DROL: A Distributed and Real-time Object-oriented Logic Environment

M. Díaz, E. Pimentel and J. M. Troya

Dpto de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Pza El Ejido, s/n 29013 Málaga, Spain

The high complexity of distributed computer systems requires new methodologies and languages especially designed for the characteristics of these systems. Declarative languages have been proposed as a promising alternative because they provide a way of leaving aside system details. However, the behaviour of reactive systems cannot be described in pure relational or functional terms. We propose a declarative environment for distributed programming based on the concurrent logic language Parlog, which has the capability of expressing concurrence, communication and non-determinism in a very natural way. That is, the intrinsic parallel semantics of the concurrent logic languages make them appropriate for distributed programming. The proposed environment is particularly suitable for loosely coupled systems and it contains mechanisms for distributed process control, and both real-time and object-oriented design. Each of these characteristics is achieved by the integration, in the framework of the underlying concurrent logic language, of real-time and distributed processing control primitives and object-oriented constructions. From this viewpoint, an operational semantics is defined and some implementation issues are discussed.

Received March 1993, revised February, 1994

1. INTRODUCTION

Software has become the most critical component in real-time distributed systems. These kinds of systems are usually very complex and they have very strong safety and reliability requirements. These systems are composed of collections of diverse computations without central coordination and without total knowledge of each other. Most languages have not been designed to support such systems, but only to describe tightly coordinated computations. In the last few years, many new proposals for distributed programming have emerged (Bal *et al.*, 1989). The majority of the models are based on extending traditional imperative languages, but others are intended to provide new paradigms suited to the characteristics of this type of system. Some of these models try to define some kind of message passing [CSP (Hoare, 1985), Ada (US DoD, 1983)] or shared memory mechanisms [Linda (Ahuja *et al.*, 1986), Orca (Bal *et al.*, 1987)], which can be used for distributed processes communication. Other approaches are based on object-oriented [Actor (Agha, 1986)], logic (concurrent logic languages) or functional [Gamma (Banatre and Lemetayer, 1990)] paradigms. So, high level languages with simple but powerful mechanisms to express features such as concurrency or non-determinism are a typical choice. Programming in these languages does not depend on the underlying machine architecture, as it is an implementation problem to map the language's abstract execution model into a concrete system.

Usually, distributed systems have to meet time constraints, because of the kind of applications they are designed for. Also, in the real-time field many approaches have been presented in the last few years, some of which use very high level languages making it possible to use formal techniques for analysis and verification. Although it may seem nonsensical to use a high level language for distributed real-time programming, real-time does not mean short response time or short interrupt handling latency. A real-time system must meet all the time constraints, and it may be that a fast system with a bad scheduling policy behaves worse, in meeting time constraints, than a slower one with a fast context switch and a well-designed scheduling policy. Among the different proposals there exist imperative languages, such as Occam (Inmos, 1988) and Esterel (Boussinot and de Simone, 1991), declarative languages, such as Lustre (Halbwachs, 1991) and Signal (Leguernic *et al.*, 1991), and graphical languages, like Statecharts (Harel, 1987) and Objectcharts (Coleman *et al.*, 1989).

Concurrent logic languages (CLLs) appear as an attempt to fit Prolog to parallel systems (Clark *et al.*, 1986), and its suitability for distributed systems has been clearly shown by Shapiro (1983). These languages have also been compared with the above mentioned models, e.g. with Actor (Kahn and Miller, 1988) and Linda (Shapiro, 1983). In fact, CLLs were used as kernel languages to construct declarative environments for distributed programming (Foster, 1990). Out of these environments, which were inefficient, CLLs evolved into new languages more oriented to the characteristics of distributed systems. Some languages which have come out of this evolution are: RGDC (Cohen *et al.*, 1991), presented as an amalgam of Prolog and Occam; Sandra (Elshiewy, 1990), a distributed language based on the idea of incorporating the concept of agent (Liskov and Scheiffler, 1983) to a CLL; and Janus (Saraswat *et al.*,

1990), a language used for concurrent constraint programming, which has the same basis as CLLs, but whose objective is to be a kernel language for distributed constraint programming.

On the other hand, CLLs have been extended providing an object-oriented view of their declarative style of programming. In fact, a number of object-oriented extensions of these languages have been proposed: Vulcan (Kahn *et al.*, 1987), Polka (Davison, 1989), Mandala (Ohki et al., 1987), A'UM (Yoshida and Chikayama, 1988) and L2||O2 (Pimentel and Troya, 1992). All these proposals include some of the main characteristics of the object-oriented paradigm: inheritance, encapsulation, information hiding, etc., and they use the logic component in the integration to obtain very expressive object-oriented programming languages. They are based on the idea of considering objects as perpetual processes and the messages between them as information transfer by means of shared variables (Shapiro and Takeuchi, 1983). The object-oriented methodology organizes the memory into local objects, instead of having a single global store; this property explains its relevance to distributed computing.

One of the drawbacks of CLLs, which is common to all the proposals in their use for distributed systems, is their lack of efficiency, because the communication mechanism is based on sharing logical variables and the implementation in distributed environments involves very high costs. In order to solve this drawback, there have been some approaches, trying to simplify the execution model of CLLs to suit it to distributed environments, such as flat CLLs (Foster and Taylor, 1988) or Strand (Foster *et al.*, 1990).

We propose a new approach for distributed programming based on Parlog (Díaz *et al.*, 1992) and oriented to coarse granularity parallelism. The environment is created on the basis of the integration of different paradigms (real-time, distributed processing and object orientation), extending CLLs to cover the different aspects of complex real-time distributed systems. In this sense the execution model of CLLs has been modified to comply with real-time requirements and distributed execution, and an object-oriented extension has been defined. Although similar extensions have been made previously, our work is new in the sense that it integrates the different paradigms under the same framework and that the extensions are particularly designed to deal with the special characteristics of complex distributed real-time systems.

These extensions have been incorporated under the same semantic framework, which is presented from an operational point of view. So, an operational semantics has been defined, in order to extract information about successful, deadlock and infinite computations. It also defines the behaviour of each component in the language and the new computational model incorporated to support the real-time primitives. In this sense, the semantics offers an integrated view of both extensions, by incorporating the time into the transition systems which define the operational semantics for the object-oriented extension (Pimentel and Troya, 1992). Although it is not the aim of this work, and therefore has not been developed here, the semantic aspects could be analyzed from other different perspectives. For example, it should be possible to define a declarative and a denotational semantics for the integration. In this sense, the declarative and denotational aspects of the object-oriented component have been studied in (Pimentel and Troya, 1992 and 1993, respectively); and it is thought that these works could be extended to deal with the distributed (Brogi and Gorrieri, 1989) and real-time (Nicollin and Sifakis, 1991) characteristics of the proposed environment.

From the implementation point of view the aspects of efficiency, scalability, resource sharing, etc., have been taken into account, following Díaz and Troya (1993).

This paper has been organized as follows. In Section 2 we introduce the CLLs. In Section 3 a language incorporating the real-time and object-oriented extensions is described, and we show how the behaviour of objects can be expressed giving some examples. Later, in the fourth section, we consider some implementation issues related to the distributed implementation of the environment.

2. CLLs AND Parlog

CLLs are high level programming languages for parallel machines, that offer a wide margin of parallel programming techniques. These languages preserve many of the advantages of the logic programming model based on Horn clauses, including logic interpretation of programs, data structures represented by terms and unification.

