Distributed Concurrent Constraint Execution in the CIAO System*

Daniel Cabeza Manuel Hermenegildo
{dcabeza,herme}@fi.upm.es

Computer Science Department
Technical University of Madrid (UPM), Spain

11 September 1995

Abstract

This paper describes the current prototype of the distributed CIAO system. It introduces the concepts of "teams" and "active modules" (or active objects), which conveniently encapsulate different types of functionalities desirable from a distributed system, from parallelism for achieving speedup to client-server applications. It presents the user primitives available and describes their implementation. This implementation uses attributed variables and, as an example of a communication abstraction, a blackboard that follows the Linda model. The functionalities of the system are illustrated through examples, using the implemented primitives. The implementation of most of the primitives is also described in detail.

1 Introduction

The distributed CIAO system provides distributed execution capabilities in both the (C)LP and CC styles of programming supported by the CIAO language [10, 2, 11]. Distributed execution can be useful for various purposes: one is to exploit coarse-grained parallelism in a distributed environment with the objective of obtaining execution speedups. This is also one possible way of implementing true concurrency between agents. Another quite different purpose is to request and/or provide remote services in a distributed network. Intuitively, the first two models involve several agents cooperating to run the same application. In distributed CIAO, the former need is addressed by the notion of "team" of workers. The latter is supported by the notion of "active module" (or active object). We believe that these two proposed concepts conveniently encapsulate many different types of functionalities desirable from a distributed system, from performance to, for example, client-server applications.

In the following we describe the current prototype of the distributed CIAO system by presenting for each concept (Teams and Active Modules) the user

*This work was funded in part by ESPRIT project 7195 "ACCLAIM" and by CICYT projects TIC93-0975-CE and TIC93-0737-C02-01 IPL-D. The authors would also like to thank F. Bueno and G. Puebla for useful discussions in the context of the paper.

primitives available, and discussing their implementation. This implementation uses attributed variables [15, 14, 4, 13, 7] and, as an example of a communication abstraction, a blackboard that follows the Linda model [6]. An interesting characteristic of the implementation is that it is done enterely at the source (Prolog) level. The functionalities of the system are also illustrated through examples, using the implemented primitives. Note that, except where otherwise noted, all the CIAO builtins have a meaning equivalent to that of SICStus Prolog v2.1 [5].

2 Teams

A team is a set of CIAO workers that share the same code and cooperate to run it. At startup, a worker belongs to a team which includes only itself. Two primitives are provided that add workers to the current team and delete workers from it.

- `add_worker(Id)`, `add_worker(Host,Id)` – Adds a worker to the team. If `Host` is provided, run the worker in it, else run the worker in the current host (this is useful, for example, to have true concurrency between threads). A unique identifier for the worker created is returned in `Id`.
- `delete_worker(Id)` – Deletes a worker from the team. If `Id` is instantiated then delete the worker with this identifier, else delete an idle worker and unify `Id` with its identifier.

The team model allows concurrency (or parallelism) between workers. Note that this is only useful if tasks are of sufficient granularity. Thus, if the parallelization is done automatically by a compiler, an analysis of granularity is of vital importance[1]. The primitives of the language provide a means for expressing independent And-parallelism and also concurrency (dependent And-parallelism). We now discuss them.

- `A &> H` – Sends out goal `A`, to be executed potentially by another worker of the team, returning in the variable `H` a handler of the goal sent.
- `H <&` – Gets the results of the goal pointed to by `H`, or executes it if it has not been executed yet. Backtracking of the goal will be done at this point.
- `A & B` – Performs a parallel "fork" of the two goals involved and waits for the execution of both to finish. This is the parallel conjunction operator used, for example, by the &-Prolog parallelizing compiler [3]. If no workers are idle, then the two goals may be executed by the same worker and sequentially, i.e., one after the other. This primitive can be implemented using the previous two:

  ```
  A & B :- B &> H, call(A), H <& .
  ```

  Note that these first three primitives are intended for independent And-parallelism, and, as such, the bindings made in the shared variables are not seen until the threads join (that is, there is an implicit `copy_term` in the goals sent out).

[1] The CIAO compiler includes a granularity control system [8].

