Proceedings of the 1995 International Conference on Logic Programming

Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems[1]

M. Hermenegildo, D. Cabeza, M. Carro
Computer Science Department
Technical University of Madrid (UPM), Spain
{herme,dcabeza,mcarro}@fi.upm.es

Abstract

Incorporating the possibility of attaching attributes to variables in a logic programming system has been shown to allow the addition of general constraint solving capabilities to it. This approach is very attractive in that by adding a few primitives any logic programming system can be turned into a generic constraint logic programming system in which constraint solving can be user defined, and at source level – an extreme example of the "glass box" approach. In this paper we propose a different and novel use for the concept of attributed variables: developing a generic parallel/concurrent (constraint) logic programming system, using the same "glass box" flavor. We argue that a system which implements attributed variables and a few additional primitives can be easily customized at source level to implement many of the languages and execution models of parallelism and concurrency currently proposed, in both shared memory and distributed systems. We illustrate this through examples and report on an implementation of our ideas.

Keywords: Implementation Techniques, Concurrency, Parallelism, Logic Programming, Attributed Variables, Generic Implementations.

1 Introduction

A number of concepts and implementation techniques have been recently introduced which allow extending unification in logic languages in a flexible and user-accessible way. One example is that of *meta-structures*, introduced by Neumerkel [22], which allow the specification by the user of how unification should behave when certain types of terms, called meta-structures and marked as such by the user, are accessed during unification. More or less at the same time, the data type *attributed variable* was introduced by Le Houitouze [18] with the purpose of implementing various memory management optimizations. Although the behavior of attributed variables during unification was not specified in this work, a number of applications were proposed including the implementation of delayed computations, reversible modification of terms, and variable typing. Earlier, Carlsson [5] used a data type called *suspension*, which was incorporated into SICStus Prolog [6] for the implementation of coroutining facilities [10]. "Attributed variables" and "suspension variables" are essentially the same objects. Le Houitouze's contribution was to put some emphasis on the data type as such and on memory management. He also used attributed

[1]The work described in this paper is supported in part by ESPRIT project "ACCLAIM", and by CICYT contract TIC93-0737-C02-01. The authors would like to thank C. Holzbaur and the anonymous referees for useful discussions in the context of the paper.

variables as a low level primitive for the implementation of mechanisms that necessitated the specification of the behavior of the data type during unification.

A refined version of the concept of meta-structures and attributed variables was used by Holzbaur in [16] for the specification and implementation of a variety of instances of the general CLP scheme [19]. By enhancing the SICStus Prolog system with attributed variables a *generic* system is provided which is basically a SICStus Prolog "clone" where the unification mechanism has been changed in such a way that the user may introduce interpreted terms and specify their unification through Prolog predicates. This approach is very attractive in that it shows that by adding a few primitives any logic programming system can be turned into a generic constraint logic programming system in which constraint solving can be user defined, at the source level – an extreme example of the "glass box" approach. Another system which implements constraint solving using similar techniques is the ECL^{i}PS^e system developed at ECRC [11].

While this approach in principle has drawbacks from the performance point of view (when fully interpreted up to an order of magnitude slowdown is possible w.r.t. native CLP systems) the convenience and generality of the approach can make it very worthwhile in many cases. Furthermore, the speed can be easily increased in interesting cases by writing the unification handlers in a lower-level language. The potential for achieving both genericity and reasonable speed is illustrated by the relatively good performance exhibited by the ECL^{i}PS^e system, which has been used in many practical applications.

Inspired by the previously discussed use of attributed variables we propose a different and novel use for such variables in a completely different context: developing *generic* parallel/concurrent (constraint) logic programming systems, using the same "glass box" flavor. Attributed variables have already been used to implement the coroutining (delay) facilities present in many Prolog systems – often what is actually being done is in fact restoring such capabilities after having "cannibalized" the delay mechanism support for implementing the attributed variables. However, we argue that a system which implements both support for attributed variables and a few additional primitives related to concurrency and parallelism can do much more than simply restoring the delay mechanism. In fact, it is our thesis that using the primitives mentioned above it is possible to easily implement many of the languages and execution models of parallelism and concurrency currently proposed. We illustrate this through examples and we discuss how quite complex concurrent languages and parallel execution models can be implemented using only such primitives. Furthermore, we argue that this can be done in a seamless and user-transparent way in both shared memory and distributed systems. Also, one additional advantage of our technique is that it relates and unifies the two main approaches currently used in concurrent logic programming, and which are seen traditionally as unrelated: "shared variable" systems, in which communication among parallel tasks is done through variables, and "distributed" or "blackboard" systems in which communication is done through explicit built-ins which access shared channels or global data areas.

