IMPLEMENTATION ISSUES OF A DISTRIBUTED REAL-TIME LOGIC LANGUAGE

M. Díaz, B. Rubio, J.M. Troya
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga
Pza. El Ejido, s/n. 29013 Málaga - SPAIN
mdr,tolo,troya@lcc.uma.es

Abstract

The integration of different paradigms within the declarative framework has been a very active research area in recent years, especially in the field of concurrent and distributed extensions for declarative languages. On the other hand, the high complexity of distributed real-time systems requires extra effort in the design of new languages, methodologies and techniques suitable for these systems. In this paper, we present some implementation issues of a new language that integrates the real-time and distributed paradigms within the framework of a concurrent logic language. Concurrent logic languages (CLLs) are capable of expressing concurrence, communication and nondeterminism in a very natural way. That is, the intrinsic parallel semantics of the concurrent logic languages makes them well-suited for distributed programming. The proposed language is particularly suitable for loosely coupled systems and it contains mechanisms for distributed and real-time process control. A new execution model for concurrent logic languages is presented, which enables efficient distributed execution and real-time control. A new abstract machine for this model and its implementation on a Unix workstations network are presented.

1. INTRODUCTION

Distributed real-time systems are very complex and they have very strict safety ar reliability requirements. These systems are composed of collections of diverse computatio without central coordination and without a total knowledge of each other. Moreover, mo programming languages have not been designed to support such systems, but rather to des tightly coordinated computations. In the last few years, many new proposals for distributed a real-time systems have emerged [4]. Some of these models are based on traditional impera languages, but others are intended to provide new paradigms suited to the characteristic these kinds of systems [1]. In this sense, many different approaches for the integration distributed and real-time paradigms in declarative languages have been proposed [2][15].

Concurrent logic languages (CLLs) have appeared as an attempt to adapt Prolog to para systems [6], and its suitability for distributed systems has been demonstrated by Shapiro [1 In fact, CLLs have been used as kernel languages to construct declarative environments distributed programming [9][12]. CLLs have evolved out of these inefficient environments int new languages more oriented towards the characteristics of distributed systems. So languages which have come out from this evolution are RGDC [7], Strand [13], Sandra [1 and Janus [18], which despite being a language used for concurrent constraint programmi has the same basis of CLLs. Something that is common to all these proposals is that they tr overcome one of the drawbacks of CLLs in their use for distributed systems: their inefficienc Distributed implementations of CLLs are not very efficient, mainly because the communicatic mechanism is based on sharing logical variables and its implementation on distribut environments involves very high costs. We propose a new approach to distribute programming based on Parlog [6] and oriented to coarse granularity parallelism: the DF language. This language has been designed on the basis of extending CLLs to cover different aspects of complex real-time distributed systems. In this sense, a new execut model for CLLs is presented which provides efficient distributed execution and real-tim control.

In the present paper we describe the implementation of this new model, which is based o distributed abstract machine. In section 2 the most relevant aspects of the DRL language presented. In section 3 we introduce the abstract machine´s execution model and some re: of the implementation. Finally, in section 4, some conclusions are presented.

2. THE DRL LANGUAGE

In the following a basic knowledge of CLLs is assumed [6][19]. In this section we will describe the DRL language's basic features. The characteristics of this language are determ by two main factors: firstly, the language must preserve the declarative nature of CLLs ar secondly, the language must be efficiently executed in distributed systems. These objectiv and the drawbacks of these languages for their implementation on distributed systems, have us to extend CLLs in different ways. The main characteristics of DRL are: flat guards (t improve efficiency [13]), explicit mode declarations, real-time control (based on tempora guards) [10], exception handling and channel based remote communications.

The language's execution model is process oriented. Processes can be grouped into gra that is the minimum execution unit that can be executed in another processor. Proc behaviour is expressed by a set of Flat Guarded Horn Clauses (i.e. the predicates that appear in the guards are predefined). Real-time constraints are expressed by means of temp guards . A temporal guard is a guard with an after(T) primitive, where T expresses time seconds. Besides the utilization of temporal guards the language provides primitives that all the constraint of process execution time.

