Specification and implementation of concurrent systems using PARLOG

David Gilbert

Department of Computer Studies
Loughborough University of Technology
Loughborough
LE11 3TU
UK

email: drg@uk.ac.lut.cs-vaxa

ABSTRACT

The specification and implementation of a class of concurrent systems is investigated in this paper using PARLOG as the implementation language, guided by specifications in CCS. This class of systems is restricted to an illustrative subset, including buffers and queues. A comparison is made between the computational models of concurrency that are expressed by each language. Illustrative programs are given which highlight the differences in operational behaviour between programs in the two languages. We investigate the ways in which equivalences in CCS programs may be used to compare PARLOG programs, and the areas of PARLOG programming which are not covered by this comparison. The advantages and disadvantages of CCS compared with PARLOG as a 'specification-implementation' language are discussed.

1. Introduction

Designers of concurrent systems are faced with the problems of firstly specifying the system and secondly implementing the design. There are several specification techniques available, including LOTOS, CCS, CSP and Petri Nets. However, the route from specification to implementation is not clear in many cases. The best that can be done is often proving that the implementation is in some way equivalent to the specification, rather than generating the implementation directly from the specification.

In this paper we wish to investigate the possibility of using the parallel logic programming language PARLOG both to specify and implement certain classes of communicating systems.

The PARLOG language has been described elsewhere in detail [Gregory87 , Gilbert87] and we will only describe its features relevant to this discussion. These are:

- asynchronous and synchronous stream based communication.

- committed choice and no output before commitment.

2. PARLOG

PARLOG as a Concurrent Logic programming language

PARLOG is a language which belongs to the family of committed choice parallel logic programming languages. It has the ability to explicitly express both OR and stream-AND parallelism. Committed choice non-determinism is implemented by the use of guards which ensure that committal is made to only one clause. These features enable the language to use non-determinism as an evaluation strategy if required by the programmer. They permit the writing of programs which require the separate use of both sequential and parallel evaluations.

The PARLOG syntax used in this article is given in the Table at the end of this paper. Note that in the examples in this paper, variable names start with a capital letter.

Mode declarations are used to specify communication restraints on shared variables, which are declared to be either **?** ("input") or ^ ("output"), thus acting as communication channels. These declarations are made once for each relation, and each argument of the relation is annotated:

mode name(a1?,..,ak^,..)

Messages sent along these channels are incrementally constructed from partially determined data structures, usually consisting of lists of terms acting as message streams.

The general form of a PARLOG clause is:

<head> ← <guard> : <body> <or-op>

where <head> is in the form *name(a1,..,ak)* , **name** being the relation name, and **a1,..,ak** its arguments. The logical implication symbol is '←' and ':' the guard operator. Both the guard and the body can be a conjunction of calls, or empty, and the calls are separated by the sequential-AND or parallel-AND operators; an OR-operator terminates the clause. In the case that a guard is empty, it is omitted along with the guard operator; if a clause has a guard but no body, the body is replaced by **true**; a clause with no guard or body is represented by just the head. A clause is a candidate for evaluation if both input matching in the head and the evaluation of the guard succeeds, whereas in a non-candidate clause either of these fail. A clause can be suspended if either the input matching or guard evaluation suspend waiting for an input variable to become instantiated. A suspended call may eventually become either candidate or non-candidate. For example, given the following PARLOG program:

Example 1.

```
mode check(pattern?).
check([H|T]).
```

a call *? check(foo)* will **fail**, a call *? check([X|Y])* will **succeed**, and a call *? check(X)* will **suspend** (until *X* has become further instantiated). Note that no output bindings are made until committal has been made to a clause (ie the input and guard conditions are satisfied), and committal may be made to only one clause of a procedure.

Stream communication

The committed choice aspect of PARLOG give it the ability to express stream communication using **stream-AND-parallelism** in a very simple manner. This enables PARLOG to be used constructively in applications where concurrency and process state are of importance. Since PARLOG allows more than one call to be evaluated concurrently, the language permits computations which comprise the execution of several concurrent processes.

The basic paradigm of stream communication is that of producers and consumers. PARLOG's ability to perform **stream-AND-parallelism** gives the language the power to express stream communication in a very simple manner. Shared variables act as communication channels and messages can be sent by the incremental construction of partly instantiated data structures such as tuples or lists of terms. This incremental communication has an analogy with the lazy and eager parallel evaluation of functional programs. Output variables are produced in an asynchronous manner, but input variables are processed synchronously if an argument contains a partially instantiated term. Mode declarations mean that in a situation involving communication only one PARLOG process can be the producer, binding shared variables, whilst there may be one or more consumers of the communication stream.