It should be noted that the use that we propose of attributed variables in the implementation of concurrency and parallelism does not necessarily prevent their simultaneous use also for other purposes, such as the original one of constraint solving. Also, note that the approach proposed, although having some similarities, differs from that of "generic objects," recently and independently discussed by the KL1 implementors [8]. The idea in generic objects is to provide an interface at the "C" level for a particular class of extensions. Our approach differs in both the level at which the extensions are made (which is completely at the source level in our case, thus really offering a reflective, glass box approach), and the nature and extent of the extensions proposed, which goes beyond those that are related to supporting KL1.

Space limitations force the presentation to cover only some basic cases and give incomplete descriptions of the implementations. For more details the reader is referred to [14]. Also, the implementations described can be obtained by contacting the authors.

2 Attributed Variables and Related Primitives

We provide a brief introduction to attributed variables. For concreteness, we follow a stylized version of Holzbaur's first implementation of attributed variables in SICStus Prolog. The reader is referred to [16, 17] for more detailed information.

2.1 General Concepts

Attributed variables are variables with an associated "attribute." Attributes are terms which are attached to variables, and which are accessed in a special way during unification and also through special built–in predicates. As far as the rest of a given Prolog implementation is concerned, attributed variables behave like variables. Special treatment for attributed variables does apply mainly during unification: as will be described later, when an attributed variable is to be unified with another attributed variable or some other non-variable term, user-defined predicates specify how this unification has to be performed. The following is a list of typical predicates which provide for the introduction, detection, and manipulation of attributed variables. In general, attributed variable related operations are correctly undone upon backtracking.

- `get_attr(X,C)`: if `X` is an attributed variable, unify the corresponding attribute with `C`, otherwise fail.
- `attach_attr(X,C)`: turn the free variable `X` into an attributed variable with attribute `C`.
- `detach_attr(X)`: remove the attribute from an attributed variable, turning it into a normal variable.

Attributed variables are dealt with specially during unification. Essentially, the different possible cases are handled as follows. A unification between an unbound variable and an attributed variable binds the unbound variable to the attributed variable. When an attributed variable is about to be bound during unification to a non-variable term or another attributed variable, the attributed variable and the value it should be bound to are collected. At the next inference step, the pending attributed variable-value pairs are supplied to user-defined handlers which are defined by the user by means of the following predicates:

- `verify_attr(C,T)`: invoked when an attributed variable with an attribute which unifies with `C` is about to be unified with the non-variable term `T`.
- `combine_attr(C1,C2)`: invoked when two attributed variables with attributes `C1` and `C2` are about to be unified.

Note that the two predicates are not called with the attributed variables involved, but with the corresponding attributes instead. The is done for reasons of simplicity and efficiency (e.g. indexing). Note also, however, that if access to the actual attributed variable is needed the variable itself can be included in the attribute when it is attached. In general, a number of other primitives are often provided which allow pretty printing and dumping of the results in a user–understandable format.

2.2 Attributed Variables And Coroutining – an Example

The following example, due to [17] serves both to illustrate the use of the primitives introduced in the previous section and also to recover the functionality of `freeze/2` [10] since attribute variables are, as mentioned before, most easily implemented in practice by "cannibalizing" an existing implementation of `freeze`:

```
freeze(X, Goal) :-
   attach_attr(V, frozen(V,Goal)),
   X = V.

verify_attr(frozen(V,Goal),Val) :-
   detach_attr(V),
   V = Val,
   call(Goal).
```

```
combine_attr(frozen(V1,G1),
             frozen(V2,G2)) :-
   detach_attr(V1),
   detach_attr(V2),
   V1 = V2,
   attach_attr(V1,
               frozen(V1,(G1,G2))).
```

The call to `attach_attr` ties the term representing the frozen goal to the relevant variable. When the variable is bound the unification routine escapes to the user-defined generic handler `verify_attr` which in turn performs the meta-call. Note the definition of `combine_attr` needed for handling the case where two variables which have frozen goals attached are unified: a conjunction of the goals is attached to the resulting variable.

Note that the *explicit* encoding of delay primitives such as `freeze/2` and their incorporation into the attributed variable handling mechanism is not to be understood as a mere substitute for the original C code. The true motivation for explicit encodings is that it enables the user to freely define the combination and interaction of such delay primitives with other uses of the attributed variables such as the implementation of a constraint solver. Note that such a solver may also itself perform some delaying, for example when dealing with non-linear constraints.

3 Kernel Concurrent Primitives

We now introduce a simple concurrent/parallel language that we call "Kernel &-Prolog" (*K&P*). The purpose of this language is to provide a small set of basic operators which will allow the implementations that we would like to propose. This language is essentially identical to the kernel language used in the shared memory [15] and distributed [13] implementations of the &-Prolog system, but it is described here for the first time.

Essentially, the *K&P* language subsumes Prolog and includes all the attributed variable primitives described in Section 2. In addition, it provides the following operators which provide for creation of processes, assignment of computational resources to them, and synchronization:

- `&/2` – Standard fork/join parallel conjunction operator (the one used, for example, by the &-Prolog parallelizing compiler [4]). It performs a parallel "fork" of the two literals involved and waits for the execution of both literals to finish (i.e. the join). If no processors are available, then the two literals may be executed in the same processor and sequentially, i.e. one after the other. For example, `..., p(X) & q(X), r(X), ...` will fork a task `p(X)` in parallel with `q(X)`. The continuation `r(X)` will wait until both `p(X)` and `q(X)` are completed.[2] The implementation of this primitive at the abstract machine level is well understood [15].

