In *Proceedings of the 1996 APPIA-GULP-ProDE Conference on Declarative Programming*

Implementing 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

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. The user primitives available are presented and their implementation described. This implementation uses attributed variables and, as an example of a communication abstraction, a blackboard that follows the Linda model. Finally, the CIAO WWW interface is also briefly described. The functionalities of the system are illustrated through examples, using the implemented primitives.

1 Introduction

Many portions of applications of interest currently have a distributed nature. Concurrent, distributed, or agent-based computation in LP and CLP has been the subject of much research, including the early concurrent logic languages and more recent approaches based on the "Concurrent Constraint" (CC) programming paradigm [18], as well as "distributed" or "blackboard" LP and CLP systems (e.g., [2, 3], and Prolog systems incorporating Linda [8, 1]). Our purpose in this paper is to address the issue of building (agent based) distributed applications which use the programming styles of both of the major paradigms mentioned above, while using to the extent possible existing LP and CLP systems. In particular, we describe a simple, distributed implementation of the CIAO language [12, 4, 13], a concurrent constraint language that is backwards compatible with traditional LP and CLP systems (basically Prolog, extended with constraints, parallelism, and concurrency). The resulting distributed CIAO system provides distributed execution capabilities in both the (C)LP and CC styles of programming.

Distributed execution can be used for various purposes: one is to build distributed networks of concurrent, communicating agents. Another one is to exploit coarse-grained parallelism in a distributed environment with the objective of obtaining execution speedups. Yet another, quite different purpose, is to request and/or provide remote services in a distributed communication network (as is often done by WWW

*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.

servers). Intuitively, the first two models involve several agents cooperating to run the same application. This is addressed by the notion of "team" of workers in CIAO. The last model is supported in CIAO by the notion of "active module" (or active object), and, also, by the WWW interface. We believe that these proposed concepts conveniently encapsulate many different types of functionalities desirable from a distributed system.

In the following we describe the current prototype of the distributed CIAO system by describing each concept (Teams, Active Modules, WWW Interface), presenting the user primitives available, and in some cases discussing their implementation. The particular implementation presented uses attributed variables [17, 15, 6, 14, 9] and, as an example of a communication abstraction, a blackboard that follows the Linda model [8]. An interesting characteristic of this implementation is that it is done entirely 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 [7].

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 (used in the following primitive).
- `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

[1] The CIAO compiler includes a granularity control system [16].

example, by the &-Prolog parallelizing compiler [5]. 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 (or constraints placed) on the shared variables are not seen until the threads join (that is, there is an implicit `copy_term` in the goals sent out).

- `A &` – Sends out goal `A` to be executed potentially by another worker of the team. No waiting for its return is performed. Updates on 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.
- `A @ Id` – Placement operator. Allows control of task placement in distributed execution: goal `A` is to be executed on worker `Id` (which may be remote). This operator can be combined with any of the parallelism and concurrency operators mentioned before.

Members of a team can communicate among each other either by shared variables or explicitly by means of a blackboard. This blackboard provides a set of Linda-like [8] 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 obtain goals, which would have been posted previously by the primitives `&>`/2, `&`/1, or `&&`/1, in order to 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 [11].

The distributed communication using shared variables follows the lines proposed in [10], where some of the proposed operators were already presented.[2] 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 (or constraints) 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, transforms these builtins into code that that automatically posts them 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.

In order to be more concrete we sketch part of the code of the implementation outlined above. The code presented is a simplified version of the actual code. The implementation of the primitives intended for independent And-parallelism is shown below. For simplicity, we show the version in which the answers are returned together. The (more efficient) alternative is to post to the blackboard each solution right after computing it, so that the continuation can proceed concurrently with the computation of the additional answers.

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

Q &> H :-
        % start blackboard if not done yet
        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).
```

[2] The discussion in presented primarily in terms of bindings, but the techniques are extensible to (variable based) constraint synchronization, specially if constraint handling is performed also via attributed variables, as in the CIAO system.

```
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).
```

Note that the builtin undo/1 above executes its argument on backtracking, as if defined by

```
undo(_).
undo(G) :- call(G), fail.
```

Now, we show the main loop that idle workers execute to get work from the blackboard, and how the $query requests produced above are handled.

```
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('$query'(N,Q),Id) :- !,
        findall(Q,Q,Answers),
        out('$answers'(N,Answers)).
...
```

Note above the use of the $idle token which ensures that if a && primitive chooses an idle worker this worker will execute the corresponding concurrent goal.

2.2 Using Parallelism: an Example

In this section we will show an example of the uses 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, processes the work, stops the started workers, and exits. The important effect 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([]).
```

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 daemons: Prolog executables which are started as independent processes at the operating system level. For simplicity we will assume communication with active modules is also implemented by means of a blackboard.[3] 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] In practice, in the case of teams with only one worker it is obviously more efficient to simply connect directly with the active module via a socket.

