DRL: A DISTRIBUTED REAL-TIME LOGIC LANGUAGE†

M. DÍAZ‡, B. RUBIO and J. M. TROYA
Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Campus de Teatinos,
s/n. 29071, Málaga, Spain

(Received 12 May 1997)

Abstract—This paper presents 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 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. The model is introduced
by giving an operational semantics for the language and the new model's implementation is discussed,
including the definition of a new abstract machine and its implementation on a network of Unix
workstations. Although the sequential core is not optimized, some previous results are discussed, showing
the feasibility of the language's execution model for distributed real-time systems. The language is
currently being used as the kernel language for a distributed simulation and validation tool for
communication protocols. © 1997 Elsevier Science Ltd. All rights reserved

1. INTRODUCTION

Distributed real-time systems are very complex and they have very strict safety and reliability
requirements. These systems are composed of collections of diverse computations, without central
co-ordination and lacking total knowledge of each other. Moreover, most programming languages
have not been designed to support such systems, but rather to describe tightly co-ordinated
computations. In the last few years, many new proposals for distributed and real-time systems have
emerged [1]. Some of these models are based on traditional imperative languages, but others are
intended to provide new paradigms suited to the characteristics of these kinds of systems [2]. In
this sense, many different approaches for the integration of distributed and real-time paradigms
in declarative languages have been proposed [3, 4].

Concurrent logic languages have appeared as an attempt to adapt Prolog to parallel systems [5],
and they have shown its suitability for distributed systems. In fact, CLLs have been used as kernel
languages to construct declarative environments for distributed programming [6–8]. These
languages have evolved out of these inefficient environments into new languages more oriented
towards the characteristics of distributed systems. Some languages which have come out of
this evolution are RGDC [9], Strand [10], Sandra [11], and Janus [12], which despite being a
language used for concurrent constraint programming, has the same basis of CLLs. Something that is common to all these proposals is that they try to overcome one of the
drawbacks of CLLs in their use for distributed systems: their inefficiency. Distributed
implementations of CLLs are not very efficient, mainly because the communication mechanism is
based on sharing logic variables and its implementation on distributed environments involves very
high costs.

We propose a new approach to distributed programming based on Parlog [5] and oriented to
coarse granularity parallelism: the DRL language. This language has been designed on the basis
of extending CLLs to cover the different aspects of complex real-time distributed systems. In
this sense, a new execution model for CLLs is presented which provides efficient distributed

†This work has been supported by the project CICYT-TIC-94-0933-C02-01.
‡Author to whom correspondence should be addressed.
execution and real-time control. These extensions have been made taking into account theoretical aspects as much as those of implementation. From the theoretical point of view the operational semantics of the proposed language is presented, maintaining the possibility of formal analysis of programs. In this sense, the semantics offers an integrated view of both extensions, by incorporating time in the transition systems which define the language's operational semantics. From the implementation point of view a new abstract machine has been designed incorporating a new communication model based on logic channels [6] and a new scheduling policy to allow real-time control. The language is currently being used as the kernel language of a distributed simulation and validation tool for communication protocols. TDL is especially well suited for these kinds of applications due to its distributed and real-time capabilities that allow the developing of distributed prototypes of complex protocols in a simple way.

The paper is organised as follows. In the rest of this section, we briefly analyse some proposals for both the distributed implementations of CLLs and the inclusion of real-time characteristics into these languages. In Section 2 we describe the language's most relevant aspects, including a few example applications. In Section 3 we give the language's operational semantics. In Section 4 we describe the new abstract machine and various details about the implementation. Some of the results of this implementation are discussed in Section 5. Finally, in Section 6, some conclusions are presented.

1.1. Related works

A variety of work has been carried out in the field of real-time extensions for CLLs. In this sense, some of the most outstanding proposals are Parlog-RT [13], Sandra [11] and Fleng [14]. Parlog-RT defines a notion of logic time by associating a local clock to each Parlog process; this clock is implemented by another Parlog process that runs in parallel. This language includes some primitives for accessing the real time clock and for the definition of time-outs (after(T) that becomes true after T time units; before(T) that becomes true before T time units; at(T,Goal) that executes Goal at instant T, etc.). The local clock processes are implemented by the use of the predefined predicate succ(C,Cs) that increments the logic clock C synchronously to the real-time system clock. The main drawbacks of Parlog-RT are: the system overload caused by the logic clock processes (they have to be reduced with every clock tick), the absence of priorities; and a real-time scheduler.

The real-time model of Sandra is similar to the previous one (in fact, it evolved from this). However, Sandra includes some extensions for exception management and fault tolerance. Fleng is a very simplified concurrent logic language; it has no guards and synchronization is achieved by the matching of clause heads only. Fleng also includes some real-time primitives that allow the definition of timeouts and delays. Like Parlog-RT it has no notion of priorities and neither does it implement any real-time scheduler.

A more recent proposal is that of RTX-Parlog [15]; this language implements a real-time scheduler and allows the definition of deadlines for the execution of real-time processes. However, this language lacks any exception management capability and neither is real-time garbage collection considered in the implementation.

With regard to the distributed implementations of concurrent logic languages, works carried out at ICOT with the KL1 programming language [8] are of great interest. The KL1 distributed implementation model has several similarities and also differences compared to our proposal. We will discuss them in the following sections. One of features of the KL1 language is the provision of simple yet powerful meta-control facilities (goal execution control, computation resource management, and exception handling). The KL1 language, like other CLLs, aims at fine-grain parallelism, and the KL1 language processor reduces a large number of goals in parallel. Therefore, it is inefficient and almost impossible for a programmer or the runtime system to control the execution of each goal. Consequently, KL1 introduces the concept of a sheen [16]. A sheen is regarded as a goal group or a task with meta-control facilities. An initial goal is given as an argument to the built-in predicate sheen. Descendant goals belonging to the sheen are controlled as a whole, which inherit the sheen of the parent goal. Shoens are possibly nested as well. A sheen has two streams as arguments of the sheen built-in predicate; one for controlling sheen execution (start, stop,...), and the other for reporting information inside the sheen (terminated, exception,...).
In the Fifth Generation Computer Systems Project, ICOT has been investigating the efficient parallel and distributed implementation of KL1 on PIM [17], a large-scale parallel machine based on a multi-cluster system. The KL1 language processor adopts automated goal-scheduling performed within a cluster and manual goal-scheduling among clusters. To carry out parallel computation mapping, priority and location specification are introduced. So, p@priority(N) tells the runtime system to execute goal p at priority N; and p@cluster(M) instructs the runtime system to execute goal p in the Mth cluster. A shoen can have several goals distributed among clusters.

The implementation model for a shoen on a distributed environment introduces a foster-parent to prevent bottlenecks and to reduce communication. A foster-parent is a kind of proxy shoen or a branch of a shoen.

The global address space is managed by using external references to remote variables. There is only one occurrence of each variable, and the other clusters have external references to it. In order to manage external references, the unification algorithm has to be modified. In the KL1 model there exist two different kinds of unification: active and passive. Passive unification occurs in the clause selection phase and active unification appears explicitly in the body of the selected clause. Passive unification only needs to read variables (guards are not permitted to instantiate variables in KL1) while active unification can instantiate variables. External references can appear in passive unification; in this case a message will be sent to the remote processor to obtain the value. Active unification is more complex, because different processes may instantiate the same variable at the same time; the most obvious solution is to use a locking mechanism for the variable, but this solution has a very high cost. The adopted solution consists of always carrying out the unification in the processor where the variable is placed. When active unification is needed and an external reference exists, a unification message is sent to the processor where the variable is placed; once the unification is completed a reply is sent to the source processor with the result of the unification.

In order to carry out local garbage collection on each cluster, without stopping the whole computation, export tables are introduced in the model. Each cluster has an export table, which has an entry for each exported variable, i.e., referred to by other clusters. In order to decrease inter-cluster traffic, the same exported data should be accessed as few times as possible. Hence, each cluster maintains an import table to register all different imported external reference identifiers. The same external references in a cluster are gathered into the same table entry. The KL1 language processor can perform three kinds of garbage collections. Firstly, a local garbage collection using the MRB incremental scheme. Since this scheme cannot reclaim all garbage objects, it is still important to implement an efficient garbage collector to supplement the MRB scheme. Secondly, a parallel execution scheme of stop-and-copy garbage collector is used for each cluster. During local garbage collection, data referred to by an export table entry should be regarded as active data, because it is difficult to know whether or not the export table entry is referred to by other clusters immediately. Therefore, and so thirdly, a global garbage collection scheme for the export table is necessary. The KL1 language processor performs a weighted export counting (WEC) scheme.

Although KL1 runs efficiently on PIM, such implementations have serious disadvantages in that they are not portable and cannot be used on commercial machines. To solve this problem, a scheme that allows a portable implementation of compiling into C, called KLIC, has recently been developed [18]. Firstly, the sequential core was designed, showing reasonable efficiency from the point of view of both time and space. Next, following the previously described scheme, two kinds of parallel implementation on a distributed memory and a shared memory were designed with the policy of retaining the efficiency of the sequential core. In order to realize this, the parallel processing part of the implementations is built based on generic objects, which provide a framework for flexible extensions without changing the core implementation whatsoever.

Some results obtained with the distributed implementation of KLIC will be compared, in Section 5, with our implementation of DL. We also compare our results with the distributed implementation of the concurrent logic language Strand [10]. This language, commercially available (and about which, therefore, we have little information) is based on variable assignment, instead of unification, the aim being to achieve a more efficient and less expensive implementation. We think that, as in the model proposed by Ian Foster for the distributed implementation of Flat Parlog [7], the global address space in the distributed implementation of Strand is also based on
external references to remote variables, and a communication protocol similar to that of KL1 is used. Export tables for carrying out local garbage collections are included. In order to achieve a global garbage collection it is necessary to stop the whole computation.