An outline program which illustrates this form of stream communication is:

```
mode eager_producer(Stream^).
eager_producer([Item|Stream])←
    produce(Item) , eager_producer(Stream).

mode naive_consumer(Stream?).
naive_consumer([Item|Stream])←
    consume(Item) , naive_consumer(Stream).

mode produce(item^) , consume(item?).

call:
```
? eager_producer(Stream) , eager_consumer(Stream).

Note that we do not give the code for the procedures *produce* and *consume*, as these are not relevant to this particular algorithm; we assume that *produce* is 'eager' (asynchronous). The *eager_producer* process can run ahead of the consumer by an arbitrary amount, and we assume the existence of system buffers to permit this. An obvious question, to which there is no explicit answer since they are not formally part of the language, is what happens when these buffers become full.

Thus the naive mode of communication in PARLOG, which uses asynchronous sends and synchronous receives, makes reasoning about communicating systems difficult since the channels act as unbounded buffers in this case. The PARLOG programmer is, however, able to utilise the completely synchronous communication facilities offered by the language using *back-communication*, which can take two forms. These are firstly *mode reversal* (lazy evaluation) and secondly *cooperative construction of binding terms*

("Incomplete messages" [Shapiro83], or "back communication by cooperative construction of binding terms"[Gregory87]).

PARLOG in the light of Milner's interpretations

We note some correspondences between the concepts Milner expresses in his recent paper [Milner86] and those implemented in PARLOG. The interested reader is referred to [Ellis86 , Hussey87] for a discussion of the translation of PARLOG into CCS.

The "naive" form of communication in PARLOG (asynchronous producer, synchronous receiver) relies on buffering within the system, and does not conform to Milner's Principle 1 (all process interactions are atomic events); co-operative construction of binding terms is excluded for the same reason that Milner discounts the 'rendez-vous' of Ada. However, reverse-mode communication can be considered to conform to the first principle of Milner if we recognise that in a practical programming language there must be some physical means by which processes can communicate.

Due to the ability of a PARLOG computation to permit the existence of more than one concurrent process, the language may be used in a manner which broadly conforms to Principle 2 (every event is an interaction among processes), although PARLOG computational style is not limited to the process view alone. Because of the semantics of logic programs, a PARLOG query which is a conjunction of calls can only succeed when all the calls succeed, and if programming by side-effects is excluded then Principle 3 (every process constructor f must be such that the behaviour of $f(P_1,...,P_n)$ depends only on the behaviours of $(P_1,...,P_n)$ is satisfied.

Milner's Principle 4 (Conjunction) is partly satisfied by PARLOG's stream-AND operator when considering the synchronous communication expressible in the language. However the language only supports a 1 to N form of communication (one producer and many consumers). Moreover, multi-way synchronisation is not the natural paradigm in the language, and it must be enforced by programming techniques, hence making it difficult to incorporate Milner's Principle 9 (Simultaneity) into the analysis of PARLOG. However, Principle 9 is supported to the extent that a PARLOG process may synchronise with more than one other process by the use of several communication channels. Encapsulation (Principle 5) is achieved to the extent that the scope rules of Logic Programming which ensure that calls made within the body of one clause are not "visible" to a procedure which calls that clause. Disjunction (Principle 6) is reflected by the parallel-OR operator in PARLOG, which coupled with the use of guards and input matching can be used to ensure either exclusive clause choice or to allow the system to arbitrarily select one clause for committal if the choice is non- exclusive. Principle 7 (Renaming) is not realisable in PARLOG. Naturally, as a language orientated towards system programming, PARLOG incorporates explicit sequencing operators to allow for such a control where needed, for example in enforcing synchronousity or in i/o operations (Principle 8).

3. Format of the examples

The example 'programs' in this paper will generally be given in the following forms:

(i) Standard PARLOG (including modes).

(ii) Calls in the form of $?goal_1,...,goal_n$ to the above PARLOG programs.

(iii) CCS with value passing.

4. Synchronous Communication.

We restrict ourselves to the synchronous form of communication possible in PARLOG using the method of "Incomplete messages", rather than mode reversal. We represent the encapsulated message by the tuple +/2 [1], the first argument being the message item itself, and the second argument the synchronisation variable:

Item + Reply

4.1. Synchronous ‘primitives’ in PARLOG

We can implement synchronous send and receive in PARLOG by the following programs:

Example 2.

```
mode synch_send(item?,tuple^).
synch_send(Item,Item+Reply)← data(Reply).

mode synch_receive(tuple?,item^).
synch_receive(Item+Reply,Item)← Reply=reply.
```

Note that *data/1* suspends until its argument is ground (ie the top level functor of a data-structure is instantiated). Thus the query *?data(X)* suspends, and *?data([H|T])* succeeds. The synchronous PARLOG programs can be composed as follows:

?synch_send(foo , Msg) , synch_receive(Msg , Item).

