On The Uses of Attributed Variables in Parallel and Concurrent Logic Programming Systems*

(Extended Abstract)

M. Hermenegildo D. Cabeza M. Carro
{herme,dcabeza,mcarro}@dia.fi.upm.es
Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid - SPAIN

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.

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 [19], 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 Hoitouze [14] with the purpose of implementing memory management optimizations such as *early reset* and *variable shunting*. 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 [2] used a data type called *suspension*, which was incorporated into SICStus Prolog for the implementation

*The work presented in this paper is supported in part by ESPRIT project "ACCLAIM".

of coroutining facilities. "Attributed variables" and "suspension variables" are essentially the same objects. Hoitouze's contribution was to put some emphasis on the data type as such and on memory management. He also used attributed 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 in [12, 13] for the specification and implementation of a variety of instances of the general CLP scheme [15]. Implementations of clp(R) (i.e. constraint solving over the "reals") and clp(Q) (i.e. constraint solving over the rationals) on top of SICStus-Prolog [3] using the concept of attributed variables have been presented by the same authors and illustrate the power of the approach. By enhancing the SICStus-Prolog system with attribute variables they essentially provide a *generic* system which is basically a SICStus-Prolog "clone" but 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 Eclipse system developed at ECRC [7].

While this approach in principle has drawbacks from the performance point of view (our measurements show that the performance obtained can be up to an order of magnitude slower than a specialized implementation of a constraint logic programming language such as e.g. CLP($\Re$)[16], CHIP [18], PrologIII [6], BNRProlog [20], etc.) 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 (such as perhaps those of the domains supported by the concrete systems mentioned before) by writing the unification handlers in a lower-level language.[1] The potential for achieving both genericity and reasonable speed is illustrated by the relatively good performance exhibited by the Eclipse system [7], 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. Our proposal shares with those previously discussed the objective of providing a generic system. But our interest focuses on implementing concurrent and parallel languages and execution models. 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

[1] In fact, the speed differential between the attribute variable / meta-term approach (with the constraint solving algorithms implemented in Prolog) and the native implementations seems to pretty much correspond to the differential between "C" and Prolog, the languages in which the constraint solving algorithms are implemented respectively.

primitives. Furthermore, we argue that that this can be done in a seamless and user-transparent way in both shared memory and distributed systems. Thus, one additional advantage of our technique is that it relates and reunites 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.

We have implemented our technique by adding support for attribute variables and the concurrency primitives (most of these primitives were already in the system) to &-Prolog, a system which efficiently supports and-parallelism on both shared memory and distributed architectures [11, 9]. It should be noted that the use that we propose of attributed variables in the implementation of concurrency and parallelism does not in any way prevent their simultaneous use also for other purposes, such as the original one of constraint solving.

The rest of the paper proceeds as follows: Section 2 provides a minimal introduction to attribute variables and the related primitives. Section 3 then describes the minimal concurrent / parallel language that we assume. Sections 4 and 5 provide two concrete examples of the application of attributed variables proposed. In Section 4 we sketch an implementation of the DDAS scheme for parallel execution of Prolog, while in Section 5 we present a way of implementing concurrent logic programming languages in a distributed environment. Finally, in Section 6 we discuss other uses of the technique and propose lines of future research.

Space limitations force the presentation to cover only some basic cases and give incomplete implementations. For more details the reader is referred to [10]. Our objective 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.

2 Attributed Variables and Related Primitives

We provide a brief introduction to attributed variables. We follow mainly Holzbaur's description of their implementation in the SICStus Prolog clone.

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. The indexing mechanism treats variables and attributed variables in the same way. Also, built-in predicates observe attributed variables as if they were ordinary variables. Special treatment for attributed variables does apply in the following situations:

- Notably, during unification. 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.

- When printed via 'print/1', a user-supplied predicate gets a chance to print the attributed variable in some customized fashion.