2.1. Remote Execution

In order to execute a set of processes in a remote processor it is necessary to group t into a grain. Grain definition has the following format:

grain grain_name(m_1,...,m_n). %(mi= ? in, ^ out, # constant)
grain_name(T_1,...,T_n)<-g_1,...,g_r: b_1,...,b_k.
<Local predicate definitions>
<Exception handling>

end grain.

A grain is executed by means of a remote metacall:

remote_call(Processor,grain_name(V_1,...,V_n), Status, Control).

Where:

-Processor is a processor's symbolic name where the grain will be executed or is integer indicating a logical processor number.

-Status is the list of states of the process that will be used in exception handling.

-Control is a list used to control the process and to propagate exceptions.

The predicate defined in the grain with the same name as the grain and all the ch processes will be executed in the processor indicated in the remote metacall. Communicat with that process will be achieved through logic channels instead of shared logic variables a the directionality of these channels is indicated in the definition of the grain. The process w not be executed until all constant arguments (declared with #) are known. A logic channel i list that will be instantiated incrementally. This list can contain: constant terms, terms wit logic variables (the transmision of terms with logic variables will be postponed until all of ther become ground) and terms including logic channels (variables annotated with ? or ^, permitti the dynamic creation of channels).

Communication between processes in the same grain is achieved through shared lo variables as in other CLLs. The Status and Control list of the remote metacall are conside logic channels too. Grain creation is dynamic, i.e. a grain can be created at any moment du execution and in any processor. The remote metacall Processor variable can be instantiate the name/number of a processor or to the terms: local (the process is executed in the s processor) and auto(X) (the system will decide where to allocate the grain).

Besides the remote metacall other primitives are defined in the language to make t development of distributed systems easier [10]. The environment is initialized from a proces (called root processor) and a system server is created in each processor in the network.

2.2. Exception Handling

Exception handling is achieved by means of a special predicate `excep(Status,Control)` whose arguments are the Status and Control logic channels of the remote metacall. Whe grain is created an exception handling process is created too. An exception is detected by predicate when the head of the Control list is instantiated. This variable can be instantia outside the grain by a process that shares the control variable with the remote metacall or by execution of the predicate `raise(Term)` by a process in the grain. When this predicate executed the head of the Control list is instantiated to Term. The exception process c instantiate the Status variable that can be consulted by a process that shares this variable i original processor. The exception process can create other processes and suspend or abo grain execution.

```
grain reader(#,^)
reader(Device,[Value|L])<-read(Device,Value,St), continue(Device,L,St).
mode continue(?,?,?).
continue(D,L,ok)<-reader(D,L).
continue(D,L,error)<-raise(error(D,L)).
excep([error(D,L)|Control],Status)<-reset(D), excep(Control,Status).
excep([abort],[aborted])<-stop.
end  grain.
mode control(?,^).
control(Disp,Read)<- wait(30,Ctrl),
                        remote_call(processor2,reader(Disp,Read),St,Ct
mode wait(?,^).
wait(T,[abort])<-after(T):true.

<- control(disk,L).
```

The above DRL program shows an example where a process reads a device that retur status (ok or error). This process is executed in a remote processor and the read data is retu to the original processor. The process is aborted after 30 seconds.

2.3. Real-Time Control

Concurrent logic languages do not provide any mechanism to express real-time constrai in the execution of processes. Sandra [11] introduces the notion of logical clocks to expre time in the field of CLLs, but the concept of time in Sandra is not actually real-time. The logica clocks only have the task of ordering the occurrence of events in the system. Another appro in the field of declarative languages is that of Erlang [2], a functional language that evolve from Strand. This language can express real-time constraints, but its semantics is at pres very different from that of CLLs.

As the execution model of CLLs is based on interleaving, the process scheduling orc cannot be known. We propose an extension of CLLs that allows expression of process timi constraints. This extension implies a modification of the execution model. In CLLs a proce may suspend in a non-instantiated variable, during matching of the clause head or in gu evaluation. In our extension, processes may also suspend in the evaluation of timed guard timed guard is a guard with an `after(T)` primitive, where `T` is a term `msec(Seconds,Miliseconds)`. This primitive makes the process to suspend for the time expressed in the term `msec(Seconds,Miliseconds)`, after the matching of the head and th guard evaluation. If there is no other clause that commits before that time has elapsed, then clause is selected to reduce the process. A clause that contains a timed guard is called a ti clause. In the example below two processes are defined: a producer, `p`, and a consumer `c` communicate via a shared variable `L`. There is a timed guard in the definition of consumer, t detects a time out error if no data arrives in the channel within a ten second period.

```
mode p(^).                      mode c(?,?,^)
p([item|L])<-p(L).              c([X|L],T,St)<-c(L,T,St).
                                c(L,T,St)<-after(T):St=time_out.
                <-p(L),c(L,10,St).
```