Their operational computational model consists of a set of concurrent processes, communicating by instantiating shared logical variables and synchronizing by suspending on non-instantiated variables. These languages also have some degree of non-determinism. A computation begins with a set of objectives, each objective being an atom $p(T_1, \ldots, T_n)$, which represents a process whose state will be given by partial substitutions of the terms T_i. The set of objectives can be considered as a communicating process network. The behaviour of a process is expressed with Guarded Horn Clauses (GHC) with the following form:

$$H \leftarrow G_1, G_2, \ldots, G_n : B_1, B_2, \ldots, B_m.$$

where the goals G_i form the guard of the clause and B_j its body.

The head and the guard specify the conditions for a reduction transition. The clause body indicates the state of the new process (or processes) after the transition. A transition is produced when a clause is selected (in a non-deterministic way) among all clauses that match with the head and with the successfully evaluated guards.

After selecting a clause the process is substituted by the set of processes provided by the body. If the body is empty then the process ends. These languages are non-

deterministic in a don't care sense. This means that once a transition is made, there is no backtracking (as in Prolog). Formally, it makes partial substitutions and communication among processes observable. The parallelism expressed by concurrent logic languages is of two different kinds: AND parallelism and OR parallelism. The first one appears when an objective is reduced to a conjunction of objectives. The second is exploited when an objective can be reduced by using more than one clause. These clauses will be tried in parallel, although only one will be selected finally.

In this family of languages, the most outstanding representatives are Parlog (Clark *et al.*, 1986), Concurrent Prolog (Shapiro, 1983) and GHC (Ueda, 1985). Another characteristic of these languages is the distinction between input and output arguments; either by means of a mode declaration (Parlog); by adding '?' after the variable (Concurrent Prolog); or by including the input constraint in the execution model (GHC).

Another significant characteristic of CLLs is back communication (incomplete messages). For example, the clause:

p(success(Ok), V) ← c(V) : Ok = ok.

will return ok on the Ok variable, if V satisfies c, although the first argument of p is declared as input.

2.1. Objects as perpetual processes

AND-stream parallelism and partial binding allow us to identify objects as perpetual processes (Warren, 1982; Koymans *et al.*, 1985). This is obtained by defining the behaviour of objects by means of predicates consisting of recursive clauses, which will create a permanently active process. On the other hand, communication among objects is achieved by sending incomplete messages on shared variables. These are the communication channels.

By applying this programming style it is possible to introduce, at an elementary level, an object-oriented methodology into the commited choice logic languages family. However, the syntactic complexity in the description of a class is very high; in addition, the exploitation of mechanisms such as inheritance, client/supplier relation, information hiding, etc., is not direct.

2.2. Process control in Parlog

Parlog has been used for system programming and, in general, for applications in which a process must control the execution of other processes. The mechanism allowing this is the control metacall (Foster, 1990). This metacall can create a process, control its execution and detect the state of the process. The format of the metacall is as follows:

call(Pred?, Status^, Control?).

Where Pred is the process to be executed and Status and Control are lists used to control the process. The control list can be instantiated with the following terms:

- *suspend*: suspends process execution; the Status list will be instantiated to [suspended|_].
- *stop*: aborts the execution of the process (Status = [stopped|_]).
- *continue*: makes a suspended process continue its execution.

The metacall is a predicate that never fails, always succeeding and returning the final state of the executed process. This feature is very important for the application of this kind of language to system programming, because one of the most important requirements of these applications is that the entire system must not break down because of a failure in a single process. This is even more important in a distributed environment, where the failure or misbehaviour of a processor must not affect the rest of the system. The control metacall can also detect termination, by waiting for the status variable to be instantiated. This can be very useful in distributed implementations where termination and deadlock detection are very difficult tasks.

3. THE DROL ENVIRONMENT

DROL is an environment designed to deal with complex distributed systems. It is based on concurrent logic languages, but the computational model has been modified to deal with real-time problems and distributed systems. Besides this, the environment includes object-oriented mechanisms to make the development of reliable software for complex systems easier. In this section, we describe the distributed execution model, the real-time and object-oriented extensions and an illustrative example.

3.1. Distributed execution

Distributed implementations of concurrent logic languages are highly complex, because they are based on communications via shared logical variables, therefore these implementations have to deal with the problems associated with sharing data in distributed environments. The usual approach to solve this problem has been to implement a distributed unification algorithm, by representing a variable with a single occurrence in only one processor and with remote references to it in the other ones. However, distributed unification algorithms (Taylor, 1989) are not very efficient and may involve the interchange of a great number of messages simply to unify two terms. This aspect is especially important in systems with high communications costs (loosely coupled systems).

Some proposals, such as Strand, try to implement more efficient CLLs. This language does not support full unification, making the implementation more efficient. However, the implementation model is still based on the representation of a variable with a single occurrence in one processor and remote references to it in others, and where access to remote variables involves

the exchange of various messages. Another approach that tries to overcome this problem, is Janus (Saraswat *et al.*, 1990). In Janus, variables are regarded as point to point unidirectional communication channels. This approach is similar to the one adopted in our environment, but we only consider the free variables shared between processors as communication channels, maintaining the same semantics for communication as in CLLs for intraprocessor communications.

We propose a new approach for systems with high communication costs, in which the simulation of data sharing mechanisms can be very expensive. Two kinds of logical variables are distinguished: the usual ones, for process communication in the same processor; and variables to link remote processes. These special variables will correspond to unidirectional communication channels, which will be mapped to physical communication channels.

This approach has been implemented by using a remote metacall. The creation and the control of Parlog processes is very easy using the control metacall, and our aim is to maintain this simplicity when the process to be created and controlled is remote, hiding any kind of protocol or implementation details of the communication system. The Parlog control metacall has been extended with a new variable that will be instantiated to the name of the host executing the process:

```
remote_call(Host, Pred, Status, Control).
```

The meaning of the control and status lists is the same as in the usual metacall. The Host variable is bound to the host symbolic name, but it may be a free variable too. If this happens the system will decide where to place the process and it will instantiate the Host variable. This scheduling will be made in relation to load balancing and communication costs. The predicate has to be defined in the remote host and the variables must be annotated as input or output variables:

```
remote_call(Host, pred(Input?, Output^), St, Ctrl).
```

Communication with the remote process will be made in the same way as with a local process (by instantiation of shared variables), with the exception of incomplete messages. In the case of incomplete messages the free variables used for back communication also have to be annotated as input or output and these variables will also be considered as communication channels:

```
Output^ = [m(X?)|T]
X? is an input communication channel
```

A remote call can be made by any process and at any time, so the network topology is dynamic. This topology is always a logical tree, i.e. the physical allocation process graph may have loops, but the logical communication graph is always a tree. The root of this tree is always the host that initializes the system. Once the system is initialized the rest of the hosts in the network can be used to execute remote metacalls. There can be more than one root host; in fact, each host in the network can initialize its own system and can be the root of a new system, using the rest of the hosts in the network for remote metacalls. However, these systems are independent and cannot exchange information (Figure 1).

This limitation can be problematical in the case of different hosts sharing resources. To solve this there exists a special class of processes, named *servers*, which can be seen from the various independent systems, so allowing information exchange among them. In this sense, each system can be seen as a *virtual machine* with as many processors as hosts in the network and sharing resources with other virtual machines via *servers* (Figure 2).

Servers are special processes which are created with the primitive create_server(process), where process is a predicate with input variables only. When a server is created the host tests whether the name of the process exists as a server (the servers are identified by their names and by the host to which they belong). The host maintains a list with the name of the servers. When a system process wants to use the service of another one it must take the following steps:

- ask_for_services(Host, L). L will be instantiated to a list containing the name of the processes in Host.
- service(Host, Name, Channels, Acknowledge). The last argument will be instantiated to ack or nack, depending whether the service is granted or not by the host, and Channels will be instantiated to a list with the variables that will be used to communicate with the server.

