Translating CBS to LML

Simon B Jones *

Abstract

CBS is a process **C***alculus* for describing **B***roadcasting* **S***ystems*. This note describes a translation of CBS expressions and agent definitions into LML expressions and function definitions. The resulting functions execute under the control of a simple bus arbitrator to produce a trace of the communications that will result from *one* of the computations possible for the original CBS agents. Each CBS agent is translated to an LML function which can be used to simulate its behaviour, and CBS agent-forming operators are translated to higher order functions which combine the functional translations of the component agents.

1 Introduction

CBS is a process **C***alculus* for describing communicating **B***roadcasting* **S***ystems* [1]. CBS is a formalism for both the description and analysis of concurrent systems, and a (prototype) programming language for expressing distributed algorithms.

A CBS *program* is a collection of *agents* (or *processes*) which communicate *signals* (or *messages*) between themselves (and with the external world, though this can be modelled as an agent too).

Agents may *transmit* signals and/or *await* signals (that satisfy some predicate). In one simple hypothetical physical model for CBS these communications take place on a shared ethernet-like *bus*. This gives the desired behaviour: each signal is *broadcast*, and *all* agents have the opportunity to *accept* or *discard* the signal; the bus acts as an *arbitrator* to ensure that there is only one signal on the bus at any time. An agent accepts a signal if it satisfies some predicate, and discards a signal if it does not satisfy the predicate (this description does not capture the full subtlety of discards; see Prasad's paper for

*Department of Computing Science and Mathematics, University of Stirling, Scotland. Email: sbj@cs.stir.ac.uk The work reported here was carried out whilst on sabbatical leave visiting the Programming Methodology Group, University of Göteborg and Chalmers University of Technology, Göteborg, Sweden, March-July 1992.

details). CBS does not define the nature of the signals themselves; for this purpose an application program must exploit the data definition capabilities of some *host language*.

In practice it is instructive, and may be useful, to be able to observe the behaviour of a CBS program. Thus implementing CBS becomes an issue. In general, since agents may be the parallel compositions or sums of other agents, any particular CBS program may give rise to many different behaviours (patterns of communications). Arranging to observe *all* the possible behaviours of a CBS program is tricky. If we decide to be content with being shown just one possible behaviour, then the implementation problem is considerably simplified. That is a restriction that we will accept here. Further we will not address efficiency questions in this paper.

One approach to implementing CBS is to build an interpreter which accepts a data structure representing a CBS program as input, and yields a trace of inter-agent communications as output; Prasad and Jenny Petersson are investigating this approach.

This note describes an alternative approach in which CBS expressions and agent definitions are translated directly into LML expressions and function definitions [4]. Each CBS agent is translated to an LML function which can be used to simulate its behaviour, and CBS agent-forming operators are translated to higher order functions which combine the functional translations of the component agents. The resulting functions execute under the control of a simple "bus arbitrator" to produce a trace of the communications that will result from *one* of the computations possible for the original CBS agents.

Section 2 describes the precise form of CBS being dealt with here, its translation into LML, and gives some examples.

Section 3 contains details of the implementation of the functions used in the translation.

Section 4 suggests further work.

The reader is assumed to be familiar with LML.

2 Customizing CBS and translating it to LML

In this implementation of CBS I decided to focus on the interesting problem of *organizing* and *synchronizing* the communications between agents; some of the CBS agent-forming operators have thus been omitted from the implementation. On the other hand, pure CBS does not define how *data* is to be handled by agents: in this implementation agents may communicate any values computable by LML expressions, and in this spirit signals will be called *messages*

in this customized form of CBS, and in its translation to LML. In reference to the translated form, agents will often be called *processes*.

The customized form of CBS to be used in this paper, CBS $_{LML}$ is an adaptation of that described in Prasad's *CAAP'91* paper. It is very close to the form adopted in more recent work by Prasad [2, 3].

2.1 The syntax of CBS$_{LML}$

The general principles of CBS remain. Some constructs are omitted, some are altered or extended, and a conditional expression has been added. Agent expressions have the following syntax:

$$\begin{aligned} E &::= X\ Arglist \mid \mathbf{Nil} \mid LMLExp\,!\,E \mid LMLPattern\,?\,E \mid \\ &\quad\ E + E \mid E \mid E \mid \mathbf{if}\ LMLExp\ \mathbf{then}\ E\ \mathbf{else}\ E \\ Arglist &::= \{LMLExp\}^{*} \end{aligned}$$

X is an *agent name* identifier. *LMLExp* and *LMLPattern*, for expressions and patterns, are adopted directly from LML. $\{\ \}^{*}$ indicates zero or more repetitions of the enclosed items. There is a restriction on the form of sum processes $E + E$; details below.