With regard to the third of the more important families of CLLs (GHC, Parlog and Concurrent Prolog) different distributed implementation models appear in the literature: [19–21]; we will only consider the last model, which is the most efficient one. The main problem in the distributed implementation of FCP is atomic unification; in this language output unifications are allowed in the guards, and because of this, output unification must be atomic. Atomic unification yields two new additional problems: the locking of shared variables and starvation of processes. Atomic unification is implemented by doing all the unifications locally; to achieve this it is necessary to first have all the data involved in the unification in the same processor. There are also two different algorithms for unification: one for reading variables, that does not need mutual exclusion, and one for writing variables. The algorithm for reading remote variables is similar to the previous models, but the other algorithm needs a locking mechanism; this mechanism is called variable migration. Every time a process needs to write a variable, which has to be localized, and the reference to the variable and its value are swapped. The mechanism used to obtain mutual exclusion in the migration of variables is based on processor priorities. This implementation model has a higher cost than the previous ones.

Finally, we must mention the Erlang language [3]. Although it is not a concurrent logic language, its real-time facilities and its distributed execution model are interesting. Erlang combines functional and concurrent programming paradigms, allowing a functional programming style and, at the same time, supporting mechanisms for modelling concurrence, distribution and real-time. This language was especially designed for programming telephonic systems and it is an evolution of the CLL Strand. In Erlang, concurrence is explicit, i.e. there exist special primitives for remote process creation, and communication is achieved by means of asynchronous message passing primitives. Every process has a single input channel where the messages are queued; the input primitive allows for the definition of time-outs in the reception of messages. The language also includes a basic exception management procedure (only for predefined exceptions). Erlang is being used for the programming of industrial applications and very efficient sequential [22] and distributed [23] implementations are available.

2. THE 9149 LANGUAGE

In this section we will describe the 9149 language’s basic features. The characteristics of this language are determined by two main factors: firstly, the language must preserve the declarative nature of CLLs and, secondly, the language must be efficiently executed in distributed systems. These objectives, and the drawbacks of these languages for their implementation on distributed systems, have led us to extend CLLs in different ways. The main characteristics of 9149 are: flat guards to improve efficiency [10]; explicit mode declarations (as in Flat Parlog); coarse granularity parallelism; explicit remote process creation; real-time control based on temporal guards [24]; exception handling; and logic channel based remote communications [6].

The language’s execution model is process oriented. Processes can be grouped into grains, which are the minimum execution unit that can be executed in another processor. Process behaviour is expressed by a set of Flat Guarded Horn Clauses (i.e. the predicates that can appear in the guards are predefined). Real-time constraints are expressed by means of temporal guards. A temporal guard is a guard with an after(T) primitive, where T expresses time in seconds. Apart from the utilization of temporal guards the language provides primitives that allow the constraint of process execution time. A 9149 program consists of a set of predicate and grain definitions and an initial goal which defines the initial state of the system.

2.1. Remote execution

Fail control is one of the problems of CLLs for their execution on distributed systems. In a CLL a network of processes is modelled by a conjunction of predicates that communicate by shared logic variables. As it is a conjunction of predicates, a failure of one process causes the failure of the whole
network; in some applications, and especially in distributed applications, this must be avoided. To solve this, different mechanisms have been proposed, most of them based on control metacalls [17]; these metacalls are metapredicates, that allow the execution and control of a process; if a process fails then the execution of the metacall succeeds returning the final state of the computation in one of its arguments.

In order to execute a set of processes in a remote processor it is necessary to group them into a grain. A grain differs from a shoen, defined for KL1, since processes of a shoen can be distributed among different clusters, incurring increasing communication. Grain definition has the following format:

```
grain grain_name (m1, ..., mn). (%(mi = ? in, ^ out, # constant)
grain_name(T1, ..., Tn) <- g1, ..., gn. b1, ..., bn.
<Local predicate definitions>
<Exception handling>
end grain.
```

A grain is executed by means of a remote metacall:

```
remote_call (Processor, grain_name(V1, ..., Vn), Status, Control).
```

where:

- **Processor** is a processor's symbolic name where the grain will be executed or is an 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 child processes will be executed in the processor indicated in the remote metacall. Communication with that process will be achieved through logic channels instead of shared logic variables and the direction of these channels is indicated in the definition of the grain. The process will not be executed until all constant arguments (declared with #) are known. A logic channel is a list that will be instantiated incrementally. This list can contain: constant terms, terms with logic variables (the transmission of terms with logic variables will be postponed until all of them become ground) and terms including logic channels (variables annotated with ? or ^, permitting the dynamic creation of channels). Communication between processes in the same grain is achieved through shared logic variables as in other CLLs. The Status and Control list of the remote metacall are considered logic channels too. Grain creation is dynamic, i.e. a grain can be created at any moment during execution and in any processor. The remote metacall Processor variable can be instantiated to the name/number of a processor or to the terms: local (the process is executed in the same 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 the development of distributed systems easier [25]. The environment is initialized from a processor (called root processor) and a system server is created in each processor in the network.

### 2.2. Logic channels

One of the most important problems in the distributed implementations of CLLs is the simulation of a global shared address space. We have seen different solutions in Section 1.1 (basically remote references and variable migration) that have been proved inefficient due to their complexity and the high number of messages involved in unifications.

In DRL, the communication mechanism between grains has been modified in order to obtain a more efficient execution of the language in loosely coupled distributed environments. On the other hand, this modification has been made trying to maintain the semantics of the language as close as possible to those of other CLLs, without affecting the declarative and expressive character of the language. As we have seen, a logic channel is a logic variable that appears in the definition of a grain with a direction; for each logic channel there exists at least two different logic variables that are the representatives of such a channel in the origin and remote processors. In Fig. 1(a), a `remote_call` is executed in processor 0 to execute process p in processor 1; a logic channel
L is created and the variables \( L^\wedge \) and \( L^? \) are the representatives of that channel in both processors. If a channel is shared between more than two processors, only one of them will be able to instantiate the channel, and the other ones become receivers (Fig. 1(b)).

A logic channel can only be instantiated with a ground logic term, possibly containing annotated variables (input \( X^? \), output \( Y^\wedge \)). If the term contains no annotated variables, its transmission is delayed until the term becomes ground; the annotated logic variables are considered new logic channels. This mechanism allows incremental communication between grains, dynamic channel creation and incomplete messages. An example will be shown where a consumer grain (cons) is executed remotely and a producer process (p) is executed locally (processes produce and consume are not defined); communication between both processes is achieved incrementally, since logic channel \( L^1 \) is annotated (example 1). If this channel is not annotated, the transmission of the list will be delayed until the producer has finished its work. This allows for the control of message length, which can be very helpful in some distributed programs.

```
grain cons(\?).
cons(\([Data \mid L]\)) <- consume(Data), cons(L).
cons([])
end grain.

mode p(\?).
p(L) <- produce(Data), L = [Data | L^1\'], p(L^1).
  p([]).
<- p(L), remote_call(1, cons(L), _, _).
```

Example 1

Another important aspect related to logic channels is interconnection. In \( \mathcal{DL} \) two logic channels can be connected by using unification, taking into account their direction. For example if an input and an output logic channel are unified, the data received by the input channel are automatically retransmitted by the output one.

Fig. 1. (a) Channel creation (b) Shared channel
2.3. 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. When a grain is created, an exception handling process is created too. An exception is detected by this predicate when the head of the Control list is instantiated. This variable can be instantiated outside the grain by a process that shares the control variable with the remote metacall or by the execution of the predicate `raise(Term)` by a process inside the grain. When this predicate is executed the head of the Control list is instantiated to `Term`. The exception process can instantiate the Status variable that can be consulted by a process which shares this variable in the original processor, so exceptions can be propagated in a simple way. With regard to KL1 language, there is no exception handling process inside a shown and control is achieved by other process outside the shown.

The JRL program of example 2 shows a situation where a process reads (process `read` is not defined) a device that returns a status (ok or error). This process is executed in a remote processor and the read data is returned to the original processor. The process is aborted after 30 seconds (in the way shown in the next section).

```
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(Status,[error(D,L)|Control])<- reset(D),
                                excep(Status,Control).
excep([aborted],[abort])<- stop.

end grain.

mode control(\?,^)).
control(Disp,Read)<-wait(msec(30,0),Ctrl),
                  remote_call(1,reader(Disp,Read),St,Ctrl).

mode wait(\?,^).
wait(T,[abort])<-after(T):true.
<-control(disk,L).
```

Example 2

2.4. Real-time control

Concurrent logic languages do not provide any mechanism to express real-time constraints in the execution of processes. Sandra [11] introduces the notion of logical clocks to express time in the field of CLLs, but the concept of time in Sandra is not actually real-time. The logical clocks only have the task of ordering the occurrence of events in the system. Another approach in the field of declarative languages is that of Erlang [22], a functional language that evolved from Strand. This language can express real-time constraints, but its semantics is at present very different from that of CLLs.
As the execution model of CLLs is based on interleaving, the process scheduling order cannot be known. We propose an extension of CLLs that allows the expression of process timing constraints. This extension implies a modification of the execution model. In CLLs a process may suspend on a non-instantiated variable, during matching of the clause head or in guard evaluation. In our extension, processes may also suspend in the evaluation of timed guards. A timed guard is a guard with an `after(T)` primitive, where T is a term `msec(Seconds,Milliseconds)`. This primitive makes the process suspend for the time expressed in the term `msec(Seconds,Milliseconds)`, after matching of the head and guard evaluation. If there is no other clause that commits before that time has elapsed, then that clause is selected to reduce the process. A clause that contains a timed guard is called a timed clause. In example 3 two processes are defined: a producer, p, and a consumer c that communicate via a shared variable L. There is a timed guard in the definition of consumer, that detects a time out error if no data arrives in the channel within a ten second period.

```
mode p(\^).
p([item|L])<-p(L).

mode c(?,?,^)
c([X|L],T,St)<-c(L,T,St).
c(L,T,St)<-after(T):St=time_out.
<-p(L),c(L,msec(10,0),St).
```

Example 3

When a timed clause exists in the definition of a process, this process must be selected for reduction as soon as possible, i.e. that process has a higher priority than ordinary ones, without a timed guard. If this does not happen then the time expressed in the guard is not real. For example, let us assume the following scenario (Fig. 2), where p and c are the processes defined above and r is a process independent from p and c. The consumer process does not detect that the producer has taken more than T time units to produce the second item of the list, because different instances of the r process have been scheduled meanwhile. However, if the c process had been scheduled before the r process, this situation would have been detected. Besides the temporal guards, other primitives have been 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 milliseconds. The process is suspended for the time units indicated by its argument.
- `timeout(Goal,Time,Status)`: A metacall executes the process indicated by Goal limiting its execution time and returning the Status (success, failed, timeout).
- `alarm(Goal,Time,Ctrl)`: The process starts its execution once the time indicated by Time has elapsed; the alarm can be cancelled using the control list.
2.5. An application example

CLLs have been used as a Formal Description Technique in different phases of protocol design (simulation, prototyping, etc.). DRL is especially well suited for these kinds of applications due to its distributed and real-time capabilities that allow the developing of distributed prototypes of complex protocols in a simple way. The language is currently being used as the kernel language of a distributed simulation and validation tool for communication protocols. We have selected a very simple application in this area to illustrate the basic features of the language: the stop & wait protocol. Example 4 and Fig. 3 show the DRL program and a graphic, respectively.