In the foreign host, when a service is requested, the variables and sockets for communication are created and sent back to the host which asked for the service. Besides the control metacall, and the primitives for dealing with servers, some other primitives for distributed programming have been implemented. The more important ones are:

- init(Host). Initializes a host to a known initial state; if Host is a free variable all the hosts in the network will be initialized.
- remote_load(Host, Database). Loads a database with predicate definitions in Host. If Host is free the Database will be loaded in all the hosts.
- statistic(Host, Results). Returns in the Results variable statistics about CPU time and Host memory.
- connected(L). Returns on the list L the symbolic names of the connected hosts.
- local_host(X). Returns in X the name of the local host.

3.2. Real-time extension

CLLs do not provide any mechanism to express real-time constraints in the execution of processes. Sandra (Elshiewy, 1990) introduces the notion of logical clocks

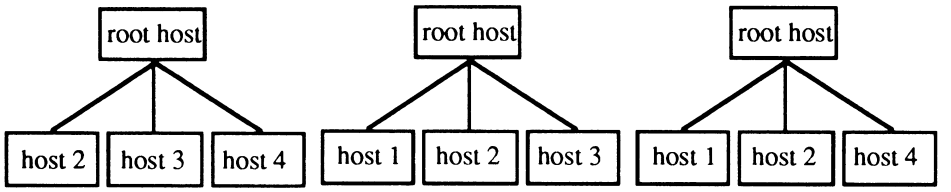

FIGURE 1. Different independent systems in the same network.

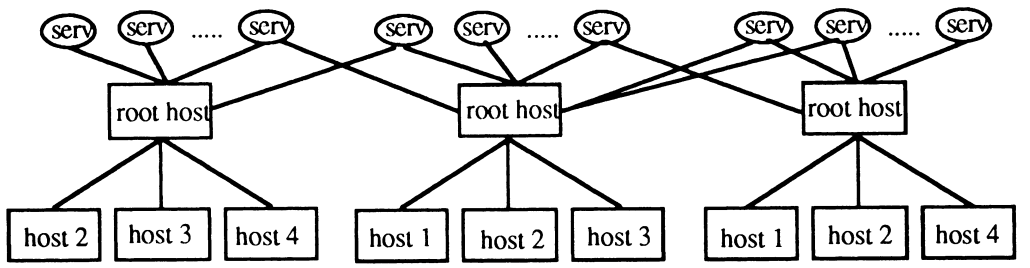

FIGURE 2. Different independent systems in the same network connected via servers.

to express time in the field of CLLs, which, however is not real-time, and the logical clocks only have the aim of ordering the occurrence of events in the system. Another approach in the field of declarative languages is the one of Erlang (Armstrong *et al.*, 1993), 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 order in process scheduling cannot be known. So, an extension of CLLs which allows for process timing constraints is proposed. This extension implies a modification of the execution model. In CLLs a process may suspend on a non-instantiated variable, during the matching of the clause head or on the 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, which makes the process suspend for T seconds, after head matching and guard evaluation. If there is no other clause which commits before that time has elapsed, then the clause is selected to reduce the process. In the example below we have the definition of two processes: a producer, p, and a consumer, c, communicating via a shared variable L. In the definition of the consumer we have a timed guard, detecting a time out error if no data arrives in the channel during a ten seconds period.

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

When a timed clause exists in the definition of a process, this process must be selected to be reduced as soon as possible, i.e. this process has higher priority than ordinary ones. For example, let us assume the following scenario, where p and c are the processes defined above and r is a process independent of p and c:

Process		L
p		[item, ...
r		
r	t > T	
r		
r		
p		[item, item, ...
c		
c		

The consumer process does not detect that the producer has taken more than T seconds to produce the second item of the list. However, if the c process had been scheduled before the r process, this situation would have been detected. If there is more than one real-time process in the same conjunction the scheduling time for them is the same as the first.

Indeed, to satisfy all the time constraints there should exist as many processes as processors, and each process should be scheduled as soon as the process becomes ready for execution (Maximal Parallelism), but this approach is unrealistic. In fact, depending on the application we could have a permitted error time-limit and a process could be scheduled in that time interval. In our model that interval can be specified and if the system violates that limit an exception is raised.

The environment can constrain the execution time of processes by using some real-time control primitives. This feature is very important in some applications like protocols and process control. Real-time control is local and there is no synchronization among the different host clocks in the network. We can access the system clock with the predicate clock(N^), that binds N with the time

elapsed (in seconds) since clock initialization. The more important primitives are:

- init_clock. Initializes the local clock.
- after(N). A delay of N seconds. The process is suspended for the number of seconds indicated by N.
- timeout(Pred, Time, Status). A metacall executes the process indicated by Pred limiting its execution time and returning the Status (*success*, *failed*, *timeout*).
- alarm(Pred, Time, Ctrl). The process starts its execution once the time indicated by Time has elapsed. The alarm can be cancelled using the control list.

The implementation of a simple protocol with timeouts is shown below as an application example:

```
mode rec(Inp_channel?, Out_channel^, timeout?).
rec([Data|Stream], O, Time)
   ← process(Data), rec(Stream, O, Time).
rec(Stream, O, Time) ← after(Time) : O = disconnect.
```

A process waits for data in a stream. If no data is received within a specified time then a disconnection message is sent.

3.3. Object-oriented extension

The absence of structuring mechanisms in CLLs necessitates the presence of characteristics which provide the programming methodology in this kind of environment. In this sense, the object-oriented paradigm has been chosen, and mechanisms such as inheritance, client/supplier relation, encapsulation, information hiding, etc., have been incorporated in the real-time extension presented previously.

The syntax, semantics and implementation of the object-oriented extension defined are based on those of Polka, although it presents significant differences (among them, the existence of dynamic binding). In addition, this system uses timed guards to describe the messages, and its implementation is based on a distributed implementation of Parlog.

The syntactic outline of a class definition in the proposed extension is:

```
<class> [<channels>][; <channels>]
[inherit <classes>]
state <state_variables_and_supplier_objects>
messages
   EGHCs
end.
```

where EGHCs are a sequence of extended GHC whose syntax and informal semantics are studied below.

The interface of a class instance is given by its input and output channels. They are declared after the class name (the output channels appear afterwards ';'). A class will always have a default input channel self(c), c being the class name, in addition to those appearing in the declaration. An object's state will be determined by the structures declared in the state clause. These will be composed of state variables, corresponding to simple data, and also by references to some other class instances, corresponding to objects providing services. Encapsulating and information hiding issues related to instance variables are similar to Smalltalk; this means that state access will be made by using incomplete messages. Message processing will be determined by the extended guarded clauses. These clauses have the following syntax:

$$[[I]\ ?]\ H \leftarrow G : B$$

where I will be an input channel of the class or an output channel of an object in use. If I is not present, the default input channel, called self(c), is considered. The non-extended GHC correspond to private predicates. Message passing primitives can appear in B, such as O!m, m being a term and O either an output channel of the class, an input channel of an object in state, or the keyword self. The message m can also be a sequence of messages. Naturally, the guard G can contain the after(T) primitive.

