences of the CML event model and CML's functional nature results in a different and somewhat surprising outcome. Section 2 presents the context of the work within a larger development method. CML is introduced in Section 3 though some small illustrative examples. The main technical content within Section 4 builds upon these earlier examples to show how support for parallel execution is prototyped within CML. Section 5 presents related work, and the paper ends with conclusions and a description of the current status of the work.

## 2 Development Method

The key idea behind our project[1] is that the same paradigms of parallelism occur in many different applications. The execution of a single application is characterised by a dynamically changing set of such paradigms. Accordingly, a number of customised harnesses for each paradigm detected dynamically within the executing application can be transparently launched.

To illustrate, consider the simple split/combine application model for parallelism where the input can be split recursively; the parts processed in parallel; and the outputs combined. Figure 1 shows how this and other generic structures can be implemented as a harness containing the parallelism ($H_{CML}$ or $H_{C++}$) in conjunction with user's application-specific code ($A''_{ML}$ or $A'_{C++}$ respectively) which does not. The Figure presents a variety of software engineering routes because there is a choice of working in CML or a more conventional language such as C++. The use of CML is not an end in itself (though there is useful software output as a side effect) but a means to rapidly prototype a C++ version of the system, which a C++ developer can use off-the-shelf. The splitting of an application into a harness and serial application code is shown both for CML and C++. However, the solid arrows represent which activities are intended exclusively for systems developers *i.e.* they are capturing the real parallelisations and then abstracting them within general harnesses so users need not be concerned with the parallelisation of specific applications. If the user is prototyping in ML, then the translation to C++ is possible either as sequential code or after fitting into a CML harness.

## 3 CML Examples

### 3.1 Example 1 : Communication-Based Remote Procedure Call

To illustrate the use of CML, the following code performs a remote procedure call (RPC). The server applies a function (`f`) to the input argument (`arg`) and sends the result down a channel (`ch`) to the RPC client. For its part, the client creates the channel; spawns the server process with its arguments and receives a result from the channel. The client is placed within a `main` process so it can be run in the CML environment using `RunCML.doit`.

[1] Rapid Parallel Prototyping in the Image/Multimedia Domain
EPSRC Contract: GR/N20980

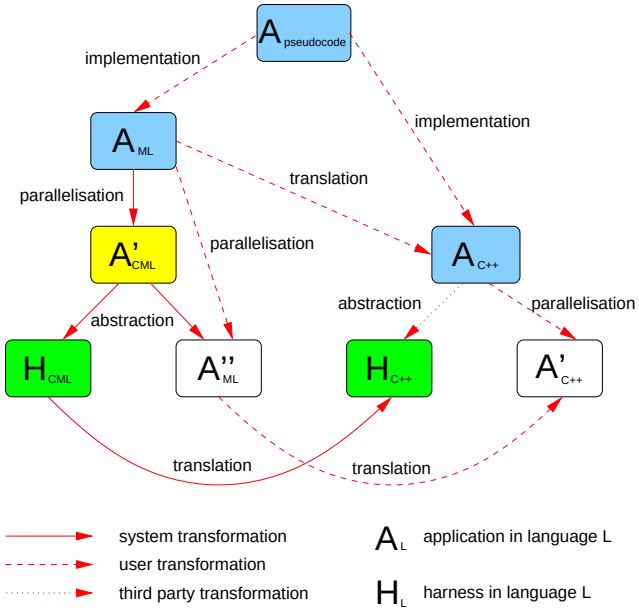


**Fig. 1.** overall development methodology

```
fun rpc_server ( ch, f, arg ) = send( ch, f(arg) );

fun rpc_client f arg =
let
     val ch = channel();
     val tid = spawnc rpc_server ( ch, f, arg );
in
     recv( ch )
end;

fun main () = print ( Real.toString( rpc_client Math.sqrt 2.0 ) ^ "\n" );

RunCML.doit ( main, NONE );
```