We model the call to synch_send/2 as the CCS form o" $\overline{\alpha}$"(foo), synch_receive/2 as (*a(x), and their composition as o" $\overline{\alpha}$"(foo) | α(x).

4.2. Producers and Consumers.

Let us consider the situation of a chained producer and consumer in PARLOG. The synchronisation primitives may be used in the formulation of a simple recursive non-terminating communicating system, where producer/1 communicates with consumer/1 via a stream consisting of a partly instantiated list of +/2 tuples. Note that the communication is constrained by the '&' operator in producer/1, and thus the stream has a maximum length of 1, ie it may consist at the most of a head element which is a +/2 tuple, and an uninstantiated tail. The producer will suspend until the consumer acknowledges receipt of the message, when it will produce the next message. The self-recursive call in the consumer will suspend until the head of the stream is instantiated, and thus "lock-step" communication is achieved.

[1] +/2 is an infix operator in PARLOG

Example 3.

```
mode producer(Stream^).
producer([Msg|Stream])←
     synch_send(item,Msg)& producer(Stream).

mode consumer(Stream?).
consumer([Msg|Stream])←
     synch_receive(Msg,Item), consumer(Stream).
```

? producer(Stream) , consumer(Stream).

The situation is analogous to the CCS :

Example 4.

$P = \overline{\alpha}(item) \ . \ P$

$C = \alpha(x) \ . \ C$

$PC = P \ | \ C$

5. Communication via buffers

5.1. One place buffers

Our first example of buffered communication is that of a simple one place buffer, which may be formulated in PARLOG as:

Example 5.

```
mode buffer1a(Ins?,Outs?).
buffer1a([In|Ins],Outs)←
     synch_receive(In,X) & buffer1a(X,Ins,Outs).

mode buffer1a(Item?,Ins?,Outs^).
buffer1a(X,Ins,[Out|Outs])←
     synch_send(X,Out) & buffer1a(Ins,Outs).
```

A CCS representation of this would be:

$B = \alpha(x) \ . \ B(x)$

$B(x) = \overline{\gamma}(x) \ . \ B$

Note that the PARLOG formulation partially evaluates to:

Example 6.

```
mode buffer1b(Ins?,Outs^).
buffer1b([In|Ins],[Out|Outs])←
     synch_receive(In,X) & synch_send(X,Out)&
     buffer1b(Ins,Outs).
```

A CCS representation of buffer1b/2 would be:

$B = \alpha(x)\ .\ \overline{\gamma}(x)\ .\ B$

These PARLOG buffers may be composed with producer/1 and consumer/1 above in the following manner:

|? - producer(A) , buffer(A,B) , consumer(B).

and the CCS buffer may likewise be composed, assuming suitable renaming of event labels:

$P \mid B \mid C$

In general, we would like to reason about the equivalence of the behaviour of two programs regarding both dynamic and completion behaviour. In this case, we note that there is no completion behaviour for the producer-buffer-consumer system, which is non-terminating. Partial evaluation can be used to demonstrate that buffer1a/2 and buffer1a/3 together are ‘equivalent’ in operation to buffer1b/2.

Another way to specify a one place buffer in PARLOG is to reformulate buffer1a to explicitly use a list rather than an extra argument as a data store:

Example 7.

```
mode buffer1c(Ins?,Outs^).
buffer1c(Ins,Outs)←
     buffer1c([],Ins,Outs).

mode buffer1c(Store?,In?,Out^).

buffer1c([],[In|Ins],Outs)←
     synch_receive(In,Item) & buffer1c([Item],Ins,Outs).

buffer1c([Item],Ins,[Out|Outs])←
     synch_send(Item,Out) & buffer1c([],Ins,Outs).
```

The CCS formulation for this is (disregarding the rules for ADTs[2]):

$B = B(nil)$

$B(nil) = \alpha(x)\ .\ B([x])$

$B([x]) = \overline{\gamma}(x)\ .\ B(nil)$

Ignoring the possible overheads of constructing a list, the programs for buffer1a, buffer1b and buffer1c are operationally equivalent.