The stop & wait protocol is one of the simplest protocols for data transmission, but it is of adequate size to be used as an example. The protocol uses two types of frames: information (info) and acknowledgement (ack) frames. The transmitter sends info frames and waits for ack frames and the receiver waits for info frames. The transmitter takes the data from a user and the receiver gives the data to another one. The transmitter has a time-out mechanism that causes the retransmission of the info frame if an ack frame does not arrive before the given time. In this implementation the transmitter can receive more data from the user before receiving the acknowledgement of the previous frames, to achieve this, it creates a new process wait_for_ack that is in charge of: controlling the time-out, the reception of the acknowledgement and the retransmission of the frame in the case of an error.

The information produced by this process and the new one produced by the user is merged by the merge process. The line is modelled by a process that includes a delay and the loss of data in the network is modelled by a non deterministic choice. The simulation can be achieved in a distributed way, and the simulation process can control the execution of each process by using the status and control lists with the exception management in the models. This protocol does not preserve the order of messages and may duplicate the information; this can be easily detected by simulating the protocol.

3. DRL SEMANTICS

In this section we present the DRL operational semantics, which is given by means of a transition system [25]. DRL semantics is based on the one given for Parlog in [26]. The operational meaning of a program is given in terms of sets of substitutions that correspond to the answers computed during the derivation. The semantics is introduced in two steps:

- Distributed execution semantics. This models grain execution, communication through logic channels and exception management.
- Real-time semantics. This is based on an extension of the distributed execution semantics and it reflects the scheduling policy, time guards and transmission delays through logic channels.

3.1. Distributed execution semantics

The distributed execution semantics must reflect the execution conditions of grains and communications via logic channels in an abstract way. The representation of a computation state must also reflect the distributed nature of the language; to achieve this we introduce a new

![Fig. 3. The stop & wait protocol](image)
remote_call(1, transmitter(Linein), St1, Ct1).
remote_call(2, receiver(Lineout), St2, Ct2).
remote_call(3, line(Linein, Lineout, St3, Ct3), simulator(St1, St2, St3, Ct1, Ct2, Ct3).

grain transmitter(^).
transmitter(L) <- user(I), trans(I, L).

mode user(^).
user([data|L]) <- user(L).

mode trans(?^).
trans([Data|Info], Linein) <-
  Linein = [sendinfo(Data, Ack?) | TLinein],
  wait_for_ack(Data, Ack, TLinein1),
  trans(Info, TLinein2),
  merge(TLinein1, TLinein2, TLinein).
trans([], []).

mode wait_for_ack(?^).
wait_for_ack(Data, ack, Line).
wait_for_ack(Data, nack, Line) <- trans([Data], Line).
wait_for_ack(Data, Ack, Line) <- after(msec(10, 0)): trans([Data], Line).
end grain.

grain line(?^).
line([sendinfo(Data, RecAck) | TLinein], Lout) <- random(T), after(msec(T, 0)):
  Lout = [sendinfo(error_frame, RecAck) | TLineout], line(TLinein, TLineout).
line([Frame | TLinein], Lout) <- random(T), after(msec(T, 0)):
  Lout = [Frame | TLineout], line(TLinein, TLineout).
line([], []). end grain.

grain receiver(?).
receiver(L) <- rec(L, I), user(I).
mode user(?).
user([Data|L]) <- user(L).
user([], []).

mode rec(?^).
rec([sendinfo(error_frame, RecAck) | TLin], Info) <- RecAck = nack, rec(TLin, Info).
rec([sendinfo(Data, RecAck) | TLin], [Data|Info]) <- RecAck = ack, rec(TLin, Info).
rec([], []). end grain.

Example 4
representation for these states; this representation is discussed in Section 3.1.2. The transition rules are discussed in Section 3.1.3.

3.1.1. Definitions. Let us introduce the following sets:

- The set \( A \) of atoms, with typical elements \( A, B, H \).
- The set \( \Sigma \) of substitutions, with typical element \( \sigma \).
- The set \( V \) of logic variables.
- The set \( C \) of logic channels.
- The set \( G \) of grains and predicates. Conjunctions have the form: \( \tilde{A} = A_1, ..., A_n \) where \( A_i \in A \).

Two special elements in \( G \) are \( \text{true} \) and \( \text{fail} \).

We also need to define the following functions:

- \( \text{cin}: G \rightarrow \mathcal{P}(C) \) returns the input channels of a term.
- \( \text{cout}: G \rightarrow \mathcal{P}(C) \) returns the output channels of a term.
- \( \text{cvar}: G \rightarrow \mathcal{P}(V) \) returns the variables of a term.
- \( \text{var}: G \rightarrow \mathcal{P}(V) \) returns the variables and logic channels of a term.

We will assume that all the clauses in a program have the following syntax:

\[
H \leftarrow G : B.
\]

For the rest of this section let \( W \) denote a fixed program. The function \( \text{imvar} \) gives for every atom \( A \) the set of variables occurring in those arguments of \( A \) that are specified as input by the mode declaration of \( W \); \( e \in \Sigma \) is the empty substitution; \( \text{mgu} \) and \( \text{mgu}_i \) denote the input and output \( \text{mgu}'s \), respectively. We also define an evaluation function for the predicates of the guard:

\[
\text{eval}: \mathcal{P}(\text{def}) \times \Sigma \rightarrow \{\text{true, fail, suspend}\} \times \Sigma
\]

3.1.2. Configurations. In order to define the system configuration we need first to define the configuration of a grain. Every instance of a grain is identified by the name of the grain and the names of the variables and channels of the remote call that created the grain. However, to simplify the notation we define a set of active grain identifiers \( I_d \) and the following function:

\[
\nu: I_d \times G \rightarrow I_d,
\]

which will assign to each grain identifier the name of the grain. We also assume that the following function exists:

\[
\nu: I_d \times \mathcal{P}(I_d) \times G \rightarrow I_d,
\]

such that \( \nu(X, g) \notin X \) and \( T(\nu(X, g)) = g \). The function \( \nu \) returns for a given set of active grain identifiers \( X \) and the name of a grain \( g \), not included in \( X \), and bound to \( g \). This function exists and can be defined in different ways [27].

The state of a grain can be represented by:

\[
g(I, \tilde{A}, E, \sigma), I \in I_d, A \in G, E \in E, \sigma \in \Sigma
\]

where \( E = \{\text{active, suspended}\} \).

The remote metacalls are represented by:

\[
\text{rc}(A, St, Ctrl), A \in G, St, Ctrl \in \mathcal{P}(G)
\]

As we also need to model the communication system, we will use the following functions (\( \tilde{\sigma} \) represents a substitutions sequence).

\[
\begin{align*}
\mathcal{P} & = \mathcal{P}(I_d \times \Sigma^+) \\
\text{prod} & : \mathcal{P} \times \Sigma \rightarrow \mathcal{P} \\
\text{cons} & : \mathcal{P} \times I_d \rightarrow \mathcal{P} \\
\text{sust} & : \mathcal{P} \times I_d \rightarrow \Sigma \\
\text{prod}(Sc, \sigma) & = \{ (I, \tilde{\theta}): (I, \tilde{\sigma}) \in \mathcal{P} \} \\
\text{cons}(Sc, I) & = \mathcal{P} - \{ (I, \sigma, \tilde{\sigma}) \} \cup \{ (I, \tilde{\sigma}) \} \\
\text{sust}(Sc, I) & = \sigma \text{ si } (I, \sigma, \tilde{\sigma}) \in \mathcal{P} \\
\lambda & \text{ otherwise}
\end{align*}
\]
The set $\mathcal{S}_c$ is used to model the communication system state; prod, cons and sust are used to modify the state of the system during the computation ($\lambda$ represents the empty sequence).

Another important feature of $\mathcal{BML}$ that needs special representation is the behaviour of the output unification (it can suspend when an output channel is instantiated with terms with free variables). Output unification is represented by the term outunif($\sigma, \sigma'$); an output unification must proceed with the unification of free variables, but it must suspend the instantiation of logic channels until all the terms become ground. We use another term to model this suspension:

$$\text{chanout}(c, A), c \in \mathcal{S}_c \times \mathcal{S}_d$$

A term like this will be added to the objective for every output logic channel that is instantiated to non-ground terms; this term will be reduced when all the free variables become ground.

Finally, the system state $\Gamma$ can be defined as:

$$\Gamma = \mathcal{S}_c^* \times \Sigma \times \mathcal{S}_c \times \mathcal{S}_d$$

where:

- The first element represents the actual objective, which is a conjunction of predicates and terms representing remote metacalls and grains in execution.
- The second term represents the current substitution in the main node.
- The third one represents the communication system state.
- The last term represents the set of current active grain identifiers.

