A Distributed Logic Programming Language and its Implementation on Transputer Networks

José A. Cardoso Cunha, Pedro A. Duarte Medeiros, Luís M. Pereira

Universidade Nova de Lisboa, Departamento de Informática
Faculdade de Ciências e Tecnologia

Abstract

Currently there is a trend towards the development of programming tools and mechanisms for the support of heterogeneous multi-agent systems on paralell computer architectures. This paper presents a contribution to this area, as far as logic programming on a distributed execution environment is concerned. We discuss the main issues on the design and implementation of the logic programming language Delta Prolog [2] [3] [6] [7] [11], extending Prolog with constructs for concurrency and communication. The work described is one of the research components of a project on the development of mechanisms for parallel logic programming support on parallel architectures, currently running in this University [8].

1. Delta Prolog: the language

In this section a simple example is used to review the main language constructs, along the following relevant dimensions: specifications of sequential and concurrent composition of Prolog goals; communication and synchronization between processes; local non-determinism (i.e. within each Prolog process) and global (or external) non-determinism (allowing the choice among multiple communication alternatives).

1.1 _-Prolog programs

A _-Prolog program is a sequence of clauses of the form: H :- $G_1,...,G_n$. (n_0). The *comma* is the *sequential* composition operator. Declaratively, the truth of goals in _-Prolog is order-dependant, so that H is true if $G_1,...,G_n$ are true in succession.

Whereas H is a Prolog goal, each G_i may be either a Prolog or a _-Prolog goal.

A _-Prolog goal is either a *split* goal (for parallelism), an *event* goal (for inter-process communication) or a *choice* goal (for external non-determinism). A _-Prolog program without _-Prolog goals is and executes like a Prolog program, so _-Prolog is an extension to Prolog.

The programming model relies on the programmer to specify the sequentiality constraints and the desirable parallelism existing in each problem (using the comma and the split operators), and the corresponding communication schemes (through event and choice goals).

The paper assumes some knowledge of logic programming languages, such as Prolog. An informal introduction to the language constructs follows.

1.2 Illustration of the main language constructs

Published in the Procs. of 12th Technical Meeting of Occam User Group, Exeter, IOS Amsterdam, April 1990.

Consider a problem that is amenable to a decomposition into three separate processes (or agents), which we will denote by P, I and C. This is an usual configuration whenever we have an intermediate processing to be performed (by I) on any items that are being produced (by P), and eventually must be sent to a consumer process C.

(i) Specifying the parallel composition of multi-agent systems

The concurrent execution of the three processes above may be specified by the top goal *prod(...) // filter(...) // consume(...)*, where // is a right associative *parallel* composition operator and *prod(...)*, *filter(...)* and *consume(...)* are the goals to be solved by each agent (as a separate Prolog process, except that communication is possible, cf. below).

Declaratively, S_1 // S_2 is true iff S_1 and S_2 are jointly true. Operationally, solving S1 // S2 corresponds to a concurrent resolution of goal expressions S1 and S2, i.e. to an arbitrary interleaving of their resolution steps.

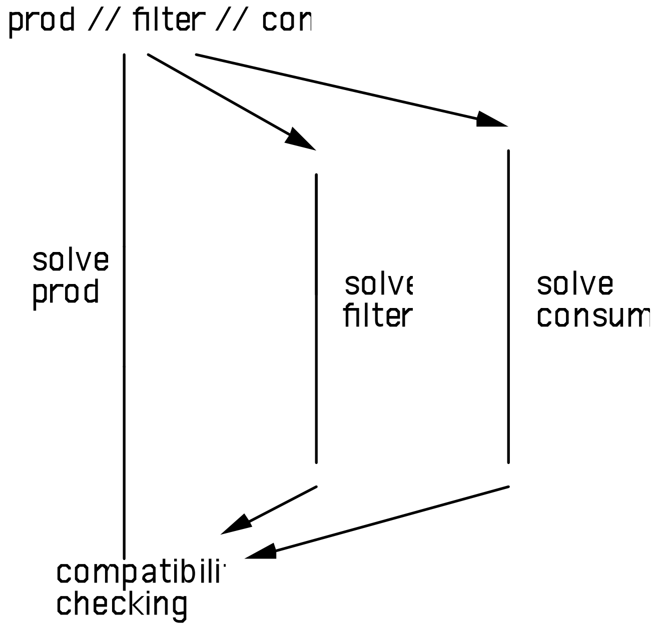

Figure 1

(ii) Specifying sequential goal composition

The sequential evolution of a system may be specified by the comma operator, e.g. within the filter process:

```
filter(...) :- communicate_with_P(...),
        intermediate_processing(...),
        communicate_with_C(...).
```

Operationally, to solve goal ′filter′ solve successively the three goals in the clause body.