When a timed clause exists in the definition of a process, this process must be selected reduction as soon as possible, i.e. that process has higher priority than ordinary ones, wit a timed guard. If this does not happen then the time expressed in the guard is not real. example, let us assume the following scenario, where p and c are the processes defined a and r is a process independent from p and c.

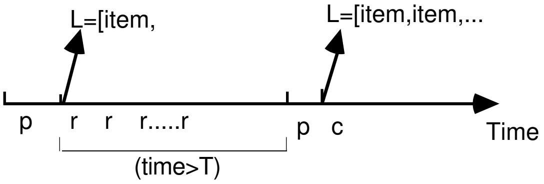

Fig. 1

The consumer process does not detect that the producer has taken more than T time uni produce the second item of the list, because different instances of the r process have b scheduled meanwhile. However if the c process had been scheduled before the r process situation would have been detected. Besides the temporal guards, other primitives have b included in the language:

- `init_clock`: Initializes the local clock.
- `clock(msec(Sec,Msec))`: Returns in Sec and Msec the time elapsed since the last clock initialization.
- `delay(msec(Sec,Msec))`: A delay of Sec seconds and Msec miliseconds. The process is suspended during the time units indicated by its argument.
- `timeout(Pred,Time,Status)`: A metacall executes the process indicated by Pred limiting its execution time and returning the Status (success, failed, timeout).
- `alarm(Pred,Time,Ctrl)`: The process starts its execution once the time indicated by Time has elapsed, the alarm can be cancelled using the control list.

3. IMPLEMENTATION

Since concurrent logic oriented architectures are not available, it is usual to design abstract machine, and use current systems to emulate its behaviour. Different abstract mach for logic languages have been proposed, most of them based on the one defined by Wa [21] for Prolog. In particular, sequential [14], shared memory [8] and distributed architectur

[16] have been proposed for CLLs on abstract machines.

The main problem that appears in the implementation of these languages on distribu systems is global shared memory space simulation. The usual solution to this problem is maintain a single occurrence of a logic variable in only one processor and to use rem references to it in the other nodes [12]. The main problem with this solution is that every time reference to a remote variable appears it is necessary to establish a communication with node where the variable is, and this is not very efficient. Other solutions exist, such as the presented in [20], where a method based on the migration of variables is proposed, but method has an even higher cost, which is only justified by the special characteristics of execution model of Concurrent Prolog [19], that needs to implement distributed unification an atomic way.

Another important problem that appears in the distributed implementation of these kinds languages is distributed garbage collection. The implementation of garbage collection distributed systems involves very high costs. Most of the algorithms need to stop the wh system to recover memory areas, bringing a large overhead into the system. In DRL a n communication mechanism is introduced for logic variables shared across different process logic channels. Remote references to variables are not necessary with logic channels, communication between processors is highly reduced. Another advantage of this model is garbage collection can be achieved locally, and it is not necessary to stop the whole syste achieve such a collection. Two previous implementations of this model exist: the first one w developed on a Parlog system using metainterpretation [9] and the second one was interpreter for a subset of the language developed in C on a Unix workstations system [10].

We will now present a new compilation based version for an abstract machine for complete version of DRL. The implementation is based on the emulation of a distribut abstract machine, and it has been developed on a Unix workstations network, using C TCP/IP sockets for communication. DRL is a flat CLL extended for real-time control, explic distributed execution, communication based on logic channels etc., so we have based implementation on an existing sequential implementation for Flat Parlog [17].