The initial configuration will be as follows:

$$< \lambda; \varepsilon; \emptyset; \{I_0\}>$$

The communication system is modelled by the empty set, and the grain identifier set will contain the identifier of the main node, $I_0$ (this identifier will be used later in the transition rules that model communication between grains). The terminal configurations can be defined in a similar way:

$$<\text{true}; \varepsilon; \emptyset; \{I_0\}> <\text{fail}; \varepsilon; \emptyset; \{I_0\}>$$

3.1.3. Transition rules. In this section we present the transition rules of the $\mathcal{BML}$ operational semantics. The rules related to the selection of candidate clauses are similar to those of the underlying CLL. We define the transition relation $\rightarrow \subseteq \Gamma^2$ as the smallest relation verifying the following rules:

3.1.3.1. Candidate Clauses

These rules model the reduction process; we will be able to solve $\rightarrow$ if we can find a renamed clause with head $H$ that can be unified with $\lambda$; apart from this, the evaluation of $\bar{G}$ must succeed.

$$\exists H \leftarrow \bar{G} : \exists \mathsf{mgu}_i (\lambda \sigma, H), \exists \mathsf{mgu}_i (\lambda \sigma, H)$$

$$\text{eval} (\bar{G}, \mathsf{mgu}_i (\lambda \sigma, H)) = (\text{true}, \sigma')$$

$$\sigma' \mid \text{invar}(\lambda \sigma) = \varepsilon$$

$$<\lambda; \sigma; C; I d> \rightarrow <\text{outunif}(\lambda \sigma, H \sigma'), \exists \mathsf{mgu}_i (\lambda \sigma, H); \sigma'; \sigma'; C; I d>$$

$$\forall H \leftarrow \bar{G} : \exists \mathsf{mgu}_i (\lambda \sigma, H) \text{ such that } \text{eval} (\bar{G}, \mathsf{mgu}_i (\lambda \sigma, H)) = (\text{true}, \sigma')$$

$$\forall H \leftarrow \bar{G} : \exists \mathsf{mgu}_i (\lambda \sigma, H) \text{ such that } \text{eval} (\bar{G}, \mathsf{mgu}_i (\lambda \sigma, H)) = (\text{false}, \sigma')$$

$$<\lambda; \sigma; C; I d> \rightarrow <\text{fail}; \varepsilon; C; I d>$$

(2)

3.1.3.2. Output Unification

Output unification succeeds if $\mathsf{mgu}_i (\lambda \sigma, H \sigma')$ exists; in addition to this, it is necessary to distinguish between logic variables and channels. In the case of logic variables unification proceeds in the same way as in other CLLs, i.e. the resulting substitution is composed with the substitution currently computed. In the case of output channels, this composition will be delayed until all the terms become ground. Another important thing to take into account is that each different channel can become ground at different moments; in order to model this, a different term is created for
every channel. Rule 3 models the creation of the terms that represent the unification of logic channels (a term chanout \((o_i, \theta_i)\) is created for each channel). The final instantiation of the channels is modeled in rule 4. When the channel becomes ground the communication system is modified adding the corresponding substitutions to the channel of every active grain. The instantiation of a logic channel is achieved immediately in the local substitution, but a new transition is needed in every active grain to achieve the local substitution. In this way we reflect the asynchronous nature of communications between grains.

\[
\exists \text{mgu}_o(\lambda \sigma, \Lambda \sigma'), \text{cout}(\text{sop}(\text{mgu}_o(\lambda \sigma, \Lambda \sigma')) = (o_1, \ldots, o_n), \theta_i = \text{mgu}_o(\lambda \sigma, \Lambda \sigma') | o_i \]

where 3mgu, (\Lambda \sigma, \Lambda \sigma')

\[
\text{cout}(\text{outunif}(\Lambda \sigma, \Lambda \sigma'), \text{B}; \sigma; C; \text{Id}) \rightarrow \\
\text{B, chanout}(o_1, \theta_1), \ldots, \text{chanout}(o_n, \theta_n); \sigma \text{mgu}_o(\Lambda \sigma, \Lambda \sigma') \text{cvar}(\Lambda \sigma); C; \text{Id} \rightarrow \\
\text{fail}; C; \text{Id} \rightarrow \\
\text{ground}(\Lambda \sigma) \rightarrow \\
\text{cvar}(\Lambda \sigma, C; \text{Id}) 
\]

3.1.3.3. Communication between Grains

These rules model the reception of messages in the grains. If there exists a substitution in the communication structure corresponding to the grain identifier, this substitution is consumed and composed with the local substitution. Rule 6 is used to model message reception in the main processor.

\[
sust(C, I) \neq \lambda 
\]

\[
<g(I, \bar{A}, E, \theta); \sigma; C; \text{Id} \rightarrow <g(I, \bar{A}, E, \theta . sust(C, I)); \sigma; cons(C, I); \text{Id} 
\]

3.1.3.4. Concurrence

We use interleaving to model concurrency. There is no distinction between predicates and grains; the execution of a grain is interleaved with the execution of the rest of the processes in the current goal. Concurrence inside the grain is modeled in a similar way (rule 16).

\[
<\bar{X}; \sigma; C; \text{Id} \rightarrow <\bar{X}; \sigma; C'; \text{Id'} \rightarrow <true; \sigma'; C'; \text{Id'} \rightarrow <true; \sigma'; C'; \text{Id'} \rightarrow <fail; \sigma'; C'; \text{Id'} \rightarrow <fail; \sigma'; C'; \text{Id'} \rightarrow \\
<\bar{Y}; \bar{X}; \sigma; C; \text{Td} \rightarrow <\bar{Y}; \bar{X}; \sigma; C'; \text{Td'} \rightarrow <\bar{Y}; \bar{X}; \sigma; C'; \text{Td'} \rightarrow <\bar{Y}; \bar{X}; \sigma; C'; \text{Td'} \rightarrow <fail; \sigma'; C'; \text{Td'} 
\]

3.1.3.5. Grain Creation

Rule 8 models grain creation. A grain is created when a remote metacall is ready for execution (i.e. all the logic variables have become ground). The grain will contain as the initial goal the conjunction of the exception management predicate and the goal specified in the remote metacall. A new identifier is assigned to the grain and the set of identifiers is modified.

\[
\text{ground}(\lambda \sigma) \rightarrow \\
I = v(\text{Id, name}(\lambda)) 
\]

\[
<\text{rc}(A, \text{St, Ct}); \sigma; C; \text{Id} \rightarrow \\
<g(I, A, \text{excep(St, Ct)}, \text{active}; \sigma \text{var}(\lambda)); \sigma; \text{CU}{(I, \lambda)}; \text{IdU}(I) > 
\]
3.1.3.6. Grain Execution Control

The execution of a grain can be controlled by using the primitives: suspend, continue and stop. A failure inside a grain has the same effect as the execution of the stop primitive, so no additional rule is needed. Rules 9 to 11 model suspension in a grain. The state of the grain changes to suspended and the only predicate that can proceed is the exception management process.

\[ \langle g(I, (\bar{x}, \text{suspend}, \bar{y}), \text{active}, \theta) ); \sigma; C; Id \rangle \rightarrow \langle g(I, (\bar{x}, \bar{y}), \text{suspended}, \theta) ); \sigma; C; Id \rangle \] (9)

\[ \langle \text{excep}(\text{St}, \text{Ct}); \sigma; C; Id \rangle \rightarrow \langle \bar{A}; \sigma'; C'; Id' \rangle \]

\[ \langle g(I, (\bar{x}, \text{excep}(\text{St}, \text{Ct}), \bar{y}), \text{suspended}); \sigma; C; Id \rangle \rightarrow \langle g(I, (\bar{x}, \bar{y}), \text{suspended}); \sigma'; C'; Id' \rangle \] (10)

\[ \langle g(I, \text{continue}, \text{suspended}, \theta); \sigma; C; Id \rangle \rightarrow \langle g(I, (\bar{x}, \bar{y}), \text{active}, \theta); \sigma; C; Id \rangle \] (11)

3.1.3.7. Termination in a Grain

Rule 12 models a successful termination in a grain. The Status channel is instantiated to the term success. The failure is modelled in rule 13; a failure is considered an exception, and the only effect is the creation of a term representing that exception (raise(fail)). The failure of the exception manager is modelled in rule 14.

\[ \langle g(I, \text{excep}(\text{St}, \text{Ct}), \text{active}, \theta); \sigma; C; Id \rangle \rightarrow \langle \text{true}; \sigma; \text{prod}(C, \{\text{St}/[[\text{success}]]\}); Id \rangle \] (12)

\[ \langle \bar{A}; \sigma; C; Id \rightarrow \langle \text{false}; \rangle \]

\[ \langle g(I, (\bar{A}, \text{excep}(\text{St}, \text{Ct})), \text{active}, \theta); \sigma; C; Id \rangle \rightarrow \langle g(I, (\text{raise}(\text{fail}), \text{excep}(\text{St}, \text{Ct})), \text{active}, \theta); \sigma; C; Id \rangle \] (13)

\[ \langle \text{excep}(\text{St}, \text{Ct}); \sigma; C; Id \rightarrow \langle \text{false}; \rangle \]

\[ \langle g(I, (\bar{A}, \text{excep}(\text{St}, \text{Ct})), \text{active}, \theta); \sigma; C; Id \rangle \rightarrow \langle \text{true}; \sigma; \text{prod}(C, \{\text{St}/[[\text{failed}]]\}); Id \rangle \] (14)

3.1.3.8. Exception Management

This rule models the management of internal exceptions. The effect of an internal exception is the instantiation of the head of the control list to the term that represents the exception.

\[ \langle g(I, \bar{A}, \text{raise}(T), \text{excep}(\text{St}, \text{Ct}), \text{active}, \theta); \sigma; C; Id \rangle \rightarrow \langle g(I, \bar{A}, \text{excep}(\text{St}, \text{Ct})), \text{active}, \theta; \{\text{Ct}/[[T\text{Ct}]]\}); \sigma; C; Id \rangle \] (15)

3.1.3.9. Computation inside a Grain

This rule models the evolution of predicates inside a grain. If a goal inside a grain can proceed following the previous rules, it will also be able to proceed inside the grain.

\[ \langle \bar{A}, \theta; C; Id \rangle \rightarrow \langle \bar{B}, \theta'; C'; Id' \rangle \]

\[ \text{fail} \]

\[ \langle g(I, \bar{A}, \text{active}, \theta); \sigma; C; Id \rangle \rightarrow \langle g(I, \bar{B}, \text{active}, \theta'); \sigma; C', Id' \rangle \] (16)

3.1.4. Operational semantics. The operational semantics can be defined as follows:

\[ O: \mathcal{G} \rightarrow \mathcal{P}(\Sigma) \]
as

\[ O[\text{true}] = \{ \varepsilon \} \]
\[ O[\overline{A}] = \{ \sigma |_{\overline{A}} \} \]
\[ \{ \overline{A}; e; \emptyset; \overline{I}_e \rightarrow^* \overline{true}; \sigma; C; \overline{I}_d \} \}

From an initial goal composed of a conjunction of predicates and remote metacalls, the corresponding substitution is obtained if there exists a transition sequence that finishes with the empty conjunction.