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 uses 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 uses 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. The "state" of the name server is implemented using the database:[4]

```
:- 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

[4]Note, however, that it could alternatively be implemented as a perpetual process, using the concurrency and communication primitives presented in the previous section.

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 Interfacing with the WWW

We would like to finalize with some comments on an issue which is also related to distributed execution and remote information access: interfacing with the WWW. Using similar techniques to those illustrated in the previous sections, we have implemented a publicly available WWW library for LP/CLP systems which enables convenient WWW access to and from programs written with current LP and CLP systems. This library provides several functionalities found to be useful in the development of LP/CLP-based WWW applications.

HTML to Herbrand syntax conversion. We provide a means of converting Herbrand terms, which are easy to manipulate in an LP/CLP system, into HTML text and viceversa. Herbrand to HTML conversion is obviously useful when a program produces output to be parsed by browsers. HTML to Herbrand conversion is useful when reading remote pages, which are often in HTML format, and manipulating their contents (for example, to perform an intelligent keyword analysis of the text within or to find pointers within the document and in turn follow them). The translation facilities should support for the creation of forms. Three builtins are provided for the task:

- `output_html(T)` – Accepts in `T` a term representing HTML code and sends to the standard output the HTML text it represents.
- `html_term(T,L)` – Accepts in `T` a term representing HTML code and produces in `L` a list of atoms which are the traduction of the term. Used by `output_html/2`.
- `parse_html(S,T)` – Accepts in `S` a string (list of characters) containing HTML code and produces in T the Herbrand term which represents it.

An example of a term representing HTML code is:

```
html([title('My page'), image('photo.gif'),
      heading(1,'Hi there'), --, 'Hello, ...'])
```

Form output parsing. It is often necessary to parse the output from forms. Such output is provided, as defined by the HTTP protocol, in a format that is reminscent of the attribute-value pair list that is usual in a symbolic language, but in a different syntax. The following builtin is provided to perform this conversion:

- `get_form_input(Dic)` – Translates form input provided by GET or POST http requests to a dictionary `Dic` of `Attribute=Value` pairs.

HTTP/FTP protocol support as a builtin. A builtin is provided which gives the programmer facilities to download documents from the net via the FTP and HTTP protocols, among others. By means of a list of options, it provides many levels of functionality.

- `get_url(URL, Opts)` – Gets the document pointed to by `URL`, and processes it according to the list of options `Opts`. Some of the options provided are:

  `content(L)` – Unifies `L` with list of characters that corresponding to the document (which can then perhaps be parsed by the HTML parsing primitives mentioned above).
  `file(F)` – Writes the document into file `F` (useful for example for caching)
  `size(S)` – Unifies `S` with the document size.
  `type(T)` – Unifies `T` with the type of the document.
  `date(D)` – Unifies `D` with the date of the document.

It is worth mentioning that the predicate determines whether a document or just its header needs to be fetched depending on the options passed (e.g., the last three options only need the header).

Standard WWW interface to active modules.
A standard form handler ("cgi-bin" application) wich can connect to one or several active modules (as presented in the previous sections) provides a generic human interface to such modules (allowing querying and modification) via the WWW.

Access to remote modules via WWW. A URL address can be provided in a standard module declaration instead of the usual file address. This allows a program to import code from a URL, so that when the module is updated the program also gets updated.

Another useful feature, in many areas but specially in the context of the WWW, is the possibility of executing Prolog scripts: i.e., simple Prolog executable files which run without need for compilation. While the intrinsic characteristics of LP and CLP systems (for example, very easy parsing via grammar rules) make them quite convenient implementation vehicles for cgi-bin applications (i.e., applications accessible by the WWW), the often large size of the resulting executables and the need to compile or consult in most systems may deter cgi-bin application programmers from using these systems. In fact, often, shell scripts or interpreted languages such as Perl are used for producing small to medium-sized cgi-bin applications, mainly for the convenience of not having to compile the source file. It appears convenient to provide a means for LP/CLP programs to be executable as scripts, even if with reduced performance. We provide a separate library which fulfills this need.

The interested reader is referred to the WWW address

`http://www.clip.dia.fi.upm.es/miscdocs/html_pl/html_pl.html`

for more details and the source for the WWW library. An example of an application developed with this library can be found at

`http://www.clip.dia.fi.upm.es/miscdocs/webchat_info.html`

5 Conclusions

We have presented the current prototype of the distributed CIAO system, introducing the concepts of "teams" and "active modules" (or active objects), and some details of the CIAO WWW interface. 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] 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. The CIAO Multiparadigm Compiler: A User's Manual. Technical Report CLIP8/95.0, Facultad de Informática, UPM, June 1995.

[5] 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.

[6] 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.

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

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

[15] S. 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.

[16] 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.

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

[18] E. Tick. The Deevolution of Concurrent Logic Programming Languages. *The Journal of Logic Programming*, 23(1–3):89–125, 1995.