2.2 Manipulation of Attributed Variables

The following is a list of typical predicates which provide for the introduction, detection, and manipulation of attributed variables:

- **get_attribute(X,C)** If X is an attributed variable, unify the corresponding attribute with C, otherwise fail.
- **attach_attribute(X,C)** Turn the free variable X into an attributed variable with attribute C. C must *not* be a variable. (Attaching an attribute to variable generally changes the identity of the variable.)
- **detach_attribute(X)** Remove the attribute from an attributed variable, turning it into a free variable. (Detaching the attribute from a variable generally changes the identity of the variable.)
- **update_attribute(X,C)** Change the attribute of the attributed variable X to C. Acts as as an attach, followed by a detach, but might be more (memory) efficient.

Note that all operations on attributed variables behave correctly (i.e. they are undone) upon backtracking.

2.3 Unification in the Presence of Attributed Variables

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 recorded internally.
- If there is more than one binding event for attributed variables between two inference steps, a list of attributed variable-value pairs is collected internally.
- At the next inference step, the pending attributed variable-value pairs are supplied to user-defined handlers which are Prolog predicates.

The handlers for the unification of attributed variables mentioned above are provided by the user by means of the following predicates:

- `verify_attribute(C,T)` This user-defined predicate is invoked when an attributed variable with an attribute which unifies with `C` is about to be unified with the non-variable term `T`.

- `combine_attributes(C1,C2)` This user-defined predicate is invoked when two attributed variables with attributes C1,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 that if access to the actual attributed variable is needed the variable itself can be included in the attribute.

2.4 Other Related Primitives

In general, a number of other primitives are often provided which allow pretty printing and dumping of the results in a user understandable format.

2.5 Attributed Variables And Coroutining – an Example

The following example, due to [12] serves both to illustrate the use of the primitives introduced in the previous section and also to recover the functionality of `freeze` since attribute variables are, as mentioned in the introduction, most easily implemented in practice by "cannibalizing" an existing implementation of `freeze`:

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

verify_attribute( frozen(Var,Goal), Value) :-
  detach_attribute( Var),
  Var = Value,
  call(Goal).

