A RELATIONAL LANGUAGE FOR PARALLEL PROGRAMMING

Keith L. Clark & Steve Gregory

Department of Computing, Imperial College
London SW7 2BZ, England

1. INTRODUCTION

A parallel program often defines a relation not a function. The program constrains the output to lie in some relation R to the input, but the particular output produced during a computation can depend on the time behaviour of component processes. This suggests the use of a relational language as an applicative language for parallel programming.

The Horn clause subset of predicate logic is a relational language with an established procedural interpretation for non-deterministic sequential computations [Kowalski 1974]. In this paper we modify and extend that interpretation to define a special purpose parallel evaluator.

We begin by restricting Kowalski's procedural interpretation to incorporate the committed (don't care) non-determinism of Dijkstra's [1976] guarded commands. We then generalize the interpreter to allow parallel and forking evaluation of a set of component processes. Shared variables become the communication channels between a single producer and several consumer processes. An annotation on each shared variable selects the producer. Time-dependent behaviour can result if a non-deterministic consumer process reads from more than one channel. Different communication rates of the different producers can then affect the consumer evaluation path. This gives us a relational and non-deterministic variant of the Kahn and MacQueen [1977] model.

Using an additional annotation we allow bounded buffer or zero buffer communication constraints to be set for each channel. The inability of a producer to send a message down a channel can now affect the producer evaluation path. With zero buffer communication on every channel, our model is an applicative analogue of Hoare's [1978] CSP.

2. THE SEQUENTIAL INTERPRETER

Notation and terminology

A program is a set of clauses.

A clause is a sentence of the form

$$P \leftarrow G_1 \& \ldots \& G_k \mid A_1 \& \ldots \& A_m \qquad k \geq 0,\ m \geq 0 \qquad (1)$$

where P, each G_i and each A_j is an atom. P is the consequent or head of the clause while $G_1 \& \ldots \& G_k \mid A_1 \& \ldots \& A_m$ is the antecedent. The antecedent consists of a guard sequence $G_1 \& \ldots \& G_k$ followed by a goal sequence $A_1 \& \ldots \& A_m$. These are separated by a clause bar '|'.

An atom is of the form $R(t_1,\ldots,t_n)$ where R is a relation name, and $t_1,\ldots,t_n$ are terms.

Data structures

The variable-free terms are the data structures of the language (we use lower case identifiers for variables). Thus {0,1,2,...} are the numeral data objects, {Nil, 2.Nil, 2.3.Nil, ...} are lists of numerals and salary(Smith,10000) is a record labelled 'salary' containing a name and a numeral.

Note that this means that the function names are always the names of data constructors. More general functions on the data structures must be treated as relations. The relation names denote the relations over the data structures that we want to compute.

Declarative reading

Let $x_1,\ldots,x_n$ be the variables of the clause (1) above. The clause can be read

> For all data structures $x_1,\ldots,x_n$:
> P if G_1 and ... and G_k and A_1 and ... and A_m

Thus, each clause can be read as a statement about the computable relations to which it refers. The program is (partially) correct if each clause is a true statement.

Example This defines the 'min-of' relation on numerals in terms of the '≤' relation.

u min-of <u,v> <- u ≤ v |
v min-of <u,v> <- v ≤ u |

Example This defines the 'merge-to' relation over lists. <x,y> merge-to z is the relation: z is an arbitrary interleaving of the lists x and y which preserves the order of their elements.

<u.x,y> merge-to u.z <- <x,y> merge-to z
<x,v.y> merge-to v.z <- <x,y> merge-to z
<Nil,y> merge-to y
<x,Nil> merge-to x

Procedural semantics

An evaluable expression is of the form

$t:B_1\&...\&B_j$

where t is a term and each B_i is an atom. The conjunction $B_1\&...\&B_j$ is a sequence of calls. Each variable in t must appear in at least one of these calls.

A terminating evaluation of the expression returns a data structure [t]s, (t with its variables replaced by data structure bindings of variables given in s). s is a set $\{u_1/t_1,...,u_n/t_n\}$ of data structure bindings for the variables $u_1,...,u_n$ of $B_1\&...\&B_j$. It is such that $[B_1\&...\&B_j]s$ is a logical consequence of the statements of the program.