5.2. Two place buffers

Building on the formulation of buffer1a, a possible specification of a two place buffer would be:

Example 8.

```
mode buffer2a(Ins?,Outs^).
buffer2a([In|Ins],Outs)←
     synch_receive(In,X) & buffer2a(X,Ins,Outs).

mode buffer2a(Item?,Ins?,Outs^).
buffer2a(X,[In|Ins],Outs)←
     synch_receive(In,Y) & buffer2a(X,Y,Ins,Outs).

buffer2a(X,Ins,[Out|Outs])←
     synch_send(X,Out) & buffer2a(Ins,Outs).

mode buffer2a(Item1?,Item2?,Ins?,Outs^).
buffer2a(X,Y,Ins,[Out|Outs])←
     synch_send(X,Out) & buffer2a(Y,Ins,Outs).
```

The buffer2a may be run in conjunction with a producer and consumer:

|? - producer(S) , buffer(S,T), consumer(T).

A CCS representation of buffer2a is:

$$\begin{aligned}
\mathrm{B2} &= \alpha(x)\ .\ \mathrm{B2}(x)\\
\mathrm{B2}(x) &= \tau\ .\ \alpha(y)\ .\ \mathrm{B2}(x,y)\ +\ \tau\ .\ \overline{\gamma}(x)\ .\ \mathrm{B2}\\
\mathrm{B2}(x,y) &= \overline{\gamma}(x)\ .\ \mathrm{B2}(y)
\end{aligned}$$

Note that the use of τ on both sides the choice operator in the CCS definition for B2(x) reflects the absence of guards in buffer2a/3.

In fact, the PARLOG buffer as formulated above does not respond to the demands of its environment. By this we mean that the buffer itself will decide whether to input or output in the case of buffer2a/2. If, for example, the second clause of the relation is selected, and the consumer is not ready to receive, the buffer will not at that point attempt to receive another item from the producer and will suspend. What we really need is a *guarded choice*, which in the CCS would be indicated by dropping the τ's, converting the choice in B2(x) to

[2] We let **nil** represent the empty list, and **[X]** represent a list containing one item.

$$\alpha(y) . B2(x,y) + \overline{\gamma}(x) . B2$$

The ‘equivalent’ PARLOG code should be:

```
mode buffer2a(Item?,Ins?,Outs^).
buffer2a(X,[In|Ins],Outs)←
        synch_receive(In,Y) : buffer2a(X,Y,Ins,Outs).

buffer2a(X,Ins,[Out|Outs])←
        synch_send(X,Out) : buffer2a(Ins,Outs).
```

Unfortunately, a property of PARLOG programs is that the guard is unsafe if it binds variables in the call, and also that no output can be made in a guard. A call to the above program for buffer2a/2 will never commit to the second clause for the second reason, and the first clause is unsafe due to the second reason. We can program around this deficiency either by using external monitor processes [Gilbert87b] or reversing the modes on the buffer-consumer stream (see the program for bufferinf_f, below).

An alternative formulation for a two-place buffer is to chain two one-place buffers. In this case, it does not matter which program for the one-place buffers is used.

Example 9. An alternative 2-place buffer

```
mode buffer2b(Ins?,Outs^) .
buffer2b(Ins,Outs) ←
    buffer1(Ins,Mid) , buffer1(Mid,Outs) .
```

In CCS, the chaining operation would be represented by:

$B2b = B \frown B$

where

$B \frown B = (B[\delta/\overline{\gamma}] \mid B[\delta/\alpha]) \backslash \delta$

We note that the buffer implemented by buffer2b/2 above has a dynamic size, due to which processes are holding items. If we represent the cells of the buffer by either 0, a or b where a comes before b in the input stream, the buffer could be in either of the following states: `<0,0>` `<a,0>` `<0,a>` `<b,a>`.

Can we say that buffer2a and buffer2b are equivalent in some sense? One transformation we can perform before answering this question is to reformulate buffer2a to use an explicit data structure as the data store rather than mutual recursion and arguments:

Example 10.

```
mode buffer2c(Ins?,Outs^).
buffer2c(Ins,Outs)←
    buffer2c([],Ins,Outs).

mode buffer2c(Store?,Ins?,Outs^).

buffer2c([],[In|Ins],Outs)←
    synch_receive(In,Item) & buffer2c([Item],Ins,Outs).

buffer2c([Item],Ins,[Out|Outs])←
    synch_send(Item,Out) & buffer2c([],Ins,Outs).

buffer2c([Item1],[In|Ins],Outs)←
    synch_receive(In,Item2) & buffer2c([Item1,Item2],Ins,Outs).

buffer2c([Item1,Item2],Ins,[Out|Outs])←
    synch_send(Item1,Out) & buffer2c([Item2],Ins,Outs).
```