The operational semantics defined in this way gives information about the success set, but it does not provide information about deadlock or failure situations. In order to incorporate this kind of information we define the following notion of observables:

\[ O_1; \Gamma \rightarrow \mathcal{P}(\Sigma^o) \]

where

\[ \Sigma^o = \Sigma^* \cup \mathcal{P}(\delta \cup \Sigma^o) \]

are the possible finite

\[ (s \in \Sigma^+) \text{ or infinite } (s \in \Sigma^o) \]

sequences of substitutions (\{\delta\} denoting failure or deadlock). The function \( O_1 \) can be defined as follows:

\[ s \in O_1(\langle \overline{A}; \sigma; C; \overline{I}_d \rangle) \]

if and only if one of the following conditions is satisfied:

1. \( s = \sigma_{j_1}...\sigma_n (n \geq 0) \)
\[ \langle \overline{A}; \sigma_n; C; \overline{I}_n \rangle \rightarrow ... \rightarrow \langle \overline{true}; \sigma; C; \overline{I}_d \rangle \]

2. \( s = \sigma_{j_1}...\sigma_n (n \geq 0) \)
\[ \langle \overline{A}; \sigma_n; C; \overline{I}_n \rangle \rightarrow ... \rightarrow \langle \overline{true}; \sigma; C; \overline{I}_d \rangle \]

3. \( s = \sigma_{j_1}...\sigma_n \)
\[ \langle \overline{A}; \sigma_n; C; \overline{I}_n \rangle \rightarrow ... \rightarrow \langle \overline{true}; \sigma; C; \overline{I}_d \rangle \]

Case (1) stands for finite normally terminated computations, (2) for finite abnormally terminated computations and (3) for infinite computations.

3.2. Real-time semantics

In the previous section we have presented the ORL operational semantics, but we have not taken into account the language's real-time features. In this section we extend the semantics to model the real-time aspects of ORL. In order to do this it is necessary to modify the configurations used to represent system states. We consider a global time scale, and the transitions will always reference this time scale which will be included in the configuration. We also assume that the time consumed in the transitions is bounded by \( \tau_{\text{max}} \).

The real-time semantics is based on a timed transition system and we use a modified interleaving concurrency model to take into account real-time process priorities.

3.2.1. Definitions and configuration. Let us define a new set \( \mathcal{F}_t \), composed of tuples formed by an atom and different instances of the special predicate wait/5. If \( A \in \mathcal{F}_t \) a typical \( \mathcal{F}_t \) element is defined as follows:

\[ A_t := A_0 \]
\[ A_t := A_t + \text{wait}(A,H,\sigma,B,1) \]
where
\[ H, B \in A, \sigma \in \sum \text{ and } t \in \mathcal{N}, \]
the set of positive integers.

We will represent
\[ A + \text{wait}(A, H_i, \sigma_i, B_i, t_i) + \ldots + \text{wait}(A, H_n, \sigma_n, B_n, t_n) \]
by \[ A + \sum_{i=1}^{n} \text{wait}(A, H_i, \sigma_i, B_i, t_i) \]. We will assume that all the clauses have temporal guards, i.e. all the guards have an \texttt{after(T)} primitive, where \( T \) is a positive integer (including 0):
\[ H \leftarrow \mathcal{G}, \texttt{after(T)}: \mathcal{B}. \]

The set of system states \( \Gamma \) is given by:
\[ \Gamma = \mathcal{S}, \times \sum \times \mathcal{F} \times \mathcal{I} \times \mathcal{N} \times \mathcal{N}. \]

In order to include real-time it is only necessary to modify a few of the rules:

- Communication through logic channels (rules 4, 5, and 6). We consider that communication via logic variables takes no time, but that transmission through logic channels takes a limited amount of time (\( \tau_{\text{max}} \)).
- The scheduling policy is not a normal interleaving. Real-time processes have a higher priority (rule 7).
- Clause selection (rule 1).

The rest of the rules are modified by simply adding a new condition to model the passage of time.

**Conditions**
\[ t < t' \leq t + \tau_{\text{max}} \]
\[ \langle \mathcal{A}; \sigma; C; \mathcal{I}; d; t \rangle \rightarrow \langle \mathcal{A}; \sigma'; C'; \mathcal{I}; d'; t' \rangle \]

Another aspect which it is necessary to modify is the representation of the communication system in order to model the delay in the transmission of substitutions. Let us define the set:
\[ \mathcal{F} = \mathcal{P}(\mathcal{I} \times (\sum \times \mathcal{N}))^+ \]
and the functions
\begin{align*}
\text{prod}: \mathcal{F} \times \sum \times \mathcal{N} & \rightarrow \mathcal{F} \\
\text{cons}: \mathcal{F} \times \mathcal{I} & \rightarrow \mathcal{F} \\
\text{sust}: \mathcal{F} \times \mathcal{I} & \rightarrow \sum \times \mathcal{N} \\
\text{prod}(Sc, \sigma, \tau) &= \{ (I, (\tilde{\sigma}, \tau, \tilde{\tau})): (I, (\tilde{\sigma}, \tilde{\tau})) \in \mathcal{F}_c \} \\
\text{cons}(Sc, I) &= Sc - \{ (I, (\sigma, \tilde{\sigma}, \tilde{\tau})) \} \cup \{ (I, (\tilde{\sigma}, \tilde{\tau})) \} \\
\text{sust}(Sc, I) &= \{ (\sigma, \tau) \in (\mathcal{F} \times \mathcal{I}) \leftarrow \mathcal{P} \} \text{ if } \mathcal{F} \end{align*}

3.2.2. The transition system. The new transition system \( \rightarrow \mathcal{F} \Gamma^2 \) will be defined based on the one described in previous sections and changing the following rules:

3.2.2.1. Clause Selection
There are two rules to model clause selection. The first one models the selection of a temporal clause. A wait predicate is generated for every clause with a deadline different from 0. This predicate
indicates that the reduction is postponed until \( t + T \). This rule has a higher priority than the other selection rule, giving priority to the reduction of real-time predicates. Rule 3 shows how a clause with a temporal guard is finally selected.

\[
A_t = A + \sum_{i=1}^n \text{wait}(A, H_i, \sigma_i, B_i, t_i), A \in \mathcal{A}, \quad (1')
\]

\[
\exists H \leftarrow \tilde{G}, \text{after}(T) : \tilde{B} \in \mathcal{W}(A), \text{con } T \neq 0, \exists \text{mgu}_1(A_\sigma, H)
\]
\[
\text{eval}(\tilde{G}, \text{mgu}_1(A_\sigma, H)) = (\text{true}, \sigma')
\]
\[
\sigma' \text{linvar}(A_\sigma) = \varepsilon
\]
\[
t < t' \leq t + t_{\text{max}}
\]
\[
\mathcal{A}_t; \sigma; C; Id; t > \rightarrow \mathcal{A}_t + \text{wait}(A, H, \sigma, \tilde{B}, t + T); \sigma; C; Id; t' >
\]
\[
\text{if rule 1 is not satisfied}
\]
\[
\exists H \leftarrow \tilde{G}, \text{after}(0) : \tilde{B} \in \mathcal{W}(A), \exists \text{mgu}_1(A_\sigma, H)
\]
\[
\text{eval}(\tilde{G}, \text{mgu}_1(A_\sigma, H)) = (\text{true}, \sigma')
\]
\[
\sigma' \text{linvar}(A_\sigma) = \varepsilon
\]
\[
t < t' \leq t + t_{\text{max}}
\]
\[
\mathcal{A}_t; \sigma; C; Id; t > \rightarrow \text{outunif}(A_\sigma, H_\sigma'); \sigma' ; C; Id; t' >
\]
\[
t_t = \min(t, i^2i^2n^2t)
\]
\[
t < t' \leq t + t_{\text{max}}
\]
\[
\mathcal{A}_t + \sum_{i=1}^n \text{wait}(A, H_i, \sigma_i, B_i, t_i); \sigma; C; Id; t > \rightarrow \text{outunif}(A_\sigma, H_\sigma'), B_i; \sigma_j; C; Id; t' >
\]

### 3.2.2.2. Output Unification

This rule is modified to model transmission delay through logic channels. The communication structure will store the time when unification was achieved. This information will be used in the rules for the reception of substitutions by the grains.

\[
\exists \text{mgu}_n(A_\sigma, H_\sigma') , \text{cout}(\text{sop}(\text{mgu}_n(A_\sigma, H_\sigma'))) = o_1, \ldots, o_n, \theta_i = \text{mgu}_n(A_\sigma, H_\sigma')|oi (5')
\]
\[
\text{sop}(\theta) = x: \theta x \neq x
\]
\[
t < t' \leq t + t_{\text{max}}
\]
\[
| \quad \ldots \exists \text{mgu}_n(A_\sigma, H_\sigma')
\]
\[
t < t' \leq t + t_{\text{max}}
\]
\[
\text{outunif}(A_\sigma, H_\sigma'), \tilde{B}; \sigma; C; Id; t > \rightarrow \text{chout}(o_1, \theta_1), \ldots, \text{chout}(o_n, \theta_n); \sigma' \text{mgu}_n(A_\sigma, H_\sigma')|\text{cvar}(A_\sigma); C; Id; t' >
\]
\[
| \quad \text{fail}; t; C; Id; t >
\]
\[
\text{ground}(\theta_\sigma)
\]
\[
\text{chout}(o, \theta), \tilde{B}; \sigma; C; Id; t > \rightarrow \text{chout}(\tilde{B}; \sigma \theta; \text{prod}(C, \theta, t); \text{Id}; t')
\]
\[
(6')
\]
3.2.2.3. Communication between Grains