Note that recursive agent expressions have been omitted; instead we will write *agent definitions* with the syntax:

$$Def ::= \{X\ LMLPattern \stackrel{\text{def}}{=} E\}^{*}$$

where X is the agent's name, and several equations may be given to define its behaviour for different argument patterns. Recursion must be guarded.

A CBS$_{LML}$ *program* will consist of one or more agent definitions; they will usually be mutually recursive.

2.2 Comments on the semantics of CBS$_{LML}$

These comments apply to the specific characteristics of the CBS $_{LML}$ form of CBS, and do not provide a complete description of the semantics of CBS.

- An agent definition $\{X\ LMLPattern \stackrel{\text{def}}{=} E\}^{*}$ is a schema for the behaviour of an agent named X. When X is invoked argument values will be supplied; these will be matched against the patterns. The first pattern that matches determines the equation to be used, and any variables in the pattern are bound to the corresponding values in the arguments. These variables are in scope in any *LMLExp*s in E, although they may be masked by receive actions within E (see next).

- An agent specified by *LMLPattern*? E *awaits* a message that matches the pattern (in the usual LML sense). It *accepts* the first matching message, subsequently behaving as specified by E. Any variables in the pattern are bound to the corresponding values in the message. These variables are in scope in any *LMLExp*s in E, unless masked by another receive action.

- An agent specified by X *Arglist* invokes a behaviour as given by the definition of agent X with actual arguments being the values of the expressions in *Arglist*.

- An agent specified by *LMLExp*! E transmits the value of the LML expression.

- In an agent expression $E + E$ *each* alternative must be either a receive or a transmit action. (This hopefully does not seriously restrict the expressiveness of the language, but it does facilitate implementation.)

- The agent expression **if** *LMLExp* **then** E **else** E is obvious.

- Agent names must not be used in *LMLExp*s. (This is just to keep us in a familiar "first order" CBS world, in particular in a world in which agents cannot be exchanged in messages. In the LML implementation which follows this restriction could be lifted, leading all sorts of exotic possibilities ....)

- The form of argument lists means that agents may be curried or uncurried. However, if the previous restriction is adhered to then there will never be any opportunity to partially apply a curried agent.

2.3 Translating CBS$_{LML}$ into LML

In the following tables $\overline{E}$ stands for "the translated form of E", where E is a CBS$_{LML}$ agent expression. Some notes on the translation follow the tables.

	CBS$_{LML}$ agent expression	LML translation
1	X *Arglist*	X *Arglist*
2	**Nil**	`stop`
3	$LMLExp\,!\,E$	`xmit` $LMLExp\ \overline{E}$
4	$LMLPattern\,?\,E$	`recv` $<$ pred $LMLPattern>$ $<$ cont $LMLPattern\ \overline{E}>$
5	$LMLPattern_{\,1}\,?\,E_1 + LMLPattern_{\,2}\,?\,E_2$	`altrr` $<$ pred $LMLPattern_{\,1}>$ $<$ cont $LMLPattern_{\,1}\ \overline{E_1}>$ $<$ pred $LMLPattern_{\,2}>$ $<$ cont $LMLPattern_{\,2}\ \overline{E_2}>$
6	$LMLPattern\,?\,E_{\,1} + LMLExp\,!\,E_{\,2}$ or $LMLExp\,!\,E_{\,2} + LMLPattern\,?\,E_{\,1}$	`altrx` $<$ pred $LMLPattern>$ $<$ cont $LMLPattern\ \overline{E_{\,1}}>$ $LMLExp\ \overline{E_{\,2}}$
7	$LMLExp_{\,1}\,!\,E_1 + LMLExp_{\,2}\,!\,E_2$	`altxx` $LMLExp_{\,1}\ \overline{E_1}$ $LMLExp_{\,2}\ \overline{E_2}$
8	$E_1 \mid E_2$	`par` $\overline{E_1}\ \overline{E_2}$
9	**if** $LMLExp$ **then** $E_{\,1}$ **else** E_2	`if` $LMLExp$ `then` $\overline{E_{\,1}}$ `else` $\overline{E_2}$