(iii) Specifying communication between two agents

Communication in _-Prolog is expressed by *event goals* . This is a message-based communication mechanism where two concurrent processes jointly solve a pair of goals (one in each process) of the form T? E and T! E; the goals must be complementary, i.e. one uses symbol ? and the other uses !, and E refers to the same name. T is a Prolog term (the *message*), ? and ! are infix binary predicate symbols (the *communication modes*), E is bound to a Prolog atom (the *event name*).

In our example:

```
prod(...) :- produce_item(X), X ! event_P.

filter(...) :- X ? event_P,
        intermediate_processing(...),
        communicate_with_C(...).
```

A solved event goal can only be said that it was true *when* it solved with its complementary goal. The declarative semantics of _-Prolog does not assign an "absolute" truth value to a goal. In general, a goal is true for some combination sequences of events (or traces) that were true, and false for others.

Operationally an event involves a form of "rendez-vous" of the two processes solving the goals X ? E and X ! E, with exchange of messages achieved by unification of X and Y: an event goal, say X?E , solves only with a complementary event goal, say Y!E, "simultaneously", whenever the latter is available in a parallel derivation (unavailability causes resolution of the goal to "suspend"). Note that failure of unifying the terms in both event goals leads to a failure which must be handled by a specific computation strategy as explained in a section below.

Term unification provides the possibility of a two-way exchange, e.g. in goals ´m(1,X)!e´ and ´m(Y,2)?e´ one process would obtain the substitution X = 2 and the other one would obtain Y = 1.

The involved processes share no common variables.

This basic synchronization mechanism of _-Prolog generalizes Hoare's and Milner's synchronous communication (Hoare 85; Milner 80) by using term unification for message exchange. Unlike CSP and Occam, no special significance is attached to the communication modes ! and ? (like "send" or "receive"), except that they are complementary in the sense above.

(iv) A complete program

```
p(X) :- prod(X) // filter // cons .   % top goal

prod(X) :- a(X), X ! event_P.         % produce a binding for X and then communicate.

a(2).                                 % unit clauses for predicate a(X).
a(3).
a(4).
a(5).

filter :- X ? event_P, (X>1, X<5), X ! event_C.
                                      % communicate, check received item, and communicate.

cons  :- X ? event_C, consume(X). % communicate and then consume an item.
```

The solutions for the top goal p(X) and the program above correspond to the substitutions {X=2}, {X=3} and {X=4}, obtained by successful resolution of event goals X!event_P and X?event_P in the communication between the producer and the filter, and of event goals

X!event_C and X?event_C in the communication between the filter and the consumer. The substitution {X=5} is ruled out by the condition in the definition of the filter predicate.

A declarative reading of this program allows us to define the solutions with no concern to any operational execution model. Several computation strategies are possible for _-Prolog as discussed in section 2.

(v) Non-determinism in _-Prolog programs

In _-Prolog the selection of the matching clauses for resolution with a selected Prolog goal is like in Prolog sequential systems (i.e. uses the textual occurrence of the clauses in the program), being independent of the state of the environment (cf the definitions for the ′a′ predicate in the example above). Therefore to allow the programming of applications where multiple communication alternatives may be simultaneously available whose selection depends (non-deterministically) on the environment, we introduced choice goals.The choice operator :: is based on its namesake introduced in (Hoare 85) (cf. also ALT construct in Occam).

These have the form $A_1 :: A_2 :: ... :: A_n$ (n_2), where :: is the *choice* operator, and the A_i are the *alternatives* of the goal. Each alternative has the form G_e,B , where G_e is an event goal (the *head* of the alternative), sequentially conjuncted to a possibly empty goal expression B (the *body* of the alternative).