These rules show how the reception of messages is achieved by the grains; the substitution can be composed with the local substitution only if the global time scale is greater than the time when the message was produced plus the transmission time $t_{\text{trans}}$.

$$sust(C, T) = (\sigma, \tau)$$

$$t \geq t + t_{\text{trans}}$$

$$<g(I, A, E, \theta); \zeta; C; Id; t > \rightarrow <g(I, A, E, \theta + \sigma .); \zeta; cons(C, I), Id; t' >$$

$$sust(C, I_0) = (\sigma, \tau)$$

$$t \geq t + t_{\text{trans}}$$

$$<a; \zeta; C; Id; t > \rightarrow <a; \sigma; cons(C, I), Id; t >$$

3.2.2.4. Concurrence

This rule shows how the predicates are selected for reduction. The rule gives a higher priority to the reduction of the predicates.

$$< \bar{X}; \sigma; C; Id; t > \rightarrow < \bar{X}'; \sigma'; C'; Id'; t' > | < \text{true}; \sigma'; C'; Id'; t' > | < \text{fail}; \varepsilon; C'; Id'; t' >$$

$$\bar{Y}$$ does not include any wait$(A, H, \theta, B, \sigma)$ atom with $s2 t$

$$< \bar{X}, \tilde{X}; \sigma; C; Id; t > \rightarrow < \bar{X}', \tilde{X}'; \sigma'; C'; Id'; t' > | < \bar{Y}; \sigma'; C'; Id'; t' > | < \bar{Y}; \sigma'; C'; Id'; t' > | < \bar{Y}; \sigma'; C'; Id'; t' >$$

$$< \bar{Y}; \sigma'; C'; Id'; t' > | < \text{fail}; \varepsilon; C'; Id'; t' >$$

The definition of the real-time semantics is achieved in the same way as in the previous section. The observables are modified to include the time when the substitutions are generated.

4. IMPLEMENTATION

Two previous implementations of subsets of $\mathcal{B}\mathcal{A}\mathcal{L}$ exist: the first one was developed on a Parlog system using metainterpretation [6] and the second one was an interpreter of the language developed in C on a Unix workstations system [24]. We will now present a new compilation based version for an abstract machine for a complete version of $\mathcal{B}\mathcal{A}\mathcal{L}$. We have based our abstract machine on the sequential abstract machine for Flat Parlog [28].

The implementation is based on the emulation of a distributed abstract machine, and it has been developed on a network of Unix workstations, using C and TCP/IP sockets for communication.

In the next subsection the $\mathcal{B}\mathcal{A}\mathcal{L}$ execution model is discussed and in the next one we present the basic features of the $\mathcal{B}\mathcal{A}\mathcal{L}$ abstract machine.
4.1. The SWRL execution model

The execution model provides the basic control mechanisms to execute SWRL programs on the abstract machine. We have selected a process oriented model, based on an AND tree execution model [29]. This has been inherited from an existing sequential abstract machine for Flat Parlog [28], but has been adapted to include SWRL’s real-time and remote execution features, and to allow execution of control metacalls which implies the control of process suspension, cancellation and failure.

Figure 4 shows process states. When a process is created its state is active, and it remains in this state until it is scheduled. A process is in the executing state until it finishes or suspends. If a process succeeds, it finishes normally returning the memory assigned to the system and the computation goes on.

A process enters the suspension on variable state when no candidate clause can be selected for reduction without instantiating variables. A process can be suspended on more that one variable at the same time, and will enter the active state when any of these variables is instantiated. A process enters the suspension on children state when it is waiting for its children to finish and when all of them finish it also enters the active state. A process enters the suspension on time state when no candidate clause can be selected for reduction and there exists at least one successfully evaluated clause with a temporal guard. Once the time indicated in the temporal guard has elapsed, the process enters the executing state.

A process can be suspended at the same time on variables and on temporal guards. In this situation it can enter the active state if one of its suspension variables is instantiated, or it can enter the executing state when the time indicated in the temporal guard has elapsed. As a special situation, the control metacall can on children and on its Control variable at the same time.

Process execution consists of two phases: Test and Spawn. When a process is scheduled after a suspension, execution will continue from where its continuation pointer indicates. In this model

Fig. 5. Test phase
there are two continuation pointers: one for the temporal guards and another for suspension on variables or children. These continuation pointers can be modified during process execution.

In the Test phase (Fig. 5), the machine tries to find a candidate clause to commit the goal. In order to achieve this it matches the input arguments and it evaluates the guards of the different clauses that define the process. In the Spawn phase (Fig. 6), once the process has committed a clause, it carries out output unification, and if this succeeds and there is a body in the clause, it also achieves the creation of the child processes.

4.1.1. The control metacall. In a pure Flat Parlog execution model, if any process fails, the whole computation fails. In the BWY model, if a process fails, and it is a descendent of a control metacall, this metacall succeeds returning a failed status, but the computation itself will not fail. Apart from
this, in BRL, the control metacall is in charge of exception handling inside the grain. In order to achieve this, the execution model of a control metacall is rather more complex. A metacall process can be in the following different states (Fig. 7):

- Active and not processed (state 0). The metacall enters this state when it is created and scheduled for the first time. When it enters this state, it will detect if an exception has occurred, handling it, after which it will suspend on the control variable.
- Suspended on control variable. The metacall will remain in this state until all its children have finished or an exception is raised, instantiating the control variable. In any case, it will enter an active state. Three different situations (substates) can be distinguished in the suspended on control variable state; in any of these substates the metacall will remain suspended on the control variable, but the behaviour of its children will change:
  - Suspended on control variable with all the children (executed until that moment) succeeding (state 1).
  - Suspended on control variable with at least one child failure (state 2).
  - Suspended on control variable after processing a suspension exception (state 5).
- Active due to the instantiation of the control variable. After the instantiation of the control variable, the metacall will wake up to manage the exception. Different substates can also be distinguished in this state (depending on the situation of the children while the metacall is in the execution queue):
  - Active in the queue with success and with pending children (state 3).
  - Active in the queue with fail and with pending children (state 4).
  - Active in the queue with success and without pending children (state 6).
  - Active in the queue with fail and without pending children (state 7).
  - Woken up by the last children (with success or fail, states 8 and 9). The last child of the metacall has finished its execution and has woken up its parent control metacall.

4.1.1.1. *Indirect Killing and Suspension*

The execution model of a metacall's descendent processes is slightly different to the model of the other processes. As we have seen, in Flat Parlog, when a process fails the whole computation fails. In BRL, if a process fails, it has to test if it is a descendent of a control metacall before finishing normally. If it is not, the whole computation fails, as in the Flat Parlog model; otherwise, it sets a flag in its control metacall parent indicating a fail (the metacall will change its state). Before starting the execution of any scheduled process, the scheduler tests whether the process is a descendent of a control metacall 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 descendent processes of a control metacall that has been cancelled (when the exception handling process executes *stop*). Suspension control inside a control metacall is achieved in a similar way (indirect suspension). A suspension flag is set in the control metacall, and the descendent processes will suspend on the metacall's Control variable when they are scheduled. When the exception handling process executes *continue* all the descendent processes will wake up, setting its state to *active*. These indirect ways of implementing suspension, cancellation and failure are very simple and efficient, and they are well-suited to the real-time features of our language.

4.1.2. *Remote execution.* A process is completely executed in a processor. When a grain process is created by a remote metacall, that process will be executed in the processor indicated by the first argument of the metacall. Each processor has a statistics table with information about the load of each processor in the system (CPU and memory). This information is updated by using some additional information that is included in each exchanged message. If the first argument of the metacall is the term `auto(X)`, then the system will allocate the process in the less loaded processor and will instantiate the X variable with the processor's identifier. When a remote metacall is executed in the i processor in order to execute a grain in the j processor, a *call/3* process will be created in the j processor, which will be in charge of executing the process indicated in the remote metacall. All the child processes of the grain will be descendants of the *call/3* process, and all of them will be controlled by this control metacall. In the original processor (i), a *remote.call* process
is created. This process will suspend on its Status variable until the remote call/3 process finishes its execution instantiating this variable. This remote call process will be the representative of the remote process execution in the original processor.

Communication between processes that are placed in different processors will be achieved by using logic channels. Remote metacall execution implies the creation of, at least, two logic channels: The Status channel (input in the original processor and output in the remote one) and the Control channel (output in the original processor and input in the remote one), to control process execution and exceptions. The exception handling process will be simply another child of the call/3 process, and will be suspended on the Control variable and be able to instantiate the Status variable. Besides these control channels, all the channels that appear in grain definition will be created, with the direction indicated in this definition.

4.2. The BRL abstract machine

Each of BRL's distributed implementation nodes is based on the Flat Parlog sequential abstract machine (which has been extended to cope with the characteristics of our language), with the addition of three new modules which are in charge of managing the language's communication and distributed execution: external communication, incoming calls and outgoing call modules (Fig. 8).

4.2.1. Data areas and structures. The abstract machine mainly uses three different data structures:

4.2.1.1. Process Structure

The process structure has a fixed size and contains enough information to execute a process. When a process is created a process structure is allocated in the corresponding area (process stack). A process is really synonymous with a process structure. In each processor, the PS register of the abstract machine points to the process structure of the current executing process. The process structure has 17 fields, as appears in Fig. 9, and only the 8 fields on the left are from the original Flat Parlog abstract machine. The rest of the fields have been added to implement BRL's new features (metacalls, priorities, real-time, etc.). The meaning of the new fields is as follows:

- **Status Flag**: this is used in metacall processes, and it indicates the state of the metacall (as was explained in the execution model).
- **Root Process Pointer**: this is a pointer to the parent metacall process. It is used to implement indirect killing and suspension.
- **Time Continuation Pointer**: this is a pointer to the next instruction to execute when a time guard succeeds.
- **Priority**: this field is used in scheduling, to allocate the process in the appropriate execution queue.
- **Deadline**: the deadline of real-time processes.
- **Creation Time**: this is used by the scheduler to compute the deadline of real-time processes.
- **Suspension on Variable**: this field indicates that the process is in the suspension on time and suspension on variable states at the same time. It is used to delete the process from the variable process suspension list when a time guard succeeds.
- **Time Queue Pointer**: this field has a symmetrical meaning to the previous one. When a process that is suspended on a variable and on a timed guard is woken up by a variable instantiation the process has to be deleted from the time queue.
- **Confirmation Count**: this is a count of the output channels created by the descendants of a metacall that has not been confirmed. It is used in termination detection.