The CCS formulation for this is (again disregarding the rules for ADTs[3]):

$$\mathtt{B} = \mathtt{B(nil)}$$
$$\mathtt{B(nil)} = \alpha(\mathtt{x}) \;.\; \mathtt{B([x])}$$
$$\mathtt{B([x])} = \tau \;.\; \overline{\gamma}(\mathtt{x}) \;.\; \mathtt{B(nil)} + \tau \;.\; \alpha(\mathtt{y}) \;.\; \mathtt{B([x]<>[y])}$$
$$\mathtt{B([x]<>[y])} = \overline{\gamma}(\mathtt{x}) \;.\; \mathtt{B([y])}$$

As with buffer2a, the choice points should be guarded (remove the τ's in the CCS, put the calls to synch_send/2 and synch_receive/2 into guards in buffer2c/3 as appropriate). We can use the same programming techniques as before to overcome this problem.

There is a fundamental difference between the dynamic behaviour of the different *producer-buffer2n-consumer* systems. This difference is due to the number of processes in existence at any one time, but is not reflected in the CCS forms. Consider the overall behaviour of

?producer(A) , buffer2a(A,B) , consumer(B).

We note that only one of producer or consumer may access the buffer at any one time because buffer2a can only respond to either the input stream or the output stream, due to the structure of the program for buffer2a/3. The potential for parallelism in this case is reduced. On the other hand, the overall behaviour of

[3] We use <> as the concatenation operator on lists.

?producer(A) , buffer2b(A,B) , consumer(B).

is quite different: the buffer may communicate with both producer and consumer simultaneously. In both cases a situation of bounded asynchronousity exists: if the consumer is slower than the producer, the latter will be constrained by the rate of the former only after the buffer has been filled up for the first time. During the input of the first two data items, the producer will only be constrained by the rate of execution of the buffer cells.

The CCS buffers can be conceived of behaving quite differently when composed with a producer and a consumer: due to the interleaving semantics of the CCS parallel operator, neither buffer can communicate with more than one other entity at a time.

6. Unbounded buffers and queues

We now consider unbounded buffers. The formulation of an unbounded buffer using the technique of arguments to store the data items is sketched out below. This method is obviously impracticable due to the inability to specify an infinite number of relations with an increasing number of arguments:

Example 11. Unbounded buffer using arguments

```
mode bufferinf_a(Ins?,Outs^).
bufferinf_a([In|Ins],Outs)←
     synch_receive(In,X1)& bufferinf_a(X1,Ins,Outs).

mode bufferinf_a(X1?,Ins?,Outs^).
bufferinf_a(X1,[In|Ins],Outs)←
     synch_receive(In,X2)& bufferinf_a(X1,X2,Ins,Outs).

bufferinf_a(X1,Ins,[Out|Outs])←
     synch_send(X1,Out)& bufferinf_a(Ins,Outs).

mode bufferinf_a(X1?,...,Xn?,Ins?,Outs^).
bufferinf_a(X1,...,Xn,[In|Ins],Out)←
     synch_receive(In,Xn+1)& bufferinf_a(X1,...,Xn,Xn+1,Ins,Outs).

bufferinf_a(X1,...,Xn-1,Xn,[In|Ins],Out)←
     synch_send(In,Xn)& bufferinf_a(X1,...,Xn-1,Ins,Outs).
```

The way to avoid the problem of specifying relations with the number of arguments ranging up to infinity is to use a list to represent the stored data items:

Example 12. ‘Equivalent’ infinite buffer

```
mode bufferinf_b(Ins?,Outs^) .
bufferinf_b(In,Out) ←
        bufferinf_b([],In,Out) .

mode bufferinf_b(Store?,Ins?,Outs^) .
bufferinf_b([],[In|Ins],Outs) ←
    synch_receive(In,Item) & bufferinf_b([Item],Ins,Outs) .

bufferinf_b([Msg|Msgs],[In|Ins],Outs) ←
    synch_receive(In,Item) &
    append([Msg|Msgs],[Item],Msgs1) ,
    bufferinf_b(Msgs1,Ins,Outs) .

bufferinf_b([Msg|Msgs],Ins,[Out|Outs]) ←
    synch_send(Msg,Out) & bufferinf_b(Msgs,Ins,Outs) .
```

The CCS formulation of the above would be (again, ignoring the ADT):

Example 13.

$$B\infty = B\infty(nil)$$
$$B\infty(nil) = \alpha(x) \, . \, B\infty([x])$$
$$B\infty([x] <> l) = \tau . \, \alpha(y) \, . \, B\infty \, ([x] <> l <> [y]) + \tau . \, \overline{\gamma}(x) \, . \, B\infty(l)$$

Again, we note that PARLOG’s inability to permit output in a guard means that the choices are in effect unguarded. In the case that the data store of messages is empty (ie, is not of the form [Msg|Msgs]), the first clause of bufferinf_c/3 would be chosen by the PARLOG evaluator. However, in the case that there was a message on the input stream, and the store was not empty, the third clause might be selected, and an attempt made to send a message. If the consumer was never ready to receive, deadlock would occur earlier than if the buffer responded fairly to the requirements of both consumer and producer.