Declaratively, $A_1 :: A_2 :: ... :: A_n$ is true iff at least one alternative is true.
Solving a choice goal consists in solving the G_e of any one alternative (whose choice is governed by the availability of a complementary goal for its G_e), and then solving its body B. If no complementary events are available for any alternative the choice suspends.

We illustrate this by modifying the example above and using a buffer process for the intermediate agent I:

```
buffer([]) :- X? event_P, buffer([]).
buffer([X|T]) :-
        (H?event_P, append([X|T], [H], NewBuffer), buffer(NewBuffer)
        ::
        X?event_C, buffer(T)
        ).
```

where the definition of append(L1,L2,L3) (L3 is the concatenation of lists L1 and L2)is omitted.

A top goal of the form *prod // buffer([]) // cons* specifies a configuration as shown below:

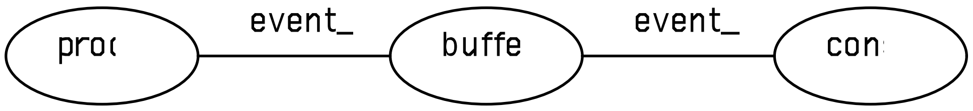

Figure 2

The buffer process non-deterministically chooses between participating in an event with the producer process (and then appending a received item to the buffer) or participating in an event with the consumer process (by offering the item at the head of its list).

1.3 Notes on the language model

We focused on the following features of Delta Prolog:

(i) the language subsumes Prolog, allowing us to gain from the enormous accumulated experience on Prolog programming;

(ii) it has a well-defined semantics, allowing the programming of distributed applications in a logic framework, with a clear declarative semantics;

(iii) Delta Prolog's communication model is based on the theoretical models of CCS [5] and CSP [4], and extends the latter with two-way synchronous communication of Prolog terms (as messages) between two processes (communication through event goals)

2. Delta Prolog: the execution models

We discuss the computation strategies and execution models that guide the implementation of Delta Prolog systems.

2.1 Operational execution models for Delta Prolog

First we informally introduce the notion of derivation for a goal expression and a program, and then discuss computation strategies for completely exploring the derivations' space.

Given an initial configuration of processes, as specified by a top goal and a _-Prolog program, the evolution of the system may be pictured as follows, for the example in 1.2 (program iv):

Figure 3

Each node represents a resolvent being executed by each individual process. An arc may be generated[1], from a current node within a process, leading to a new node (that will represent the newly obtained resolvent), whenever the expansion of the leftmost goal in the corresponding resolvent is possible (the cases of split and choice goals are omitted for simplicity [7]):

-- for a Prolog goal this occurs if there is a matching clause and unification is successful.
-- for an event goal, a complementary event goal must be available on another process, and term unification must be successful.

Each process performs independent derivation steps, as long as the resolution of Prolog goals is involved, thus working exactly like a sequential Prolog system. A process reduces to ´true´ when it solves its initial goal with success.

When evaluating event goals a joint derivation step must be performed, as illustrated by the thick line between processes. On a sucessful event, both processes proceed with their individual computations reflecting the outcome of the exchange (in the example the filter process got its private variable X instantiated to 2).

The solutions to a given top goal and program correspond to the configurations where all the processes have reduced to true.

Consider the following configuration:

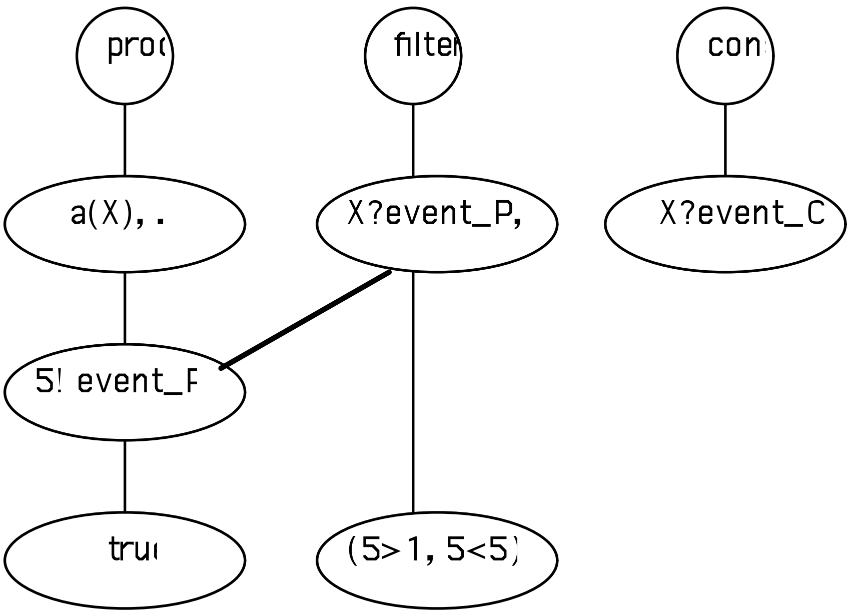

Figure 4

This is a configuration where a received item by the ´filter´ process, corresponding to the binding of X to 5, does not satisfy the local condition (X>1, X<5). So, although process ´prod´ has reduced itself to true, the remaining processes are not expandable anymore (´filter´ fails while

[1] The arc enumeration is explained in section 2.2.

´cons´ hangs waiting for a partner). The shown configuration is unsucessful, and a correct computation strategy must abandon it, and look for alternative configurations.

A complete computation strategy for _-Prolog must generate all the configurations corresponding to the defined solutions of a given top goal and program.

A sequential strategy is based on a single executor while a concurrent strategy allows the parallel execution of multiple derivation steps by distinct executors. An informal presentation of both strategies is given in the following. A more detailed presentation is given in [2].

2.2 Sequential execution of Delta Prolog programs

A sequential execution scheme simulates concurrency, as defined in a program, by a single (threaded) interpreter using a co-routining strategy. In fig. 3 a possible ordering of expansion of the nodes is shown, corresponding to an execution scheme where the interpreter tries to expand the leftmost active process until it ends or reaches an event goal with no partner process currently active. In the latter case the process suspends and control is switched to another process.

On failure of expansion of the current node (e.g. see configuration in fig. 4), a backtracking strategy uses the reverse order of node expansion in its search for alternatives.

A deadlock configuration, where all the existing processes are suspended waiting for communicating partners, is easily detected by the interpreter, and recovery uses the same reverse order as in the failure situation.

2.3 Concurrent execution of Delta Prolog programs

Parallel evolution of an initial configuration is achieved by coordinated execution of distinct threads of control, currently implemented as separate interpreter instances, each one being responsible for the expansions of an individual process, locally following Prolog's sequential search strategy.

The cooperating Delta Prolog processes must achieve agreement on the (possibly intermediate) results of their local computations, by exchanging unifiable terms on communication points (events). If they do not reach this agreement, or if one of them locally backtracks till a previous communication point, then a strategy is devised where alternative local computations are tried, aiming to provide a solution to a top goal (i.e. the original problem to be solved).

Failure of Prolog goals is handled by standard backtracking. Failure of _-Prolog goals is governed by an innovative distributed backtracking strategy, which triggers a mechanism for searching alternative configurations.

Two aspects are thus involved:

- a local search is performed by each process relative to the doing and undoing of derivation steps of Prolog goals; this follows Prolog's depth-first search and uses local backtracking within each process; a gain is obtainable here, vis-à-vis a purely sequential system, as we have multiple Prolog processes that may be executing in parallel both in the forward and backward directions;
- a global coordination of the search is required whenever a joint derivation step (involving an event or a split goal) must be done or undone; this follows a distributed backtracking strategy demanding the cooperation of all processes that may depend on the joint step.

The main purpose of this distributed backtracking strategy [2] [7] is to implement an exhaustive search for the set of successful derivations for a program and goal, where a distinction between local and global search is made and no centralized control component is required. The computation path of each process is sliced down into consecutive segments, identified as derivation paths between two consecutive _-Prolog goals. The concept of segment allows concentrating on interaction points only, and ignore local backtracking within segments (which is dealt with by each process alone). An ordering amongst segments is defined that guides an exhaustive search through all alternative derivations.

The global strategy is invoked only when a process, in local backtracking, reaches a previously solved _-Prolog goal, or when the conditions for success of a synchronous event fail. Issues of handling multiple failures by concurrent processes, and the handling of deadlocks are not detailed here ([2]).

Experimentation with this novel strategy for distributed backtracking finds its motivation in A.I. applications where a coordination is required for systems of cooperating problem solvers (each written in Prolog style). One may see it as an extension of the backtracking facility currently provided by existing sequential Prolog systems, onto distributed execution environments.