- `&&/2` – "fair" fork/join parallel conjunction operator. It performs a parallel fork of the two literals involved and waits for the execution of both literals to finish (join). A "thread" is assigned to each literal. The execution of the two literals will be interleaved either by executing them on different processors (if they are available) or by multiplexing a single processor. Thus, even if no processors are available, the two literals will be executed with (apparent) simultaneity in a fair way.
- `&/1` – Standard fork operator. It performs a parallel fork of the literal involved. No waiting for its return is performed (unless explicitly expressed using the `wait` primitive – see below). For example, `..., p(X) &, q(X), r(X), ...` will fork a task `p(X)` in parallel with the rest of the computation.
- `&&/1` – "fair" version of the fork operator.
- `&@/2` – "Placement" standard fork operator. It performs a parallel fork of the literal involved, assigning it to a given node. No waiting for its return is involved. If that node is busy, then the literal will eventually be executed in that node when it becomes idle. For example, `..., p(X) &@` *node* `, q(X), ...` will fork the task `p(X)` in parallel with the rest of the computation and assign it to node *node*.[3] The second argument can be a variable. If the variable is instantiated at the time the literal is reached, its value is used to determine its placement. If the variable is unbound at that time, then the goal is not assigned to any particular node and the variable is bound to the node id. of the node that picks up the task, when it does so.
- `&&@/2` – "fair" placement fork operator. It performs a parallel fork of the literal involved, assigning it to a given node and finding (or, if not available, creating) a thread for it in that node.
- `wait(X)`: This primitive suspends the current execution thread until `X` is bound. `X` can also contain a *disjunction* of variables, in which case execution waits for either one of such variables to be bound.
- `lock(X,L)/unlock(L)`: This primitive gets/releases a lock `L` (on the term `X`).

Note that in the discussion above a (parallel) conjunction of literals can always be used in place of a literal, i.e. the expression `..., (a,b) & (c, d & e, f), ...` is supported. Also, note that the "placement" primitives (`&@/2` and `&&@/2`) and the wait/lock primitives are sufficient to express all the other primitives.

In addition to the "placement" operators described above, which can be directly used in distributed environments, the language also provides as base primitives a Linda-like [7] library, and a lower-level Unix socket interface both of which reproduce the functionality of those of SICStus Prolog. In fact, in distributed environments the primitives described above are implemented using the Linda library [13]. However, the Linda interface can also be used directly: there is a server process which handles the blackboard. Prolog client processes can write (using `out/1`), read (using `rd/1`),

[2] Note that the goals do not need in any way to be independent – this is only necessary if certain efficiency properties of the parallel execution are to hold. However, unlike in the source language, in the kernel language care must be taken to lock properly concurrent accesses to shared variables (see locking primitives).

[3] This is implemented by having a private goal stack for each agent, from which other nodes cannot pick work, and putting the goal being scheduled on the private goal stack of the appropriate agent.

and remove (using `in/1`) data (i.e. Prolog terms) to and from the blackboard. If the data is not present on the blackboard, the process suspends until it is available. Alternatively, other primitives (`in_noblock/1` and `rd_noblock/1`) do not suspend if the data is not available – they fail instead and thus allow taking an alternative action if the data is not in the blackboard. The input primitives can also wait on disjunctions of terms.

4 Implementing Concurrent (Constraint) Languages in Distributed Environments

We now describe a concrete application of our ideas. Our objective in this example is to combine the two main approaches currently used in concurrent logic programming, and which are seen traditionally as unrelated: "shared variable" systems, in which communication among parallel tasks is done through logical variables (e.g. Concurrent-Prolog [25], PARLOG [12], GHC [30], Janus [24], AKL [20], Oz [27], etc.), and "distributed" or "blackboard" systems, in which communication is done through explicit built-ins which access shared channels or global data areas (e.g. Multi-Prolog [2], Shared Prolog [3], and Prologs incorporating Linda [7], being one of the most popular Linda implementations the one bundled with SICStus [1]). In order to do that, we will sketch a method for implementing communication through shared variables by means of a blackboard. We assume the availability of the primitives introduced in the first sections. We also assume that we want to implement a simple concurrent (constraint) language which basically has a sequential operator, a parallel operator (which, since we are in a distributed environment could actually mean execution in another node of the net), and "ask" and "tell" unification primitives. The sort of system that we have in mind could perhaps be a local area net, where the nodes are workstations. The incorporation of the sequential operator (to mark goals that should not be "farmed out") and the special marking of "(remote) communication variables" that will be mentioned later is relevant in the environment being considered. Note that it would be extremely inefficient to blindly run a traditional concurrent logic language (creating actual possibly remote tasks for every parallel goal and allowing for all variables to be possibly shared and worked on concurrently by goals in different nodes) in such a distributed environment. A traditional concurrent language can of course be *compiled* to run efficiently in such an environment after granularity analysis [9, 32, 21] — in fact, this can be seen as a source level transformation to a language of the type we are considering.