We give below the PARLOG program for the guarded choice formulation indicated by the CCS:

$$B\infty([x] <> l) = \alpha(y) \, . \, B\infty \, ([x] <> l <> [y]) + \overline{\gamma}(x) \, . \, B\infty(l)$$

In the following PARLOG program, we use a consumer which *sends* requests to remove items from the buffer, and hence both consumer/1 and producer/1 have an output mode. The consumer produces a stream of -/2 tuples which have as the first argument the instruction to remove an item and as the second argument the item removed from the buffer. The buffer in this case has input modes on both the input and the output stream, and employs suspension coupled with pattern matching to repond to requests to add or remove items from the data store. Note that *fairness* is not guaranteed by this version, and we would need to use a technique similar to a *fair merge* [Shapiro84] to ensure fair attention by the buffer to each stream of requests.

Example 14.

```
mode bufferinf_b(Ins?,Outs?) .
bufferinf_b(In,Out) ←
    bufferinf_b([],In,Out) .

mode bufferinf_b(Store?,Ins?,Outs?) .
bufferinf_b([],[In|Ins],Outs) ←
    synch_receive(In,Item) &
    bufferinf_b([Item],Ins,Outs) .

bufferinf_b([Msg|Msgs],[Item + R|Ins],Outs) ←
    R=reply &
    append([Msg|Msgs],[Item],Msgs1) ,
    bufferinf_b(Msgs1,Ins,Outs) .

bufferinf_b([Msg|Msgs],Ins,[remove - X |Outs]) ←
    X=Msg &
    bufferinf_b(Msgs,Ins,Outs) .

mode consumer(Requests^).
consumer([H|T]) ←
    request(H,X)&
    consumer(T).

mode request(Request^,Item^).
request( remove - Item , Item)  ← data(Item).
```

Reformulating the unbounded buffer using individual processes to store the data items, we face the problem of creating buffers which never output anything. One such program is presented below, which inputs items, but can never output them due to infinite tail recursion creating an unbounded buffer ahead of the most recently created buffer1 cell:

Example 15.

```
mode bufferinf_d(Ins?,Outs^).
bufferinf_d(Ins,Outs)←
    buffer1(Ins,Mids), bufferinf_d(Mids,Outs).
```

The CCS for this buffer is:

```
B∞ = B ∩ B∞
```

Alternatively, we can create an unbounded buffer which never outputs since it can never input. In this case, there is an unbounded buffer between the most recently created buffer1 cell and the input stream.

Example 16.

```
mode bufferinf_d(Ins?,Outs^) .
bufferinf_d(Ins,Outs) ←
    bufferinf_d(Ins,Mids)  , buffer1(Mids,Outs) .
```

The CCS for the above is:

```
B∞ = B∞ ∩ B
```

We attempt to rectify the problem of no output on an infinite buffer by ensuring that items are input, and then are output whilst at the same time the buffer grows in length.

Example 17.

```
mode bufferinf_e(Ins?,Outs^) .
bufferinf_e([In|Ins],Outs) ←
    synch_receive(In,Item) &
    bufferinf_e(Ins,Mids) , buffer1a(Item,Mids,Outs) .
```

The CCS for this program is:

```
B1∞ = α(x) . ( B1∞ ∩ B(x) )
```

The infinite buffer may be rewritten as:

Example 18.

```
mode bufferinf_f(Ins?,Outs^) .
bufferinf_f([In|Ins],Outs) ←
    bufferinf_f(Ins,Mids) , buffer1([In|Mids],Outs) .
```

This program is not readily expressible in CCS since we are explicitly manipulating the communication streams in the PARLOG program. We note that the buffers bufferinf_e/2 and bufferinf_f/2 always increase in length with each data item processed; operationally this may swamp the system with processes. Of more importance is the fact that in an implementation, communication between buffer cells takes a finite time, and the overall delay associated with the buffer will increase with the number of items that it has ever input[4]. CCS ignores such implementation oriented problems.

[4] The buffer may be implemented with individual cells on different processors, so that introducing more cells may not necessarily introduce greater inefficiency due to the processing time required by each cell regardless of intercell communication.