The following is a recursive description of the computation of the output substitution s.

Case 1: j>1

The computation of s for $B_1\&...\&B_j$ is reduced to the computation of a substitution r of data structure bindings for the variables of B_1 followed by the computation of a substitution s' of data structure bindings for the variables of $[B_2\&...\&B_j]r$. s is then the set of bindings (r union s'). Notice that this means that the output binding for a variable is always computed by the evaluation of the first call in which it appears. This output binding is then passed on to later calls.

Case 2: j=1

The computation of the substitution s of data structure bindings for the variables of a single call $R(t_1,...,t_n)$ is non-deterministic. Let

$$R(t'_1,...,t'_n)\leftarrow G_1\&...\&G_k \mid A_1\&...\&A_m$$

be a program clause. This clause is a candidate clause for the computation of s if

1. There is a (partial) output substitution r of bindings for some or all of the variables of $R(t_1,...,t_n)$ and an input substitution r' of bindings for all the variables of $R(t'_1,...,t'_n)$ such that

   $$[R(t_1,...,t_n)]r = [R(t'_1,...,t'_n)]r'$$

   Here (r union r') is a most general unifier of the two atoms; see [Robinson 1965].

2. The instance $[G_1\&...\&G_k]r'$ of the guard sequence is a true conjunction of variable-free atoms. Notice that this means that all the variables in the guard sequence must be bound to data structures by the input substitution r'.

Using any candidate clause the computation of the substitution s for the call $R(t_1,...,t_n)$ is reduced to the computation of a substitution s' for variables of the input instance $[A_1\&...\&A_m]r'$ of the goal conjunction of the clause. s is then the bindings of the output substitution r evaluated with respect to s'. (If r has the binding y/t, s has the binding y/[t]s' .) If m=0, i.e. the goal sequence of the candidate clause is empty, the evaluation of the call terminates with s equal to r.

If there are no candidate clauses for $R(t_1,...,t_n)$, the computation aborts.

Example Consider the single atom evaluable expression

w: <2.3.Nil,4.Nil> merge-to w

Each of the clauses

<u.x,y> merge-to u.z <- <x,y> merge-to z
<x,v.y> merge-to v.z <- <x,y> merge-to z

is a candidate clause.

For the first of these clauses the input substitution is {u/2, x/3.Nil, y/4.Nil} and the output substitution is {w/2.z}.

For the second clause the input substitution is {v/4, x/2.3.Nil, y/Nil} and the output substitution is {w/4.z}.

Neither clause has a guard sequence to check. In each case, the input substitution decomposes one of the data structure arguments of the call and the (partial) output substitution gives the first approximation to the data structure that will be the output binding for w if that clause is used.

Example w: w min-of <2,3>

This time there is just one candidate clause:

u min-of <u,v> <- u ≤ v |

The input bindings are {u/2, v/3} and the output binding is {w/2}. The reason that the second 'min-of' clause is not a candidate is that its guard '3 ≤ 2' is not true.

Concurrent search for a candidate clause

We assume that given the call $R(t_1,...,t_n)$ all of the program clauses for R are tested in parallel for candidacy. The first clause to pass the test is the one that is used. There is no backtracking on this choice.

Notice that this is the committed choice nondeterminism of Dijkstra's [1976] language of guarded commands. Indeed, our sequential language

is the applicative analogue of his language.

What we have described is a special purpose resolution theorem prover featuring "don't care" non-determinism [Kowalski 1979].

Example evaluation

The following sequence of calls represents one possible evaluation of the call
<2.3.Nil,4.Nil> merge-to w

<2.3.Nil,4.Nil> merge-to w
↓ partial output {w/2.z}
<3.Nil,4.Nil> merge-to z
↓ partial output {z/4.z'}
<3.Nil,Nil> merge-to z'
↓ partial output {z'/3.z"}
<Nil,Nil> merge-to z"
↓ partial output {z"/Nil}
(empty)

As the evaluation proceeds the sequence of partial output bindings

w/2.z, z/4.z', z'/3.z", z"/Nil

give us a sequence of partial approximations

2.z, 2.4.z', 2.4.3.z", 2.4.3.Nil

to the final output data structure. Each new approximation gives a further element of the list. Lists approximated to in this way we shall call streams. This will be an important concept in our extension to parallel evaluations.