combine_attributes( frozen(V1,G1), frozen(V2,G2)) :-
  detach_attribute( V1),
  detach_attribute( V2),
  V1 = V2,
  attach_attribute( V1, frozen(V1,(G1,G2))).
```

The call to attach attribute 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_attribute` which in turn performs the meta-call. Note the definition of `combine_attributes` 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 The Kernel Concurrent Language

We now introduce a simple concurrent and parallel extension of Prolog, that we call "Kernel &-Prolog" (K&P). The purpose of this language is to provide a reduced (hopefully minimal) set of 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 [11] and distributed [9] 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 "gas" (computational resources) to them, and synchronization:

- `&/2` – Standard fork/join parallel conjunction operator (the one used, for example, by the &-Prolog parallelizing compiler [1]). 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. This is a "parallelism" operator: it is used to indicate where parallel execution can be profitable, with speed as the main objective. All tasks created with this primitive will eventually be run, unless one such task goes into an infinite loop. It is defined as an infix operator. 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]

- `&&/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.[3] Thus, even if no processors are available, the two literals will be executed with (apparent) simultaneity in a fair way. It is defined as an infix operator. For example, `..., p(X) && q(X), r(X), ...` will fork a task `p(X)` in parallel with `q(X)`, both tasks getting a "thread" allocated to them. The continuation `r(X)` will wait until both `p(X)` and `q(X)` are completed. This approach of distinguishing specially the cases where "gas" (or, more formally, a notion of fairness) is to be attached to parallel processes has also been followed in the concurrent constraint language Oz [25]. There are also variants of the primitives which allow handing levels of "gas" down to lower levels for implementing priority schemes.

- `&/1` – Standard fork operator. It performs a parallel fork of the literal(s) involved. No waiting for its return is involved (unless explicitly expressed using the `wait` primitive – see below). If no processors are available, then the literal may be executed in the same processor and sequentially, i.e. after the rest of the computation finishes. It is defined as a postfix operator.

[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.

[3] The implementation of these tasks is identical to that of the standard tasks in the &-Prolog system except that while standard tasks are put on a goal stack and picked up when there is an idle thread, in the case of "fair" tasks an idle thread is directly attached to the task. Notably, a new thread is created if no idle thread is available. When using the standard operators the maximum number of threads is never larger than the number of processors on the system, while when using fair operators, many more threads than processors may be created.

For example, `..., p(X) &, q(X), r(X), ...` will fork a task `p(X)` in parallel with the rest of the computation.

- `&&/1` – "fair" fork operator. It performs a parallel fork of the literals involved. No waiting for its return is involved (unless explicitly expressed using the `wait` primitives – see below). A "thread" is assigned to the literal. It is defined as a postfix operator. For example, `..., p(X) &&, q(X), r(X), ...` will fork a task `p(X)` in parallel with the rest of the computation and a thread will be attached to it.

- `&@/2` – "Placement" standard fork operator. It performs a parallel fork of the literal(s) 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. It is defined as an infix operator. 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*. 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. Processor names can also be of the format *network_node#processor*.[4]

- `&&@/2` – "fair" placement fork operator. It performs a parallel fork of the literal(s) 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)` / `unlock(X)`: This primitive gets (releases) a lock on the (address of the) object `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.

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 [5, 4] 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 [9]. 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`), 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 wait on conjunctions or disjunctions of data.

[4] 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. Alternatively, the general goal stack of that agent can be used, in which case that agent will execute the goal unless another agent becomes idle first and steals the goal.

4 Implementing the DDAS Model

We now discuss the implementation of a model for parallel execution of Prolog, the DDAS model of Shen [24]. We choose this model both because it is interesting and also because it has quite complex behavior and synchronization rules and therefore it should put to the test our thesis. In the following for simplicity we will discuss mainly forward execution in the model. However we argue that backward execution can be implemented in a similar way.

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.[5] Parallelism in this model is controlled by means of "Extended Conditional Graph Expressions" 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. Bindings to these variables by the goals in the ECGEs can only be performed if certain conditions hold. Otherwise the computation must suspend until such conditions do hold. In particular, 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).

For example, when the ECGE `..., (dep(X) => p(X) & q(X)), ...` is executed the variable `X` is marked as "dependent" and the goals `p(X)` and `q(X)` are scheduled to execute in parallel. During their execution bindings to `X` behave according to the above mentioned rules.

In order to support this model in K&P we assume a source to source transformation (using `term_expansion/2`) of ECGEs. The intuition behind the transformation (and the implementation of the built-ins used by it) is as follows. 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 to them attributes. 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.[6] 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 (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

[5] More precisely, independent and-parallelism, but where the independence rule is applied at a much lower level of granularity.

[6] Locking (using the `lock/1` and `unlock/1` primitives) must obviously be done while performing such bindings, since other threads running the parallel goals can be performing simultaneous accesses. However, we have left out all locking from the description for simplicity.

of the goal itself and a call to the predicate `pass_token/1` 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.

For example, the ECGE

```
..., (dep(X), ground(Y) =>
       a(X,Y) & b(X,Y) & c(X)), ...
```

is transformed into