	CBS$_{LML}$ agent definition	LML translation
10	$\{X\ LMLPattern \stackrel{\text{def}}{=} E\}^*$	$\{X\ LMLPattern = \overline{E}\}^*$ equations separated by ||

Notes

- X, *Arglist*, *LMLExp* and *LMLPattern* are already valid LML and do not need translating.
- Additional parentheses may be necessary to ensure correct structure.
- Variable identifiers and data constructor identifiers may need renaming to avoid clashes with LML reserved words, etc.
- The agent constructors `stop`, `xmit`, `recv`, `altrr`, `altrx`, `altxx` and `par` are predefined functions (described in detail in Section 3).
- In rule 1 function application is used to invoke a named process.
- Rules 5, 6 and 7 cover the only cases that are allowed for agent sums. Note that each of the functions `altrr`, `altrx` and `altxx` has *four* arguments.

- Receive actions are specified by a pattern that a message must match and a continuation agent. However, in LML a pattern cannot be passed as an argument to a function, so `recv`, `altrr` and `altrx` cannot be depend directly on a pattern extracted from the CBS $_{LML}$. The pattern has two rôles: it determines whether a message is accepted or discarded, and it extracts data values from an accepted message and binds these values to variables. In the translation these two rôles are played by a pred**i**cate, constructed from the pattern, which indicates whether a message is to be accepted or discarded (returning `true` and `false` respectively), and a cont**i**nuation function, constructed from the pattern and the continuation process, which extracts data from a message and then executes the continuation process. These are represented in the translation rules by $<$ pred $LMLPattern>$ and $<$ cont $LMLPattern\ \overline{E}>$ and are defined as follows:

	translates to
$<$ pred $LMLPattern>$	`(\m.case m in` $LMLPattern$ `: true || _ : false end)`
$<$ cont $LMLPattern\ \overline{E}>$	`(\`$LMLPattern.\overline{E}$`)`

  [This mechanism is quite powerful: the use of a predicate rather than just a pattern to select messages allows arbitrary properties to be tested (such as that two data fields are equal).]

- LML program transformation rules may be applied to the result of translation: the introduction of `let` blocks, for example. It may also occasionally be expedient to exploit other features of LML: guarded equations in function definitions, for example.

The predefined functions (and types) are held in a separate module. To form a complete LML program the translated process definitions must be packaged in the usual way as a main program, with the inclusion of:

- Appropriate `import` details for the CBS $_{LML}$ module. Usually a statement of the form `#include "cbs.t";`
- (If necessary) A type definition for the messages to be communicated by processes.
- (If necessary) The definition of a show function for the message type.

- A main expression specifying the execution required. This should have the form:

  `go` *show_fn extern program*

  where *show_fn* is a suitable show function for the messages exchanged by the processes, *extern* is a process expression describing the external world's behaviour, and *program* is an process expression for the program to be executed. The distinction between *extern* and *program* is perhaps artificial; `go` executes them in parallel using the `par` operator, and so there is no additional functionality. The distinction is sometimes useful, but either *extern* or *program* can be replaced with `stop` (both are treated the same way). The output from `go` *show fn extern program* is a trace of the communications that occur up to the point when the "bus" first becomes quiet (no further messages will be sent after this point [1]).

2.4 Examples

2.4.1 Counting and echoing

Here are some CBS $_{LML}$ agents:

$$
\begin{array}{rcll}
\mathit{from}\ n & \stackrel{\text{def}}{=} & n\,!\,\mathit{from}\,(n+1) & \text{Count up from } n \\
\mathit{fromto}\ m\ n & \stackrel{\text{def}}{=} & \textbf{if}\ m > n & \text{Count up a range} \\
 & & \textbf{then Nil} & \\
 & & \textbf{else}\ m\,!\,\mathit{fromto}\,(m+1)\,n & \\
\mathit{echoinc} & \stackrel{\text{def}}{=} & x\,?\,(x+1)\,!\,\mathit{echoinc} & \text{Echo incremented values} \\
\mathit{echoincrange}\ m\ n & \stackrel{\text{def}}{=} & x\,?\ \textbf{if}\ m <= x\ \&\ x <= n & \text{Only if in a range} \\
 & & \quad \textbf{then}\ (x+1)\,!\,\mathit{echoincrange}\ m\ n & \\
 & & \quad \textbf{else}\ \mathit{echoincrange}\ m\ n &
\end{array}
$$

[1] This is a consequence of the discrete time model of CBS $_{LML}$ realized by the implementation: all processes are always either awaiting or attempting to transmit; if there is a quiet "slot" on the bus then no processes (if there are any) were ready to transmit, all are only able to receive; therefore none made progress, and this state will persist indefinitely.