To implement this language on *K&P* we start by observing that the sequential and parallel operators of the source language map directly into the sequential ("`,`") and `&@` (or `&&@/2`, if fairness is needed) operators of *K&P*. However, while this allows creating remote tasks, it does not by itself implement the communication of values between nodes through shared variables. We propose to do this by placing before the concurrent call a call to a predicate which will attach an attribute to the shared variables marking them as "communication variables". Also, a unique identifier is given to each communication variable. All bindings to these variables are *posted on the blackboard* (using the `out/1` primitive) as *(variable_id,value)* pairs, where if values contain themselves new variables, such variables are represented by their identifiers. Thus, substitutions are represented as explicit mappings. When bound to a communication variable, a non-communication variable is turned into a communication variable. Tell and ask operations on ordinary variables, which are handled in the standard way, are distinguished from tell and ask operations to (remote) communication variables by the fact that the latter have the corresponding attribute attached to them. Thus, tell and ask unifications to communication variables will be handled by the attributed variable unification. A tell will be implemented by actually performing the binding to the variable in the manner explained above using the `out/1` blackboard primitive. An ask will wait until a binding for the variable is posted on the blackboard. This will be commonly implemented using the blocking `rd/1` blackboard primitive, since in general a variable can have multiple readers and thus `in/1` cannot be used. On the other hand, if a threadedness analysis is performed and a variable is determined to have only one producer and one consumer then `in/1` can be used performing on the fly garbage collection on the blackboard.[4] Else, when a remote goal finishes, a call to a tidying-up predicate can be used to erase the entries in the blackboard corresponding to the bindings of variables which are not used as communication variables any more (and are not linked to other active communication variables) and creating the corresponding term in the heap of the process which continues with the execution.

A Concrete Implementation in SICStus Prolog

In order to be more concrete we sketch our implementation of the ideas outlined above in a widely available environment: SICStus Prolog, enhanced with attributed variables, and using the Linda library provided with recent versions of the system. We hope that this detailed presentation of a concrete implementation will clarify the issues that appear in practice when using the techniques proposed.

The implementation of the basic operators such as `&` and `&@` in a Linda based environment is not our current subject but is in any case relatively straightforward (details of a particular implementation, also available by `ftp`, can be found in [13]): a number of Prolog processes running in different network nodes are started as Linda clients and thus share the blackboard, which is accessible through the normal Linda primitives. Goals that are followed by `&` are simply posted on the blackboard. Idle processes are waiting for work to be posted, which they then execute. Goals that are followed by `&@ x` are posted on the blackboard with an identifier that indicates they are meant to be run on a given machine `x`. This allows, for example, the following query to start a "producer" goal in a remote machine "`alba`" and a consumer locally:

```
?- N=10, producer(N,L) &@ alba, consumer(L).
```

As mentioned before, in order to implement communication between nodes through the variable `L` we would like to mark that variable as shared by attaching an attribute to it. In general `L` may be bound to a complex term with intervening variables, and then each such variable has to be marked in turn. On the other hand, in the blackboard implementation we are considering, variables posted to the blackboard lose their identity. Thus, a unique identifier needs to be given to each one. Note that since attribute attachment operations are local to each process, identifying the shared variables and giving them identifiers (which can be done once) is best separated from the action of actually attaching attributes to them (which has to be repeated in each node sharing the variable).

We implement a predicate `var_ids(LVars, Pairs)` which given a set of lexical variables appearing in the forked goal(s) returns the set of intervening run-time variables, assigns a unique identifier to each of them, and returns the information in the form of $(Variable, Id)$ pairs. In our example, a call to this predicate is placed before the call to `producer(L)` as follows:

```
?- var_ids([L],Ps), ( assign_ids(Ps), producer(N,L) ) &@ alba,
                      assign_ids(Ps), consumer(L).
```

[4] This illustrates how the attributed variable approach also allows performing low-level optimizations as source to source transformations.

