QUEEN MARY
AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

Department of Computer Science

Research Report No. RR-02-03 ISSN 1470-5559 October 2002

Coordination using a Single-Writer Multiple-Reader Concurrent Logic Language

Matthew Huntbach

Coordination using a Single-Writer Multiple-Reader Concurrent Logic Language

Matthew Huntbach

Department of Computer Science, Queen Mary University of London
mmh@dcs.qmul.ac.uk

Abstract The principle behind concurrent logic programming is a set of processes which co-operate in monotonically constraining a global set of variables to particular values. Each process will have access to only some of the variables, and a process may bind a variable to a tuple containing further variables which may be bound later by other processes. This is a suitable model for a coordination language. In this paper we describe a type system which ensures the co-operation principle is never breached, and which makes clear through syntax the pattern of data flow in a concurrent logic program. This overcomes problems previously associated with the practical use of concurrent logic languages.

23 October 2002

Introduction

Concurrent logic programming was once considered one of the leading models for coordination and open systems (see for example, [Ka&Mi 88]), but in recent years has been overshadowed by other approaches, particularly those originating from Linda. This paper serves three purposes. Firstly to introduce concurrent logic programming to those who may be unfamiliar with it, or who may share the common misconception that it is "parallel Prolog". Secondly to show how concurrent logic programming may be considered a coordination language. Thirdly, to give a simple type system for it involving modes and linearity which ensures all variables in a concurrent logic program have a single writer which can be identified from the syntax. This type system and new clearer syntax overcome many of the problems previously associated with concurrent logic programming.

Logic Programming

Logic programming, as defined by Kowalski [Kowa 74], has the basis that a set of clauses in predicate logic of the form

$H \leftarrow B_1, \ldots, B_n \qquad n \geq 0$

can be viewed as a program. Here H and each B_i consists of a predicate name and an argument list, and an argument is a variable name, a constant or a tuple of further arguments. Execution of a program starts with an initial set of goals, $A_1,\ldots,A_m \quad m \geq 1$, using the single computation rule, known as *resolution*, that at any time the set of goals

$A_1,\ldots, A_{i-1}, A_i, A_{i+1},\ldots,A_p$

can be rewritten to

$(A_1, \ldots, A_{i-1}, B_1, \ldots, B_n, A_{i+1}, \ldots, A_p)\theta$

given the existence of a clause $H \leftarrow B_1, \ldots, B_n$ where θ is the most general unifier between A_i and H (the minimal set of variable substitutions which makes A_i and H identical). Each time a clause is used in this way, its variables are renamed to fresh ones not already used in the set of goals, so "variable capture" is avoided. Execution continues either until the set of goals is reduced to the empty set (success), or the situation is reached where there is no possible application of the resolution rule (failure).

This model of computation was used for the programming language Prolog which has had a successful, albeit niche, existence since the 1970s. Prolog works by specialising the above model, so that only the current leftmost goal, A_1, can ever be rewritten by resolution. It was Kowalski's intention however [Kowa 79] that more sophisticated control mechanisms could be employed. Continually rewriting the goals at the left of the list until the set of goals derived from A_1 becomes empty could be considered execution of the procedure call A_1. However, if before the set of goals derived from A_1 becomes empty, a goal outside that set, A_k, is rewritten, then execution can be considered as co-routining between A_1 and A_k. Since the resolution rule could be applied to several goals simultaneously, the possibility of parallel execution exists.

Concurrent Logic Programming

The logic programming model above has no concept of a fixed input/output pattern for procedures. It makes no distinction between testing whether a variable has a particular value and assigning it a value. Testing occurs when matching a constant in a goal against a constant in a clause head H, assignment occurs when matching a variable in a goal against a constant in a clause head. In the case of testing, if the two constants are not equal, resolution fails and another clause has to be tried. In the case of assignment, adding the substitution of the variable for the constant against which it is matched to the global substitution communicates to other goals which share that variable.

When attempts were made to parallelise the logic programming model, it was discovered that Prolog's sequential left-to-right resolution of the goals was more fundamental to its success as a practical language than first assumed. If two goals, A_i and A_j, $i<j$, shared a variable, the occurrence of a constant in that variable's place in a clause head for A_i would be an assignment occurrence, while a constant in the variable's place in a clause head for A_j would be a testing occurrence. Breaking Prolog's guarantee that A_i was resolved before A_j meant what was intended to be a test could become an assignment, termed by Ringwood the *premature binding problem* [Hu&Ri 99]. A prematurely bound variable in a parallel processing situation could cause a whole computation to proceed with the wrong argument. To overcome this, it was proposed that in concurrent logic programming, the resolution rule would only be applied when a unification θ of a goal A and a clause head H could be found where $H\theta = A$ (in the place of Prolog's $H\theta = A\theta$).