And here is a complete LML file containing the translated forms, and a request to execute the CBS $_{LML}$ expression *fromto* 1 10 | *echoincrange* 4 8 :

```
#include "cbs.t";
let
rec from n = xmit n (from (n+1))
and fromto m n = if m>n then stop
                         else xmit m (fromto (m+1) n)
and echoinc = recv (\x.true) (\x.xmit (x+1) echoinc)
and echoincrange m n = recv (\x.m<=x & x<=n)
                            (\x.xmit (x+1) (echoincrange m n))
in go show_int (fromto 1 10) (echoincrange 4 8)
```

Notes:

- *echoinc* receives unconditionally; this is encoded as the predicate `(\x.true)`, which is a trivial simplification after application of the general rule, which would have produced

  ```
  (\m.case m in x : true || _ : false end)
  ```

- The translated form of *echoincrange* has taken advantage of the power of using a predicate for message selection: the test rejects messages which are "out of range", whereas the original CBS $_{LML}$ used explicit recursion.

- The messages are just integers, so the show function used is the standard `show_int`.

The behaviour of the CBS $_{LML}$ expression *fromto* 1 10 | *echoincrange* 4 8 is that the agent *fromto* transmits the integers $1 \ldots 10$, which are thus observable on the bus, and the agent *echoincrange* receives those integers in the range $4 \ldots 8$ and retransmits each incremented by 1. Thus the output from this example program, the observable bus traffic, is

```
[1; 2; 3; 4; 5; 5; 6; 6; 7; 7; 8; 8; 9; 9; 10]
```

2.4.2 Broadcast sorting

The following description and CBS program are taken from Prasad's paper (with minor modifications):

> Consider first the simpler case where all the numbers input by the user are distinct.
>
> Broadcast sort is a parallelised insertion sort. The input so far is held in a sorted list, maintained by cells each holding a number

u and a "link" l, the next lower number. The correct place for the next number thus always splits exactly one cell. Let $\bot$ and $\top$ be sentinel values, respectively less than and greater than all numbers. At all times, there is exactly one cell with $l = \bot$, and exactly one with $u = \top$. Output is done by the first of these two transmitting u, the head of the sorted list. Each cell (l, u) changes to $(\bot, u)$ when it hears l, thus giving the tail of the list.

$$
\begin{array}{rcll}
\mathit{sorter} & \stackrel{\text{def}}{=} & \mathit{in}\,(\bot, \top) \setminus \{\, \mathit{Out}? \,\} & \\
\mathit{in}\,(l, u) & \stackrel{\text{def}}{=} & \mathit{Go}?\ \mathit{out}\,(l, u) + \mathit{In}\,(n)\,? & \textbf{if } l < n \ \&\ n < u \\
 & & & \textbf{then } \mathit{in}\,(l, n) \mid \mathit{in}\,(n, u) \\
 & & & \textbf{else } \mathit{in}\,(l, u) \\
\mathit{out}\,(\bot, \top) & \stackrel{\text{def}}{=} & \mathit{in}\,(\bot, \top) & \\
\mathit{out}\,(\bot, u) & \stackrel{\text{def}}{=} & \mathit{Out}\,(u)\,!\,\textbf{Nil} & \\
\mathit{out}\,(l, u) & \stackrel{\text{def}}{=} & \mathit{Out}\,(l)\,?\ \mathit{out}\,(\bot, u) &
\end{array}
$$