The behaviour of a controller, which will be explained in the next section, can be described in this language as follows:

```
controller C; G, T
state Car
messages
  ? measure(D)    ← D = = approaching : G!down(Ok),
                    wait(Ok, train).
  ? measure(D)    ← D = = travelling : G!up(Ok),
                    wait(Ok, gate).
  C ? measure(D) ← Car becomes D.
  mode wait(?, ?).
  wait(Ok, gate)  ← after(20) : G!error.
  wait(ok, gate).
  wait(Ok, train) ← after(10) : T!alarm.
  wait(ok, train) ← Car/ = = ingate : true.
end.
```

A controller is defined by considering two input channels, self(controller) and C, and two output channels, G and T. The variable Car will determine the state of an instance, and its updating is made with the becomes primitive (Davison, 1989), whose behaviour is similar to an assignment. However, the result is not effective until whole message is processed. Note that private messages, such as wait/2, will be processed by Parlog clauses.

The class instances are created from the state section of a class or by means of a query. The queries will be made from what is called top, and that will be considered a special class (**T**), such that its state section will contain the part of the query which creates the instances, and whose messages section will be composed of non-extended guarded Horn clauses. For example, a query like:

$$?- c_1(o_1), \ldots, c_n(o_n), q_1, \ldots, q_m.$$

c_i being classes, will be made in a class **T** frame where the state section is: $c_1(o_1), \ldots, c_n(o_n)$. The rest of the

calls, $q_1, \ldots, q_m$, will be considered goals to solve. The class top is only considered for technical reasons related to the operational semantics definition.

3.4. Operational semantics

We have presented the environment as currently developed; two different extensions of the concurrent logic language Parlog have been proposed. However, the result is an integration of both extensions. The operational semantics defined below shows how two paradigms are combined to obtain an integrated language, incorporating real-time primitives and object-oriented constructions for distributed programming. In this sense, the real-time primitives inside the object-oriented context are defined by means of transition systems, and the effect of these primitives on the object-oriented mechanisms are studied. Only the after primitive is considered, because the other time primitives can be defined in the same terms. The operational semantics for the object-oriented extension given in Pimentel and Troya (1992) is extended, in order to incorporate the behaviour derived from using timed guards, so defining an operational semantics for the proposed language. This approach is used instead of extending the operational semantics of the real-time extension, because the after primitive modifies the computational model, whereas the object-oriented mechanisms do not affect the functioning of timed guards.

Thus, the operational semantics gives an integrated view of the environment, based on the semantics of the underlying concurrent logic language, i.e. Parlog. In this sense, we will assume a transition relation $\rightarrow_{\text{Parlog}}$ as defined in Boer *et al.* (1989).

The general description of a class is assumed to be as follows:

$c\ i_1, i_2, \ldots, i_n; o_1, o_2, \ldots, o_m$
inherit $c_1, c_2, \ldots, c_h$
state $x_1, x_2, \ldots, x_s, d_1(\overline{j_1}, \overline{p_1}), d_2(\overline{j_2}, \overline{p_2}), \ldots, d_u(\overline{j_u}, \overline{p_u})$
messages
M_c
end

where c, c_k, $d_k \in \Sigma^c$ (set of class names); i_k, $o_k \in Ch$ (set of channels); $x_k \in SVar$ (set of state variables); $\overline{j_k}, \overline{p_k} \in Ch^+$ (sequences of channels); and M_c is a set of EGHCs. The variables appearing in the clauses of M_c can be state or logical variables; we will denote each one by *SVar* and *LVar*, respectively.

A class $c \in \Sigma^c$ will be characterized by:

$I_c = i_1 \cdot \ldots \cdot i_n \in Ch^+$, representing input channels,
$O_c = o_1 \cdot \ldots \cdot o_m \in Ch^+$, representing output channels,
$S_c = \{x_i\}_{i=1\ldots s} \in \mathcal{P}(SVar)$, representing state variables,
$H_c = \{c_i\}_{i=1\ldots h} \in \mathcal{P}(\Sigma^c)$, representing inherited classes,
$U_c = \{d_k(\overline{j_k}, \overline{p_k})\}_{k=1\ldots u} \in \mathcal{P}(At(Ch, \Sigma^c))$, representing suppliers,

and M_c. $At(Ch, \Sigma^c)$ denotes the set of atoms composed by functors of Σ^c, and arguments of *Ch*. Of course, $h = 0$ implies $H_c = \varnothing$, which means the absence of inherited classes. Similarly, $n = 0$ is equivalent to $I_c = \varepsilon$ (empty sequence). A program will be given by a set of class definitions.

The class hierarchy associated with a class c will be represented by the set:

$$\overline{H_c} = \{c, \mathbf{T}\} \cup \left(\bigcup_{a \in Hc} \overline{H_a} \right),$$

Note that I_c and O_c are defined as sequences while S_c, H_c and U_c are sets. This distinction is not arbitrary: the position of the channels is relevant when instances are created. Although the inheritance order is also important to the programmer (we could have considered the set $\overline{H_c}$ with an order relation), we will model a strict inheritance in which the graph inheritance search will depend on the implementation.

In the following, we will consider all becomes operations in an EGHC grouped at the end of its body. It does not assume a restriction in the language because the result of all becomes operation has no effect until the next message is processed. In addition, all clauses in a class are assumed to have a timed guard such as after(τ), τ being an integer (including 0). That is, EGHCs will have the following appearance:

$$[I]?H \leftarrow G, \text{after}(\tau) : B, \bar{x} \text{ becomes } \bar{t}.$$

We will base the operational semantics of the language on the transition systems. The transition systems permit us to determine the program's semantics in terms of transitions between configurations. Program execution is modelled by a configuration sequence with transitions between them, starting with an initial configuration, and finishing with a terminal one.

An instance is identified by its class name and its channels. In fact, when an object is created, a call such as:

$$c(i_0, i_1, \ldots, i_n, o_1, o_2, \ldots, o_m),$$

is used, where $c \in \Sigma^c$ and $i_k, o_l \in Ch$.

However, to make object representation easier we assume a set *Obj* containing names for active objects, and a function

$$T : Obj \rightarrow \Sigma^c,$$

which assigns to each object $\alpha \in Obj$ the class to which it belongs. Furthermore, we consider the function

$$v : \mathcal{P}(Obj) \times \Sigma^c \rightarrow Obj,$$

such that $v(X, c) \notin X$ and $T(v(X, c)) = c$, for finite $X \subseteq Obj$ and $x \in \Sigma^c$. The v function gives a new name for an instance of c for a finite set X of object names and a class name c.

3.4.1. Configurations

In order to describe the state of an object different aspects need to be considered. Firstly, it is necessary to

know the current values of its state variables. So, one of the state components of an object $\alpha \in Obj$ ($a = T(\alpha)$) is a mapping from

$$\overline{S_a} \to Tm,$$

where $\overline{S_a} = \cup_{d \in \overline{Ha}} S_d$, and Tm is the set of terms. In general, the state of a system of objects can be modelled by:

$$S_{var} = Obj \to Eval,$$

where $Eval = SVar \to Tm$. But the state of an instance is also characterized by its connection with other system objects by means of its channels. It can be expressed by

$$S_{ch} = Obj \to Ch \to Obj.$$

For each $\alpha \in Obj$ ($a = T(\alpha)$), an element $\sigma \in S_{ch}$ will determine the correspondence between its output channels and the objects taking them as input channels. That is,

$$\sigma(\alpha) : \overline{O_a} \cup \overline{OU_a} \to Obj,$$

where the sequence $\overline{O_a}$ contains all output channels defined in the class a and its ancestors. Similarly, $\overline{OU_a}$ denotes the output channels corresponding to the suppliers.