With this restriction, resolution only passes data into the code for a procedure, so it is a form of pattern matching similar to that seen in functional languages. The rule is that with a clause $H \leftarrow B_1, \ldots, B_n$ if there is a θ such that $H\theta = A_i$ then $A_1,\ldots,A_{i-1},A_i,A_{i+1},\ldots,A_p$ becomes $A_1,\ldots,A_{i-1},(B_1, \ldots, B_n)\theta,A_{i+1},\ldots,A_p$. However, in order for data to be passed out of a procedure, an implicit assignment is needed, Any B_i in a clause may take the form $X = v$, where X is a variable name and v a value, and then $A_1,\ldots,A_{i-1},X=v_i,A_{i+1},\ldots,A_p$ rewrites to $[v/X](A_1,\ldots,A_{i-1},A_{i+1},\ldots,A_p)$. At any point where matching a goal A_j against clause head H results in a variable from A_j being matched against a constant or tuple from H, the attempt to rewrite A_j is suspended until an assignment to one of the variables in A_j causes the match to become a constant-to-constant match.

A further restriction on the logic programming model involves the abandonment of the *backtracking* mechanism which is a fundamental aspect of Prolog. In Prolog, if at any point in the execution there is no clause with head H such that there is a unification θ where $H\theta = A\theta$, the global state of computation reverts to the point before the previous resolution was applied. Then resolution using a clause different from that originally applied takes place, or further backtracking if there is no such clause. In concurrent computation if one rewrite causes an assignment which results in several further rewrites, possibly distributed in some way, taking place, a global backtracking of the whole computation to the state before the original rewrite may be impractical. The effective use of backtracking in Prolog turned out to be dependent on Prolog's execution being a single sequence of steps. In a coordination context, if assigning a value to a variable sends an external signal, withdrawing that value in backtracking may not make sense, in other words, you cannot backtrack over real time.

The languages which developed an efficient specialisation of the logic programming model in this way became known as the *committed choice* logic languages due to application of resolution being a commitment rather than something reversible through backtracking. A more detailed description of their development and its rationale can be found in [Hu&Ri 99].

The abandonment of features that were fundamental to Prolog, described by Tick as a "de-evolution" [Tick 95], was a process which involved a plethora of languages, as described in Shapiro's survey [Shap 89], as intermediate forms attempting to hold onto some of the features of Prolog were proposed. However, as noted by Ueda [Ueda 99], the weakest form proved surprisingly stable and versatile. Under the name KL1 [Ue&Ch 90] it was described as the most significant contribution of the Japanese Fifth Generation Project [Sh&Wa 93], and it was promoted under the name "Strand" [Fo&Ta 90] as a coordination language [Ca&Ge 92]. In [Hu&Ri 99] it is referred to as *Guarded Definite Clauses* (GDC), and we shall use this name.

An operational semantics for GDC is given by Ueda [Ueda 99]. A semantics in terms of pi calculus is given by Hirata [Hira 95].

Concurrent Logic Programming as a Coordination Language

Around the same time that research on concurrent logic programming was converging on this simple model, the coordination model Linda was launched. Some of the paper which did most to launch it [Ca&Ge 89] makes odd reading now, it lists concurrent logic programming as one of the "big three" parallel programming paradigms, against which Linda was a small overlooked competitor: very much the reverse of the current situation. The strange death of concurrent logic programming can be attributed to a number of factors, some more sociological than technical. Some of these are discussed in [Sh&Wa 93] and [Ueda 99]. Although the model was very different from Prolog, it continued to be classed by many as a "parallel Prolog". It was thus often dismissed as suitable only for those applications where Prolog had obtained a usage or seen in terms of what it lost from general logic programming compared to Prolog rather than what it added. The range of minor variations produced in the de-evolution made the paradigm look far more complex than it really was. In the field of declarative programming languages, widespread agreement on Haskell [HPJW 92] as a common standard revived interest in functional languages which unlike most forms of logic programming offered the power of higher order programming. The verbosity of the Prolog-like syntax was also a big problem [Cian 92]. Perhaps the major problem, however, was that the syntax did not make clear the location of binding of any variable [YKS 90], and hence the pattern of data flow.