Here is its translation into LML, with a final expression requesting that the integers 5, 1, 3, -9 and 7 be sorted into order:

```
#include "cbs.t";
let

rec type Message = In Int + Go + Out Int     -- message contents

and show_Message :: Message -> String
and show_Message (In n) = "In(" @ show_int n @ ")"
 || show_Message Go = "Go"
 || show_Message (Out n) = "Out(" @ show_int n @ ")"

and inject :: List Int -> Process Message
                                    -- to be the "external world"
and inject [] = xmit Go stop
 || inject (x.xs) = xmit (In x) (inject xs)

and smallnum = -1000000                 -- to be the lower and
and bignum   =  1000000                 -- upper sentinels

and sorter :: Process Message           -- CBS agent "sorter"
and sorter = inp smallnum bignum
```

```
and inp :: Int -> Int -> Process Message     -- CBS agent "in"
and inp l u = altrr (\m.m=Go) (\m.outp l u)
                    (\m.case m in In n : true || _ : false end)
                    (\(In n). if l<n & n<u
                                then par (inp l n) (inp n u)
                                else inp l u)

and outp :: Int -> Int -> Process Message    -- CBS agent "out"
and outp l u & (l=smallnum & u=bignum) = inp smallnum bignum
 || outp l u & (l=smallnum) = xmit (Out u) stop
 || outp l u = recv (\m.case m in Out n & (n=l) : true
                                  ||              _ : false end)
                    (\(Out n).outp smallnum u)

in go show_Message (inject [5;1;3;-9;7]) sorter
```

Notes:

- Messages exchanged on the bus are of the form `In` n, `Go` and `Out` n. Hence the type definition for `Message` and the definition of the function `show_Message`.
- A simple agent, `inject`, has been added to act as the external world in triggering the sorter.
- There are no predefined constants to be used as $\bot$ and $\top$ in *LMLPattern*s. They have been translated into the global identifiers `smallnum` and `bignum`, and guards have been added to the first two equations of `outp`. (This inelegance could be overcome by suitable use of a new `datatype`.)
- In the third equation for *out* the pattern for receiving, *Out(l)*, is not in fact a correct LML pattern since it contains an identifier which has already been bound to a value. In the translation an unbound identifier has been substituted, and an equality test added as a guard in the predicate.

The output from this program is:

```
[In(5); In(1); In(3); In(-9); In(7); Go;
          Out(-9); Out(1); Out(3); Out(5); Out(7)]
```

We see the sequence of `In` messages followed by `Go` produced by `inject`, followed by the `Out` messages as the sorter produces its result.

3 Implementation of the LML functions

3.1 The implementation strategy

The implementation of the CBS $_{LML}$ functions is directly based on the ethernet-like *bus* model mentioned in the Introduction. All CBS $_{LML}$ processes wishing to transmit messages compete for access to the bus: one will be selected and the others must re-try (this is implicit in the CBS $_{LML}$ semantics). CBS $_{LML}$ processes awaiting signals consume the next available message which matches the pattern specified; and every message is available for reception by every process.

Although the semantics of CBS $_{LML}$ does not deal with time, in this implementation it was necessary to introduce a notion of discrete time intervals. Time is divided into notional intervals by organizing the bus as a series of *slot*s, each of which either contains a message or is null.

This discretization of time induces an "execution cycle" as follows:

1. At the start of a slot each process nominates zero or more messages for transmission on the bus: zero since the process may only be able to receive; one or more since a process may be the parallel or choice composition (recursively) of more than one process.

2. The "bus arbitrator" selects *one* message from all those nominated.

3. Each process is informed (via "`Success`" or "`Failure`" `Grant`s) which of the messages that it nominated, if any, was selected (so that non-selected processes can re-try).

4. The bus slot is filled with the selected message, and *all* processes are able to either accept it or discard it.

5. The cycle starts again.

Note that processes which are the composition of other processes must collect together the transmission requests of each component process at each time slot, must return the selection grants from the bus to the correct component processes[2], and must inform all components of the actual contents of the bus slots. Thus each composite process has exactly the same interface as a simple process, and much of the complexity of the organization of the implementation is distributed amongst the process forming functions `xmit`, `recv`, `altrr`, `altrx`, `altxx` and `par`; the bus arbitrator (`go`) essentially deals with *one* process, the internal details of which it is unaware of.

[2]Note that there is no simplification in making the selection locally in composite processes, since the selection is not valid until confirmed *globally* by the bus arbitrator.

A CBS$_{LML}$ process is thus implemented as a stream processing function, where the stream elements are synchronized with the bus slots:

```
type Process *a == Stream (List Grant) -> Stream (Slot *a)
                                       -> Stream (List (Request *a))
```

- One output stream: each element is a list of the messages requesting transmission in the current slot.
- The first input stream is structurally identical to the output stream: it carries the grants indicating the success or failure of each of the transmission requests.
- The second input stream carries the actual contents of the bus slots.