- `A &` – Sends out goal `A` to be executed potentially by another worker of the team. No waiting for its return is performed. Bindings in the variables of `A` (tells) will be exported to other workers sharing them.
- `A &&` – "Fair" version of the `&/1` operator. If there is no idle worker, create one to execute goal `A`. This way, fairness among concurrent threads is ensured.
- `wait(X)` – Suspends the execution until `X` is bound.
- `ask(`C`)` – Suspends the execution until the constraint C is satisfied.

Members of a team can communicate among them either by shared variables or explicitly by means of a blackboard. This blackboard provides a set of Linda-like [6] primitives, essentially reproducing the functionality of the Linda library present in SICStus Prolog v2.1 [1]. Workers can write (using `out`/1), read (using `rd`/1), and remove (using `in`/1) data to and from the blackboard. If the data is not present on the blackboard, the worker 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. There are also input primitives that wait on disjunctions of terms (`in`/2 and `rd`/2).

2.1 Implementation Issues

In the current implementation, each worker is implemented by a process in the corresponding host, and the `add_worker` primitives are ultimately implemented with the UNIX command `rsh` (which executes a command in a remote host). All the communication between workers is implemented via the blackboard, which is created the first time a distributed primitive is executed. The blackboard itself is implemented using a UNIX sockets interface, which is also available at the user level.

Idle workers listen to the blackboard in order to get posted goals sent by the primitives `&>`/2, `&`/1, or `&&`/1, and execute them. If the goal was issued by the `&>`/2 primitive, the worker returns its solutions back to the blackboard. The solutions are gathered in turn by the `<&`/1 primitive. This essentially implements in a distributed fashion a goal stealing scheduling algorithm similar to that of the &-Prolog system [12].

The distributed communication using shared variables follows the lines proposed in [9], where some of the proposed operators were already presented. The distributed concurrency primitives `&`/1 and `&&`/1 take care of marking with an attribute of "communication variable" the variables of the goal (note that if an analysis is done this can be optimized by marking only the relevant variables). Then, when they are involved in a unification, the hook for attributed variables posts the bindings to the blackboard to inform other workers about them (after ensuring their consistency). Also, when a `wait` is done on a communication variable, the worker suspends until a binding for this variable is posted to the blackboard.

Also, the requisite that the workers in the same team share the code implies that builtins that modify it (e.g. `compile`, `assert`, etc) must be performed by all the workers ("globalized"). Thus, the compiler, when processing code that

deals with distributed execution, compiles these builtins in such a way that they are automatically posted to the blackboard to be executed by all the members of the team (using `expand_term`). Also, when a new worker is added to the team, it must be put in the same state as its siblings. This is managed by recording the execution of "global" builtins, so that the `add_worker` primitive can send to the new worker the series of such builtins to execute. Care must be taken with nested global executions: during global compilation of a file, "global" builtins must be executed locally. This is easily implemented with a flag that tells whether we are executing in the scope of a global builtin or not.

2.2 Using Parallelism: an Example

In this section we will show an example of the use of the primitives introduced in the previous section. The program in the example first gets some work, then finds out somehow which hosts in the local network are not too loaded, starts a worker in each one, process the work, stops the started workers and exits. The important thing here is that the elements of the list `L` in `process_list(L)` are processed in parallel by the team of workers. Thus, if N workers were started, the list would be processed N times faster (modulo communication overhead).

```
main :-
        get_list(L),
        collect_unloaded_hosts(Hosts),
        add_workers(Ids, Hosts),
        process_list(L),
        delete_workers(Ids),
        halt.

get_list(L) :- ...

process_list([H|T]) :-
        process(H) &
        process_list(T).
process_list([]).

collect_unloaded_hosts(Hosts) :- ...

add_workers([Id|Ids],[Host|Hosts]) :-
        add_worker(Id,Host),
        add_workers(Ids,Hosts).
add_workers([],[]).

delete_workers([Id|Ids]) :-
        delete_worker(Id),
        delete_workers(Ids).
delete_workers([]).
```

2.3 Code

In order to be more concrete we sketch our implementation of the ideas outlined above in a widely available environment: SICStus Prolog v2.1, enhanced with attributed variables, and using the Linda library provided with that version. The code is a simplified version of our implementation, the real code can be

requested by e-mail to the authors. Note that the code presented herein must not be expanded by the compiler.