Modes

We have seen that the candidacy condition for a clause can only be satisfied if all the variables of a guard sequence are bound to data structures by the input substitution. In practice this means that a set of clauses for a relation are a usable program only for a restricted set of calls. Thus, the 'min-of' program can only be used for calls of the form u min-of <v,w> where v and w are given.

Let us call each allowed pattern of call a mode. A declaration of the different modes in which the clauses for a relation will be used provides useful documentation. As with the Edinburgh Prolog compiler [Warren 1977], it can also be used to optimize the compilation of the clauses.

A mode declaration for the n-ary relation R takes the form

mode $R(m_1,\ldots,m_n)$

where each m_i is either '?' or '^'. '?' means that the corresponding argument in the call will be a data structure, while '^' means that it will be a variable. There may be several mode declarations for a relation.

The following are two mode declarations for the 'merge-to' program:

mode <?,?> merge-to ^, mode <^,^> merge-to ?

3. THE PARALLEL INTERPRETER

We now extend the language to allow the goal sequence of a clause (and the initial evaluable expression) to be split into a number of sequential components, separated by the '//' symbol:

$P \leftarrow G_1 \& \ldots \& G_k \mid S_1 // \ldots // S_p$

A sequential component S_i is a conjunction $A_1 \& \ldots \& A_m$ of atoms.

The declarative reading of the clause is unchanged by the splitting: '//', like '&', is read as 'and'. Operationally, however, the sequential components are intended to be evaluated concurrently on independent processors. The evaluation of a sequential component is a process.

Shared variables

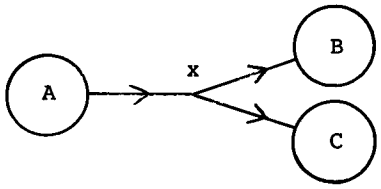
Provided the sequential components of a clause have no variables in common there are no synchronization problems. However, if a number of components share a variable, the binding of this variable represents an inter-process communication. In the sequential language one call, the first in the sequence, generates the data structure binding for any shared variable. In the parallel language we shall similarly assume that there is exactly one producer of a shared variable x. Which one is the producer is signalled by annotating the producer occurrence of x with '^'. All other sequential components in which x appears are now consumers of the data structure binding that will be generated by the producer.

Networks

We can think of a set of parallel processes with shared variables as a network of processes connected by one-way channels. For example:

A(x^) // B(x) // C(x)

corresponds to the network



If, as here, there is more than one consumer, any values sent by the producer are replicated and conveyed to all consumers.

We shall use the term channel variable to refer to a variable shared between processes.

Procedural semantics of the parallel language

This is an elaboration of the sequential interpreter.

An evaluable expression is now of the form

$$t{:}S_1//.../S_q$$

where each variable in t has one of the sequential components S_i as its designated producer.

Each sequential component is evaluated in parallel. With each one, S_i, we associate a set I of input channels and a set O of output channels. The set I comprises those variables that the evaluation of S_i cannot bind because another sequential component is the producer for the variable. The set O comprises the variables for which S_i is the producer. I and O are the incoming and outgoing arcs, respectively, of the process node for S_i in the graph representation of the parallel evaluation.

Each process will generate a substitution s for its output channels. The answer computed by the evaluable expression is $[t]r$, where r is the union of the output substitutions of the component processes.

To generate an output substitution s for the output channels of a single process $\langle S_i, I, O\rangle$ (I the input channels of the process, O its output channels), we proceed as follows:

Case 1: $S_i = B_1\&...\&B_j$, $j>1$

As in a strictly sequential evaluation, we first evaluate $\langle B_1, I, O\rangle$ to get a substitution r for the variables $y_1,...,y_i$ of O that appear in B_1. s is then the union of r and s' where s' is the output substitution for $\langle [B_2\&...\&B_j]r, I, O-\{y_1,...,y_i\}\rangle$.