Strand was promoted as a coordination language without reference to its logic programming background, and developed further into PCN [FOT 92], losing Strand's Prolog-like syntax, but maintaining the same computation model underneath. The features which make this model suitable for coordination are described in a further paper [Fo&Ta 94]. The concurrent interleaving of code execution is one such feature. The only restriction which causes any necessary ordering in computation is the flow of data from assignment to matching against clause heads. The monotonicity of variable assignment is another powerful feature for concurrency. Many of the problems of concurrency over shared memory are due to a computation accessing a variable whose value may be simultaneously changed by another. Logic variables are useful for concurrent and distributed programming [Hari 99] because once they are assigned a value they keep that value, there is no reassignment, nor even change in value through backtracking in concurrent logic programming. A third feature is that concurrent logic programming is indeterminate, enabling it to coordinate events which may be happening together with an unpredictable ordering. A fourth feature is the ability to separate code for the different components of a concurrent logic program, due to its declarative nature. A goal in a concurrent logic program has a simple and well-defined interaction with other goals only by shared variables. So long as the external behaviour of a goal fits in with concurrent logic programming's monotonic binding of variables, its internal code could be anything, it needn't itself be in logic clauses, and it could involve interaction with an external environment.

Concurrent Logic Programming as a Constraint Language

A clearer basis for describing concurrent logic languages, leading to the development of a formal semantics, was given by Maher [Mahe 87] and elaborated by Saraswat et al [SRP 91]. The idea is that there is a global set of constraints, and a set of agents which have access to the set of constraints. The actions on the set of constraints are to *ask* whether a particular constraint is consistent with the set, and to *tell* a constraint, that is to add the constraint to the set. A tell agent adds a constraint c to the global set of constraints, or fails if c is inconsistent with that set. An ask agent takes the form $c \rightarrow A$ where c is a constraint. It converts to agent A if c is entailed by the global set of constraints, it fails if the negation of c is entailed by the global set of constraints, and it is blocked until further constraints are added to the set if neither c nor its negation is entailed. An agent may be a conjunction of agents, or a procedure call. To allow for indeterminacy, agents of the form

$$c_1 \rightarrow A_1 \, c \, c_2 \rightarrow A_2 \, c \ldots c \, c_n \rightarrow A_n \qquad n \geq 1$$

are allowed. In this case, the agent transforms to A_i if c_i is entailed by the global set of constraints, if this is true for a number of values of i, any i may be chosen. The agent fails if the negation of every c_i is entailed by the global constraints otherwise it suspends if no c_i is entailed.

This general framework may be used with any suitable constraint system. For Guarded Definite Clauses, the only constraints that may be told are to constrain a variable from a global set of variables to be equal either to a constant, to another variable, or to a tuple which may contain further variables and constants. Similar constraints may be asked: whether a variable is equal to a particular constant or a tuple. These are represented by pattern matching in GDC. Further ask operations from a limited set can be used, represented by the guard operation in GDC. These include to ask whether two variables have been given values that order them arithmetically, to ask whether a variable has been given a value of a particular type, to ask whether a variable has been given any value. The type asks give *dynamic typing*, since Guarded Definite Clauses, like Prolog is not a statically typed language, though the introduction of a type inference system is not impractical [Ya&Sh 87]. The “any value” ask is equivalent to *wait*, it waits for a variable to be bound. Note the converse *var* or *unknown* guard test [YKS 90] breaks the requirement that constraints are monotonic, since it succeeds when a variable is unbound, but fails after it becomes bound.

Some of the complexities described in [Shap 89] come about because of a desire to maintain *atomic tell*, that is when an agent transforms using $c_i \rightarrow A_i$ any tell operations in A_i are published before any other transformation of the whole computation. However, this is impractical in a distributed implementation and not in practice widely needed. The simplest alternative, in which no guarantee for the timing of tell operations exists (or more specifically when agents suspended on a variable are informed the variable has become bound) is referred to as *eventual tell*, and is used in GDC. Communication in GDC is asynchronous, but Brim et al [BGJK 96] point out that a synchronous variant can be obtained by blocking tell operations until there is a matching ask.

As an example of GDC, here is the code for indeterminate stream merger:

```
merge(cons(Head,Tail),List,Merge) :-
     Merge=cons(Head,Merge1), merge(Tail,List,Merge1).
merge(List,cons(Head,Tail),Merge) :-
     Merge=cons(Head,Merge1), merge(List,Tail,Merge1).
merge(nil,List,Merge) :- Merge=List.
merge(List,nil,Merge) :- Merge=List.
```

Here a list is either the constant `nil`, or a tuple `cons(Head,Tail)` where `Head` and `Tail` are further variables. GDC can use the list notation derived from Prolog but it is not used here to demonstrate tuples more generally. `Merge=cons(Head,Merge1)` sets the variable `Merge` to the tuple `cons(Head,Merge1)`. The example shows how an ask may wait for a variable to be bound to a tuple, but need not wait for variables in the tuple to be bound. A goal `merge(List1,List2,Merge)` will transform as soon as either variable `List1` or `List2` becomes bound to a tuple `cons(H,T)` without requiring `H` or `T` to be bound. Similarly, the tell of the tuple `cons(Head,Merge1)` to variable `Merge` in the clause bodies does not require `Head` or `Merge1` to have been given values. If both `List1` and `List2` have been constrained to hold values then there are two possible clauses that may be used and it is indeterminate which. However, if an attempt is made to resolve the goal when either `List1` or `List2` have been constrained, there is no necessity to wait for the other to be constrained. Hence this is stream merging because `H` is sent on stream `S` by assigning `cons(H,T)` to `S`, `T` could be later bound to send another value on the stream, and with two streams being merged whichever gets a value sent first gets that value sent on the merged stream.