So, S_{ch} determines the receivers associated with each output channel of an object, but no information exists about what input channel will be used to make the connection. To cover this aspect, we define:

$$R = \{r \in Obj \to Ch \to Ch : supp(r(\alpha)) \cap ran(r(\alpha)) = \varnothing \wedge r(\alpha) \text{ injective } \forall\, \alpha \in Obj\},$$

where *supp* and *ran* represent the support and the range, respectively, of a mapping. An element $r \in R$ represents, for each object α ($a = T(\alpha)$), a renaming of channels:

$$r(\alpha) : \overline{I_a} \cup \overline{IU_a} \to Ch,$$

where $\overline{I_a}$ and $\overline{IU_a}$ are defined in a similar way to $\overline{O_a}$ and $\overline{OU_a}$, respectively. Given an object, a renaming can be extended to sequences, so that if $\bar{o} = o_1, \ldots, o_k$ and $\bar{p} = p_1, \ldots, p_k$, the renaming $\{\bar{o}/\bar{p}\}$ will represent $\{o_1/p_1, \ldots, o_k/p_k\}$.

So far only the functional aspects of the objects have been modelled. The logical component will be given by substitutions. Let *Subst* be the set of substitutions. A substitution is a total function from logical variables to terms, $\theta : LVar \to Tm$, such that $supp(\theta)$ is finite. The empty substitution will be denoted by ε. The composition of substitutions and the most general unifier (mgu) of terms are defined in the usual way.

Finally, it is necessary to maintain a set containing the active objects of the system.

Definition 1 (State) We define the set of states St by:

$$St = S_{var} \times Subst \times S_{ch} \times R \times \mathcal{P}(Obj)$$

Each component of $\sigma \in St$ is denoted by $(\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5)$. The first component stores the values of the state variables in each object, and the second one contains the binding of logical variables occurring in the system. The third and fourth components establish connections between objects. The last one includes the names of the active objects.

The following variant notation is used. By $\sigma\{\alpha, x \to t\}$ (with $\alpha \in Obj$, $x \in SVar$ and $t \in Tm$) we shall denote the state σ' that is the same as σ, but for the value of $\sigma'_1(\alpha)(x)$, which is t. Similarly, we denote the new state σ' by $\sigma\{\alpha, o \to \beta\}$ (with $\alpha \in Obj$, $o \in Ch$ and $\beta \in Obj$}, which is equal to σ but for $\sigma'_3(\alpha)(o)$, which is β. Likewise, by $\sigma\{\alpha, r_\alpha\}$ ($r_\alpha \in Ch \to Ch$) we represent the state σ' that is the same as σ but for the value of $\sigma'_4(\alpha)$, which is r_α. This notation can be extended to sequences: if $\bar{x} = x_1 \cdot \ldots \cdot x_s$ is a sequence of state variables, $\bar{t} = t_1 \cdot \ldots \cdot t_s$ is a sequence of terms, and $\bar{o} = o_1 \cdot \ldots \cdot o_m$ is a sequence of channels:

$$\sigma\{\alpha, \bar{0} \to \beta\} = \sigma\{\alpha, o_1 \to \beta\} \ldots \{\alpha, o_m \to \beta\}, \quad \text{and}$$

$$\sigma\{\alpha, \bar{x} \to \bar{t}\} = \sigma\{\alpha, x_1 \to t_1\} \ldots \{\alpha, x_s \to t_s\}.$$

Also, by $\sigma\theta$ ($\sigma \in St, \theta \in Subst$) we denote $(\sigma_1, \sigma_2\theta, \sigma_3, \sigma_4, \sigma_5)$.

Definition 2 (Configurations) A typical configuration will be a triad composed of a set of pairs, a state, and a global clock. Each pair in the first component will model the pending goals for each active object, and these are represented by a list of goals. So, the set of configurations will be

$$\Gamma = \mathcal{P}(Obj \times \mathcal{P}(At_\tau)) \times St \times \mathbb{N},$$

where At_τ is a set of tuples composed of an element from At, and different instances of a special predicate wait/5. If $p \in At$, a typical element of At_τ has the form

$$p_\tau ::= p, \quad \text{or}$$

$$p_\tau ::= p_\tau + \text{wait}(C, t),$$

where C is an EGHC, and $t \in \mathbb{N}$. When $p + \text{wait}(C_1, t_1) + \ldots + \text{wait}(C_n, t_n) \in At_\tau$, we will write $p + \Sigma_{1=1\ldots n}\text{wait}(C_i, t_i)$. An element of At_τ denotes the candidate clauses to solve a goal, or to process a message.

Given a configuration $(X, \sigma, t) \in \Gamma$, when an object α is created the set of goals to solve is empty. In this case, the pair $(\alpha, \varnothing)$ will belong to X. But the absence of goals can also be produced because all goals have been solved. In fact, a pair $(\alpha, [\,]) \in X$, $[\,]$ being the singleton set {true}, represents this situation.

The third component of a configuration represents time. We will assume the maximum time to perform a transition is $\tau_{\max}$.

Definition 3 (Terminal configurations) A configuration $(X, \sigma, t) \in \Gamma$ is terminal if and only if:

$$\forall\, (\alpha, P_\alpha) \in X,\ P_\alpha = \varnothing \vee P_\alpha = [\,]$$

The set of terminal configurations will be denoted by Γ'.

3.4.2. Transition rules

Before describing the transition relation, we will explain when a clause is a candidate to solve a goal or to process a message, which is similar to that in concurrent logic languages. For the rest of the section we let W denote a fixed program. Given a set of logical variables V, W_V denotes the program whose clauses are variants, with respect to V, of the clauses of W. The set of variables occurring in a term t is indicated by $var(t)$. Similarly, if $p \in At$, $ivar(p)$ denotes the set of variables corresponding to input arguments of p. The notions of input and output mgu's of two predicates (mgu_i and mgu_o, respectively) are introduced by considering the mgu restricted to input and output arguments, respectively.

Definition 4 (Candidate clauses) Given a class c and an atom g, a clause $C \equiv (h \leftarrow F, \text{after}: B) \in (M_d)_{ivar(g)}$ $(d \in \overline{H_c} \cup \{\mathbf{T}\})$ is *candidate* to solve the goal g in the context of c, if and only if:

$$\exists\, \theta = \text{mgu}_i(g, h): \langle F, \theta\rangle \rightarrow^*_{\text{Parlog}} \langle [\,], \theta'\rangle \wedge \theta'_{|ivar(g)} = \varepsilon$$

Let $M_c(g)$ be the set of candidate clauses, in the context of c, to solve g. $\theta_C(g)$ denotes θ'.

Likewise, if $\xi \in Eval$, an extended clause $C \equiv (i?h \leftarrow F, \text{after}: B) \in (M_d)_{var(m)}$ $(d \in \overline{H_c})$ is *candidate*, in the environment (c, ξ), to process the message m along the channel i, iff:

$$\exists\, \theta = \text{mgu}(m, h\xi): \langle F\xi, \theta\rangle \rightarrow^*_{\text{Parlog}} \langle [\,], \theta'\rangle \wedge \theta'_{|var(m)} = \varepsilon$$

Let $M_{c,\xi}(m, i)$ be the set of candidate clauses, and $\theta_{C\xi}(m)$ the substitution θ'.

Now we can define $\rightarrow_\tau \subseteq \Gamma \times \Gamma'$ as the smallest relation verifying these two rules:

$$\frac{\langle X, \sigma, t\rangle \rightarrow \langle X', \sigma', t'\rangle}{\langle X, \sigma, t\rangle \rightarrow_\tau \langle X', \sigma', t'\rangle}$$

$$\frac{\langle X, \sigma, t\rangle \not\rightarrow \text{ and } \langle X, \sigma, t\rangle \text{ is not terminal}}{\langle X, \sigma, t\rangle \rightarrow_\tau \langle X, \sigma, t + \tau_{\max}\rangle}$$

where $\rightarrow \subseteq \Gamma \times \Gamma'$ is an auxiliary transition relation defined below. Note how the transition relation $\rightarrow_\tau$ inserts a silent transition by increasing the current time when the configuration cannot proceed under $\rightarrow$.