(this is handled automatically by a simple lexical expansion of the original query). Only one pair would be generated in this case, since `L` is a free variable (of course, this important case can be treated specially, but the general purpose primitive is used for illustration purposes). Note that `assign_ids(Ps)`, defined by

```
assign_ids([]).
assign_ids([(X,Id)|Ps]) :-
   assign_id(X,Id),
   assign_ids(Ps).
```

```
assign_id(X,Id) :-
   attach_attr(X,shv(Id)).
```

does the actual attachment of the attribute `shv(Id)` to each shared variable, and that this is done both in the local and the remote machine (`alba`, in this case). Once suitably marked, all the unifications involving these communication variables are handled through the blackboard. The appropriate handlers are given in the following "blackboard unification" code (recall that `verify_attr/2` is called when an attributed variable is unified with a term, and `combine_attr/2` is called when two attributed variables are bound to each other):

```
verify_attr(shv(Id),Term) :-
   trans_shterm(Term, NewTerm),
   shv_unify('$shv'(Id),NewTerm).
```

```
combine_attr(shv(I1), shv(I2)) :-
   shv_unify('$shv'(I1),'$shv'(I2)).
```

The predicate `trans_shterm/2` transforms a term into its blackboard representation:

```
trans_shterm(X,'$shv'(Id)) :-
   var(X), !,
   ( get_attr(X,shv(Id)) -> true ;
     new_shv_id(Id),
     attach_attr(X,shv(Id)) ).
trans_shterm(Term,NewTerm)  :-
   functor(Term,F,N),
   functor(NewTerm,F,N),
   trans_shterm(N,Term,NewTerm).
```

```
trans_shterm(0,_,_) :- !.
trans_shterm(N,Term,NewTerm) :-
   N > 0,
   arg(N,Term,Arg),
   arg(N,NewTerm,NewArg),
   trans_shterm(Arg,NewArg),
   N1 is N-1,
   trans_shterm(N1,Term,NewTerm).
```

This predicate uses the primitive `new_shv_id/1`, which returns a new shared variable identifier different from any other in any process participating in the computation.

The predicate `shv_unify/2` performs the actual unification of terms that are already in the blackboard (we have left out all explicit locking in the unification for simplicity):

```
shv_unify(A, B) :-
   dereference(A, VA),
   dereference(B, VB),
   shv_unify_values(VA,VB).

dereference(X, V) :-
   X = '$shv'(_),
   sh_get_bind(X,Binding), !,
   dereference(Binding, V).
dereference(V,V).

shv_unify_values(X1,X2) :-
   X1='$shv'(Id1),
   X2='$shv'(Id2),
   ( Id1=Id2 ; sh_bind(X1,X2) ), !.
shv_unify_values(X1,X2) :-
   X1='$shv'(_), !,
   sh_bind(X1,X2).
```

```
shv_unify_values(X1,X2) :-
   X2='$shv'(_), !,
   sh_bind(X2,X1).
shv_unify_values(X1,X2) :-
   functor(X1,F,N),
   functor(X2,F,N),
   shv_unify_args(N, X1, X2).

shv_unify_args(0, _, _) :- !.
shv_unify_args(N, X1, X2) :-
   N > 0,
   arg(N, X1, A1),
   arg(N, X2, A2),
   shv_unify(A1, A2),
   N1 is N-1,
   shv_unify_args(N1, X1, X2).
```

The unification routine uses the following primitive operation, which returns the immediate binding of a shared variable or fails if it does not exist:

```
sh_get_bind(Id,T) :- linda:rd_noblock(shbinding(Id,T)).
```

The following operation is used when writing out a binding for a variable:

```
sh_bind(Id,T) :- linda:out(shbinding(Id,T)).
```

For example, given the query

```
?- var_ids([L],Ps), assign_ids(Ps), producer(3,L).
```

and the following definition of a simple producer, the contents of the blackboard after execution are listed to its right:

```
producer(0,T):- !, T = [].
producer(N,T) :- N>0,
        T = [N|Ns],
        N1 is N-1,
        producer(N1,Ns).
```

```
shbinding($shv(0),[3|$shv(1)])
shbinding($shv(1),[2|$shv(2)])
shbinding($shv(2),[1|$shv(3)])
shbinding($shv(3),[ ])
```

In order to support synchronization some blocking ("ask") primitive has to be provided. For simplicity, we only describe the implementation of the *K&P* `wait` primitive in this context, which is in any case sufficient for most purposes:

```
wait(X) :-
   get_attr(X,shv(Id)), !,
   sh_wait_bind('$shv'(Id),Binding),
   wait_shnonvar(Binding).
wait(_).
```

```
wait_shnonvar(X) :-
   X = '$shv'(_), !,
   sh_wait_bind(X, Binding),
   wait_shnonvar(Binding).
wait_shnonvar(_).
```

The following primitive is used above:

```
sh_wait_bind(Id,T) :- linda:rd(shbinding(Id,T)).
```

A simple stream communication based consumer using these primitives can be constructed as follows:

```
consumer(T) :-
   wait(T),
   consumer_body(T).
```

```
consumer_body([]).
consumer_body([H|T]) :-
   consumer(T).
```

Note that the above producer and consumer can also be seen as the result of a straightforward compilation of the following fragment of GHC code:

```
producer(0,T) :- T = [].
producer(N,T) :-
   N>0 | T = [N|Ns],
         N1 is N-1,
         producer(N1,Ns).
```

```
consumer([]).
consumer([H|T]) :-
       | consumer(T).
```