3.2 Annotated CBS$_{LML}$ module

The entire CBS$_{LML}$ module is given next. Sections of the LML code are followed by explanatory comments.

```
module
export Process, Message, Pattern,
                        -- required for typechecking applications
       recv, xmit, par, go, stop,      -- for use in applications
       altrr, altrx, altxx;
```

- `Process`, `Message` and `Pattern` need never be used by the applications programmer, but they need to be exported otherwise applications programs cannot be typechecked.

```
rec
    type Message *a == *a
                 -- message data *a determined by the application
and type Slot *a = Null + Data (Message *a)          -- Bus slots
and type Grant = Success + Failure
and type Request *a == Message *a        -- Transmission requests
and type Stream *a == List *a
and type Process *a == Stream (List Grant) -> Stream (Slot *a)
                                      -> Stream (List (Request *a))
and type Pattern *a == Message *a -> Bool
```

- `Process` is the key type: two input streams and one output stream as described above.
- `Pattern` is the type of the predicates used for message selection by receive actions.

```
and show_Message :: (*a -> String) -> Message *a -> String
and show_Message f = f

and show_Slot :: (*a -> String) -> Slot *a -> String
and show_Slot f Null = "Null"
 || show_Slot f (Data x) = show_Message f x

and show_Stream :: (*a -> String) -> Stream *a -> String
and show_Stream = show_list
```

- The show functions are not very exciting.

```
and go :: (*a->String) -> Process *a -> Process *a -> String
and go show_data extern prog =
       ((show_Stream (show_Slot show_data) live_bus) @ "\n"
        where rec live_bus = fst (take ((~=) Null) bus)
                                        -- system dies at first Null
/**/          and reqs = par extern prog sigs bus
/**/          and (sigs,bus) = select 0 reqs
                                -- pick one from each collection
                                -- of requested transmissions

              and select :: Int -> Stream (List (Request *a))
                   -> ((Stream (List Grant))#(Stream (Slot *a)))
              and select n [] = ([],[])
               || select n ([].reqs) =
                      (([].sigs,Null.bus)
                       where (sigs,bus) = select n reqs )
               || select n (reqlist.reqs) =
                      ((siglist.sigs,Data msg.bus)
                       where rec numreqs = length reqlist
                             and n' = (n+1)%numreqs
                             and msg = reqlist??n'  -- pick n' th req
                             and siglist =
                                   for 0 (numreqs-1)
                                         (\i.if i=n' then Success
                                                     else Failure)
                             and (sigs,bus) = select n' reqs ) )
```

- `go` is the main function — it is the bus arbitrator: it selects between transmission requests and formats the bus traffic for output.

  The key lines are labelled `/**/`; *these depend crucially on lazy evaluation*: `extern` and `prog` are executed in parallel to obtain the stream of requests for each bus slot. A selection is made from each list of requests, producing a stream of selection grant lists and the actual bus contents. The grants and bus are conveyed to `extern` and `prog` to determine their subsequent behaviour [3].

- `select` makes the slot-by-slot selection; it has an integer argument which is used in a very rough and ready way to give a degree of randomness to the message selection by cycling through the request lists (it could be replaced by any number of other mechanisms).

- The bus is formatted for output up to, but not including, the first null slot (see footnote on page 7 which explains this truncation) using the show function for the message data supplied by the user.

[3] The mutual dependence of these two equations may look like magic, but it works.

```
and stop :: Process *a  -- no further transmissions
and stop sigs bus = []

and recv :: Pattern *a -> (Message *a -> Process *a) -> Process *a
and recv patok p sigs bus =
        [] .                        -- no transmission requests
             case (sigs,bus) in   -- just listen for message
                  (_.sigs',Null.bus') : recv patok p sigs' bus'
               || (_.sigs',Data m.bus') & (patok m) : p m sigs' bus'
               || (_.sigs',Data m.bus') : recv patok p sigs' bus'
             end

and xmit :: Message *a -> Process *a -> Process *a
and xmit m p sigs bus =
        [m] .                      -- request transmission of m
              case (sigs,bus) in -- and see if successful
                   ([Success].sigs',_.bus') : p sigs' bus'
                || ([Failure].sigs',_.bus') : xmit m p sigs' bus'
              end
```

- These are the three basic process forming operators.
- `stop` simply produces no further request lists (this allows `par` to discard such processes).
- `recv` outputs an empty request list [4] (reasonable!) and tests the next slot on the bus for an acceptable message. If the slot is empty or contains an unacceptable message then the receive action is repeated with the subsequent bus slot(s), otherwise the continuation process is executed. In each case, since no transmission has been requested, the returned grant list is ignored.
- `xmit` requests the transmission of a single message, and then either continues or repeats depending on the selection grant returned. Note how the structure of the selection grant list exactly matches the structure of the request list.
- **Note** that in `recv` and `xmit`, and in subsequent function definitions
  - the grants and bus streams *are not pattern matched* on the left hand side of the equation; *doing so would introduce extra, and impossible to satisfy, eagerness to the evaluation* [5], and would cause (in process synchronization terms) a form of deadlock;

[4] A *list* because, in general, a process may nominate any number of messages for the next bus slot.