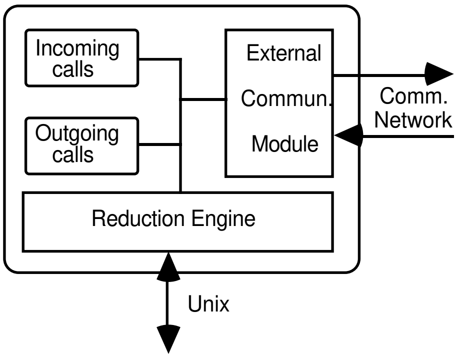

Fig. 2

Each of DRL distributed implementation nodes is based on this sequential version (wh has been extended to cope with the characteristics of our language), with the addition of th new modules which are in charge of managing the language's communication and distribu execution: external communication, incoming calls and outgoing call modules (Fig. 2).

3.1. Execution Model

The execution model provides the basic control mechanisms to execute DRL programs the abstract machine. We have selected a process oriented model, based on an AND execution model [8]. This has been inherited from the sequential abstract machine for F Parlog, but has been adapted to include DRL's real-time and remote execution features, an allow execution of control metacalls which implies the control of process suspensio cancellation and failure. In a pure Flat Parlog execution model, if any process fails, the who computation fails. In the DRL model, if a process fails, and it is a descendent of a contr metacall, this metacall succeeds returning a failed status, but the computation will not fail.

When a process is created its state is active , and it remains in this state until it is schedu A process is in the executing state until it finishes or suspends. If a process succeeds finishes normally returning the memory assigned to the system and the computation goes On the other hand, if a process fails, it has to test if it is a descendent of a control meta before finishing normally. If it is not, the whole computation fails, as in the Flat Parlog mode Otherwise, it sets a flag in its control metacall parent indicating a fail. When any process scheduled, before starting its execution it tests whether it is a descendent of a control meta with the fail flag set, in which case it finishes directly without executing (indirect killing). This "indirect killing" mechanism is also used to implement the cancellation of the descende processes of a control metacall that has been cancelled (when the exception handling pro executes stop). Suspension control inside a control metacall is achieved in a similar w (indirect suspension). A suspension flag is set in the control metacall, and the descend processes will suspend on the metacall´s Control variable when they are scheduled. When exception handling process executescontinue all the descendent processes will wake up, se its state to active.

These indirect ways of implementing suspension, cancellation and failure are very simp and efficient, and they are well-suited to the real-time features of our language. A proce enters the suspension on variable state when no candidate clause can be selected for redu without instantiating variables. A process can be suspended on more that one variable at same time, and will enter the active state when any of these variables is instantiated. A proc enters the suspension on children state when it is waiting for its children to finish and when of them finish it also enters the active state. A process enters the suspension on temporal gu state when no candidate clause can be selected for reduction and there exists at least successfully evaluated clause with a temporal guard. Once the time indicated in the tempo guard has elapsed, the process enters the executing state. A process can be suspended same time on variables and on temporal guards. In this situation it can enter the active sta one of its suspension variables is instantiated, or it can enter the executing state when the indicated in the temporal guard has elapsed. As a special situation, the control metacall ca can be suspended on children and on its Control variable at the same time.

Process execution consists of two phases: Test and Spawn. When a process is schec after a suspension, execution will continue from where its continuation pointer indicates. In tł model there are two continuation pointers: one for the temporal guards and another suspension on variables or children. These continuation pointers can be modified duri process execution.

In the test phase, the machine tries to find a candidate clause to commit the goal. In orde achieve this it matches the input arguments and it evaluates the guards of the different clau that define the process. In the spawn phase, once the process has committed a clause, it c out output unification, and if this succeeds and there is a body in the clause, it also achieves creation of the child processes.