Consider now another move from Prolog syntax, replacing pattern matching by explicit asks:

```
merge(list1,list2,merge)
{
 list1=cons(head,tail) ||
    merge=cons(head,merge1), merge(tail,list2,merge1);
 list2=cons(head,tail) ||
    merge=cons(head,merge1), merge(list1,tail,merge1);
 list1=nil || merge<-list2;
 list2=nil || merge<-list1
}
```

We feel this style removes some of the verbosity of concurrent logic programming, that becomes evident when processes have large numbers of arguments. A similar step was part of the transformation of Strand to PCN [FOT 92]. Note we have also moved from GDC's Prolog-inherited need to initialise variable names with capital letters. This necessitates a distinction between the tell operation `merge<-list2`, which sets the variable `merge` to `list2` and `merge=list2`, which would set the variable `merge` to the constant `'list2'`. The double bar operator separates asks from clause bodies, we use a semicolon as the clause separator rather than GDC's period, but it does not indicate any change in semantics.

Modes in Concurrent Logic Programming

One of the most curious hangovers from Prolog in the concurrent logic languages is the lack of a syntactic distinction between variables used for input and variables used for output. In Prolog it is common for predicates to be multimodal, but this is almost never the case in the concurrent logic languages [Ueda 01]. Values are constructed cooperatively rather than competitively in concurrent logic programming, meaning that while one agent may constrain a variable to a tuple, and leave another agent to constrain variables in the tuple, it is very unusual for two separate agents both to attempt to constrain a single variable. If it were to happen, it would cause failure (unless both constrained the variable to the same value). Failure is a normal part of Prolog execution, leading to backtracking, but is abnormal in GDC programming, and would lead to catastrophic collapse of a whole system if used in a coordination context.

So almost all GDC programs are well moded, meaning that at any one time there is exactly one agent that may write to any variable (if no agents may write to a variable, it would result in permanent suspension). Extensive work has been done on mode analysis of GDC programs, with the assignment of input/output modes to all variables leading to more efficient execution, and the detection of ill-moded variables (with zero or more than one writer) used for automated debugging [AUC 98]. However, this analysis would not be needed if programs were written in a way that guaranteed all variables had a single writer. Furthermore, if it were syntactically clear which agent was the writer of any particular variable, programs would be much easier to understand. As it is put in [YKS 90] a common question frustrated concurrent logic programmers ask is “who the hell instantiated this variable?”. A system in which all variables were given a mode indicated by the syntax would mean that question could be easily answered. It would also remove the possibility of program failure caused by two agents trying to bind a variable to different values.

A simple mode system works as follows. The arguments to a call of a procedure are divided into those to be used for reading and those for writing. This is done in a heading to the procedure. An ask may not be used to ask the value of any argument indicated as used for writing. Each clause must have exactly one write occurrence for each write argument to the procedure, where a write occurrence is either a tell which constrains the value of that variable or the occurrence of the variable in a write position in a procedure call. A clause may have any number of read occurrences of any argument to the procedure, where a read occurrence is the variable occurring in the read position of a procedure call, or as an argument to a tuple in a tell which constrains another variable. An argument indicated as used for reading may not be used in any write occurrence.

There are two forms of local variable, which must have different names from the procedure arguments. A variable local to a clause occurs as an argument in the tuple of an ask. Its value may itself be asked in the same clause, and it may only be used in read occurrences in the body of the clause (the part to the right of the **||**). A variable local to the body of a clause must appear in exactly one write occurrence, and can have any number of read occurrences.

Checking that the above conditions hold is done at compile time, code is rejected as invalid if they don't. This ensures that every variable has exactly one writer and any number of readers.

The annotation we suggest uses `->` to separate read and write arguments, making our stream merger procedure look like:

```
merge(list1,list2)->merge
{
 list1=cons(head,tail) ||
     merge=cons(head,merge1), merge(tail,list2)->merge1;
 list2=cons(head,tail) ||
    merge=cons(head,merge1), merge(list1,tail)->merge1;
 list1=nil || merge<-list2;
 list2=nil || merge<-list1
}
```

Here, `list1` and `list2` are read arguments to the procedure, `merge` is a write argument to the procedure. In the first clause, `head` and `tail` are local variables to the clause, while `merge1` is a local variable to the clause body.