The difference in the number of items stored in the two types of buffers (ie using a list or processes) is not apparent operationally here. Both buffers have a store in the range of $0 \leq \text{size} < \infty$; however the size of the data-structure store can range dynamically within these limits, whilst the size of the process buffer monotonically increases during its execution.

Another way of formulating the infinite buffer is to use a variation of the one slot buffer, but allowing unconstrained recursion so that the producer can run ahead of the consumer. Note that the difference with buffer1b/2 is the use of the parallel-AND operator between the receive_send calls (which we bracket for clarity) and the self-recursive call to the buffer. The size of such a buffer may grow or shrink according to the rates of the producer and consumer. Messages are held in the suspended calls to synch_send/2 rather than in a system buffer as in PARLOG's naive asynchronous form of communication. The clausal form is the same as the regular one-slot buffer. In this case there is no easy inductive reasoning path towards the formulation of an n-place buffer from the one-place buffer.

Example 19.

```
mode bufferinf_g(Ins?,Outs^).
bufferinf_g([In|Ins],[Out|Outs])←
     ( synch_receive(In,X) & synch_send(X,Out) ) ,
     bufferinf_g(Ins,Outs).
```

This program is harder to model in CCS than the previous ones. If we naively translate bufferinf_g/2 as follows, we get an *infinite bag*, ie the order of output of items is not preserved:

$$B\infty = (\alpha(x) \ . \ \overline{\gamma}(x)) \ | \ B\infty$$

We note that one way of programming an infinite bag in PARLOG is:

Example 20.

```
mode bag(Ins?,Outs^).
bag([In|Ins],Out_all)←
     synch_receive(In,X)&
     synch_send(X,Out),
     bufferinf_g(Ins,Outs),
     merge([Out],Outs,Out_all).
```

where *merge/3* is non-deterministic.

The problem is that the streams in PARLOG impose an ordering which is lost in the CCS version. Thus in the CCS formulation above, we cannot guarantee in which order the items input into the bag will be output. We can impose an order, however, by using synchronisation signals:

Example 21.

```
B∞ = Start ⌒ B´∞
Start = $\overline{\delta}$ . $\overline{\delta}$
B´∞ = Cell ⌒ B´∞
Cell = $\varepsilon$ . $\alpha$(x) $\overline{\delta}$ . $\varepsilon$ . $\overline{\gamma}$(x) . $\overline{\delta}$ .

where def
    P ⌒ Q = ( P[$\gamma/\beta$] | Q[$\gamma/\alpha$] ) \ $\gamma$
```

In this case, the ε and $\overline{\delta}$ signals are used to sequence the input and output of items; each α(x).$\overline{\gamma}$(x) pair is guarded by an ε which is not activated until a previous α(x).$\overline{\gamma}$(x) pair has input a value. Likewise, the output of the pair cannot take place until a synchronisation signal has been received from the previous pair on successful output of its value. The renaming in the definition of the chaining operation '⌒' is the equivalent of an infinite number of different synchronisation flags.

However, this formulation allows the 'eager' creation of buffer cells, a condition which is not strictly necessary, since more buffer cells may be created than are needed. This is not the case in the PARLOG program above, where the spawning of new buffer cells is restricted by the rate at which the input stream is being partially instantiated. Thus in the PARLOG version, no more buffer cells are spawned than are required by the producer.

To force the spawning of buffer cells to be in step with the production of messages from the producer, we formulate the following CCS 'program'. The main point to note is that the spawning of new buffer cells is guarded by the initial ε action, which waits until it receives a synchronisation signal from a previous buffer pair.

Example 22.

```
B∞ = Start ⌒ B´∞
Start = $\overline{\delta}$ . $\overline{\delta}$
B´∞ = $\varepsilon$ . (Cell ⌒ B´∞ )
Cell = $\alpha$(x) $\overline{\delta}$ . $\varepsilon$ . $\overline{\gamma}$(x) . $\overline{\delta}$
```

7. Bounded buffers

We may formulate a bounded buffer using either the data-structure technique or an explicit number of one place buffer processes. The buffer formulated using processes as stores is:

Example 23. Bounded Buffer - processes

```
mode bounded(Bound?,Ins?,Outs^).
bounded(Size,Ins,Outs) <-
    Size > 0 :
    Size1 is Size-1 ,
    buffer1(Ins,Mids) , bounded(Size1,Mids,Outs).

bounded(0,Outs,Outs).
```

Note that this program is quite different to that proposed by [Takeuchi87], which is closer to a program which uses data-structures as stores. The following example uses a **difference list** as a store, making the operation of adding to the end of the list more efficient (the second clause of *bounded/5*), and doing away with the need for a call to *append*.