(a) *Creation of instances*

$$\frac{\begin{array}{l} d \in \Sigma^c,\ \beta = \nu(\sigma_5, d),\ t < t' \leqslant \tau_{\max} \\ \sigma' = \sigma\{\alpha, i_0 \cdot \bar{i} \rightarrow \beta\} \\ \quad \{\beta, \bar{o} \rightarrow \alpha\}\, \{\beta, \{\text{self}(d)/i_0, \overline{I_d}/\bar{i}, \overline{O_d}/\bar{o}\}\}, \\ \quad \text{with } \sigma'_5 = \sigma_5 \cup \{\beta\} \end{array}}{\langle \{(\alpha; d(i_0, \bar{i}, \bar{o}))\}, \sigma, t\rangle \rightarrow \langle \{(\alpha; [\,]), (\beta; \varnothing)\}, \sigma', t'\rangle}$$

The creation of an instance is made by a goal composed of a class identifier (d) and the channels ($i_0, \bar{i}, \bar{o}$). The first renames the self channel ($\text{self}(d)/i_0$) and the rest are connected to the input and output channels of the class ($\overline{I_d}/\bar{i}, \overline{O_d}/\bar{o}$). A new object is created (β) without goals to solve, by considering its input channels as output channels of α ($\alpha, i_0 \cdot \bar{i} \rightarrow \beta$) and vice versa ($\beta, \bar{o} \rightarrow \alpha$).

This transition models the creation of an object produced by a predicate call made in the context of another object (α). The new object is represented by adding a new pair to the first component of the current configuration, given by a new name (β) and a set of initially empty goals. In fact, this set should include the predicates in U_d, corresponding to the providers of the class d. However, to avoid infinite loop transitions derived from recursive class definitions, we add these predicates only when the object β receives the first message (see the rule (c), corresponding to the message processing).

(b) *Private predicates.* The reduction of a goal depends on the environment in which is must be solved:

$$\frac{\begin{array}{l} C_i \equiv (h \leftarrow F, \text{after}(t_i): B) \in M_{T(\alpha)}(g\sigma_2) \\ t < t' \leqslant t + \tau_{\max} \end{array}}{\langle \{(\alpha; g)\}, \sigma, t\rangle \rightarrow \langle \{(a; g + \Sigma_i\, \text{wait}(C_i, t + t_i))\}, \sigma, t'\rangle}$$

The resolution method applied to solve a private predicate is given in two parts. Firstly, all candidate clauses, C_i, are considered, adding them to the initial goal. For each clause C_i we also consider the time when it will be ready to proceed. This time is given by the current time (t) plus the time expressed in the guard (t_i). It is assumed that this sort of transition is made in less time than $\tau_{\max}$.

The goal can be reduced by using one of the candidate clauses when the time of some of them has expired:

$$\frac{\begin{array}{l} \exists\, t_j = \min\{t_i\} \leqslant t \\ \theta = \theta_{Cj}(g\sigma_2),\ C_j \equiv h \leftarrow F: B \end{array}}{\begin{array}{l} \langle \{(\alpha; g + \Sigma_i\, \text{wait}(C_i, t_i))\}, \sigma, t\rangle \\ \qquad \rightarrow \langle \{(\alpha; \text{outunif}(g\sigma_2, h\theta), B)\}, \sigma\theta, t\rangle \end{array}}$$

The clause with the minimum time is selected, and the second component of σ is modified by the substitution θ. The new goals generated are the clause body, and the ouput unification of the original goal and the cluster head. The output unification is given by:

$$\frac{\exists\, \theta = \text{umg}_o(g, h)}{\langle \{(\alpha; \text{outunif}(g, h), B)\}, \sigma, t\rangle \rightarrow \langle \{(\alpha; B)\}, \sigma\theta, t\rangle}$$

(c) *Messages on an output channel.* Messages can be sent along self, or an output channel, including the channels corresponding to the input channels of suppliers (objects defined in the state section). Mesage processing along output channels is modelled in a similar way to private predicates. So, the candidate extended clauses are again considered as in the previous rule.

$$\frac{\begin{array}{l} \sigma_3(\alpha)(o) = \beta,\ \sigma_4(\beta)(p) = o,\ b = T(\beta),\ \eta = \sigma_1(\beta) \\ C_i \equiv (p?h \leftarrow \text{after}(t_i), F: B) \in M_{b,\eta}(m\sigma_2, p) \\ t < t' \leqslant t + \tau_{\max} \end{array}}{\begin{array}{l} \langle \{(\alpha; o!m), (\beta; G)\}, \sigma, t\rangle \\ \qquad \rightarrow \langle \{(\alpha; o!m + \Sigma_i\, \text{wait}(C_i\eta, t + t_i)), (\beta; G)\}, \sigma, t'\rangle \end{array}}$$

The processing of the message m, sent along an output channel, can succeed after all *wait* atoms have been generated:

$$\sigma_3(\alpha)(o) = \beta,\ \sigma_4(\beta)(p) = o,\ b = T(\beta),\ \eta = \sigma_1(\beta)$$
$$\exists\ t_j = \min\{t_i\} \leqslant t$$
$$\theta = \theta_{Cj}(m\sigma_2),\ C_j \equiv p?h \leftarrow F : B,\ \bar{x} \text{ becomes } \bar{t}$$
$$B' = \begin{cases} B,\ U_b \text{ si } G = \varnothing \\ B \text{ si } G \neq \varnothing \end{cases}$$
$$\sigma' = \sigma\{\beta, \bar{x} \rightarrow \bar{t}\}$$

$$\langle\{(\alpha;\ o!m + \Sigma_i \text{ wait}(C_i, t_i)), (\beta;\ G)\},\ \sigma,\ t\rangle \rightarrow \langle\{(\alpha;\ [\,]), (\beta;\ G, B')\},\ \sigma'\theta,\ t\rangle$$

In this case, the clause C_j is chosen for processing the message m because its suspension time has expired. The clause body is added to the list of pending goals in the object β, and if β has been recently created ($P = \varnothing$) the goals in U_b are also added. The state is modified by the becomes primitive and the substitution θ.

(d) *Messages on self.* Another way to send messages is by using the special variables self, super or an inherited class. When the send primitive '!' is used with some of these entities, the messages are put on the self channel, to be consumed by the object itself. The difference in each case is the scope of the candidate clauses search. Only the rule corresponding to self messages is presented here:

$$a = T(\alpha),\ \eta = \sigma_1(\alpha)$$
$$C_i \equiv (?h \leftarrow F, \text{after}(t_i) : B) \in M_{a,\eta}(m\sigma_2, \text{self}(a))$$
$$t < t' \leqslant t + \tau_{\max}$$

$$\langle\{(\alpha;\ \text{self}!\ m)\},\ \sigma,\ t\rangle \rightarrow \langle\{(\alpha;\ \text{self}!\ m + \Sigma_i \text{ wait}(C_i\eta, t_i))\},\ \sigma,\ t'\rangle$$

Note that the candidate clauses are considered from $M_{a,\eta}$. So, if the message is sent to self the whole inheritance graph is used to find a convenient clause. In other case, when the message is sent by using one of the inherited classes, $c \in H_a$, candidate clauses should be inspected in $M_{c,\eta}$. The super entity corresponds to $\cup_{c \in H_a} M_{c,\eta}$.