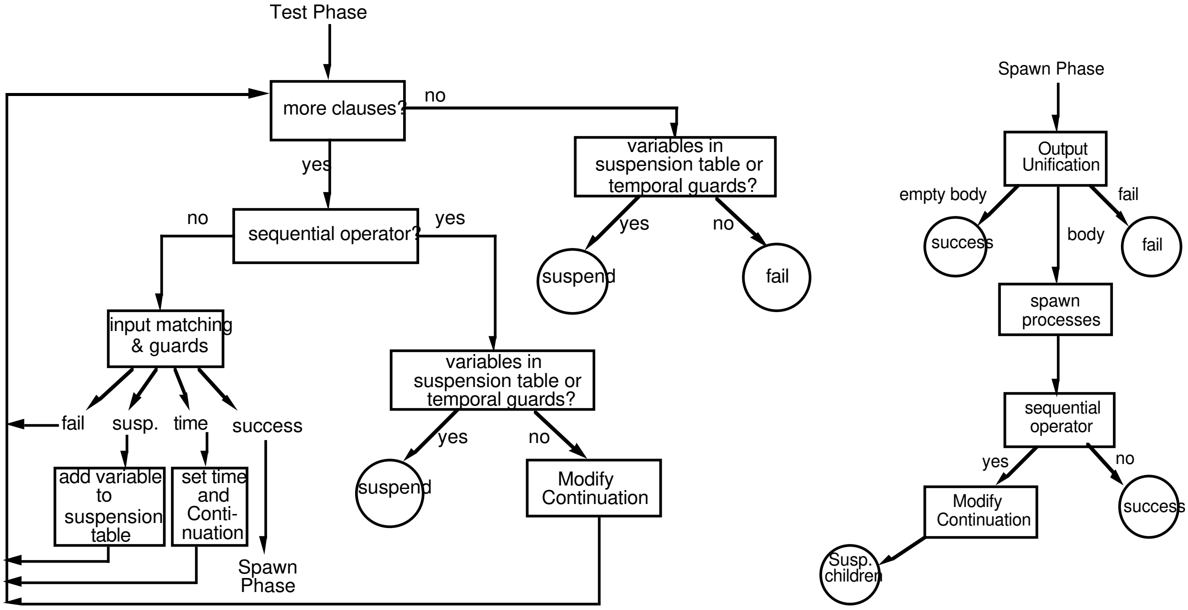

Fig. 3. Test and Spawn Phases

Remote Execution

A process is completely executed in a processor. When a grain process is created b remote metacall, that process will be executed in the processor indicated by the first argum of the metacall. Each processor has a statistics table with information about the load of e processor in the system (CPU and memory). This information is updated by using sor additional information that is included in each interchanged message. If the first argument of t metacall is the term auto(X), then the system will allocate the process in the less load processor and will instantiate the X variable with the processor´s identifier. When a remo metacall is executed in the i processor in order to execute a grain in the j processor, a ca process will be created in the j processor, which will be in charge of executing the proce indicated in the remote metacall. All the child processes of the grain will be descendents of t call/3 process, and all of them will be controlled by this control metacall. In the origina processor (i), a remote_call process is created. This process will suspend on its Status vari until the remote call/3 process finishes its execution instantiating this variable. This remote_c process will be the representative of the remote process execution in original processor.

Communication between processes that are placed in different processors will be achie by using logic channels. Remote metacall execution implies the creation of, at least, two lo channels: The Status channel (input in the original processor and output in the remote one) the Control channel (output in the original processor and input in the remote one), to cont process execution and exceptions. The exception handling process will be simply another c of the call/3 process, and will be suspended on the Control variable and able to instantiate Status variable. Besides these control channels, all the channels that appear in grain defin will be created, with the directionality indicated in this definition. A logic channel, betwee processors i and j, will have an output (input) channel variable in the i processor and an in (output) channel variable in the j processor. Besides the logic channels that appear in g definition, other logic channels can be automatically created during process execution. T permits the use of incomplete messages between processes placed in different processors also the creation of communication channels between two processors via other intermed processors.

3.2. Data Areas and Structures

The abstract machine use mainly three different data structures. The process structure: it a fixed size and contains enough information to execute a process: parent pointer, refere count, argument count, state flags for the control metacall, time information , etc. When process is created a process structure is allocated in the corresponding area (process sta process is really a synonymous of a process structure. In each processor, the PS register abstract machine point to the process structure of the current executing process. The instru structure is compossed by a field indicating the instruction code, and the necessary argum for that instruction. The working word of the abstract machine is composed by two fields: t flags field, that indicates the kind of data of the word (list, atom, channel,...) and the val field, that contains a value or a pointer to an data area, depending on the content of the f field.

The memory of each abstract machine processor is divided into different data areas. code area, that contains the compiled code of a program. The process stack, that contains structures of the processes. The argument stack, that contains the environments and argu of the processes i.e, the structures of variable size of a process. The environments contain variables that appear before and after a sequential operator in a clause. The heap contain lists, variables, terms and channels created during the execution. Three execution queues: for the processes suspended on temporal guards, another one for high priority processes the last one for the rest of the processes.