Case 2: $S_i = R(t_1,...,t_n)$

Again, as in a sequential evaluation we can only use a candidate clause. However, there is an additional restriction on the use of a candidate clause to take account of the read-only constraints on the variables in the set I.

Suppose there is a program clause with head $R(t'_1,...,t'_n)$ which satisfies the first condition of a candidate clause, i.e. there are substitutions r and r' such that

$$[R(t_1,...,t_n)]r = [R(t'_1,...,t'_n)]r'$$

To satisfy the read-only constraints r must not bind any variable of input set I to a non-variable term. If it does, we shall say that the clause is input suspended. The attempt to bind the input variable we shall call a read match on that input channel.

Given a call $R(t_1,...,t_n)$ we now have three disjoint categories of program clause:

1. the non-candidate clauses (there is either a match failure or a false guard sequence),
2. the input suspended clauses,
3. the candidate clauses.

There are also three possibilities for the next step in the evaluation of $\langle R(t_1,...,t_n), I, O\rangle$:

1. The evaluation is aborted if there are only non-candidate clauses.
2. It is suspended if there are input suspended clauses but as yet no candidate clause.
3. It is reduced to the computation of a substitution s' for

   $$[S_1//.../S_p]r'$$

   using any candidate clause

   $$R(t'_1,...,t'_n) \leftarrow G_1\&...\&G_k \mid S_1//.../S_p$$

   r' is the input substitution of the match between $R(t_1,...,t_n)$ and $R(t'_1,...,t'_n)$. The answer s is then the output substitution r of the match evaluated with respect to s'.

If there are parallel components in the new goal sequence, the evaluation of the current process forks. Each sequential component $[S_i]r'$ of the new goal sequence becomes a new process

$$\langle [S_i]r', I', O'\rangle$$

Let I" be the set of variables of $[S_i]r'$ that have a producer annotation in another component $[S_j]r'$, and let O" be the set of variables having a producer annotation in $[S_i]r'$. Then I' is (I union I") and O' is ((O union O") - I").

Note that the rules for updating the I and O sets for offspring processes represent the reconfiguration of the corresponding network.

Example Figure 1 shows the network corresponding to the evaluable expression

$$: P(y\hat{}) \;//\; C(y)$$

together with the reconfigured network which results from the $\langle P(y),\{\},\{y\}\rangle$ process using the candidate clause

$$P(x) \leftarrow Q(x\hat{}) \;//\; R(x,z\hat{}) \;//\; S(z)$$

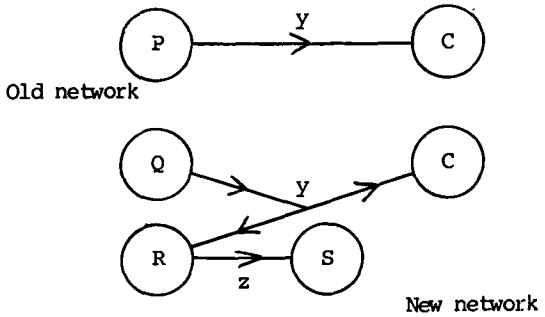


Figure 1

The use of the clause reduces the call P(y^) to Q(y^) // R(y,z^) // S(z) (the input substitution binds x to y). This forks, with <Q(y),{},{y}> taking on the role of the producer for y. <R(y,z),{y},{z}> is an extra consumer of y added to the network.

Inter-process communication

It remains to specify when data that is generated for an output channel y of a process <P,I,O> is communicated to a process <P',I',O'> which has y as an input channel. There are two cases to consider: the first corresponding to the transmission of a single message, the second to the transmission of one of a stream of messages.

Case 1:

An evaluation step of process <P,I,O> binds a channel variable y in O to a data structure s. The item s is communicated to each process <P',I',O'> with y in I' by binding y to s in the input set I'. In a subsequent evaluation step of the consumer process a read match on y accesses this binding and deletes it from the input set I'. As s is communicated, y is deleted from the output set O of the producer process <P,I,O>.

In terms of our network representation, the deletion of y from the output set O closes down the y channel. The binding y/s held in each consumer process is a buffer holding s.