### 3.2 Example 2 : Event-Based Remote Procedure Call

The second example performs the same actions as the first, only it is written in terms of the primitive event functions of CML: `sendEvt` and `recvEvt` instead of the derived `send` and `recv`. The communication only happens after the event returned by `sendEvt` and the corresponding event returned by `recvEvt` are both explicitly synchronised using the `sync` operator. In the server, the send event is directly synchronised; whereas the client actually returns a receive event which is synchronised at a higher level in the code. These useful client and server routines will be used later in Sections 4.3, 4.4 and 4.5.

```
fun rpc_server( ch, f, arg ) = sync( sendEvt( ch, f(arg) ) );

fun rpc_client f arg =
let
     val ch = channel();
     val tid = spawnc rpc_server ( ch, f, arg );
in
     recvEvt( ch )
end;

fun main () = print ( Real.toString( sync ( rpc_client Math.sqrt 2.0 )  ) ^ "\n" );

RunCML.doit ( main, NONE );
```

### 3.3 Example 3

A `guard` event operator implements a delay in terms of the absolute temporal event primitive `atTimeEvt`. The `guard` operator performs pre-synchronisation actions or may be informally considered to "push" an action back [12]. The absolute time for the output event is only evaluated when the `sync` is called and not when `timeout` is called. The result is that synchronising on event `e` always causes a delay of exactly one second.

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

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

RunCML.doit ( main, NONE );
```

## 4 Split/Combine Application Harness

CML allows the rapid prototyping of a parallel harness which implements the split/combine model described in Section 2. This is shown below in five stages starting with a serial harness.

### 4.1 Serial Hierarchical Harness - SH_harness

The harness calls itself twice recursively if the problem is still to be split. Otherwise the input data is processed normally. Recursion is limited by simple recursion depth, though in a practical implementation this would be dynamic recursion through monitoring of performance as a function of granularity. Note the combine function is an output of the split function, so it can be a customised inverse operation.

```
(*
 * argument     description
 * --------     -----------
 * depth        current recursion depth
 * max_depth    maximum recursion depth allowed
 * f            function that does processing of input to output
 * split        function that splits input AND provides matching
 *              function which combines the corresponding outputs
 * input        input data to application
 *)

fun SH_harness ( depth, max_depth ) ( f, split ) input =
if ( depth < max_depth )
then
     let
          val ( combine, input1, input2 ) = split input;
          val daughter = SH_harness ( depth+1, max_depth ) ( f, split );
          val output1 = daughter input1;
          val output2 = daughter input2;
     in
          combine output1 output2
     end
else
     f input;
```

### 4.2 Communicating Parallel Hierarchical Harness - CPH_harness

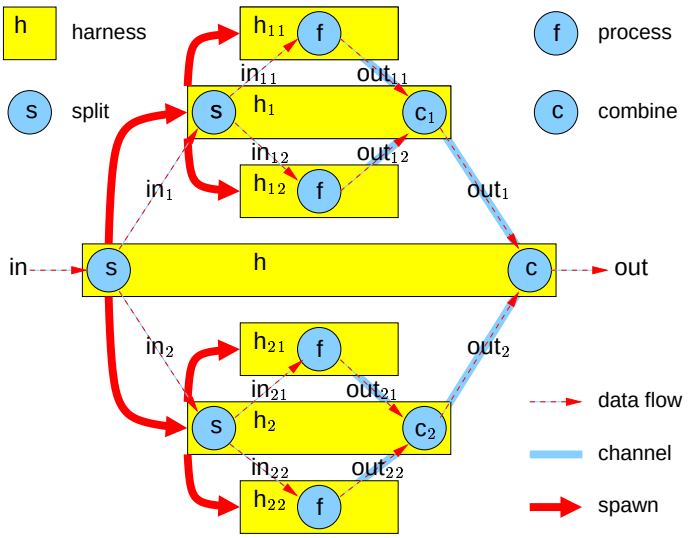


**Fig. 2.** communicating parallel hierarchical split/combine harness

An extra channel argument is supplied for the harness to send its result rather than returning it. Parallel execution is obtained by spawning two further versions of the harness instead of calling them sequentially in turn. To achieve

parallelism the results must be received *after* both processes have been spawned. Note that this code will have to naïvely wait for the first harness spawned to finish even though the second could have finished earlier. Corresponding sends and receives occur confusingly between different recursion levels of the harness. Figure 2 shows the process diagram. The subscript sequences indicate where each item is in the binary tree hierarchy. Each harness is spawned with its input parameters, while the output results are communicated by channel.