The input channels table will contain information about the channels: the remote process and the heap index of the variable corresponding to the input channel. The output channels t will contain: the remote processor and the number of the input channel associated in t processor. The output channel variables will point to its corresponding entry in this table. It possible that some of this information can be unknown when a logic channel is creat (because an output channel variable need a confirmation from the remote process with the i channel number corresponding in that processor), so, if we try to send data by one of th channels before that the confirmation arrives, the sending of this data is postponed. To achi this, we need another field in this table, that will point to the location in the heap of the data be sent.

The active processors table contains the processors identifiers that can be used in the re metacalls. This table and the two channels table are managed by incoming/outgoing calls the communication management modules.

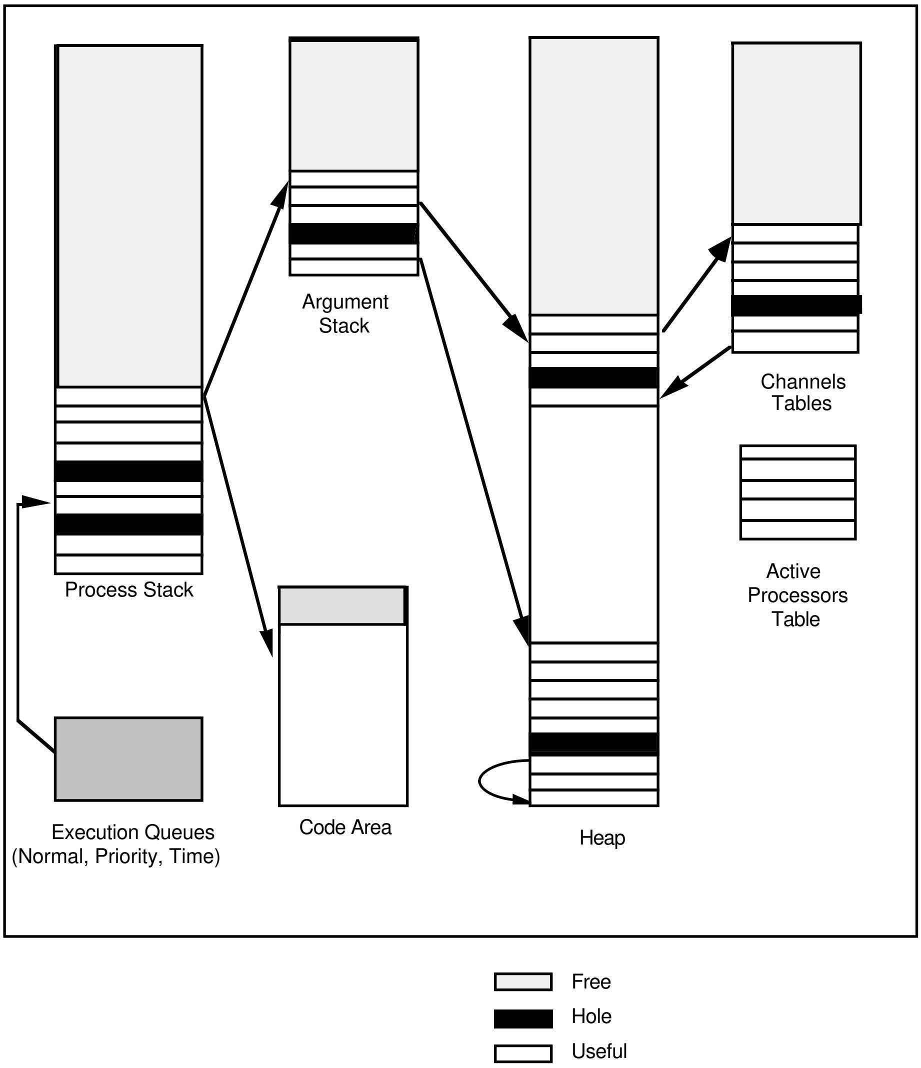

Fig. 4. Data areas

Beside these data areas, the abstract machine will have a set of internal registers in e processor:

- -PC. Program counter.
- -PS. Current process.
- -HP. The top of the heap.
- -A0,A1,...,An. Argument registers.
- -X0,X1,...Xn. Temporal registers.
- -etc.