Case 2:

An evaluation step of process <P,I,O> binds y in O to a first approximation s.y' of a stream of messages that begins with the data structure s. This is communicated to each consumer <P',I',O'> by binding the y in I' to s.y' and by adding y' to the input set I'. Simultaneously y is replaced by y' in the output set O of the producer process.

In our network representation, this is the communication of s as a first message down the channel connecting the two processes. The channel remains and is logically identified with the stream of messages y' that are yet to be generated. See Figure 2. The binding y/s.y' held in the consumer process is a buffer holding the first message. Again, when an evaluation step of the consumer has a read match on y the binding is accessed and then deleted from I'.

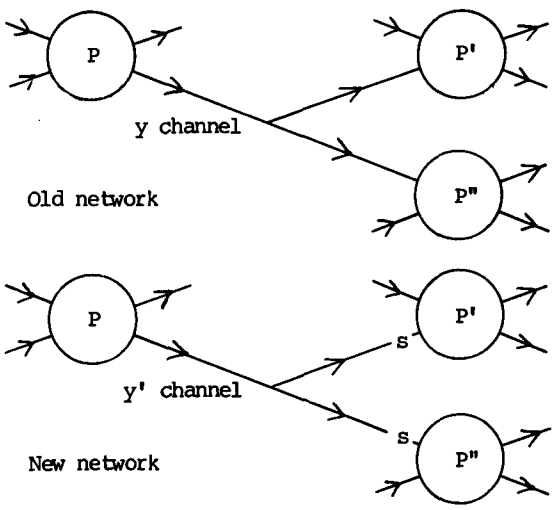


Figure 2

Unbounded buffers

It can happen that the producer process generates a new binding s'.y" for y' before the consumer has read and deleted the y/s.y' binding. This running ahead by the producer will build up a queue of messages at the consumer process. We impose no limit on the size of this stream build-up.

In allowing an unbounded build-up of messages in a channel our language is similar to Kahn and MacQueen's applicative language for parallel programming [Kahn and MacQueen 1977]. Ours differs in being relational rather than functional, and in allowing non-deterministic evaluations.

Example This program is a solution to Hamming's problem (given in [Dijkstra 1976]) of generating multiples of 2, 3 and 5 in ascending order:

```
mode multiples(^)
multiples(x) <- times(2,1.x,r^) //
        times(3,1.x,s^) // times(5,1.x,t^) //
        amerge(r,s,y^) // amerge(t,y,x^)

mode times(?,?,^)
times(n,u.x,y) <-
          v = n * u & y = v.z & times(n,x,z)

mode amerge(?,?,^)
amerge(u.x,u.y,u.z) <- amerge(x,y,z)
amerge(u.x,v.y,u.z) <- u < v |amerge(x,v.y,z)
amerge(u.x,v.y,v.z) <- v < u |amerge(u.x,y,z)
```

amerge(x,y,z) names the relation: z is an order-preserving merge of lists x and y with duplicates deleted.

multiples(x) names the relation: x is the infinite list of numbers in the set $\{2^i3^j5^k \mid i,j,k \geq 0\} - \{1\}$ in ascending order. The clause for this relation is a recursive description of such an x. It tells us that x is a fixed point of the operations

multiply the list 1.x by 2,
multiply the list 1.x by 3,
multiply the list 1.x by 5,
then merge the results, deleting duplicates.

Figure 3 is a network representation of the evaluable expression

x: multiples(x)

The '1' placed on the input arcs to the three 'times' processes comes from the 1.x which they take as input. It is the seed for the entire computation.

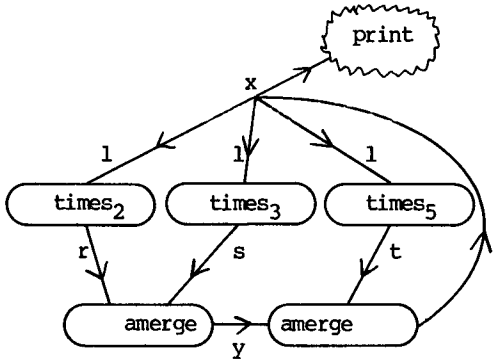


Figure 3

4. TIME-DEPENDENT PROGRAMS

Consider the case when different clauses for a relation accept input from different arguments. (Our two recursive clauses for the 'merge-to' relation in the mode <?,?> merge-to ^ are an example.) A situation can arise when one clause is input suspended while the other is a candidate clause. In this way, the availability of input to a process determines which clause is used, which in turn influences the evaluation path of the process. The relative timing of the parallel processes then determines which instance of the relation is computed. This time-dependence is essential for applications such as real-time systems.