Of course, an equivalent distributed producer-consumer situation (in which the elements are consumed in the same order as they are produced, as in the program above) can be easily implemented making direct use of the Linda primitives using the query:

```
?- N=10, lproducer(N) &@ alba, lconsumer.
```

and the following program:

```
lproducer(N) :- lproducer(N,1).

lproducer(0,C) :- !,
   linda:out(message(C,end)).
lproducer(N,C) :-
   N>0,
   linda:out(message(C,number(N))),
   N1 is N-1,
   C1 is C+1,
   lproducer(N1,C1).
```

```
lconsumer :- lconsumer(1).

lconsumer(C) :-
   linda:rd(message(C,T)),
   lconsumer_data(T,C).

lconsumer_data(end,_).
lconsumer_data(number(N),C) :-
   C1 is C+1,
   lconsumer(C1).
```

However, arguably this program lacks the elegance of the shared variable communication based program: for example, if we want to run simultaneously several instances of producers and consumers, the generation of new identifiers for the messages must be explicitly encoded. The shared variable communication approach sketched allows in some ways having the best of both worlds or, in any case, being able to choose between them. It certainly provides the expected functionality. Its performance of course depends heavily on the performance of the blackboard implementation. However, this is certainly also the case if the Linda primitives are used directly.

5 Implementing Other Models Using Attributed Variables

Lack of space does not allow elaborating further but we argue that using techniques similar to those that we have proposed it is possible to implement many other parallel and concurrent models at the source level. For example, while and-parallelism can be supported in or-parallel implementations by folding it into or-parallelism, no communication among and-parallel tasks is possible. Our technique could be used to provide this communication, for example by "escaping" shared variable unifications and asserting them to the common database. We believe it is as well quite possible to encode the determinacy driven synchronization and-parallelism of the Andorra-I system [23] in terms of our `wait` primitive and the concurrency operators. We also believe it is quite possible to implement languages with deep guards and/or those based on the Extended Andorra Model [31], such as AKL [20].

For example, one of the most characteristic features of deep guard languages is precisely the behavior of the guards, and one of the main complications in implementing such languages is in implementing the binding rules that operate within such guards. If the Herbrand domain is used, the guard binding rules require in principle that no bindings to external variables be made. Thus, it is necessary to keep track of the level of nesting of guards and assign to each variable the guard level at which it was created. Note that this can be done by assigning to each guard a hierarchical identifier and attaching to each variable such an identifier as (part of) its attribute. Unifications in the program are labeled with the identifier of the guard in which they occur (the level is passed down recursively through an additional argument). Such unifications are handed over to the attributed variable handler which makes computation suspend unless the variable and the binding have the appropriate relative identifiers. The binding rules for domains other than Herbrand can be more complex because they often use the concept of entailment. But note that in the proposed approach all constraint solving would possibly be implemented through attributed variables anyway. Thus, it is not difficult to imagine that a correct entailment check can be written at the source level using the same primitives and `wait`. Some models are more involved: in AKL, for example, there is a notion of local bindings and there is an additional rule controlled by the concept of "stability" (closely related to that of independence) which allows non-deterministic bindings to propagate at "promotion" time. We believe however that there is also potential for the use of attributed variables for the implementation of AKL. For example, promotion rules can also be implemented by updating the identifiers (the attributes) of all the local variables to higher levels.

Another model for parallel execution of Prolog is the DDAS ("Dynamic Dependent And-parallel Scheme") model of K. Shen [26]. In a very simplified form the DDAS model is an extension to (goal level) independent and-parallel models which allows fine grained synchronization of tasks, implementing a form of "dependent" and-parallelism. Parallelism in this model is controlled by means of "Extended Conditional Graph Expressions" (ECGE for short) which are of the form: `(` *conditions* `=>` *goals* `)`. As such, these expressions are identical to those used in standard independent and-parallelism: if the *conditions* hold, then the goals can be executed in parallel, else, they are to be executed sequentially. The main difference is that a new builtin is added, `dep/1`. This builtin can appear as part of the conditions of an ECGE. Its effect is to mark the variable(s) appearing in its argument specially as "shared" or "dependent" variables. This character is in effect during the execution of the goals in the ECGE and disappears after they succeed. Only the leftmost *active* (i.e. non finished) goal in the ECGE (the "producer") is allowed to bind such variables. Other goals which try to bind such variables (the "consumers") must suspend until the variable is bound or they become leftmost (i.e. all the goals to their left have finished). In order to support this model in *K&P* we assume a source to source transformation (using `term_expansion/2`) of ECGEs: an ECGE is turned into a Prolog if-then-else such that if the conditions succeed then execution proceeds in parallel (using the `&/2` operator, which directly encodes the fork-join parallelism implemented by the ECGEs), else it proceeds sequentially. Dependent variables shared by the goals in a ECGE are renamed. The `dep/1` annotation is transformed into a call to a predicate that marks the variables as dependent by attaching attributes to them. Such attributes also encode whether a variable is in a producer or a consumer position. Unification is handled in such a way that bindings to variables whose attribute corresponds to being in the producer position are actually bound. Note that if the variable is being bound to a complex term with variables, these variables also have to be marked as dependent. Bindings to variables whose attribute corresponds to being in a consumer position suspend the execution of the associated process (using `wait/1`). The change from producer to consumer status is implemented as follows: each parallel goal containing a dependent variable (except the last one) is replaced by the sequential conjunction of the goal itself and a call to a predicate which will "pass the token" of being leftmost to the next goal (or short-circuiting the token link if it is an intermediate goal). This predicate also takes care of restoring the connection lost due to the variable renaming.