3.3. Instruction Set

The abstract machine code for a DRL procedure consists of instructions to try alternati clauses followed by the code for each clause. The code for a clause can be further divided test phase and spawn phase instructions. The instruction set is very similar to the one define [17]. Instructions have been added to deal with logic channels, and the implementation of so of the original instructions have also been modified. We can divide the instruction set follows:

a) For trying alternative clauses: try, trust, suspend_process.

try is a jump instruction to the begining of clause´s code.

trust is similar to previous instruction, but for the last clause of the procedure.

suspend_process implements the sequential clause operator.

b) Test phase:wait,read,builtin.

wait instructions corresponds with the clause input arguments.

read instructions try subterms of lists and structures on input arguments.

builtin instructions corresponding to the primitives allowed in the guards.

c) For selecting a clause: commit

d) Spawn phase: get,unify,proceed,fail,call,put,push.

get instructions correspond with output arguments.

unify instructions correspond with the subterms of lists and structures.

A process that executes the proceed instruction terminates successfully

The fail instruction terminates the process with failure.

call instructions create processes.

put instructions build the arguments of the primitives in the guards and the subterms of li and structures in the arguments of the body goals.

push instructions build the arguments of body goals

e) For allocating and deallocating environments: allocate, deallocate.

3.4. Garbage Collection

One of the most important problems that arises in the distributed implementation of CLLs garbage collection. Some algorithms exist for garbage collection in distributed systems [1 but a common drawback to many of them is that computation has to be stopped to achiev global garbage collection. Algorithms also exist that collect garbage incrementally, but th incur very high costs. In our model, there are no remote references to variables and garb collection can be achieved locally, in the same way as in a monoprocessor implementation. efficiency of garbage collection algorithms is especially important in real-time systems becau a garbage collection phase can lose deadlines to real-time processes. Some proposed algor for real-time garbage collection [2] also exist.

In our implementation we have used the stop & copy garbage collection algorithm [3], wi some modifications. In the original algorithm a garbage collection phase starts when the hea the argument stack is full and the whole data area is inspected for garbage collection. In a r time environment it is necessary to take into account the process deadlines before starti garbage collection phase. We have modified the algorithm to achieve incremental garb collection (data areas can be garbage collected in various phases) and to take into accour process deadlines before starting a garbage collection phase. To achieve this, we have defi critical threshold, that depends on the amount of free space in the stack and heap and o amount of processes ready for execution. A garbage collection phase will start only if there no process with maximum priority or the deadlines of real-time processes allow a garba collection phase without losing any of them. A garbage collection phase can also start if all processes are of low priority. To implement this algorithm it is necessary to know a limit to t maximum time for garbage collection.

3.5. Results

The system has been successfully implemented on a Unix workstations network, usin and TCP/IP sockets for communication. Some examples, some of which were taken from CLL literature (nqueens, matrix multiplication, quicksort,...) and adapted to DRL, have bee used to test the system. Other typical distributed programming problems have also be succesfully implemented, especially those related to communication protocol specification a simulation (token ring, alternating bit protocol, daemon game,...). The real-time aspects of t language have also been tested with some common problems. Periodic and aperiodic proce have been used and the system has performed well, meeting the deadlines in the order of a of a second.

Although the main aim of the system has not been to improve the execution of parall programs, but to design a language for distributed and real-time programming, so measurements of the system's efficiency in executing parallel algorithms have also been ca out. The problems were adapted for execution in the actual system. The sequential versi timings have been taken in a previous sequential implementation of a real-time sequential Parlog version.

The number of process migrations (only the grain processes) is very low and the lo balancing is quite good (the number of reductions achieved by each processor is very simil The number of messages is very low, in the order of 5 messages for each 1000 reductions. speed up obtained is nearly lineal, and we even obtained a superlineal one with the th problem, due to the great number of garbage collections made in the sequential version.

	nqueens(7)	qsort(3000)	matmult(40)
Sequential	90.3	56.5	32.4
3 processors	47.3	21.9	13.6
Speed up	1.9	2.6	2.4
6 processors	20.8	10.6	4.5
Speed up	4.3	5.3	6.4

4. CONCLUSIONS

The implementation of the DRL language has been presented. DRL has been designed the aim of integrating distributed and real-time programming under a declarative langua framework. The basis for this language's development has been CLLs, which have be extended to cope with the requirements of distributed and real-time languages. A ne distributed abstract machine for the execution of DRL has been proposed, and t implementation's main issues have been discussed.