Back-communication

The above leads to a simple functional-like language, in fact we have considered syntactic sugar not described here which gives a more functional appearance, and cuts down further on the verbosity [Hunt 00]. Unlike conventional functional programming, it can handle indeterminacy. However, procedure names are not first-class citizens of the language, neither is there any way of returning a procedure as a result. So it lacks functional programming's higher order abilities. But it also lacks a crucial aspect of concurrent logic programming, *back-communication* [Greg 87]. Back-communication means that an agent may bind a variable to a tuple containing unbound variables, and other agents may bind those variables. In the ask-tell formulation, it means that an agent may pass tell access to part of a value to another agent which has ask access on that value.

As an example, `w=t(u,v)` in a clause body is a tell constraining a variable to hold a tuple value with functor `t` and arguments `u` and `v`. It counts as a read occurrence of `u` and `v`. Thus `u` and `v` must either be input arguments to the procedure, further inputs local to the clause as arguments to a tuple in an ask, or local to the clause body so having write occurrence in the clause. The ask `w=t(u,v)` gives a clause body read access to `u` and `v`, but no write access.

What we want, however, is to allow a tell of the form `w=t(u)->v`. This would pass write access to `v` to whichever agent has read access to `w`, and would count in the clause body as a write occurrence of `v`. We then need a matching ask of the form `w=t(u)->v`. Then `v` is treated in the clause which has this ask as if it were a write argument, it must have exactly one write occurrence in the body of the clause, but can have any number of read occurrences.

The problem with this is that our variables have a single writer but possibly several readers. As indicated above, a variable may be in more than one read occurrence in a clause body. If it were assigned a tuple containing back communication, each read occurrence would create a write occurrence of the back communication variable. The concurrent logic programming language Janus [SKL 90] uses a mode system similar to ours, but overcomes the problem of back-communication breaking the single writer property by insisting that all variables have just a single reader as well. Variables which have this property are referred to as *linear*. In our terminology, Janus's restrictions would mean that all variables must have just a single read occurrence in a clause body. An exception is made if the guard of the clause has an ask which means the variable cannot contain unassigned variables, for example, an arithmetic test ask means the clause will not be used until the variable has been bound to a number.

We believe Janus's insistence on linearity for all variables is too restrictive. Being able to use a variable before it is bound is an important part of logic programming, enabling us to build data structures with "holes" which are later filled in. A logic variable used as an argument or incorporated into structure concurrently with another process computing a value for it is equivalent to a *future* [Hals 85] in functional programming.

In order to maintain both back-communication and the use of logic variables as futures, we extend the system of mode types with linearity types. Arguments to procedures and tuples must be declared not only as either read or write, but also linear or non-linear. Non-linear variables are treated as we describe the mode system above. But linear variables have additional restrictions. A linear read argument variable must be used exactly once in each clause: either in an ask which asks whether it has been constrained to a particular tuple, or in a read occurrence. A linear write argument variable must be used exactly once in each clause in a write occurrence. A linear variable local to a clause may be introduced in an ask, in which case it must be used exactly once, treated as either an input or output linear variable as indicated by its position to the left or right of the `->` in the tuple of the ask. A linear variable local to a clause body must have exactly one read occurrence and one write occurrence in the clause body. Only linear variables may be assigned tuples containing back communication or further linear variable arguments. Attempting to match a linear variable against a non-linear variable in an ask will always fail.

We have already mentioned how concurrent logic programs, unlike Prolog programs, should never fail. The mode system we propose eliminates one way failure can occur: a single variable bound to conflicting values due to multiple writers. Failure due to lack of any matching clause can be overcome by succeeding but binding arguments to which the otherwise failing goal has write access to special values indicating an exception. With a linear read variable, once it has become bound, any write or further linear read arguments in it should be bound in this way. This could also be used to deal with failure due to broken links in a distributed system, and other reasons external to GDC.