In order to ease the experimentation with a wider range of applications, Delta Prolog currently allows for parallel execution and inter-process communication where no distributed backtracking strategy is automatically implied if a failure occurs on a communication point [3]. This is the case for systems programming where deterministic processes are usual (e.g. no backtracking is usually required for a process directly interacting with the real world).

3. Implementation issues

Implementation issues are discussed by defining a system layer for low-level process control and port-based communication, suitable for the support of Delta Prolog constructs. The implementation is organized in three layers:

(i) Delta Prolog or other distributed processing paradigms.
(ii) Port-Prolog layer.
(iii) System layer.

The System level provides the operating system support (handling low-level system configuration and process creation, control and inter-process communication), offering an interface for C programs.

Port-Prolog [1] supports the execution of multiple Prolog processes communicating through ports, offering a generalized i/o interface. Each Prolog process may synchronously or asynchronously send / receive Prolog terms through ports. This layer offers a portability level for the experimentation with concurrency and communication constructs at the language and application levels.

Besides its use for Delta Prolog implementation, layer (ii) is also being used for the development of distributed transputer-based environments for systems of multiple agents (programmed as Prolog processes) in Robotics applications [9] [10].

3.1 Delta Prolog level

There are two approaches for the implementation of the abstract models of execution discussed in section 2:

i) An interpreter-based approach, obtained through the extension of existing Prolog interpreters. This offers portability and debugging facilities and allows the writing in Prolog of almost all the code supporting the _-Prolog construct.

ii) A compiler-based approach, aiming at efficiency, consisting of a compiler (basicly extending a Prolog compiler with the processing of _-Prolog goals) plus a _-Prolog abstract machine (e.g. preferably based on a Prolog abstract machine [12] suitably extended for the support of split, event and choice goals).

3.2 Port-Prolog interface layer.

This is an intermediate system layer providing a set of primitives for process control and inter-process communication and assuring portability to distinct operating system environments.

It supports facilities for:

i) virtual _-Prolog process (thread) creation

ii) port-based inter-process communication, which generalizes the i/o interface of a conventional Prolog process

iii) an inter-process signaling mechanism which may or not generate an interruption

iv) mechanisms for the control of a sequential Prolog machine, namely a facility for the identification of specific activation frames in the stack and for the forced backtracking of a thread to a specified stack frame.

This allows alternative implementations on different environments:

- in a single-process model all the virtual _-Prolog processes (in a co-routining fashion) share the same address space,
- in a multiprocessed model under a single-processor operating system (e.g. Unix) they are implemented by relying on the underlying system primitives
- in a multiprocessed model under a multiple-processor system, they may exploit a shared-memory communication model or use the network communication facilities.

We will now give a brief description of Port-Prolog mechanisms.

The process concept

From our point of view a process provides the support for the execution of a Prolog program, regardless of the way this is actually implemented (i.e. a Prolog interpreter or a compiler and intermediate machine emulator). We consider the following components in the process concept:

• a built-in Prolog (virtual) processor, responsible for the Prolog computation, following a sequential execution model, and using depth-first search with backtracking;

• a set of input/output channels establishing the communication of the process to the outside world;

• a Prolog program (a set of definite Horn clauses) is specified to be associated with the process, on its creation and activation.

We provide several operations on a virtual processor, regarding the control of the execution, and on the input/output channels, regarding the communication between independent Prolog processes. No special operations will be discussed regarding the virtual memory abstraction, as consisting of the Prolog workspace (local and global stacks, and trail areas) and the memory for the clauses of the Prolog program. Further we assume no common memory between the processes.

We want to be able to model systems of cooperating Prolog processes, communicating through well defined interfaces. This communication is supported by an extension of the i/o interface of a Prolog process to encompass the notion of communication ports, i.e. inter-process communication channels for message passing. Thus our model applies to a distributed environment with no shared memory, in which the distinct processes may be executing on separate physical processors.

The Prolog interface to an operating system environment is defined in the form of a set of system predicates .

The creation of a Prolog process is achieved by invoking the following system predicate:

forkp(P, I, O)

where **P** stands for the process name, **I** and **O** stand for the standard input and output channels for that process.