[5] It would undo the magic in the definition of `go` alluded to earlier.

– the grants and bus streams *are both decomposed by the* `case` *expression*, although, on occasions, it appears that one or other of the streams could be omitted from the `case` and `tl` could be applied to the stream instead; *this is to avoid very expensive space leaks* [6].

[6]Which I had to track down most meticulously when they arose in an early version of the code.

```
and par :: Process *a -> Process *a -> Process *a
and par p1 p2 sigs bus =
/**/   (combine p1reqs p2reqs
/**/    where rec p1reqs = p1 p1sigs bus
/**/          and p2reqs = p2 p2sigs bus
/**/          and (p1sigs,p2sigs) = splitsigs sigs p1reqs p2reqs

              and combine :: Stream (List (Request *a))
                          -> Stream (List (Request *a))
                          -> Stream (List (Request *a))
              and combine [] p2reqs = p2reqs
               || combine p1reqs [] = p1reqs
               || combine (p1.p1reqs) (p2.p2reqs) =
                                (p1@p2) . combine p1reqs p2reqs

              and splitsigs :: Stream (List Grant)
                            -> Stream (List (Request *a))
                            -> Stream (List (Request *a))
                            -> ((Stream (List Grant))
                               #(Stream (List Grant)))
              and splitsigs sigs [] p2reqs = ([],sigs)
               || splitsigs sigs p1reqs [] = (sigs,[])
               || splitsigs (ls.sigs) (l1.p1reqs) (l2.p2reqs) =
                      (head (length l1) ls . p1sigs,
                       tail (length l1) ls . p2sigs)
                      where (p1sigs,p2sigs) =
                                        splitsigs sigs p1reqs p2reqs
    )
```

- `par` builds a composite process which behaves as two component processes executed in parallel. The key lines are labelled with `/**/`. The request streams, `p1reqs` and `p2reqs`, from the two components are combined to produce the request stream from the composite process: in each slot the list of requests is the concatenation of the separate request lists. The selection grants returned by the bus arbitrator are split up to match the request lists, giving `p1sigs` and `p2sigs`. These grants and the bus contents are conveyed to the component processes.

- If either of the component processes `stop`s then its request stream ends, and `par` discards the stopped process; subsequent processing is carried out by the remaining process.

```
and altrr :: Pattern *a -> (Message *a -> Process *a)
          -> Pattern *a -> (Message *a -> Process *a) -> Process *a
and altrr pat1ok p1 pat2ok p2 sigs bus =
     [] .                           -- no transmission requests
     case (sigs,bus) in            -- just listen to the messages
          (_.sigs',Null.bus') : altrr pat1ok p1 pat2ok p2 sigs' bus'
       || (_.sigs',Data m.bus') & (pat1ok m) : p1 m sigs' bus'
       || (_.sigs',Data m.bus') & (pat2ok m) : p2 m sigs' bus'
       || (_.sigs',Data m.bus') : altrr pat1ok p1 pat2ok p2 sigs' bus'
     end

and altrx :: Pattern *a -> (Message *a -> Process *a)
                        -> Message *a -> Process *a -> Process *a
and altrx patok p1 m p2 sigs bus =
     [m] .                              -- request to transmit m
     case (sigs,bus) in                -- and see if successful
             ([Success].sigs',_.bus') : p2 sigs' bus'
          || ([Failure].sigs',_) : -- No, so listen to message on bus
                   case bus in
                        Null.bus' : altrx patok p1 m p2 sigs' bus'
                     || Data m.bus' & (patok m) : p1 m sigs' bus'
                     || Data m.bus' : altrx patok p1 m p2 sigs' bus'
                   end
     end

and altxx :: Message *a -> Process *a
          -> Message *a -> Process *a -> Process *a
and altxx m1 p1 m2 p2 sigs bus =
     [m1;m2] .                  -- request transmission of m1 and m2
     case (sigs,bus) in        -- and see if either is successful
             ([Failure;Failure].sigs',_.bus') :
                                  altxx m1 p1 m2 p2 sigs' bus'
          || ([Success;Failure].sigs',_.bus') : p1 sigs' bus'
          || ([Failure;Success].sigs',_.bus') : p2 sigs' bus'
     end

end
```

- These three functions implement the restricted forms of the agent sum operation. The first is a choice between two receive actions, and so requests no transmissions. The second is a choice between a transmit and a receive, and so one transmission request is output. The third is a choice between two transmit actions, and so two transmissions are requested.