When all candidate clauses to process the message have been found, we choose a clause C_j with a suspension time less than or equal to the current time. The new state will be given by the becomes operation of the clause, and by the substitution generated by the unification.

$$\exists\ t_j = \min\{t_i\} \leqslant t,\ C_j \equiv ?h \leftarrow F : B,\ \bar{x} \text{ becomes } \bar{t}$$
$$\theta = \theta_{C_j}(m\sigma_2)$$
$$\sigma' = \sigma\{\beta, \bar{x} \rightarrow \bar{t}\}$$

$$\langle\{(\alpha;\ \text{self}!\ m + \Sigma_i \text{ wait}(C_i, t_i);)\},\ \sigma,\ t\rangle \rightarrow \langle\{(\alpha;\ \text{B})\},\ \sigma'\theta,\ \text{t}\rangle$$

(e) *Interleaving.* The parallel execution will be modelled by interleaving. Firstly, the objects can progress in parallel (inter-objects parallelism), as shown by the following rule:

$$\frac{\langle X, \sigma, t\rangle \rightarrow \langle X', \sigma', t'\rangle}{\langle X \cup Y, \sigma, t\rangle \rightarrow \langle X' \cup Y; \sigma', t'\rangle}$$

On the other hand, the inner activity of an object can progress concurrently, because the goal conjunction is also modelled by interleaving:

$$\frac{\langle\{(\alpha;\ G_1)\} \cup Y, \sigma, t\rangle \rightarrow \langle\{(\alpha;\ G_1')\} \cup Y', \sigma', t'\rangle}{\langle\{(\alpha;\ G_1, G_2)\} \vee Y, \sigma, t\rangle \rightarrow \langle\{(\alpha;\ G_1', G_2)\} \cup Y', \sigma', t'\rangle}$$

In fact, part of the goals to be solved in the context of an object can proceed independently, as the previous rule illustrates.

3.4.3. *Observables*

The transition rules previously defined do not provide information, at least directly, about deadlock or failure situations, and infinite computations. In order to incorporate this kind of information we define the following notion of observables.

Let, for $St_\partial^\infty = St^+ \cup St^* \cdot \{\partial\} \cup St^\omega$ and ∂ denoting fail (or deadlock), the function

$$\mathcal{O}_1 : \Gamma \rightarrow \mathcal{P}(St_\partial^\infty),$$

be given as follows:

$$s \in \mathcal{O}_1(\langle X, \sigma\rangle),$$

with $(X, \sigma, t) \in \Gamma$ if and only if one of the following conditions is satisfied:

(i) $s = \sigma_0 \cdot \sigma_1 \cdot \ldots \cdot \sigma_n$ $(n \geqslant 0)$ and
$\exists\ X_0, X_1, \ldots, X_n, t_0, t_1, \ldots, t_n$ such that
$\langle X, \sigma, t\rangle = \langle X_0, \sigma_0, t_0\rangle \rightarrow_\tau \ldots \rightarrow_\tau \langle X_n, \sigma_n, t_n\rangle$, and
$\langle X_n, \sigma_n, t_n\rangle$ is terminal.

(ii) $s = \sigma_0 \cdot \sigma_1 \cdot \ldots \cdot \sigma_n \cdot \partial$ $(n \geqslant 0)$ and
$\exists\ X_0, X_1, \ldots, X_n, t_0, t_1, \ldots, t_n$ such that
$\langle X, \sigma, t\rangle = \langle X_0, \sigma_0, t_0\rangle \rightarrow_\tau \ldots \rightarrow_\tau \langle X_n, \sigma_n, t_n\rangle \not\rightarrow_\tau$,
and $\langle X_n, \sigma_n, t_n\rangle$ is not terminal.

(iii) $s = \sigma_0 \cdot \sigma_1 \cdot \ldots$ and $\exists\ X_0, t_0, X_1, t_1 \ldots$ such that
$\langle X, \sigma, t\rangle = \langle X_0, \sigma_0, t_0\rangle \rightarrow_\tau \langle X_1, \sigma_1, t_1\rangle \rightarrow_\tau \ldots$

Note that each case corresponds to one of the three parts of St_∂^∞. If $s \in St^+$ (case i), it stands for a finite normally terminating computation; if $s \in St^* \cdot \{\partial\}$ (case ii), it reflects a finite abnormally terminating computation, which is indicated by the symbol ∂ (fail or deadlock); and if $s \in St^\omega$ (case iii), it represents an infinite computation.

Definition 5 (Operational semantics) We define $\mathcal{O} : P(At) \rightarrow \mathcal{P}(St_\partial^\infty)$ by

$$\mathcal{O}(G) = \mathcal{O}_1(\langle X, \sigma, 0\rangle),$$

where $X = \{(v(\varnothing, \mathbf{T});\ G)\}$ and $\sigma = (id, \varepsilon, id, id, \{v(\varnothing, \mathbf{T})\})$. The first, third and fourth components of σ correspond to functions with empty support, i.e. totally undefined.

This operational semantics gives information about the sequence of states produced by an initial query at the time $t = 0$. This initial call is considered in the context of the special class **T**. Each state corresponds to a timed transition, and stores the inner state of all objects in the system, and their relation with each other. The logic component is given by the substitutions corresponding to the second component of the state. So, in terms of logic programming, the success set for a sequence of goals $G \in P(At)$ is given by $\mathcal{O}_2 : \mathcal{P}(At) \to \mathcal{P}(Subst)$:

$$\mathcal{O}_{ss}[G] = \{\sigma_2 : \exists\, s \in St^*,\ s \cdot \sigma \in \mathcal{O}(G),\ \sigma \neq \partial\}$$

On the other hand, we can also extract the information related to deadlock computations by defining the mapping:

$$\mathcal{O}_{ad}[G] = \{s_2 : s \cdot \partial \in \mathcal{O}(G)\}$$

where s_2 represents the sequence derived from each second component of s. That is, if $s = \sigma \cdot s'$, then $s_2 = \sigma_2 \cdot s'_2$. Finally, a map modelling infinite computations could be defined in a similar way.

3.5. An example

The features of the proposed environment are exemplified by simulating the control of a level crossing with a gate or barrier, by defining the behaviour of some objects and their relations. The example simulates the message flow established between a train and a gate by means of a controller. To obtain a more realistic situation, we have also considered cars. The train, car and gate objects are characterized by their state: a train can be 'travelling', 'approaching' the gate or 'ingate' (crossing the gate); the only interesting states of a car are 'travelling' and 'ingate': and a gate can be 'down' or 'up', depending whether the train is 'ingate' or not (Figure 3).

When a car is 'ingate' and a train is 'approaching', a dangerous situation is produced. In this case, the train must be stopped. Thus, the state 'stop' must be added to a train.