Examples: Simulating Higher Order Programming, Dynamic Communication, and the Dining Philosophers

We now consider some examples. In these examples, we adopt the convention that linear variables are indicated by initial capital letters. The first shows how back-communication can give a higher-order programming effect as in functional languages. We represent a function by a procedure which takes a single linear variable as a read argument. The effect of applying the function is given by binding this variable to `apply(arg,Cont)->res` where `arg` is the function's argument, and `Cont` is used for the next application of the function, the reply variable `res` will then be bound to the result. For example, the following procedure represents the square function:

```
square(Func)
{
 Func=apply(arg,Cont)->res || res<-arg*arg, square(Cont);
 Func=halt ||
}
```

Below is a procedure representing a map operation so that the conjunction of calls

```
square(Func), map(list)->(Func,result)
```

causes `result` to be bound to the list whose contents are the squares of the numbers, in the same order, as the contents of the list to which variable `list` is bound:

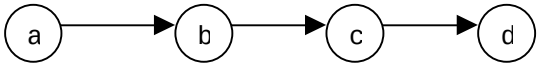
```
map(list)->(Func,result)
{
 list=cons(head,tail) ||
     Func=apply(head,Func1)->head1,
     map(tail)->(Func1,tail1),
     result=cons(head1,tail1);
 list=nil || Func=halt, result=nil
}
```

We could have procedures of similar construction to `square` to provide any function, so that `map` is higher order. The procedure `map` would be written in a different way, however, if we wanted it to be useable as the argument to a higher-order procedure. The `Func` argument is given as a write argument because `map` binds it to give the effect of function application. Again, we can use syntactic sugar to give code that closer resembles conventional functional programming.

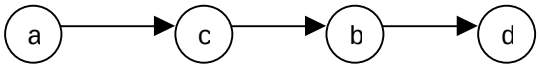
Our next example shows how communication links can be altered dynamically. The following conjunction of calls:

```
a(aargs)->S1, b(bargs,S1)->S2, c(cargs,S2)->S3, d(dargs,S3)
```

can be considered as representing a situation in which `a` communicates with `b` through `S1`, `b` communicates with `c` through `S2`, and `c` communicates with `d` through `S3`. Diagrammatically, this can be shown as:



Suppose we wish to change the ordering of communication, so that `b` and `c` are reversed, giving:



This can be achieved by the clauses below, initiated by binding `aargs` to `rev(aargs1)`:

```
a(aargs)->Out
{
 aargs=rev(aargs1) || Out=rev(C1), a(aargs1)->C1;
}

b(bargs,In)->Out
{
 In=rev(C1) || Out=rev(C1,C2)->C3, b(bargs,C3)->C2;
}

c(cargs,In)->Out
{
 In=rev(C1,C2)->C3 || c(cargs,C1)->C3, Out<-C2;
}
```

As a final example, the Dining Philosophers is often used to illustrate coordination issues. It was used, for example, in the paper which introduced Linda [Ca&Ge 89], and compared there with one particular representation in GDC [Ring 88]. That representation was rather complex, and was not the only way the problem could be tackled in GDC. We give another representation below, using our mode and linearity notation:

```
fork(One,Two)
{
 One=up(One1)->ok || ok=yes, upFork(One1,Two);
 Two=up(Two1)->ok || ok=yes, upFork(Two1,One)
}

upFork(Up,Down)
{
 Up=down(Up1) || fork(Up1,Down)
}

place(Phil)->(Left,Right)
{
 Phil=hungry(Phil1)->ok ||
     Left=up(Left1)->okleft, Right=up(Right1)->okright,
     waitingPlace(okleft,okright,Phil1)->(Left1,Right1,ok)
}

waitingPlace(okleft,okright,Phil)->(Left,Right,ok)
{
 okleft=yes, okright=yes ||
     ok=yes, eatingPlace(Phil)->(Left,Right)
}
```

```
eatingPlace(Phil)->(Left,Right)
{
 Phil=eaten(Phil1) ||
     Left=down(Left1), Right=down(Right1),
     place(Phil1)->(Left1,Right1)
}
```

We haven't included the modifications required to avoid deadlock (restrict the number of philosophers attempting to eat to a maximum of four to prevent the possibility of all five simultaneously taking up one fork), but that would not be difficult to add.

The standard dining philosophers set-up is initiated by:

```
place(Phil1)->(Rfork5,Lfork1), place(Phil2)->(Rfork1,Lfork2),
place(Phil3)->(Rfork2,Lfork3), place(Phil4)->(Rfork3,Lfork4),
place(Phil5)->(Rfork4,Lfork5), fork(Lfork1,Rfork1),
fork(Lfork2,Rfork2), fork(Lfork3,Rfork3),
fork(Lfork4,Rfork4), fork(Lfork5,Rfork5), phil()->Phil1,
phil()->Phil2, phil()->Phil3, phil()->Phil4, phil()->Phil5
```

Code for the procedure `phil` was not given above since this could be used for coordinating processes which are not themselves described in GDC. All that is needed is a front-end that is compatible with GDC. Having write access to a linear variable `Phil`, a `phil` process must bind it to a tuple `hungry(Phil1)->ok`, that is a tuple with two arguments, `Phil1` a linear variable to which it has write access, and `ok` a non-linear variable to which it has read access. After it has detected that `ok` has been bound to `'yes'`, it must bind `Phil1` to `eaten(Phil2)`, and then repeat this with `Phil2` replacing `Phil`. The process will determine for itself the amount of time between bindings, representing philosophers taking arbitrary amounts of time to think and eat. If `phil` were to be coded in GDC, it would be:

```
phil()->Phil
{
 || Phil=hungry(Phil1)->ok, hungryPhil(ok)->Phil1
}

hungryPhil(ok)->Phil1
{
 ok=yes || Phil1=eaten(Phil2), phil()->Phil2
}
```

although the only time delays here would be any caused by the underlying GDC implementation.