```
..., (ground(Y), dep(X, [X1, X2]) ->
       (a(X,Y), pass_token(X)) &
       (b(X1,Y), pass_token(X1)) &
       c(X2)
     ;
       a(X,Y), b(X,Y), c(X) ), ...
```

The new variables produced after the renaming of the original shared variable form a "group". That group is logically sorted in the same order in which the goals where the variables appear become producers, and is updated at run–time to reflect the success of the goals or the execution of new and–parallel goals sharing a variable belonging to the group. Thus, the group contains the variables that appear in the computation tree frontier that correspond to the shared variable, in left–to–right order.

Each dependent variable's attribute in the group shares a common field from where the aforementioned group can be accessed, i.e., from each variable's attribute one can consult (and update) the group itself. Moreover, each variable's attribute includes a suspension–oriented field whose aim is to allow the suspension of goals trying to bind a dependent variable when they are in consumer position. This suspension is performed, as mentioned above, using the `wait/1` primitive. As an example, the attribute for variable `Var` can be `depatt(Var, Susp, Group)`.

In what follows we explain more in depth the operations sketched above.

The `dep/2` predicate takes two arguments, the first is the original dependent variable and the second is a list of the remaining variables in the group (that belong to consumer goals initially). `dep/2` copies the term given in the first argument to each of the elements of the second argument. For each free variable appearing in the term this variable and the corresponding free variables of the copied terms are placed in a new "group". In addition, the suspension field of the producer variable is marked as `producer`. Variables (initially) in consumer positions are left unbound so that `wait/1` will suspend on them. A special case occurs if the dependent variable in the ECGE was already dependent. In that case, the group to which this variable belongs is expanded to include the new renamed variables, and all inserted variables are left as consumers.

The `pass_token/1` predicate is called after the execution of a dependent goal, and binds each variable in the argument to the next variable in its group, removing the first variable from the group, and passing the suspension field of the attribute of the first variable to the second.

When a dependent variable is about to be bound to a non-variable, the predicate `verify_attribute/2` is invoked since such a variable is attributed. `verify_attribute/2` performs a `wait` on the suspension field of the attribute, to avoid unification of the variable if it is not in a producer position. The suspension field can be instantiated to two values. A first case is when the suspension field has the value `producer`. This means that this goal is now the producer and thus we can proceed with the unification. The remaining consumer variables are deprived of

their attributes, and their suspension fields are collected. In order to initialize the attributes of the free variables that possibly appear in the value given to the variable, the `dep/2` predicate is called again. Finally, the consumers are woken by binding their previously collected suspension fields with the atom `consumer`. Thus, the other possible case is when, after proceeding from a `wait`, the suspension field has the value `consumer`. In this case, since the variable attributes were already managed by the producer, only the standard unification has to be performed.

The listing of the predicates mentioned above is given schematically below. While this code does not take into account all the details present in an actual implementation, it does on the other hand give an idea of how the operations can be implemented with the primitives provided.

```
dep(X, Xs) :-
   copy_term_to_list(Xs, X),
   dep_term_vars(X, Xs).

copy_term_to_list([], _X).
copy_term_to_list([Y|Ys], X) :-
   copy_term(X, Y),
   copy_term_to_list(Ys, X).

dep_term_vars(X, Xs) :-
   {calls dep_var for each variable inside X
   together with the corresponding variables in Xs}

dep_var(X,Xs) :-
   get_attribute(X, depatt(_X, _Susp, Group)) →
      insert_to_right(Group, X, Xs),
      put_depatt_attributes(Xs, Group)
   ;
      create_group_structure([X|Xs], Group),
      attach_attribute(X, depatt(X, producer, Group)),
      put_depatt_attributes(Xs, Group).

put_depatt_attributes([], Group).
put_depatt_attributes([X|Xs], Group) :-
   attach_attribute(X, depatt(X, _Susp, Group)),
   put_depatt_attributes(Xs, Group).

pass_token(X) :-
   {calls pass_token_var for each variable
   inside the term X}

pass_token_var(X) :-
   get_attribute(X, depatt(_X, Susp, Group)),
   dettach_attribute(X),
   delete_and_get_right_var(X, Group, Xr),
   get_attribute(Xr, depatt(_Xr, Susp_r, _Group)),
   X = Xr,
   Susp = Susp_r.
```

```
verify_attribute(depatt(X, Susp, Group), Val) :-
   wait(Susp),
   (
      Susp = consumer →
         X = Val
      ;
         dettach_attribute(X),
         X =Val,
         get_right_vars(Group, X, Xs),
         detach_and_get_susps(Xs, Susps),
         dep(X, Xs),
         signal_consumers(Susps)
   ).

detach_and_get_susps([],[]).
detach_and_get_susps([X|Xs],[Susp|Susps]) :-
   get_attribute(X, depatt(_X, Susp, _Group)),
   dettach_attribute(X),
   dettach_and_get_susps(Xs, Susps).

signal_consumers([]).
signal_consumers([consumer|S]) :-
   signal_consumers(S).