Example 24. Bounded buffer - data structures

```
mode bounded(Bound?,Ins?,Outs^) .
bounded(Bound,In,Out) ←
    bounded(Bound,0,Var/Var,In,Out) .

mode bounded(Bound?,Size?,Store?,Ins?,Outs^) .

bounded(Bound,Size,Msgs/Tail,[In|Ins],Outs) ←
    Size =< Bound :
    synch_receive(In,Item) ,
    Tail=[Item|NewTail] , Size1 is Size + 1 ,
    bounded(Bound, Size1, Msgs/NewTail,Ins,Outs) .

bounded(Bound, Size, [Msg|Msgs]/Tail,Ins,[Out|Outs]) ←
    Size > Bound :
    synch_send(Msg,Out) ,
    Size1 is Size - 1 ,
    bounded(Bound, Size1, Msgs/Tail,Ins,Outs) .
```

8. Conclusions

We have discussed a class of communication based algorithms, restricted to a producer-buffer-consumer scenario. PARLOG programs have been formulated, and informal equivalences between them noted. In order to enable such comparisons, formulations in CCS have been given for the same algorithms. We note that PARLOG lacks a formal denotational semantics based on logic, and that the most recent work on semantics has been to model PARLOG in CCS [Ellis86 , Hussey87]. The inability of a PARLOG clause to produce output before committal means that in some cases choice is unguarded, and thus the behaviour of resulting programs may not be desirable. There are ways to avoid this, based on the use of external stream monitoring processes [Gilbert87b], mode reversal, or by metalevel programming. The above inability is a deficiency that PARLOG needs to rectify before it could be used effectively as a specification language. Advantages that PARLOG has over CCS as a ‘specification-implementation’ language include its ‘truly parallel’ model of execution whereby any number of processes may be in existence at any instant. We also note that the stream based model of communication employed by PARLOG permits easier specification of a process-based infinite buffer with non-monotonically increasing size than does the event gate model of CCS.

Acknowledgements

This paper was written while the author was funded by Alvey on "Implementation and Applications of PARLOG", Project number 043/098.

I would like to thank Reem Bahgat, Chris Hogger, Iain Phillips and Steve Vickers of Imperial College for all the time that they gave me, and the helpful comments that they made during the preparation of this paper.

References

Ellis86.
Mark R. Ellis, “A Relational Language into ECCS,” M.Sc, Imperial College of Science & Technology, London. UK, 12th September 1986.

Gilbert87.
David Gilbert, “PARLOG: a tutorial introduction.,” *Proceedings of Parallel Processing and Supercomputing*, Begian Institute for Automatic Control, Antwerp, Belgium, November 19-20, 1987.

Gilbert87b.
David Gilbert, “Executable LOTOS: Using PARLOG to implement an FDT,” *Proceedings of IFIP Protocol Specification, Testing and Verification: VII, Zurich, Switzerland, 5-8 May 1987*, Elsevier Science, North-Holland, Amsterdam, Netherlands, 1987.

Gregory87.
Steve Gregory, *Parallel Logic Programming in PARLOG: The Language and its Implementation,* Addison-Wesely, London, UK, 1987.

Hussey87.
Charlie Hussey, “Interpreting PARLOG Programs as CCS agents,” Report for MSc degree in Engineering, Imperial College of Science and Technology, London, September 1987.

Milner86.

Robin Milner, “Process Constructors and Interpretations,” *Proceedings of IFIP 10th International World Computer Congress*, vol. 10, pp. 507-514, North Holland, Dublin, Ireland, September 1-5, 1986.

Shapiro83.

Ehud Shapiro and Akikazu Takeuchi, “Object Oriented Programming in Concurrent Prolog,” CS83-08, Department of Applied Mathematics, Weizmann Institute of Sciences, Rehovot, Israel, June 1983.

Shapiro84.

E. Shapiro and C. Mierowsky, “Fair, Biased, and Self-Balancing Merge Operators: Their Specification and Implementation in Concurrent Prolog,” CS84-07, Weizmann Institute, Rehovot, Israel, 1984.

Takeuchi87.

Akikazu Takeuchi and Koichi Furukawa, “Bounded Buffer Communication in Concurrent Prolog,” in *Concurrent Prolog*, ed. Ehud Shapiro, vol. 1, pp. 464-475, MIT Press, 1987.

Tables

The PARLOG syntax used in this article is given in the Table below.

Symbol	**Meaning**
$\leftarrow$	logical implication
&	sequential-AND
	parallel-AND
;	sequential-OR
.	parallel-OR
:	guard operator
?	input mode annotation
^	output mode annotation

Table 1 PARLOG syntax