Consider the following use of the 'merge-to' program:

z: proc1(x^) // proc2(y^) // <x,y> merge-to z^

where 'proc1' and 'proc2' each generates a stream of messages.

If there is a message in the x channel from 'proc1' but no message in the y channel from 'proc2' we must use the clause

<u.x,y> merge-to u.z <- <x,y> merge-to z

This passes on the message from 'proc1'. Only if both channels have messages pending will the choice of which message is passed on be time-independent. Thus, the instance of the 'merge-to' relation that is computed depends upon the time behaviour of the generating processes.

The Friedman and Wise 'frons' extension to Lisp [Friedman and Wise 1980] similarly allows results to be time-dependent. However, their extension to this functional programming language destroys its referential transparency. We avoid that problem by explicitly referring to the input-output relation of the many-valued function.

Example The following is an outline of a simple relational operating system with two user terminals. Each terminal is represented by a 'keyboard' process and a 'screen' process. Each command generated by a keyboard is tagged by the identifier of that terminal and the commands from both keyboards are merged into one stream. This stream is processed by a 'monitor' which produces a stream of tagged responses. The responses are routed to the appropriate screen according to their tag. The system is depicted in Figure 4.

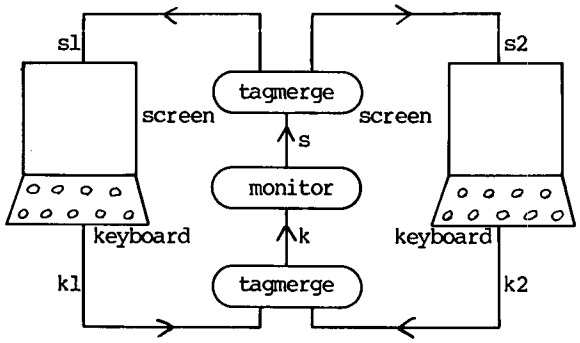


Figure 4

```
mode keyboard(^)
mode screen(?)
mode monitor(?,^)

mode tagmerge(?,?,^)
mode tagmerge(^,^,?)
tagmerge(u.x,y,<1,u>.z) <- tagmerge(x,y,z)
tagmerge(x,v.y,<2,v>.z) <- tagmerge(x,y,z)

: keyboard(k1^) // keyboard(k2^) //
  tagmerge(k1,k2,k^) // monitor(k,s^) //
  tagmerge(s1^,s2^,s) //
  screen(s1) // screen(s2)
```

keyboard(x) names the relation: x is a list of commands, in fact those which are typed at a particular keyboard.

screen(y) names the relation: y is a list of responses, which will be displayed on a particular screen.

monitor(x,y) names the relation holding between a list of tagged commands x and a list of tagged responses y. It is intended to provide all the required facilities of an operating system and give appropriate responses to commands from the users.

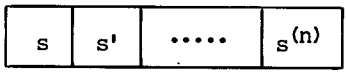
tagmerge(x,y,z) names the relation: z is some order-preserving interleaving of the lists x' and y', where x' is list x with each of its elements tagged by a '1' and y' is list y with each of its elements tagged by a '2'.

Notice that we have used the 'tagmerge' relation in two modes. In the mode tagmerge(?,?,^) the first two streams are tagged by 1 and 2 respectively and merged, in arrival order, to give

the third argument. Using the same relation in reverse, i.e. in the mode (^,^,?), the tag is removed from each item in the incoming stream and used to route the item to either the first or second argument stream.