First, we show the code that deals with creation and deletion of workers.

```
add_worker(Id) :-
        get_blackboard_address(BBAddr),
        new_worker_id(Id),
        start_worker(Id,BBAddr).

add_worker(Host,Id) :-
        current_host(Host), !,
        add_worker(Id).
add_worker(Host,Id) :-
        get_blackboard_address(BBAddr),
        new_worker_id(Id),
        start_worker(Id,BBAddr,Host).

get_blackboard_address(BBAddr) :-
        blackboard_address(BBAddr), !.
get_blackboard_address(BBAddr) :-
        start_blackboard(BBAddr),
        % Connect me with the blackboard
        initialize_client(0,BBAddr),
        out('$worker'(0)),
        out('$last_worker'(0)),
        retract(global_counter(C)),
        out('$global_counter'(C)).

new_worker_id(Id) :-
        in('$last_worker'(N)),
        Id is N+1,
        out('$last_worker'(Id)).

% Start the blackboard process returning its address in BBAddr
start_blackboard(BBAddr) :-
        BBAddr = Host:Port,
        current_host(Host),
        ciao_blackboard(CiaoBB),
        unix(popen(CiaoBB, read, S)),
        read(S, Port),
        assert(blackboard_address(BBAddr)).

start_worker(Id,BBHost:Port):-
        executable(Exec),
        atom_append([Exec," worker ",Id," ",BBHost," ",Port," &"],
                    StartCommand),
        start_worker_executable(Id,StartCommand).

start_worker(Id,BBHost:Port,Host):-
        executable(Exec),
        atom_append(["rsh -n ",Host," ",
                     Exec," worker ",Id," ",BBHost," ",Port," &"],
                    StartCommand),
        start_worker_executable(Id,StartCommand).
```

```
start_worker_executable(Id,StartCommand) :-
        % start worker...
        unix(system(StartCommand)),
        setof(global_builtin(N,B), global_builtin(N,B), Bs),
        out('$global_history'(Id,Bs)),
        % ... and wait until it is ready
        rd('$worker'(Id)).

initialize_client(Id,BBAddr) :-
        % connect to the blackboard
        linda_client(BBAddr),
        assert(blackboard_address(BBAddr)),
        assert(worker_number(Id)),
        assert(query_counter(Id)),
        assert(shv_counter(Id)).

delete_worker(Id) :-
        var(Id), !,
        rd_noblock('$idle'(Id)), % Choose an idle worker
        in('$worker'(Id)),
        out('$halt'(Id)).
delete_worker(Id) :-
        in_noblock('$worker'(Id)),
        out('$halt'(Id)).
```

Now, this is the code executed by started workers. For simplicity, we show the version in which remote parallel calls are implemented using `findall`. The alternative would be to post to the blackboard each solution right after computing it.

```
% This is the entry point of the distributed ciao executable, its
% argument is the list of command-line arguments
boot1([worker,Id,BBHost,Port]) :- !,
        atom_chars(Port,SPort), number_chars(NPort,SPort),
        atom_chars(Id,SId), number_chars(NId,SId),
        initialize_client(NId,BBHost:NPort),
        in('$global_history'(NId,Builtins)),
        execute_global_history(Builtins,NId),
        out('$worker'(NId)),
        idle_worker_loop(NId).
boot1(_) :-
        current_dir(WD),
        % other workers start in the same directory
        assert(global_builtin(0,unix(cd(WD)))),
        assert(global_counter(1)).

idle_worker_loop(Id) :-
        out('$idle'(Id)),
        in([
               '$concurrent'(Id,_,_), % concurrent goal for me
               '$halt'(Id),           % halt
               '$global'(Id,_,_),     % global call
               '$concurrent'(_,_),    % concurrent goal
               '$query'(_,_),         % query
          ], Command),
```

```
        (
            in_noblock('$idle'(Id)) ->
                process_command(Command,Id)
        ;   Command = '$concurrent'(_,_,_) ->
                process_command(Command,Id)
        ;   out(Command),
            NCommand = '$concurrent'(Id,_,_),
            in(NCommand),
            process_command(NCommand, Id)
        ),
        fail.
idle_worker_loop(Id) :- idle_worker_loop(Id).

process_command('$halt'(_),_) :- !,
        % close connection to blackboard
        close_client,
        halt.
process_command('$global'(Id,N,Q),_) :- !,
        execute_global(N,Q),
        out('$done_global'(Id,N)).
process_command('$concurrent'(Q,Ps),Id) :- !,
        assign_ids(Ps),
        once(Q).
process_command('$concurrent'(_,Q,Ps),Id) :- !,
        assign_ids(Ps),
        once(Q).
process_command('$query'(N,Q),Id) :- !,
        findall(Q,Q,Answers),
        out('$answers'(N,Answers)).

execute_global_history([global_builtin(N,B)|RestBuiltins],Id) :-
        execute_global(N,B),
        execute_global_history(RestBuiltins,Id).

execute_global(N,B) :-
        assert(global_builtin(N,B)),
        assert('$inside_a_global_call'),
        maybe(B),
        retract('$inside_a_global_call').

once(G) :- call(G), !.

maybe(Q) :- call(Q), !.
maybe(_).
```