```
fun CPH_harness ( depth, max_depth ) ( f, split ) ch input =
if ( depth < max_depth )
then
     let
          val ch1 = channel();
          val ch2 = channel();

          val ( combine, input1, input2 ) = split input;
          val daughter = CPH_harness ( depth+1, max_depth ) ( f, split );

          val _ = spawnc (daughter ch1) input1;
          val _ = spawnc (daughter ch2) input2;

          val output1 =  recv(ch1);
          val output2 =  recv(ch2);

     in
          send( ch, combine output1 output2 )
     end
else
     send( ch, f input );
```

### 4.3 Event-Explicit Parallel Hierarchical Harness - EEPH_harness

Explicit communication is now done within the RPC functions, so the harness is abstracted from channel use. `recvEvt` and `sendEvt` appear together in the RPC calls using the same channel, so less channel management is required. If the `recvEvt` in the client were a `recv` the harness would spawn the daughter harnesses sequentially - not what is required!

```
fun EEPH_harness ( depth, max_depth ) ( f, split ) input =
if ( depth < max_depth )
then
     let
          val ( combine, input1, input2 ) = split input;
          val daughter = EEPH_harness ( depth+1, max_depth ) ( f, split );

          val event1 = rpc_client daughter input1;
          val event2 = rpc_client daughter input2;

          val output1 = sync(event1);
          val output2 = sync(event2);
     in
          combine output1 output2
     end
else
     f input;
```

### 4.4 Event-Implicit Parallel Hierarchical Harness - EIPH_harness

If the harness purely deals in events, synchronisation can be done at the last possible moment just before the data is about to be sent on to provide maximum decoupling. The `event_and` binary event operator "fires" only when both individual events would fire. The `alwaysEvt` operator is necessary to turn a value into an (always enabled) event of that value.

```
fun event_and combine event1 event2 =
let
        val event1' =  rpc_client sync event1;
        val event2' =  rpc_client sync event2;
in
         guard ( fn () => alwaysEvt ( combine ( sync event1' ) ( sync event2' ) ) )
end;

fun EIPH_harness ( depth, max_depth ) ( f, split ) input =
if ( depth < max_depth )
then
        let
                val ( combine, input1, input2 ) = split input;
                val daughter = EIPH_harness ( depth+1, max_depth ) ( f, split );
                val event1 = rpc_client daughter input1;
                val event2 = rpc_client daughter input2;
        in
                event_and combine event1 event2
        end
else
        alwaysEvt( f input );
```

### 4.5 Unified Harness - U_harness

The previous parallel harness is identical to the serial harness apart from the use of three higher order functions in three places to abstract the parallelism. Both the serial and parallel harnesses can thus be instanced as parameterised versions of the same generic harness as below. It is difficult to visualise an event-based solution compared to a message-passing one, but the communication patterns are still those shown in Figure 2. However, the timing and software engineering characteristics are considerably better.

```
fun U_harness ( exec, merge_using, return ) ( depth, max_depth ) ( f, split ) input =
if ( depth < max_depth )
then
     let
          val ( combine, input1, input2 ) = split input;
          val daughter = U_harness ( exec, merge_using, return )
                                   ( depth+1, max_depth ) ( f, split );
          val output1 = exec daughter input1;
          val output2 = exec daughter input2;
     in
          merge_using combine output1 output2
     end
else
     return ( f input );

val id = fn a => a; (* identity operator *)