6 Performance

The objective of the technique presented is achieving a certain functionality through the use of attributed variables, rather than any increase in performance. However, it is still interesting to make some observations regarding the resulting implementations. Table 1 presents some results obtained with the concurrent extension to SICStus Prolog described in Section 4. A set of programs involving producers and consumers was run with each process running in a different workstation (Sun IPC) connected over an Ethernet network. The performance of our implementation of concurrent logic programming is compared with equivalent programs written di-

	shared var.	linda	shared var./linda
Incomplete message protocol (30 messages)			
Time	17.5	6.4	2.7
Space	$3n+1$	$2n+1$	$\simeq 1.5$
Operations	456	125	3.6
Bounded buffer protocol (30 messages, 3 places)			
Time	21.0	6.6	3.2
Space	$6n-5$	$2n+4$	$\simeq 3$
Operations	699	129	5.4
One to many communication with acknowledge (20 messages, two readers)			
Time	25.1	4.6	5.45
Space	$10n+1$	$2n+1$	$\simeq 5$
Operations	910	109	8.3

Table 1: Distributed Shared Variable Communication

Benchmark	1 Proc.	2 Proc.	3 Proc.	5 Proc.	7 Proc.	9 Proc.
qs_iap	90 (1)	50 (1.8)	50 (1.8)	50 (1.8)	50 (1.8)	50 (1.8)
qs_conc_1	320 (1)	170 (1.8)	120 (2.6)	80 (4.0)	70 (4.5)	60 (5.3)
qs_conc_2	400 (1)	210 (1.9)	150 (2.6)	100 (4.0)	90 (4.4)	80 (5.0)
nrev	120 (1)	80 (1.5)	60 (2.0)	30 (4.0)	30 (4.0)	30 (4.0)

Table 2: &-Prolog Performance for Concurrent Benchmarks

rectly by hand in Linda in the most efficient way possible. "Time" gives the execution time in seconds. "Space" is an expression which gives the number of blackboard items generated by the programs with an input of size n (experimental results confirm this expression). "Operations" is the number of operations performed in the blackboard during the execution of the programs. This number was measured by instrumenting the SICStus implementation of the Linda library. The incomplete message program is the standard program implementing a two-way communication between processes [12, 28, 29]. The bounded buffer program is also the standard one. The one to many communication with acknowledge program allows several processes to read a stream produced by another, the latter being informed of which process read each element. We argue that, despite the amount of metainterpretation of terms happening when using shared variables for communication the resulting performance is still reasonable, specially if we take into account that the current implementation using attributes is very naïve, and many optimizations can be made to improve performance, both in the low–level implementation and in the compilation techniques.

In order to also see if actual performance improvements from parallelism are attainable with the technique, we have performed some measurements on a prototype implementation of communication through shared variables in the &-Prolog system using similar techniques. It should be noted that the results are based on rather inefficient implementations of the `wait/1`, `lock/2` and `unlock/1` primitives.

Table 2 shows the results. Times are in ms. and speedups between parentheses. **qs_iap** is the independent and–parallel version of quicksort, where the two recursive calls are executed in parallel, provided for reference. A small list is used in order to somewhat limit the parallelism available using (goal level) independent and-parallelism. The benchmark uses append rather than difference lists. **qs_conc_1** is again quicksort, in which the list splitting is performed concurrently with the recursive calls, acting as a producer. In **qs_conc_2** the concurrency is extended also to the append call, and all the goals in the recursive clause are run concurrently. Finally, **nrev** is the standard naïve reverse benchmark. The results are interesting in that they show that even with a naïve implementation of the concurrency primitives reasonable speedups can be achieved with our techniques in programs with small granularity (for example, nrev), even if not (yet) speed, when compared to sequential Prolog. Again, our objective herein is simply to point out and substantiate to some extent the great potential that in our view the concept of attributed variables has in the implementation of generic parallel, concurrent, and distributed logic programming systems.

7 Conclusions

We have proposed a different and novel use for the concept of attributed variables: developing a generic parallel/concurrent (constraint) logic programming system, using the same "glass box" flavor as that provided by attributed variables and meta-terms in the context of constraint logic programming implementations. We argue that a system which implements attributed variables and the few additional primitives which have been proposed constitutes a kernel language which can be easily customized at source level to implement many of the languages and execution models of parallelism and concurrency currently proposed, in both shared memory and distributed systems. We have illustrated this through a few examples.