% These predicates deal with the group structure

create_group_structure(Xs, Group) :-
   {makes a group with the variables in the list
   Xs, in that order}

insert_to_right(Group, X, Xs) :-
   {given a group Group that contains the
   variable X, inserts the list of variables
   Xs to its right}

delete_and_get_right_var(X, Group, Xr) :-
   {removes X from Group, and gives
   in Xr the variable to the right of X
   in the group}

get_right_vars(Group, X, Xs) :-
   {gives in Xs the list of the variables
   to the right of X in Group}
```

5 Implementing Concurrent (Constraint) Languages in Distributed Environments

We now sketch a relatively different application. Our objective here 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 [23], PARLOG [8], GHC [26], Janus [22], AKL [17], Oz [25], etc.), and "distributed" or "blackboard" systems [5] (for which there are many implementations, one of the most popular being the one bundled with [3]) in which communication is done through explicit built-ins which access shared channels or global data areas. 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 previous 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 will actually mean execution in another node of the net), and "ask" and "tell" unification primitives. The sort of net 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 — 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. Thus, if a goal `p(X)` is to be executed concurrently and remotely, it can simply be replaced with a call to "`p(X) &&@ _`" (or a value or non-anonymous variable can be used in place of the "`_`" if it is important or useful to impose or know in which particular node the goal is going to be executed). However, while this allows creating remote tasks, it does not by itself implement the communication of values between nodes through the variable `X`. We propose to do this by placing before the call to `p(X) &&@ _` a call to a predicate which will attach an attribute to the variable `X` marking it as a "communication variable". If `X` is bound to a complex term with variables, then each such variable is marked in that way. 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 distiguished 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 such variables will be then 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 primitives. An ask will wait (using the blocking `read/1` blackboard primitive)[7] until a binding for the variable is posted (i.e. a *(variable_id,value)* pair where *variable_id* is the identifier of the variable is present in the blackboard). Finally, it may often be possible to tidy up things when a remote goal finishes by erasing 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 originating heap (as in the case of the DDAS model, a sequential conjunction of the parallel goal with a call to a tidying-up predicate can be used for this purpose).

While the approach sketched certainly has the potential of providing the functionality required of the language being implemented, the performance of the approach will of course depend heavily in turn on the performance of the blackboard implementation.

Space limitations do not allow us to describe this approach in more detail. For more information the reader is referred to [10].

6 Discussion and Future Work

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.

Lack of space does not allow elaborating further but we believe 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 of a kernel language. We believe it is quite possible to encode the determinacy driven synchronization of the Andorra-I system [21] 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 [27], such as AKL [17]. 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[8] 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

[7] Note that a variable can have multiple readers and thus `in/1` cannot be used in general. 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. This illustrates how the attributed variable approach allows performing low-level optimizations as source to source transformations also in the context!

[8] Some models are more complicated: 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 in the implementation of AKL.

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 computation 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.[9] 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 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`.

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 much 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] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. Technical Report TR Number CLIP7/93.0, T.U. of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, October 1993.

[2] 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.

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

[4] N. Carreiro and D. Gelernter. How to Write Parallel Programs – A Guide to the Perplexed. *ACM Computing Surveys*, September 1989.

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

[6] A. Colmerauer. Opening the Prolog-III Universe. In *BYTE Magazine*, August 1987.

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

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

[9] M. Hermenegildo. A Simple, Distributed Version of the &-Prolog System. Technical report, School of Computer Science, Technical University of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, April 1994.

[9] Promotion rules can also be implemented by updating the identifiers (the attributes) of all the local variables to higher levels.

[10] M. Hermenegildo, D. Cabeza, and M. Carro. On The Uses of Attributed Variables in Parallel and Concurrent Logic Programming Systems. Technical report CLIP 5/94.0, School of Computer Science, Technical University of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, June 1994.

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

[12] C. Holzbaur. *Specification of Constraint Based Inference Mechanisms through Extended Unification.* PhD thesis, University of Vienna, 1990.

[13] 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.

[14] Serge Le Huitouze. 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.

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

[16] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Fourth International Conference on Logic Programming*, pages 196–219. University of Melbourne, MIT Press, 1987.

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

[18] H. Simonis M. Dincbas and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1 & 2):72–93, 1990.

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

[20] W. Older and A. Vellino. Extending Prolog with Constraint Arithmetic in Real Intervals. In *Canadian Conference on Electrical and Computer Engineering*, September 1990.

[21] 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.

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

[23] 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.

[24] 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.

[25] Gert Smolka. Feature Constraint Logics for Unification Grammars. *Journal of Logic Programming*, 1991.

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

[27] 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.