A new instance of a Prolog virtual processor is created when **forkp** is invoked. This means that a skeletal environment for a Prolog computation is thus created. The process will start up and hang waiting for goals to be sent to it. A separate operation will be required in order to specify a set of clauses for the process memory, typically by sending the name of a file to be consulted by the new process.

Sending signals to a Prolog process

The synchronization of the processes in a Port-Prolog program is achieved through the use of ports, as explained above, and through the use of specific mechanisms for signaling events to a Prolog process. Our signaling mechanism supports, in an uniform way, two different semantics for the synchronization of independent processes, namely through the polling of event counters or through the delivering of signals with process interruption, the interrupt handlers being specified by Prolog predicates.

Communication ports

We introduce a special device for inter-process communication allowing for the message passing in the form of Prolog terms. The name of a port is system-wide, i.e. is unique within a system of cooperating processes. Processes communicate through ports by sending and receiving Prolog terms. This is the only data structure currently supported for information transfer through ports. We provide several variants (blocking, non-blocking, and receiving on a set of ports).

A process may send a term by invoking the predicate:

send(M, T)

where M is a port name and T stands for a Prolog term.

A process receives a term from a port, through the predicate:

receive(M, T)

where M is a port name.

Locks

For simplicity and flexibility a mutual exclusion mechanism is supported based on locks, with the semantics of binary semaphores.

3.3 Implementation issues

The first _-Prolog implementations were based on modified C-Prolog interpreters running on multiuser systems (VAX/VMS, Unix).

Each _-Prolog program was supported by multiple _-Prolog executors. They were mapped to separate processes in a multiprocessed operating system (e.g. Unix) for a single CPU computer, where each process executes an instance of a modified Prolog (e.g. a C-Prolog interpreter plus extensions for process creation and inter-process communication). Alternatively, they were supported by system processes running on distinct CPUs on multiprocessor machines (shared or non-shared memory).

We are currently developping alternative approaches. A sequential model is based on a single executor (process) that provides multiple execution contexts (each with its private stack, as in a sequential Prolog machine). Each virtual process in a _-Prolog derivation is mapped to one of those contexts.