The implementation is based on the emulation of this distributed abstract machine, whi has been developed on a Unix workstations network, using C and TCP/IP sockets communication. Some examples have been used to test the system, of which some were t from the literature about CLLs and have been adapted to the DRL language. Other typi distributed programming problems have also been succesfully implemented, especially th related to communication protocol specification and simulation. The real-time aspects of t language have also been tested with some common problems. Periodic and aperiodic proce have been used and the system has performed well, meeting the deadlines in the order of a of a second.

REFERENCES

[1] AGHA, G. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press. 1986.

[2] ARMSTRONG, J.; VIRDING, R. AND WILLIANS, M. "Concurrent Programming in ERLANG". Prentice Hall. 1993.

[3] BAKER, H.G. "List processing in real time on a serial computer". Communications of the ACM 21,4 (April 1978), pp. 280-294.

[4] BAL, H.E.; STEINER, J.G.; TANENBAUM, A.S. “Programming Languages for Distributed Computing Systems”, in ACM Computing Surveys, vol. 21, 3. 1989. pp. 261-322.

[5] BANATRE, J-P.; LEMETAYER, D. "The GAMMA Model and its Discipline of Programming", in Science of Computer Programming, 15 (1). 1990. pp. 55-77.

[6] CLARK, K.; GREGORY, S. "PARLOG: Parallel Programming in Logic" in ACM Transactions on Programming Languages and Systems, vol 8, 1. 1986.

[7] COHEN, D.; HUNTBACH, M.M.; RINGWOOD, G.A. "Logical Occam". Tech. Report. Dept. of Computer Science, Qeen Mary and Westfield College. London. 1991.

[8] CRAMMOND, J. “The abstract machine and implementation of parallel Parlog”. Tech. Report Dept. o Computing of Imperial College. 1990.

[9] DIAZ, M.; PIMENTEL, E.; TROYA, J.M. "A Declarative Environment for Distributed Programming", in Proc. of 12th IFIP, 1992. pp. 163-169.

[10] DIAZ, M; TROYA, J.M. "A Parlog based real-time distributed logic environment", Future Generation Computer Systems, 9, 1993, pp. 201-218

[11] ELSHIEWY, M.A. “Robust Coordinated Reactive Computing in Sandra”. PhD. SICS, 1990.

[12] FOSTER, I. System Programming in Parallel Logic Languages. Prentice Hall, 1990.

[13] FOSTER I, TAYLOR S. "Strand: New Concepts in Parallel Programming". Prentice-Hall, 1990.

[14] FOSTER, I.; GREGORY, S.; RINGWOOD, G.; SATOH, K.; "A sequential implementation of Parlog", Proc. of the 3rd ICLP, London, UK, LNCS-225, Springer-Verlag, pp. 149-156. 1986.

[15] HALBWACHS, N.; CASPI, P.; RAYMOND, P.; PILAUD, D. "The Synchronous dataflow programming language LUSTRE", in Proc. IEEE, vol. 79. 1991. pp. 1305-1320.

[16] ICHIYOSHI, N.; ROKUSAWA, K; NAKAJIMA, K; INAMURA, Y. "A new external reference management and distributed unification for KL1". Proc. of Int. Conf. on 5th Generation Computer System ICOT. 1988.

[17] RUBIO, B. ; TROYA J.M.; "A Sequential Implementation of Flat Parlog". Technical Report. Dpto. Lenguajes y Ciencias de la Computación. Universidad de Málaga. 1993.

[18] SARASWAT, V.A.; KAHN, K.; LEVI, J. "Janus: A step towards distributed constraint programming", in Proc. of ICLP. 1990.

[19] SHAPIRO, E. "The Family of Concurrent Logic Programming Languages" in ACM. Computing Survey Vol 21, 3. 1989. pp. 413- 510.

[20] TAYLOR, S. "Parallel Logic Programming Techniques". Prentice Hall, 1989.

[21] WARREN, D.H.D. "An Abstract Prolog Instruction Set". Technical Note 309, SRI International, Stanforc 1983