### 4.2.1.2. Instruction Structure

The instruction structure is composed of a field indicating the instruction code, and the necessary arguments for that instruction.

### 4.2.1.3. Working Word

The working word of the abstract machine is composed of two fields: the flags field, which indicates the kind of data of the word (list, atom, channel,...) and the value field, which contains a value or a pointer to a data area, depending on the content of the flags field.

The memory of each abstract machine processor is divided into different data areas. These data areas will contain the data structures that have been described. The data areas are shown in Fig. 10; we will only describe the new data areas added to the Flat Parlog abstract machine.

<table>
<thead>
<tr>
<th>Parent Pointer</th>
<th>Status Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Count</td>
<td>Root Process Pointer</td>
</tr>
<tr>
<td>Continuation Pointer</td>
<td>Time Continuation Pointer</td>
</tr>
<tr>
<td>Argument Pointer</td>
<td>Priority</td>
</tr>
<tr>
<td>Argument Count</td>
<td>Deadline</td>
</tr>
<tr>
<td>Environment Pointer</td>
<td>Creation Time</td>
</tr>
<tr>
<td>Suspension List Pointer</td>
<td>Suspension on Variable</td>
</tr>
<tr>
<td>Free List Pointer</td>
<td>Time Queue Pointer</td>
</tr>
<tr>
<td></td>
<td>Confirmation Count</td>
</tr>
</tbody>
</table>

**Fig. 9. Process structure**
4.2.1.4. **Input Channels Table**

The *input channels table* (Fig. 11) will contain information about the input channels; there is an entry in the table for each channel and each channel variable will point to its entry in this table. The information of each entry is the following:

- **Heap Index:** this field points to the heap index where the data received by the channel will be stored.
- **Processor:** the remote processor identifier of the channel.
- **Root Metacall:** this points to the root metacall process. It is used to maintain the count of confirmations of output channels when these channels are created dynamically.
- **Free List Pointer:** this points to the following free hole in the table.
- **Hole:** this indicates that this entry in the table is free.

4.2.1.5. **Output Channels Table**

This table (Fig. 12) is symmetrical to the previous one. The output channel variables will point to its corresponding entry in this table; each entry will contain the following information:

- **Channel:** this contains the identifier of the associated remote input channel. This value is unknown when the channel is created and it will be filled when a confirmation message for the output channel is received.
- **Pending Message:** when an output channel is instantiated before it is confirmed, the heap index of the data to be sent is stored in this field. This data will be sent immediately when the confirmation arrives.

The rest of the fields have the same meaning that the last four fields of the input channels table.
4.2.1.6. **Active Processors Table**

The *active processors table* contains the processor identifiers that can be used in the remote metacalls. It is used in system initialization to establish the communication links between the processors. It also contains statistics about the processor's load, which will be used when the processor is not specified in a metacall where the process has to be executed.

4.2.1.7. **Execution Queues**

Three different execution queues exist, one for each different priority. The time queue is ordered by deadlines. All the queues contain pointers to the process stack.

4.2.2. **Instruction set.** The abstract machine code for a DRL procedure consists of instructions to try alternative clauses followed by the code for each clause. The code for a clause can be further divided into Test phase and Spawn phase instructions. The instruction set is similar to the one defined in [28]. Instructions have been added to deal with logic channels, and the implementation of some of the original instructions have also been modified to take into account the language's new features.

4.2.3. **Unification and logic channels management.** We have seen in Section 1.1 that one of the key features of the distributed implementation of CLLs is distributed unification. In DRL all the unifications are achieved locally, so a distributed unification algorithm is not necessary. The unification algorithm of Flat Parlog has been extended to cope with logic channels.

The DRL implementation does not distinguish between active and passive unification like other distributed implementations: the unification algorithm is the same in the Test phase and in the clause body. The main difference between Flat Parlog and DRL unification algorithms is that output unification can suspend, just like the input one; if a logic term contains free variables when an output channel is instantiated this unification will suspend until the term becomes ground.

![Fig. 11. Input channels table](image-url)
Another key difference is the instantiation of non-confirmed channels; in this case unification will succeed, but the transmission of the term will be delayed as we described previously. Finally, the algorithm has to deal with the unification of logic channels. We will briefly describe the behaviour of the algorithm in this case:

\[ \text{Output}_\text{channel} = \text{Input}_\text{channel} \]

When an input and an output channel are unified it is necessary to distinguish two different situations:

- The input channel has been created by the same processor.
- The input channel corresponds to a remote output channel.

In the first case, the input channel will be a dynamically created channel and the channel will have to be transmitted in the data message; when this channel is received in the remote processor, a new output channel will be allocated. In the second case, the unification will suspend until the input channel is instantiated.

\[ \text{Output}_\text{channel} = \text{Output}_\text{channel} \]

When two output channels are unified, a reference between them is created, in such a way that when any of the channels is instantiated, the term is also sent by the other channel.

\[ \text{Input}_\text{channel} = \text{Input}_\text{channel} \]
\[ \text{Input}_\text{channel} = \text{Output}_\text{channel} \]

The behaviour of the unification algorithm is the same in both cases, and the unification proceeds as in the case of two common variables. The unification will succeed if, later, both variables are instantiated to the same term but will otherwise fail.
4.2.3.1. Channel Transmission

Another important aspect to take into account in unification is the transmission of logic channels. If the channels have been created locally, the channels are managed in the way we have previously described (for each input channel the remote processor will create an output channel and vice versa), but in other cases (the channels are the pairs of other remote channels), it is necessary to take into account other different factors. We will describe each of the cases:

Output$_{\text{channel transmission}}$

If the output channel has been assigned to another processor, it is necessary to create a new output channel, that will be sent to the remote processor as a new locally created channel, and both the new channel and the old one will be unified. In this way, when any of the channels are instantiated, both processors will receive the same term.

Input$_{\text{channel transmission}}$

When a previously assigned input channel is transmitted to a remote processor, we want to send the term that we will receive by this channel to the remote processor. To achieve this, a new output channel is locally allocated and sent to the remote processor, and the new output channel and the old one are unified. This unification will suspend until the input channel is instantiated.

4.2.4. Garbage collection. As we have discussed in previous sections, one of the most important problems that arises in the distributed implementation of CLLs is garbage collection. Some algorithms exist for global garbage collection in distributed systems, but a common drawback to many of them is that computation has to be stopped to achieve a global garbage collection. Algorithms also exist that collect garbage incrementally [17], but they incur very high costs. In our model, there are no external references to remote variables and garbage collection can be achieved locally, in the same way as in a monoprocessor implementation, without needing a global garbage collection, as it is needed, for example, for KL1 system to claim export table entries.

The efficiency of garbage collection algorithms is specially important in real-time systems because a garbage collection phase can lose deadlines to real-time processes. Some proposed algorithms for real-time garbage collection [3] also exist. At first thought, it may seem that the most appropriate algorithms for real-time garbage collection are incremental ones, but there a great problem exists with these kinds of algorithms: efficiency. In incremental garbage collection algorithms access to the heap is very much slower; for example, in an incremental stop & copy algorithm, any time the heap is accessed, whether the data has been copied to the new heap has to be tested and, if so, the new heap accessed. In CLLs access to the heap affects to the efficiency considerably (in some tests it assumes an overhead of 30–40%).

In our implementation we have used a normal stop & copy garbage collection algorithm [30], with some modifications. In the original algorithm a garbage collection phase starts when the heap or the argument stack is full and the whole data area is inspected for garbage collection. In a real-time environment it is necessary to take into account the process deadlines before starting a garbage

![Fig. 13. (a) GC. in Flat Parlog (b) GC in DRLP](image)
collection phase. We have modified the algorithm to take into account that. To achieve this, we have defined a critical threshold, which depends on the amount of free space in the stack and heap and on the amount of processes ready for execution. A garbage collection phase will only start if there is no process with maximum priority or the deadlines of real-time processes allow a garbage collection phase without losing any of them. A garbage collection phase can also start if all the processes are of low priority. To implement this algorithm it is necessary to know a limit to the maximum time for garbage collection.

Apart from this modification, the algorithm has been altered to take into account the new data areas of the abstract machine (Fig. 13). In Flat Parlog, the stop & copy algorithm has to sweep the argument stack, and after that it has to copy the useful information to the new heap. In our implementation it is also necessary to sweep the process stack, and the input and output channels tables, because there may be other useful positions in the heap (the ones pointed to from this data areas: the heap indices in the channel tables, the control variables of the metacalls, etc.).

4.2.5. Termination and deadlock detection. Another important aspect in a distributed implementation is termination and deadlock detection. In other distributed implementations of CLLs, termination and deadlock detection is achieved by using traditional distributed algorithms. To solve deadlock detection in DRL, a similar approach has been taken, and a modification of termination detection proposed in [31] has been used. However, in DRL, termination detection has been solved in a very easy and efficient way, as it is based on the control metacalls. Termination detection is achieved by the root processor in a similar way to sequential CLL implementations. An execution has finished when all the processes in the root processor have finished (including remote metacalls). From the point of view of the origin processor, a remote metacall will have finished when its representative in the origin processor has finished, i.e. when all the Status variables of the control metacalls are instantiated. In this way termination can be detected without using a specific algorithm. A metacall will instantiate its status variable when it finishes its execution, i.e. all its children have been executed.