val    SH_harness = U_harness   ( id        , id       , id        );
val  EIPH_harness = U_harness   ( rpc_client, event_and, alwaysEvt );
```

## 5 Related work

The concept of parallel processing paradigms is almost as old as parallel processing itself. For example, in [11] the 'divide-and-conquer' paradigm is characterized as a suitable for architectures in which dynamic process creation is possible. The concept of a harness which encapsulates or abstracts the underlying parallel paradigm or communication pattern seems to have emerged from occam and the transputer, for example in the TINY communications harness [2, 9]. It also seems to have preceded the pattern movement in conventional software [7]. The parallel paradigm is expressed in its clearest form, sometimes called an 'algorithmic skeleton' [3], in functional languages. Arising from the algorithmic skeleton approach has come the claim that parallel processing can be executed by functional composition of a restricted number of paradigms [5], such as the pipeline and the task queue, which claim we remain neutral on.

Unfortunately, attempts to implement a parallel architecture suitable for processing functional languages seem to have floundered [15]. Therefore it has been suggested [4], that functional languages are best used as a way of expressing parallel decomposition in a non-parallel language. In [14], this approach was indeed taken, but the prototype was in the sequential (*i.e* Standard) version of ML, before transferring to occam. In this paper, the approach has been taken one stage further, in that a concurrent functional language, CML, has been used to express parallelism. Hammond & Michaelson [8] give an overview of recent work in the parallel functional field.

## 6 Conclusion and Future Work

CML has proved an effective tool for the development of parallel harnesses: the split/combine harness shown is only one of a number of harnesses that have been developed in this way. The CML model of parallelism is small, clean and elegant providing an ideal porting layer that should enable the harnesses to be moved to a variety of parallel architectures in future. Sadly, execution within the current CML environment is strictly serial. Modern occam in the form of kroc [1] was considered as an implementation layer, and this would have had the advantage of execution in a multi-process environment though prototyping would have been less rapid.

In the absence of a genuinely distributed implementation of CML, current research is looking at mixed language solutions (C++ & CML) to establish whether the harnesses can remain in CML or need to be ultimately implemented directly in C++. Some of the harnesses have been implemented in C++ for a distributed processor architecture and have been evaluated for performance on pre-existing image processing applications [13] but this is outside the scope of this paper.

The developmental method shifts the ball-park of parallelisation significantly. Instead of writing parallel mechanism code to some external system specification, the user writes application-oriented sequential code, leaving the parallelisation to be described by a separate harness. By freeing the user from the concerns and difficulties of parallelisation, time is available to evolve existing software to match harness templates. The split/combine harness presented can clearly

dynamically adapt to the appropriate granularity and scalability for the parallel hardware on which it finds itself running. All the user has to supply are suitable application-specific split and combine routines for the input and output datatypes respectively of a particular function, for that function to be transparently executed in parallel.

It was a surprise that the parallel split/combine harness developed under CML emerged homologous to the serial harness, and it is the expressive power of CML that permits such strong unifications. With the appropriate high level constructs it may be that supporting parallel execution is no more complex than writing serial code to the same pattern.

## References


1. F. Barnes and P. Welch. Prioritised dynamic communicating processes - part 1. In *Communicating Process Architectures - 2002*, pages 331–362, 2002.
2. L. Clarke. TINY: Discussion and user guide, 1989. Newsletter 7, Edinburgh Concurrent Supercomputer Project.
3. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT, Cambridge, MA, 1989.
4. M. Cole. Writing parallel programs in non-parallel languages. In R. Perrott, editor, *Software for Parallel Computers: Exploiting Parallelism through Software Environments, Tools, Algorithms and Application Libraries*, pages 315–325. Chapman & Hall, London, 1992.
5. J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, and R. L. While. Parallel processing using skeleton functions. In *PARLE93, Parallel Architectures and Languages Europe*, 1993. LNCS 694.
6. Hoare C.A.R. Ed. *Occam 2 Reference Manual.* Prentice-Hall, 1988.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP'93*, pages 406–431, 1993.
8. K. Hammond and G. Michelson. *Research Directions in Parallel Functional Programming.* Springer Verlag, first edition, 1999. ISBN: 1852330929.
9. A. J. G. Hey. Experiments in MIMD parallelism. In *PARLE89, Parallel Architectures and Languages Europe*, pages 28–42, 1989. LNCS 366.
10. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, first edition, 1985. ASIN: 0131532715.
11. E. Horowitz and A. Zorat. Divide and conquer for parallel computing. *IEEE Transactions on Computers*, 32(6):582–585, 1983.
12. D Johnston. `privatewww.essex.ac.uk/~djjohn/rappid` A Concurrent ML Tutorial, 2002. Essex University.
13. D. J. Johnston, M. Fleury, and A. C. Downton. A functional methodology for parallel image processing development. In *Visual Information Engineering 2003*, 2003.
14. G. Michaelson and N. Scaife. Prototyping a parallel vision system in Standard ml. *Functional Programming, Concurrency, Simulation, and Automated Reasoning*, 5(3):345–382, 1995.
15. S. L. Peyton Jones and Lester D. *Implementing Functional Languages.* Prentice Hall, New York, 1992.
16. J. H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, Cambridge, UK, 1999.
17. M. Spivey, I. Page, and W. Luk. How to program in Handel, 1995. Oxford University Hardware Compilation Unit.