While the wide applicability of the ideas presented is very attractive, a clear issue is the performance of the systems built using them. Of course, such performance is bound to be slower than that of the corresponding native implementations. It is clear that the native implementation approach is both sensible and practical, and simply the way to go in most cases. On the other hand we also feel there it is interesting to be able to have a generic system which can be easily customized to emulate many implementations. On one hand, it can be used to study in a painless way different variations of a scheme or to make quick assessments of new models. On the other hand the loss in performance is compensated in some ways by the flexibility (a tradeoff that has been found acceptable in the implementation of constraint logic programming systems), and such performance can be improved in a gradual way by pushing the implementation of critical operations down to C.

References

[1] J. Almgren, S. Andersson, L. Flood, C. Frisk, H. Nilsson, and J. Sundberg. *Sicstus Prolog Library Manual.* Po Box 1263, S-16313 Spanga, Sweden, October 1991.

[2] K. De Bosschere. Multi–Prolog, Another Approach for Parallelizing Prolog. In *Proceedings of Parallel Computing*, pages 443–448. Elsevier, North Holland, 1989.

[3] A. Brogi and P. Ciancarini. The Concurrent Language, Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, 1991.

[4] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.

[5] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In *Fourth International Conference on Logic Programming*, pages 40–58. University of Melbourne, MIT Press, May 1987.

[6] M. Carlsson. *Sicstus Prolog User's Manual*. Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[7] N. Carreiro and D. Gelernter. Linda in Context. *Communications ACM*, 32(4), 1989.

[8] T. Chikayama, T. Fujise, and D. Sekita. A Portable and Efficient Implementation of KL1. In Springer Verlag, editor, *Proc. 6th. Int'l Symposium on Programming Language Implementation and Logic Programming*, September 1994.

[9] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[10] A. Colmerauer et Al. *Prolog II: Reference Manual and Theoretical Model*. Groupe D'intelligence Artificielle, Faculté Des Sciences De Luminy, Marseilles, 1982.

[11] European Computer Research Center. *Eclipse User's Guide*, 1993.

[12] S. Gregory. *Parallel Logic Programming in PARLOG: the Language and its Implementation*. Addison-Wesley Ltd., Wokingham, England, 1987.

[13] M. Hermenegildo. A Simple, Distributed Version of the &-Prolog System. CLIP TR, School of Computer Science, Technical University of Madrid (UPM), 1994. Available from `http://www.clip.dia.fi.upm.es`.

[14] M. Hermenegildo, D. Cabeza, and M. Carro. On the Uses of Attributed Variables in Parallel and Concurrent Logic Programming Systems. CLIP TR, School of Computer Science, Technical University of Madrid (UPM), 1994. Available from `http://www.clip.dia.fi.upm.es`.

[15] M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[16] C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *1992 International Symposium on Programming Language Implementation and Logic Programming*, pages 260–268. LNCS631, Springer Verlag, August 1992.

[17] C. Holzbaur. *SICStus 2.1/DMCAI Clp 2.1.1 User's Manual*. University of Vienna, 1994.

[18] Serge Le Houitouze. A New Data Structure for Implementing Extensions to Prolog. In P. Deransart and J. Małuszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 136–150. Springer, August 1990.

[19] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *ACM Symp. Principles of Programming Languages*, pages 111–119. ACM, 1987.

[20] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[21] P. López and M. Hermenegildo. Efficient Term Size Computation for Granularity Control. In *Proc. of the Twelfth International Conference on Logic Programming*. The MIT Press, June 1995.

[22] U. Neumerkel. Extensible Unification by Metastructures. In *Proceeding of the META'90 workshop*, 1990.

[23] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, April 1990.

[24] V. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie Mellon, Pittsburgh, 1989. School of Computer Science.

[25] E. Y. Shapiro. A Subset of Concurrent Prolog and Its Interpreter. Technical Report TR-003, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, January 1983.

[26] K. Shen. Exploiting Dependent And-Parallelism in Prolog: The Dynamic, Dependent And-Parallel Scheme. In *Proc. Joint Int'l. Conf. and Symp. on Logic Prog.* MIT Press, 1992.

[27] G. Smolka. The Definition of Kernel Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), November 1994.

[28] S. Taylor. *Parallel Logic Programming Techniques*. Prentice-Hall, Englewood Cliffs, NY, 1989.

[29] E. Tick. *Parallel Logic Programming*. MIT Press, Boston, Mass., 1991.

[30] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, Cambridge MA, 1987.

[31] D.H.D. Warren. The Extended Andorra Model with Implicit Control. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.

[32] X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A.V.S. Sastry, and R. Sundararajan. Towards an Efficient Compile-Time Granularity Analysis Algorithm. In *Proc. of the FGCS'1992 International Conference*. Institute for New Generation Computer Technology (ICOT), June 1992.