It is also necessary to take into account another aspect: confirmation of the output channels. As we have seen, every time an output channel is created, it is necessary that the remote processor sends a confirmation with the corresponding input channel; if before receiving this confirmation the output channel is instantiated message sending is delayed until the confirmation arrives. Because of this it is possible that all the children of a control metacall finish, but the metacall cannot finish (until the confirmation has arrived). To control this, every time that an output channel is created, the field confirmation.count of the parent metacall process is incremented; every time that a confirmation arrives the same field is decremented (the parent metacall of the process that created the channel is pointed to from the channel table); a metacall will not finish until its confirmation.count is null.

5. PERFORMANCE MEASURES

The system has been successfully implemented on a network of SUN-4 Unix Sparc-Station with SunOS 4.1.3, using ANSI C and TCP/IP sockets for communication. Some examples, some of

---

Table 1. Real-time performance. Deadline 5 ms

<table>
<thead>
<tr>
<th></th>
<th>go(1,L)</th>
<th>go(30,L)</th>
<th>go(300,L)</th>
<th>go(600,L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reductions</td>
<td>40 256</td>
<td>2 128 077</td>
<td>12 090 749</td>
<td>24 173 215</td>
</tr>
<tr>
<td>Real-time processes</td>
<td>277</td>
<td>8318</td>
<td>82 374</td>
<td>170 285</td>
</tr>
<tr>
<td>Missed deadl.</td>
<td>118</td>
<td>302</td>
<td>320</td>
<td>310</td>
</tr>
<tr>
<td>Mean GC time (ms)</td>
<td>150</td>
<td>150</td>
<td>320</td>
<td>310</td>
</tr>
<tr>
<td>GC number</td>
<td>2</td>
<td>25</td>
<td>130</td>
<td>130</td>
</tr>
</tbody>
</table>

Table 2. Real-time performance. Deadline 50 ms

<table>
<thead>
<tr>
<th></th>
<th>go(1,L)</th>
<th>go(30,L)</th>
<th>go(300,L)</th>
<th>go(600,L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reductions</td>
<td>42 829</td>
<td>1 234 947</td>
<td>12 420 478</td>
<td>244 784 20</td>
</tr>
<tr>
<td>Real-time processes</td>
<td>31</td>
<td>952</td>
<td>9537</td>
<td>90 285</td>
</tr>
<tr>
<td>Missed deadl.</td>
<td>7</td>
<td>12</td>
<td>210</td>
<td>112</td>
</tr>
<tr>
<td>Mean GC time (ms)</td>
<td>150</td>
<td>150</td>
<td>210</td>
<td>112</td>
</tr>
<tr>
<td>GC number</td>
<td>1</td>
<td>16</td>
<td>210</td>
<td>112</td>
</tr>
</tbody>
</table>
which were taken from the CLL literature (nqueens, matrix multiplication, quicksort,...) and adapted to 9RL, have been used to test the system. Other typical distributed programming problems have also been successfully implemented, specially those related to communication protocol specification and simulation (token ring, alternating bit protocol, daemon game,...). The real-time aspects of the language have also been tested with some common problems. We have also tested the performance of the distributed implementation.

5.1. Real-time performance

In order to measure the real-time performance of the implementation we must take into account its capacity to meet process deadlines. We will use the program showing in example 5, with periodic and aperiodic real-time processes and non real-time processes.

```
mode periodic(?,-^).
periodic(C,T,[T|L])<-> after(msec(0,T)):periodic(C,T,L).
periodic(cancel,_,[]).

mode aperiodic(?,-^).
aperiodic(C,[T|L])<-> random(10,T),after(msec(0,T)):
aperiodic(C,L).
aperiodic(cancel,[]).

mode temp(?,-^).
temp(T,cancel)<-> after(msec(T,0)): true.

mode p(?).
p(cancel);
p(X)<-p(X).

mode go(?,-^).
go(T,L)<-> temp(T,C),
periodic(C,3,L1),
periodic(C,5,L2),
merge(L1,L2,LL),merge(LL,L3,L),
aperiodic(C,L3),
p(C),p(C).

<-go(300,L).
```

Example 5

The behaviour of the program is the following:

- Periodic processes will be executed every T milliseconds. Each execution will consist of instantiating the head of a list (third argument) with the value of its period.
- Aperiodic processes will behave in the same way but with a random deadline between 0 and 10 milliseconds.
- The process temp (C, T) will control the execution time of the rest of the processes, cancelling its execution when T seconds have elapsed.
- The p (C) process is used to overload the system.
Every time the goal go \((T, L)\) is executed the system will run for \(T\) seconds. The list of the periodic and aperiodic processes is built to measure the effect of garbage collection in the execution. In Tables 1 and 2 information about the execution of processes appears, with deadlines in the order of 5 and 50 milliseconds during different amounts of time.

In the tables we can see how the number of missed deadlines is very low (around 1 for each 1000 real-time processes), even in the case of deadlines in the order of milliseconds. Failure to meet these deadlines is due to the high load of the processor and the number of garbage collections, but even in these extreme cases the behaviour of the system is acceptable. In other tests with a lower load and with less garbage generated, the missing of deadlines is nearly zero for times in the order of 50 ms.

Another important aspect to take into account in the real-time performance analysis is the execution overhead due to real-time process execution. In order to measure this aspect the same example has been used, but without using time guards. The number of reductions per second is nearly the same, and the overhead is always less than 10%.

5.2. Distributed execution performance

Although the main aim of the system has not been to improve the execution of parallel programs, but to design a language for distributed and real-time programming, some measurements of the system's efficiency in executing parallel algorithms have also been carried out (Table 3). The problems were adapted for execution in the current system. The sequential version's timings have been taken from a previous sequential implementation of a real-time sequential Flat Parlog version, which has been the basis to built the sequential core of the distributed implementation of DPL. We have considered three typical examples in logic programming: matrix multiplying, quicksort and the nqueens problem. The naive version of the nqueens problem has been used, generating
all the permutations of the positions and selecting only the right ones (this is much less efficient, but it is more appropriate for parallelization).

The speed-up (Fig. 14) obtained is nearly linear, and we even obtained a superlinear one with the third problem, due to the great number of garbage collections carried out and the availability of more memory in distributed execution.

The number of process migrations (only the grain processes) is very low and the load balancing is quite good; Fig. 15 shows the number of reductions achieved by each processor, for the case of three processors. The number of messages is also low, in the order of 5 messages for each 1000 reductions. This is because in DRL the size of the messages can be controlled and adapted depending on the program.

5.3. Comparison with other implementations

It is difficult to compare our distributed implementation of DRL with the implementations of similar languages, since the most of these implementations of CLLs have been developed on different environments (shared memory multiprocessor systems [29], special purpose machines [17], etc.). In spite of that problem, we have carried out some comparisons with three systems: Strand, KLIC and Erlang.

Despite Strand and Erlang are commercial implementations and KLIC has a very efficient sequential core based on compiling into C the results of this non-optimized version of DRL are reasonably good. With respect to the real-time features, DRL has better performance than Erlang; this language is capable to meet deadlines in the order of tens of milliseconds as DRL, but DRL behaviour is better when garbage collections are achieved. In the Erlang version of the example 5 program, Erlang loses a 70% of the deadlines.

With respect to execution model, the Erlang, Strand and KLIC sequential cores have better performance than DRL. However, the speed-up obtained by DRL are much higher. Even the absolute execution time is better in DRL than in Strand when the number of processors is six; Table 4 shows the execution times and speed-ups for the qsort and matmult programs in Strand;

<table>
<thead>
<tr>
<th>Table 4 Time and speed-up with strand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential time (s)</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>qsort(5000)</td>
</tr>
<tr>
<td>matmult(50)</td>
</tr>
</tbody>
</table>
Fig. 16. (a) Execution time with qsort. (b) Execution time with multimet).

Fig. 16 shows speed-ups for both implementations (nqueens problem does not appear because of memory problems).

6. CONCLUSIONS

DRL, a new language that integrates the real-time and distributed paradigms within the framework of a concurrent logic language has been presented. Its new execution model has been introduced by giving an operational semantics for the language. The basis for this language's development has been CLILs, which have been extended to cope with the requirements of distributed and real-time languages. A new distributed abstract machine for the execution of the language has been proposed, and the implementation's main issues have been discussed.

The implementation is based on the emulation of this distributed abstract machine, which has been developed on a network of Unix workstations, using C and TCP/IP sockets for communication. The results of current version of DRL are reasonably good, despite its non-optimized sequential core. The implementation is currently available by anonymous ftp to lcc.ctima.uma.es.
Some examples have been used to test the system, of which some were taken from the literature about CLLs and have been adapted to the DRL language. Other typical distributed programming problems have also been successfully implemented, specially those related to communication protocol specification and simulation. In fact, DRL is being used as the kernel of a distributed protocol simulation and validation tool. The real-time aspects of the language have also been tested with some common problems. Periodic and aperiodic processes have been used and the system has performed well, meeting the deadlines in the order of milliseconds.

REFERENCES

**About the Author—MANUEL DÍAZ** received his M.S. and Ph.D. degrees in Computer Science from the University of Málaga (Spain) in 1990 and 1995, respectively. From 1990 to 1995 he was an Assistant Professor at the Department of "Lenguajes y Ciencias de la Computación" of the University of Málaga. Since 1995 he has been an Associate Professor in the same department. His research interests are in the areas of parallel and distributed programming and real-time systems.

**About the Author—BARTOLOMÉ RUBIO** received his B.S. and M.S. degrees in Computer Science from the University of Málaga (Spain) in 1989 and 1990, respectively. Since 1991 he has been an Assistant Professor at the Department of "Lenguajes y Ciencias de la Computación" of the University of Málaga. He is currently working to obtain his Ph.D. degree and his research interests are in coordination models and parallel and distributed system programming.

**About the Author—JOSÉ M. TROYA** received his M.S. and Ph.D. degrees from the University Complutense of Madrid (Spain) in 1975 and 1980, respectively. From 1980 to 1988 he was an Associate Professor in that University, and since 1988 he has been a Professor at the Department of "Lenguajes y Ciencias de la Computación" of the University of Málaga. He has worked on parallel algorithms for optimization problems using dynamic programming and branch-and-bound techniques, and on concurrent logic programming. His current research interests are in coordination languages, software architectures and programming environments for heterogeneous and distributed systems.