- The subsequent action is determined by inspection of the selection grants returned and the bus contents; in each case the decision is a combination of the mechanisms in `recv` and `xmit`, above.

- There is, almost inevitably, some bias in the choice: `altrr` prefers the first alternative if both are prepared to accept the next message; `altrx` seems to give priority to a transmit attempt [7]; `altxx` exhibits no bias, since it is not possible for both transmit requests to be successful (any bias would come from the bus arbitrator).

3.3 General comments on the implementation

- The types `Message`, `Request` and `Stream` do not achieve anything special; it is just nice to have mnemonic type names as "wrappers".

- The implementation depends crucially on being evaluated lazily. This is evident from the key equations in the definition of `go`, in which streams have mutually recursive definitions.

- The functions do not contain pattern matching for streams on their left hand sides. This is to avoid the introduction of undesirable degrees of eagerness, which would cause failure of the mutually recursive key equations in the definition of `go` [8].

- `case` expressions are used to decompose streams into their head and tail, even when simple `hd` and `tl` would seem adequate. This is to avoid expensive space leaks; these would arise if argument expressions in recursive function calls used `hd` or `tl` but were not immediately forced. An early version of the implementation suffered from this problem rather badly (though the severity depended on the application program).

[7] Though whether this is the case is arguable: if the transmit is unsuccessful ("quite likely") then the receive has the option of accepting whatever message succeeded in occupying the bus slot; the only circumstance under which both can succeed is if the transmitted message is acceptable to the receiving pattern (a strange program?). There is no genuine bias if the bus arbitrator is reasonably fair. It is a subtle issue. It was the implementation of this particular case of process choice that necessitated the restriction of agent sums; it was hard to see how it could be any other way.

[8] Giving a run time error which may or may not be reported.

4 Suggestions for further work

As far as it goes, the implementation of CBS $_{LML}$ described in this note is quite effective and is an interesting approach. However, some perhaps arbitrary decisions were made, and there are some questions to be asked and some loose ends to be tidied up:

- Can the restriction on the forms of agent sum be removed? It would be nice to have just one `alt` function with type

  ```
  Process *a -> Process *a -> Process *a
  ```

  and a translation rule:

$E_1 + E_2$	`alt` $\overline{E_1}$ $\overline{E_2}$

  (I found no way to make the sum of two agents translate to the general composition of two processes: the composition needed to know *something* about their internal workings, viz. their first receive or transmit action. Perhaps I was just not seeing clearly.)

- The omitted feature of CBS needs adding: morphism.

- All the processes are obliged to run in synchrony with the bus slots: if any process performs a lengthy computation before determining whether it will transmit or receive at the next step (for example evaluating the boolean expression in an `if then else`), then the entire system must wait. Can this restriction be removed whilst remaining with this style of implementation?

- Can the implementation be extended to display *all* possible computations?

- Perhaps the simple strategy in the bus arbitrator `go` for "randomizing" the selection of the next message can be improved? There are probably pathological cases for which the current strategy is wildly unsuitable.

- Is the implementation a correct implementation of CBS $_{LML}$? (The question needs formalizing and then answering.) Is it any easier or harder to prove this implementation correct than those described by Prasad and Petersson?

There's plenty for someone to do ...

References

[1] K. V. S. Prasad. A calculus of broadcasting systems. In *Volume I: CAAP'91*. Springer Verlag LNCS 493. April 1991.

[2] K. V. S. Prasad. *A Calculus of Value Broadcasts.* Technical Report. Department of Computer Science, Chalmers University of Technology. 1992.

[3] K. V. S. Prasad. *Programming with Broadcasts.* Technical Report. Department of Computer Science, Chalmers University of Technology. 1993.

[4] L. Augustsson and T. Johnsson. *Lazy ML Users' Manual.* Department of Computer Science, Chalmers University of Technology.