We now go over the primitives which send out work to other workers in the team. As said before, distributed communication of concurrent goals via shared variables is implemented following the lines of [9]. Thus, `assign_ids/2` has the same meaning as there, and `get_var_ids/2` is essentially `var_ids/2`, changing the first argument from a list to a term.

```
:- op(950, xfx, '&>').
:- op(950, xf,  '<&').
:- op(950, xfy, '&').
:- op(950, xf,  '&').
```

```
Q &> H :-
        get_blackboard_address(_),
        new_query_id(N),
        % clean blackboard on backtracking
        undo(in([
                    '$answers'(N,_),
                    '$query'(N,_)
                ], _)),
        out('$query'(N,Q)),
        H = query(N,Q).

query(N,Q) <& :-
        in([
               '$answers'(N,_),
               '$query'(N,_)
           ], Data),
        (
            Data = '$query'(_,Qr) ->
                findall(Qr,Qr,As)
        ;   Data = '$answers'(_,As)
        ),
        % restore data in blackboard on backtracking
        undo(out('$answers'(N, As))),
        member(Q,As).

new_query_id(N) :-
        retract(query_counter(N)),
        N1 is N+100, % This limits the number of workers to 100
        assert(query_counter(N1)).

A & B :- B &> H, call(A), H <& .

Q & :-
        get_blackboard_address(_),
        get_var_ids(Q,Ps),
        local_assign_ids(Ps),
        out('$concurrent'(Q,Ps)).

Q && :-
        get_blackboard_address(BBAddr),
        get_var_ids(Q,Ps),
        local_assign_ids(Ps),
        (
            in_noblock('$idle'(Id)) ->
                out('$concurrent'(Id,Q,Ps))
        ;   new_worker_id(Id),
            out('$concurrent'(Id,Q,Ps)),
            start_worker(Id,BBAddr)
        ).
```

Suspension primitives are more involved, especially `ask`. The code for `wait` is shown in [9]. We will sketch here the implementation of `ask` in the Herbrand domain with (dis)equality constraints.

First we address the question of local variables (those not marked with an attribute of "communication variable"). As such variables are local to a worker they can be considered existential variables. Thus, an `ask` regarding one of these variables can always succeed without suspension.

For other cases we will need a multiple `wait`, that is, a predicate that waits until one of several variables is bound. Fortunately, since bindings on communication variables are posted to the blackboard, and since we have available a multiple `rd`/2 primitive that waits for one term out of a list of terms to be posted, this is easily implemented.

Thus, the `ask` primitive proceeds as follows: if it involves the equality of two communication variables, it waits for a binding on one of them, checks equality, and, if needed, calls itself recursively. If it is comparing a communication variable and a structure, waits for a binding on the communication variable and recurses. Finally, if it is comparing two structures, compares their functors and then compares their arguments pairwise. If the comparisons did not fail, but the structures are not yet equal, it waits for a binding on one out of the several communication variables that can make the structures different. Then, if that binding does not lead to failure, it recurses in the list of pairs of arguments not made already equal. Thus, in fact the procedure must accept a list of pairs to be compared, waiting for any of the bindings that can make a pair different.