A `phil` process can be considered as sending messages to a `place` process on their shared linear variable, asking it to reserve a place with a reply stating that the place has been gained. It then becomes a `hungryPhil` process waiting for that reply. Once that reply has been received it will send a message on the shared linear variable indicating it is relinquishing the place. Note that each message sent has a variable initially unbound to send further messages, thus giving a stream effect. A `place` process similarly sends messages to two `fork` processes, then becomes a `waitingPlace` process waiting for replies to those messages saying the forks have been picked up. It then becomes an

`eatingPlace` process, waiting for a message indicating the philosopher has finished eating, on receiving this it reverts to a `place` process.

The code for `fork` indicates indeterminism, and blocking. It can react to a message from either of its two inputs, but on doing so becomes an `upFork` process which can react only to a message from the source of the previous message, thus blocking reception of messages from the other source. When it receives this second message, it reverts to a `fork` process . This represents a fork on the table which can be picked up from either side, but once picked up can only be put down by the person who picked it up.

Conclusion and Further Work

Guarded Definite Clauses is a model for a programming language that is both declarative and interactive. It is interactive both through its natural concurrency, and through its back-communication, which provides easy co-ordinated communication between concurrent entities. This fits in with modern programming requirements, where interaction is as important as computation [Ca&Ge 92].

We have proposed a type system for Guarded Definite Clauses which restricts it so that any variable can be guaranteed to have exactly one writer. In practice, almost all programs written in GDC languages have this property, but as the syntax does not indicate it, complex mode analysis is required to determine it. Making it apparent through a revised syntax not only avoids the need for such analysis, it makes programs in GDC much easier to understand, since the data flow pattern which was hidden in the Prolog-like syntax becomes apparent. If GDC programs are guaranteed to be well-moded, they can also be implemented with much more efficiency than otherwise [Ue&Mo 94].

In the example code we have given, common patterns can be seen, and we propose further work to develop more extensive syntactic sugar to provide a style of programming which resembles functional programming, but with interactivity a natural part of it. For example, although GDC is a first-order language, we have one example showing how a higher-order function could be simulated, and we could encapsulate the technique used here in a syntactic form which compiles down to the kernel GDC language given here.

GDC makes a suitable model for a coordination language, and a version has already been marketed as such [Fo&Ta 90], because it is not difficult to put a GDC interface on a component and link it in with other components with a similar interface using GDC as a glue. There is a clear communication pattern using only shared variables which can be guaranteed to change only by becoming further partially instantiated.

GDC can be considered a model for a programming language rather than a language because the exact order of computation has been underspecified. While naturally concurrent, it is not necessary that every computation that in principle could be run concurrently has its own thread. An orthogonal notation for mapping GDC computations onto real parallel architectures could be considered, as in PCN [FOT 92]. If parallelisation is more automated, a priority system could be considered, as in KL1 [Ue&Ch 90], and argued for in [Hunt 91]. A more sophisticated form of prioritising computations could involve a form of computational economy [Wald 92].

As noted in [BGJK 96], restricting the application of the eventual tell to a time when it is definitely needed by a matching ask gives a synchronous form of the model. If we restricted applying tell until we knew it wrote to a variable shared by an external component with a GDC interface, or to a variable whose value is asked in a clause guard for such a tell, we would in effect achieve a form of lazy evaluation.

References

[AUC 98] Y.Ajiro, K.Ueda and K.Cho. Error-correcting source code. *Principles and Practice of Constraint Programming - CP98* M. Maher, J.-F. Puget (eds), Springer *LNCS 1520*, 40-54.

[BGJK 96] L.Brim, D.R.Gilbert, J-M.Jacquet, M.Kretinsky, A process algebra for synchronous concurrent constraint programming, *Proc. 5th Int. Conf. on Algebraic and Logic Programming*, Michael Hanus and Mario Rodríguez-Artalejo (eds), Springer *LNCS 1139*, 165-178.

[Ca&Ge 89] N.Carriero and D.Gelernter. Linda in context. *Comm. ACM* 32(4), 444-458.

[Ca&Ge 92] N.Carriero and D.Gelernter. Coordination languages and their significance. *Comm. ACM* 35(2), 97-107.