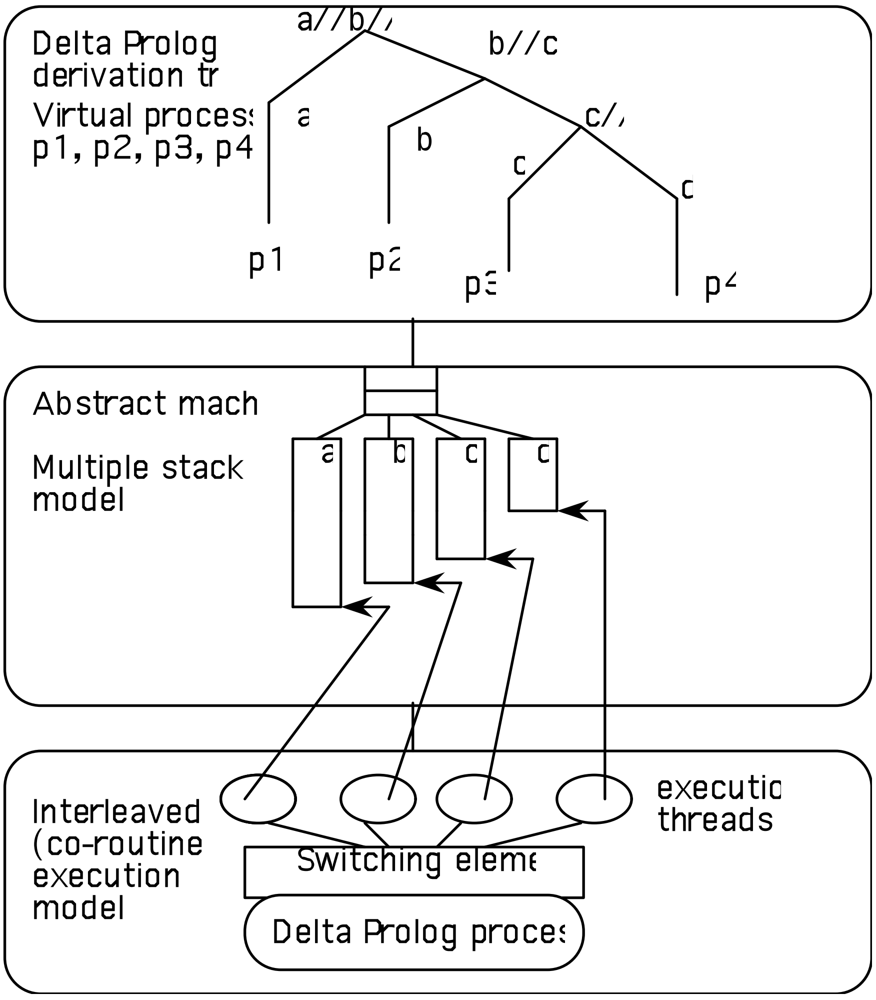

Figure 5

A single execution context is active at each instant. The remaining execution contexts are suspended on _-Prolog goals (events or completion of split goals), or they are ready but must wait for their turn. The switching element decides which virtual process will be executed by the single executor, at each point in a computation.

Parallel execution model and its implementation on transputer networks

The communication through event goals is particularly suited for transputer systems due to its similarity with the CSP/OCCAM_ model.

In a distributed _-Prolog system synchronous events require an inter-node communication facility in order to implement rendez-vous and term exchanges between two remote threads (thread intra-node communication is supported by the single-process address space). Distributed backtracking uses the same facility for the passing of control messages between threads, the relevant issue being the relative amount of inter-process communications across the network when compared to the amount of local computations (i.e. Prolog derivation steps or intra-node communications) performed by the threads in each node.

Suitable models for the structuring of the concurrency in _-Prolog programs may help in localizing the amount of distributed backtracking in each node and among the nodes, e.g. by creating teams of threads if they exhibit heavy interactions, and mapping the corresponding threads into the same nodes.

A suitable execution model for a network of transputers addresses issues of communication mechanisms and strategies for goal sheduling.

i) In each node we have a _-Prolog system, supporting multiple Delta Prolog virtual contexts. Work progresses on a single active thread at each instant, but a queue of pending threads is kept. A new thread is added to the queue whenever a split goal is activated.

ii) Several scheduling strategies may be conceived for such a system, allowing all the processors in a network to be busy working on existing threads. An example of such a scheduling policy would be one where a currently idle Delta Prolog processor, on a specific node, tries to execute a remote goal by stealing it from another processor queue of pending threads.

This is sketched in the following figure:

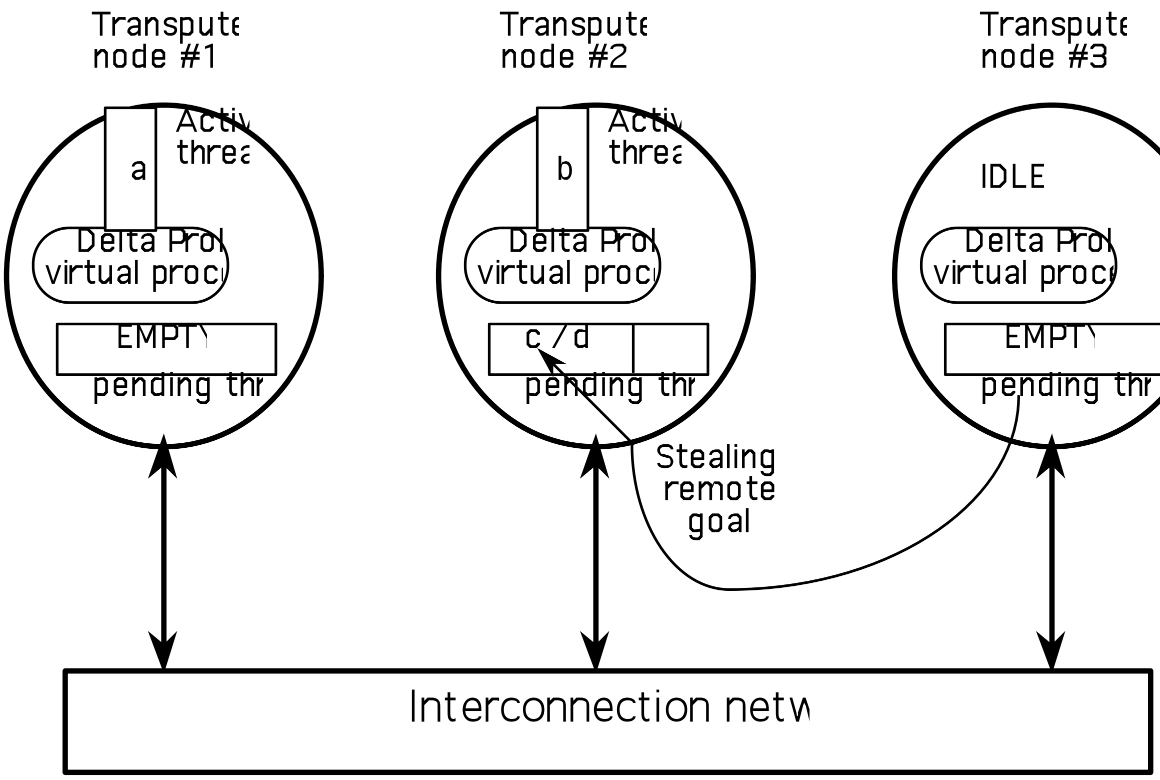

Figure 6

4. Current work

Our current prototype is running on a hardware configuration with 4 processors (cards with 1 T800 and 4 Mbytes of memory develloped at this University). The host is an IBM-PC/AT compatible and we are using 3L Parallel C for the development of the above layers.

At a first stage a system layer that provides the basic inter-process communication, file system and interactive input/output requirements was implemented on the above transputer prototype. Port-Prolog is currently running on top of this system layer.

The second stage corresponds to an implementation of a _-Prolog system that explores the suitability of the language model to a transputer-based architecture, with focus on an efficient mapping between the event goal execution requirements and the transputer model.

Acknowledgements

To colleagues of the Logic Programming and A.I. Group, and to A.Steiger-Garção and Manuel M. Barata of the Intelligent Robotics Group. To Luís Silva and João Cabral for their work on the implementation of the system level on the transputer prototype.

References

[1] Cunha, J.; Medeiros P.; Carvalhosa M. : Interfacing Prolog to an operating system environment: mechanisms for concurrency and parallelism control. Internal Report, UNL, Computer Science Department, April 1987.

[2] Cunha, J.: Concurrent execution of a logic programming language (in portuguese), PhD. Thesis, UNL, Computer Science Department, September 1988.

[3] Cunha, J.C., Ferreira, M.C., Pereira, L.M. 1989. Programming in Delta Prolog. (presented at the Sixth International Logic Programming Conference, Lisbon, June 1989). ICLP'89, MIT Press.

[4] Hoare, C.A.R. 1985. *Communicating sequential processes.* Prentice-Hall, Englewood Cliffs, New Jersey.

[5] Milner, R. 1980. *A calculus of communicating systems.* LNCS **92**, Springer-Verlag, New York.

[6] Pereira, L.M.; Monteiro L.; Cunha J.C.; Aparício J.N. : Delta-Prolog: a distributed backtracking extension with events. (Internal Report, UNL-29/85, Computer Science Department, November 1985), In "Proc. 3rd Int. Conf. on Logic Programming", Lecture Notes in Computer Science #225, pp 69-83, Springer-Verlag, New York, 1986.

[7] Pereira, L.M.; Monteiro L.; Cunha, J.C.; Aparício, J.N., Concurrency and communication in Delta Prolog. In IEE International Specialists Seminar on "The design and applications of parallel digital processors", pp, 94-104, Lisbon, 1988.

[8] Cunha, J.C.; Pereira, L.M; Medeiros, P.D.; Carvalhosa, M.B.; Paralogism: mechanisms for parallel logic programming support, UNL-33/89, Computer Science Department, July 89.

[9] Stgeiger-Garção, A. ; Barata, M. M.; Gomes, L.; Integrated environment for prognosis and monitoring systems support, UNIDO 1st Workshop, Lisbon, September 89.

[10] Barata, M.M.; Cunha, J.C.; Steiger-Garção, A. ; Transputer environment to support heterogeneous systems in Robotics, submitted for publication.

[11] Pereira, L., Nasr, R. : Delta Prolog: a distributed logig programming language, Procs. Fifth Generation Computer Systems. Tokyo.1984.

[12] Warren, D.H.D. : An Abstract Prolog instruction set. Technical Note 309, Artificial Intelligence Center, SRI International, Menlo Park. 1983.