5. BOUNDED BUFFERS

In an implementation of stream communication between processes a set of bindings $y/s.y'$, $y'/s'.y''$, ..., $y^{(n)}/s^{(n)}.y^{(n+1)}$ generated by a producer and not yet consumed by the consumer would be held in a buffer



y channel buffer

By default the buffer is unbounded. To specify a fixed sized buffer we attach a positive integer to the '^' annotation on the channel variable. For example, writing y^10 puts a buffer size of 10 on the y communication channel.

In the communication between processes there is now an extra case to consider. For an unbounded buffer channel a clause was either a candidate, input suspended or a non-candidate. Now a clause can also be output suspended. This occurs when the (partial) output substitution that would be generated by the use of the clause binds an output variable y to a term s.y' and some consumer for y cannot take s without overflowing its buffer.

Just as the non-availability of input on a channel can affect the evaluation path, the inability to transmit output down a particular channel can now affect the evaluation path. In CSP terms [Hoare 1978] we have both read guards and write guards on evaluation steps. A process can also become suspended because of its inability to write. The suspension occurs when there is at least one clause that is output suspended and there are no candidate clauses.

With an unbounded buffer the process that generates output for the channel simply transmits the data down the channel. The data is guaranteed to be accepted by each consumer. For a bounded buffer channel the use of a clause that generates output is conditional upon the output being accepted. To determine acceptability of the output, that is to determine whether the clause in question is a candidate, the producer must poll each consumer of the channel to see if it can accept a message. If all consumers can accept the message the clause can be used and the message can be sent.

Synchronized communication

As a special case of a bounded buffer, we allow a zero size buffer. This gives us tight synchronization of processes. When two processes are connected by a zero-buffer channel, communication can occur only when the producer evaluation step that writes down the channel is synchronized with each consumer evaluation step that reads from the channel. The languages of both Hoare [1978] and Milne and Milner [1979] are designed specifically for this kind of communication.

Our next example illustrates both finite-buffer and zero-buffer communication. Suppose we have a computer system with two printers and a process 'files' generating a stream of files to be printed. Our 'merge-to' relation can be used in its splitting mode

```
mode <^,^> merge-to ?
```

to distribute files between the two printers:

```
: files(z^20) // <x^0,y^0> merge-to z //
  printer(x) // printer(y)
```

printer(x) is the relation: x is a list of files. It has the side effect of printing these files on a particular printer. The logic of the program guarantees that each file in z will be printed exactly once.

The buffer size of 20 on channel z allows a queue of 20 files to accumulate before the 'files' process suspends. Each file in z is allocated to a printer only when one of the printers is ready to accept it, since until then both recursive 'merge-to' clauses are output suspended. Thus, the goal sequence annotated as above gives a single queue, multiple server spooling system. A multiple queue system with the same capacity would result from

```
: files(z^0) // <x^10,y^10> merge-to z //
  printer(x) // printer(y)
```

in which each printer has its own input buffer of size 10.

Buffers as processes

Zero buffer communication is the primitive in terms of which both bounded and unbounded buffer communication can be implemented. If we have an unbounded buffer b then the relationship buffer(b,x,y) between b, the list x of all the input messages to come and the list y of all the subsequent output messages is satisfied when y is b<>x (i.e. b appended to x). The following clauses describe this relation and in so doing implement an unbounded buffer between channels x and y.

```
mode buffer(?,?,^)
buffer(b,u.x,y) <-
            b' = b <> u.Nil & buffer(b',x,y)
buffer(u.b,x,u.y) <- buffer(b,x,y)
```

The goal sequence

```
: prod(x^0) // buffer(Nil,x,z^0) // con(z)
```

gives exactly the same behaviour as

```
: prod(x^) // con(x)
```

By adding the guard length(b) < K to the first clause of the program we define a bounded buffer of size K. In a similar way, we can define a process that acts as a register.

6. CONCLUDING REMARKS

There are several extensions to the above model of parallel computation that we are investigating. One is the use of multiple producers. These multiple producers can either compete or co-operate. If they compete, the shared variable is a two-way channel. The first producer to generate the next message sends it down the channel. It must be accepted by the co-producers. If the producers co-operate, they share a communication channel to the consumer processes. Each message sent must be agreed by each producer, but a producer can generate a "don't care" message represented by an unbound variable.