[Cian 92] P.Ciancarini. Parallel programming with logic languages: a survey. *Computer Languages* 17 (4) 213-239.

[Fo&Ta 90] I.Foster and S.Taylor. *Strand: New Concepts in Parallel Programming*, Prentice Hall.

[FOT 92] I.Foster, R.Olson and S.Tuecke. Productive parallel programming: the PCN approach. *Scientific Programming* 1 (1), 51-66.

[Fo&Ta 94] I.Foster and S.Taylor. A compiler approach to scalable concurrent program design. *ACM Trans. on Programming Languages and Systems* 16 (3) 577-604.

[Greg 87] S.Gregory. *Parallel Logic Programming in PARLOG*. Addison-Wesley.

[Hals 85] R.H.Halstead. Multi-Lisp – a language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems* 7 (4) 501-538.

[Hari 99] S.Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient Logic Variables for Distributed Computing. *ACM Trans. on Programming Languages and Systems* 21 (3) 569-626.

[Hira 95] K.Hirata. *Pi-Calculus Semantics of Moded Flat GHC*, Technical Report ISRL-95-3, NTT Basic Research Labs, Japan.

[HPJW 92] P.Hudak, S.L.Peyton-Jones and P.Wadler (eds). Report on the programming language Haskell. *ACM SIGPLAN Notices* 27 (5).

[Hunt 91] M.M.Huntbach. Parallel branch and bound search in Parlog. *Int J.of Parallel Programming* 20 (4) 299-314.

[Hunt 00] M.Huntbach. The concurrent language Aldwych. *Proc. 1st Int Workshop on Rule-Based Programming*, Montreal Canada,

[Hu&Ri 99] M.M.Huntbach and G.A.Ringwood. *Agent-Oriented Programming: from Prolog to Guarded Definite Clauses*, Springer *LNCS 1630*.

[Ka&Mi 88]. K.M.Kahn and M.S.Miller. Language design and open systems. In *The Ecology of Computation*, B.A.Huberman (ed), Elsevier, 291-312.

[Kowa 74] R.A.Kowalski. Predicate logic as a computational formalism, *Proc IFIP*, North-Holland, 569-574.

[Kowa 79] R.A.Kowalski. Algorithm = Logic + Control, *Comm. ACM* 22 (7), 424-436.
[Mahe 87] M.J.Maher. Logic semantics for a class of committed-choice programs. *Proc. 4th Int. Conf. on Logic Programming*, MIT Press, 858-876.
[Ring 88] G.A.Ringwood. Parlog86 and the dining logicians. *Comm. ACM* 31 (1), 10-25.
[SRP 91] V.A.Saraswat, M.Rinard and P.Panangaden. Semantic foundations of concurrent constraint programming. *Proc. Principles of Programming Languages Conf. (POPL'91)* 333-352.
[Shap 89] E.Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys* 21 (3), 412-510.
[Sh&Wa 93] E.Shapiro and D.H.D.Warren (eds) The Fifth Generation project: personal perspectives. *Comm. ACM* 36 (3), 46-101.
[SKL 90] V.Saraswat, K.Kahn and J.Levy. Janus: a step towards distributed constraint programming. *Proc. North American Logic Programming Conf. 1990*, MIT Press, 431-446.
[Tick 95] E.Tick. The de-evolution of concurrent logic programming languages. *J. Logic Programming* 23 (2), 89-123.
[Ueda 99] K.Ueda. Concurrent logic/constraint programming: the next 10 years. In *The Logic Programming Paradigm: A 25-year Perspective*, K.R.Apt, V.W.Marek, M.Truszczynski and D.S.Warren (eds), Springer.
[Ueda 01] K.Ueda. Resource-passing concurrent programming. *4th International Symposium TACS 2001*,Springer *LNCS 2215*, 85-126,
[Ue&Ch 90] K.Ueda and T.Chikayama. Design of the kernel language for the parallel inference machine. *Computer Journal* 33(6) 494-500.
[Ue&Mo 94] K.Ueda and M.Morita. Message-oriented parallel implementation of moded flat GHC. *New Generation Computing* 13 (1) 3-43.
[Wald 92] C.A.Waldspurger, T.Hogg, B.A.Huberman, J.O.Kephart and W.S.Stornetta. Spawn: a distributed computational economy. *IEEE Trans. on Software Engineering* 18 (2) 103-116.
[YKS 90] E.Yardeni, S.Kliger and E.Shapiro. The languages FCP(:) and FCP(:,?). *New Generation Computing* 7 89-107.
[Ya&Sh 87] E.Yardeni and E.Shapiro. A type system for logic programs. In *Concurrent Prolog: Collected Papers Volume 2* E.Shapiro (ed) MIT Press 211-244.