Finally, we show the management of builtins that need to be "globalized". Each builtin `B` as such is translated to `execute_global(B)`, whose definition is:

```
execute_global(Call) :-
        '$inside_a_global_call', !,
        call(Call).
execute_global(Call) :-
        blackboard_address(_BBAddr), !,
        next_global_number_BB(N),
        other_workers_list(OWs),
        send_global(OWs,N,Call),
        execute_global(N,Call),
        receive_global(OWs,N).
execute_global(Call) :-
        next_global_number(N),
        execute_global(N,Call).

next_global_number(N) :-
        retract(global_counter(N)),
        N1 is N+1,
        assert(global_counter(N1)).

next_global_number_BB(N) :-
        in('$global_counter'(N)),
        N1 is N+1,
        out('$global_counter'(N1)).

other_workers_list(OWs) :-
        worker_list(Ws),
        worker_number(MyId),
        delete_one(Ws, MyId, OWs), !.
other_workers_list([]).
```

```
worker_list(Ws) :-
        rd_findall(W, '$worker'(W), Ws).

send_global([],_N,_Global).
send_global([W|Ws],N,Global) :-
        out('$global'(W,N,Global)),
        send_global(Ws,N,Global).

receive_global([],_N).
receive_global([W|Ws],N) :-
        in('$done_global'(W,N)),
        receive_global(Ws,N).
```

3 Active Modules

An active module (or an active object, if modularity is implemented via objects) is a module to which computational resources are attached (in our case, a CIAO process or a CIAO team). In a distributed environment, this is useful to provide remote services to other members of the network. In principle, every module can be activated, and from the programmer point of view an active module is like an ordinary module. An active module has an address (network address) that must be known to use it. Thus, the only difference between an ordinary module and an active module is that to use an active module one has to know its address.

Now we present the constructions of the language that implement active modules. Note that for concreteness and compatibility in the description of modules we mainly follow the same scheme as SICStus Prolog.

- `:- use_active_module(Module,Predicates)` – A declaration used to import the predicates in the list `Predicates` from the (already) active module `Module`. From this point on, the code should be written as if a standard `use_module/2` declaration had been used. The declaration needs the following hook predicate to be defined.

- `module_address(Module,Address)` – This predicate must give, for each active module imported in the code, its address.

- `save_active_module(Name, Address, Hook)` – Saves the current code as an active module, into executable file `Name`. When the file is executed (for example, at the operating system level by "`Name &`"), `Address` is unified with the address of the module, and `Hook` is called in order to export this address as required.

Note that this scheme is very flexible. For example, the predicate `module_address/2` itself could be imported, thus allowing a configurable standard way of locating active modules. One could, for example, use a directory accessible by all the involved machines to store the addresses of the active modules in them, and this predicate would examine this directory to find the required data. A more elegant solution would be to implement a name server, that is, an active module with a known address that records the addresses of active modules and supplies this data to the modules that actively import it. Later we will show how such a name server can be implemented and used.

3.1 Implementation Issues

Active modules are essentially deamons: Prolog executables which are started as independent processes at the operating system level. Communication with active modules is also implemented by means of a blackboard. Each active module has its own blackboard. Requests to execute goals in the module are put into the blackboard by remote programs. When such a request arrives, the process running the active module takes it and executes it, returning to the blackboard the computed results. These results are then taken by the remote processes. When an active module is run by a team, the blackboard of the team can be used for both inter-team communication and outer communication. The address of an active module is then the address of its blackboard (in particular, in the current implementation it is a UNIX socket in a machine).

Thus, when the compiler finds a `use_active_module` declaration, it defines the imported predicates as remote calls to the active module. For example, if the predicate *P* is imported from the active module *M*, the predicate would be defined as

```
P :- module_address(M,A), remote_call(A,P)
```

A remote call to an active module involves sending the predicate to its corresponding blackboard and waiting for its results to be posted to the blackboard. For this procedure the address of the blackboard of the active module must be known, and this is achieved by the predicate `module_address`/2.

The predicate `save_active_module`/3 saves the current code like `save`/1, but when the execution is started a blackboard is created whose address is the first argument of the predicate, and the expression in the second argument is executed. Then, the execution goes into an idle worker loop of reading execution requests from the blackboard, executing them, and returning the solutions back to the blackboard.

3.2 Using Active Modules: an Example

In this section we will show the implementation of a remote database server using the primitives introduced in the previous section, and how the server would be used.