We also need to address the issues of correctness and termination. Correctness is relatively straightforward. We can use the standard techniques for logic programs (see [Clark 1979]). Termination is more of a problem. For sequential programs we need to show that for each allowed mode of call there is a terminating computation no matter what candidate clause is used. For parallel programs we are investigating conditions on the communication network that will guarantee termination of an unbounded buffer parallel evaluation if the sequential evaluation terminates. Loop network programs, such as that for the Hamming problem, must be handled differently. One must reason about the flow of messages around the network. Thus, for the Hamming program, one must show that the 'seed' of a single value on the incoming arcs of the three 'times' processes is sufficient to ensure that a new seed is generated on each of these arcs. Finally, bounded buffers and synchronized communication introduce new complexities. But proofs of termination, or proofs of deadlock-free evaluation, should not be harder than such proofs for CSP programs.

Related work

MacQueen's paper [MacQueen 1979] is an excellent survey of various models of parallel computation. Many of these correspond to special cases of our model.

The use of Horn clause logic as a parallel programming notation has been investigated by van Emden, de Lucena and Silva [1981]. However, they only treat the deterministic parallelism of Kahn and MacQueen's model. Dausmann, Persch and Winterstein [1979] use an annotation '/B' on shared variables to synchronize processes: an occurrence of 'x/B' delays the process until x is bound.

IC-Prolog [Clark, McCabe and Gregory 1981] is a Horn clause evaluator with a pseudo-parallel execution mode. It time-shares the evaluations of the different processes. Annotations on shared variables are also used to impose communication constraints, and a process is suspended if those constraints cannot be met.

Hogger [1980] discusses various aspects of concurrent logic programming. A backtracking parallel evaluator is assumed in which all shared variables represent two-way channels. He shows how disjunctive concurrency can be obtained by introducing extra shared variables between the parallel calls, and using "head annotations" of IC-Prolog.

ACKNOWLEDGEMENTS

The ideas presented in this paper germinated in discussions with Maurice Bruynooghe and developed in work with Frank McCabe. This work was supported by the UK Science and Engineering Research Council.

REFERENCES

Clark K.L. [1979], Predicate logic as a computational formalism. Draft monograph, Dept. of Computing, Imperial College, London.

Clark K.L., McCabe F.G. and Gregory S. [1981], IC-Prolog reference manual. Research report, Dept. of Computing, Imperial College, London.

Dausmann M., Persch G. and Winterstein G. [1979], Concurrent logic. In Proc. 4th Workshop on AI, Bad Honnef.

Dijkstra E.W. [1976], A Discipline of Programming. Prentice-Hall, Englewood Cliffs, NJ.

van Emden M.H., de Lucena G.J. and Silva H. de M. [1981], Predicate logic as a language for parallel programming. Research report, Dept. of Computer Science, Univ. of Waterloo, Ontario.

Friedman D.P. and Wise D.S. [1980], An indeterminate constructor for applicative programming. In Record 7th ACM Symp. on Principles of Programming Languages, 245-250.

Hoare C.A.R. [1978], Communicating sequential processes. Comm. ACM 21, 8 (August 1978), 666-677.

Hogger C.J. [1980], Logic representation of a concurrent algorithm. Research report, Dept. of Civil Engineering, Imperial College, London.

Kahn G. and MacQueen D.B. [1977], Coroutines and networks of parallel processes. In Proc. IFIP Congress 77, North Holland, Amsterdam, 993-998.

Kowalski R.A. [1974], Predicate logic as programming language. In Proc. IFIP Congress 74, North Holland, Amsterdam, 569-574.

Kowalski R.A. [1979], Logic for Problem Solving. North Holland, New York.

MacQueen D.B. [1979], Models for distributed computing. Rapport de Recherche 351, INRIA, France.

Milne G. and Milner R. [1979], Concurrent processes and their syntax. J. ACM 26, 2 (April 1979), 302-321.

Robinson J.A. [1965], A machine oriented logic based on the resolution principle. J. ACM 12, 1 (January 1965), 23-41.

Warren D.H.D. [1977], Implementing Prolog - compiling predicate logic programs. Research reports 39, 40, Dept. of AI, Edinburgh Univ.