The code for the server uses the primitive `save_active_module` to make the `dbserver` executable, assigning it address `alba:888`:

```
:- module(database, [stock/2]).

stock(p1, 23).
stock(p2, 45).
stock(p3, 12).

:- save_active_module(dbserver, alba:888, true).
```

At this point the executable "dbserver" would be started as a process ("`dbserver &`", at the unix level) and it would be ready for other modules to import it. The code of a module that use the previous active module could start like this:

```
:- module(sales)
```

```
:- use_active_module(database, [stock/2]).

module_address(database, alba:888).

replenish(P) :-
        stock(P, S),
        ...
```

Calls to `stock/2` in the previous module will be executed remotely by the active module "dbserver". Except for the `module_address` definition, the code would be identical if `use_module` replaced `use_active_module` (but not the execution, of course).

3.3 A Name Server for Active Modules

As a more complex example of the use of active modules, let us now sketch how a name server such as the one mentioned earlier could be implemented.

First we program the name server module that will be active, and whose address ought to be fixed and known. Assume we want it to be run in host `clip` at socket number 999.

```
:- module(name_server, [dyn_mod_addr/2, add_address/2]).
:- dynamic dyn_mod_addr/2.

add_address(Module, Address) :-
        retractall(dyn_mod_addr(Module,_)),
        assert(dyn_mod_addr(Module, Address)).

:- save_active_module(name_server, clip:999, true).
```

Then, we make a module that uses this, and that will be imported in the standard way by modules that want to use active modules and this name server. Note that the call to `dyn_mod_addr` below will be executed by performing a remote call to the name server.

```
:- module(locate_module_addresses, [module_address/2]).
:- use_active_module(name_server, [dyn_mod_addr/2]).

module_address(name_server, clip:999).
module_address(Module, Address) :-
        dyn_mod_addr(Module, Address).
```

Thus, modules which will become active and want the name server to be notified must proceed as follows. Note that again the `add_address` goal below will be in fact executed as a remote call to the name server.

```
:- module(flight_reservation, [find_connections/4]).
:- use_active_module(name_server, [add_address/2]).

find_connections(Origin, Destination, Date, Flights) :- ...

:- save_active_module(flight_reservation, Address,
                      add_address(flight_reservation, Address)).
```

Finally we show how to import active modules managed by the name server:

```
:- module(travel_agency).
:- use_module(locate_module_addresses, [module_address/2]).
:- use_active_module(flight_reservation, [find_connections/4]).
    ...

airplane_trip(from_to(Origin, Destination), Date, Trip) :-
        find_connections(Origin, Destination, Date, Flights),
    ...
```

In this case, the call to `find_connections/4` will be executed as a remote call to the active module `flight_reservation`.

4 Conclusions

We have presented the current prototype of the distributed CIAO system, introducing the concepts of "teams" and "active modules" (or active objects). These concepts conveniently encapsulate different types of functionalities desirable from a distributed system, from parallelism for achieving speedup to client-server applications. We have presented the user primitives available, sketching their implementation. This implementation uses attributed variables and, as an example of a communication abstraction, a blackboard that follows the Linda model. An interesting characteristic of the implementation is that it is done enterely at the source (Prolog) level. We are currently working on adding new functionality to the system. The code is also being provided as a public domain standard library for SICStus Prolog and other Prolog systems (please contact the authors or `http://www.clip.dia.fi.upm.es` for details).

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] F. Bueno. The CIAO Multiparadigm Compiler: A User's Manual. Technical Report CLIP8/95.0, Facultad de Informática, UPM, June 1995.

[3] 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.

[4] 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.

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

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

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

[8] P. López García, M. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In *Proc. of First International Symposium on Parallel Symbolic Computation, PASCO'94*, pages 133–144. World Scientific Publishing Company, September 1994.

[9] M. Hermenegildo, D. Cabeza, and M. Carro. Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming. In *Proc. of the Twelfth International Conference on Logic Programming*, pages 631–645. MIT Press, June 1995.

[10] M. Hermenegildo and the CLIP group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, LNCS 874, pages 123–133. Springer-Verlag, May 1994.

[11] M. Hermenegildo and the CLIP group. The CIAO Multiparadigm Compiler and System: A Progress Report. In *Proc. of the Compulog Net Area Workshop on Parallelism and Implementation Technologies*. Technical University of Madrid, September 1995.

[12] M. V. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In *Fourth International Conference on Logic Programming*, pages 556–575. University of Melbourne, MIT Press, May 1987.

[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 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.

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