The vehicle class defines the common behaviour of trains and cars. So, a vehicle will have an input (default) and an output channel, to receive and send messages, respectively. A vehicle can receive an init message, which will initialize its state to 'travelling' and it will start the vehicle. The argument of the init message represents the distance to the gate. The movement of the vehicle is produced by the *travel* message, which updates the state *D* (by means of becomes) and continues to move the vehicle.

```
vehicle ; M
state D
messages
  ? init(Dist)       ← self!travel(Dist), D becomes
                       travelling.
  ? travel(Dist)     ← self!(update(Dist), cont(Dist)).
  ? update(Dist)     ← D = = ingate, outgate(Dist) :
                       M!measure(travelling), D becomes
                       travelling.
  ? update(Dist)     ← Dist ⩽ 20 :
                       M!measure(ingate), D becomes
                       ingate;
  ? update(Dist).
  ? cont(Dist)       ← D/= = stop :
                       step(Dist, New_Dist),
                       self!travel(New_Dist).
  ? cont(Dist)       ← D = = stop : true.
end.
```

The two kinds of vehicle considered, trains and cars, have the following behaviour:

```
train inherits vehicle
messages

  ? alarm            ← D becomes stop.
  ? travel(Dist)     ← D = = travelling, Dist < 2000 :
                       M!measure(approaching),
                       self!cont(Dist),
                       D becomes approaching;
  ? travel(Dist)     ← self!(update(Dist), cont(Dist)).
mode outgate(?)
  outgate(Dist)      ← Dist ⩽ −20.
  mode step(?, ^).
  step(Dist, NwD)    ← NwD is Dist-4.
end.
car inherits vehicle
```

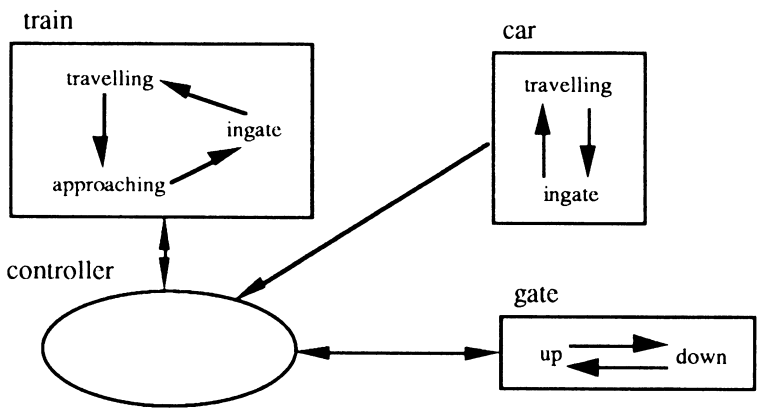

FIGURE 3. The interaction between a train, a car and a gate.

```
messages
  mode outgate(?).
  outgate(Dist)    ← Dist ≤ −10.
  mode step(?, ^).
  step(Dist, NwD) ← NwD is Dist-1.
end.
```

Both classes inherit from the vehicle class. In the first case, the travel messsage is redefined, and the possible values for the state *D* are 'travelling', 'approaching', 'stop', 'ingate'. So, for example, when the distance to the gate is less than 2000 units, and the state is 'travelling', then this changes to 'approaching', and a message is sent to the controller reporting the change along the channel *M*. In the two classes the step predicate is used to simulate the speed of the vehicles: in this example, a train runs four times faster than a car. The *outgate* predicate is used to detect the moment when the gate has been left behind.

Another object integrated in the simulation is the controller, whose behaviour was described in the previous section. An instance of that class receives messages from the train and from a device enabled to detect the cars crossing the gate. A controller can send messages to the gate and to the train. Thus, the previous class definition includes two input channels and two output channels. Its state is composed of a variable containing the value 'ingate' if a car is crossing the gate.

An instance of the controller class can receive the message *measure* from a train reporting its state: if the train is approaching, a down message is sent to the gate, and it waits for a confirmation. Similarly, when the train changes its state from the 'ingate' to 'travelling', the up message is sent to the gate. In both cases, the wait predicate is used to raise an exception if the confirmation arrives later than the established time. If the gate has some problem in going up an error message is sent to the gate, and if the gate has some problem in coming down or there is a car 'ingate', then an alarm message is sent to the approaching train (the alarm message stops the train).

Finally, the gate class is defined.

```
gate
message
  ? up(Ok)      ← go_up & Ok = ok.
  ? down(Ok)    ← go_down & Ok = ok.
  ? error       ← write('The gate is down').
  end
```

go_up and go_down predicates correspond to external processes simulating the physical actions of a gate, and they can be implemented as C functions.

A goal showing a particular simulation could be:

```
← train([init(90000)|T], M1), car(C, M2), gate(G),
  controller(M1, M2, G, T), C!init(5000).
```

A train T is created with the message init(90000) (note that the input channel of the train is treated as a list). At the same time an instance of car is created due to the message init(5000). The controller takes care of the safe working of the gate. In the example, if a dangerous situation is produced an exception is raised, but if there are no problems the computation does not finish.

If we want to create objects in different processors, we would make the following call:

```
← train([init(90000)|T], M1) @ host1,
  car(C, M2) @ host1, gate(G) @ host2,
  controller(M1, M2, G, T) @ host2, C!init(5000).
```

In this case, the train and car objects are created in host1, and the gate and controller instances are created in host2.

4. IMPLEMENTATION ISSUES

4.1. Distributed implementation

The environment has been implemented in a Sun-4 workstation network, by using TCP/IP for communication among hosts. A *virtual machine* can be created from each host in the network, and these machines can share resources by using servers. When a virtual machine is created an interpreter process is started in each host in the network.

These interpreters are connected by sockets in order to exchange system information. The interpreter of the root host also tries to create a communication channel with the other systems (root interpreters) in the network to manage server information. The structure of each interpreter is shown in Figure 4.

The interpreter of the root host also has a server module described later in this section. For now, the function and structure of each module and process in the interpreter is briefly described:

- *Incoming calls.* When the communication module receives an incoming metacall the channel sockets and the corresponding logic variables are created, and the process name and the variables are passed to this module. This module creates a local metacall that is executed by the execution module.
- *Outgoing calls.* This process is in charge of managing the creation of a remote process. When a remote

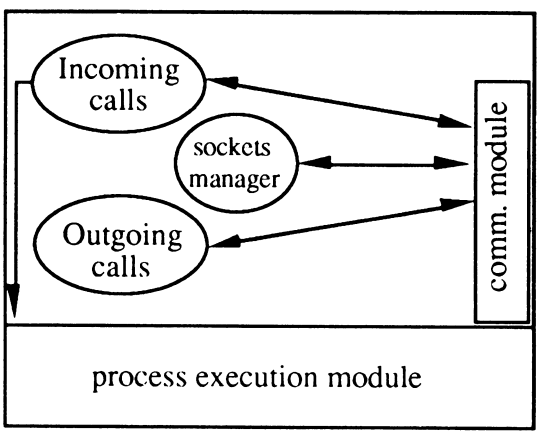

FIGURE 4. Interpreter structure.

predicate call is detected by the execution module a message is passed to this module which will interact with the communication module and with the remote host to execute the process. If the Host variable in the remote metacall is free, then this module is in charge of the assignment of a host for its execution.

- *Sockets manager.* This process is in charge of the assignment of socket numbers to variables and of maintaining the socket-variable table. The structure of this table is the following:

(Variables, ?/^,Host-sock. number)

When a variable is destroyed the socket is closed but can be reused.

- *Communication module.* This module is composed of the following processes:
  - *System messages process.* As there are some channels for communicating with the system interpreters and with the root hosts of other systems, this process sends and receives these messages from the other modules.
  - *Transmission process.* This process is initiated when any variable of the socket-variable table is instantiated, and it sends the value of the variable. If it contains free variables (incomplete messages) these variables are sent to the socket manager for the assignment of new sockets (its value is transmitted instead of the socket number).
  - *Reception process.* This process is initiated when any data arrives and it instantiates the variables to this received data.
  - *Process execution module.* This is an interpreter for the concurrent logic language Parlog. It also implements the real-time and distributed execution primitives interacting with the rest of the modules of the interpreter.

In the case of the interpreter that runs in the host of the system there is another module in charge of server management. This server manager creates the server and merger processes after the execution of create_server (process) primitive (Figure 5).

The server manager maintains a table with the name and the rights of each server (not all the hosts in the network can access a server). When a request for a server is received new variables are created and sent to the merger processes. After this, these variables are sent to the communication module to obtain socket numbers. If the service is granted a system message is sent to the host which asked for the service, including the socket number.

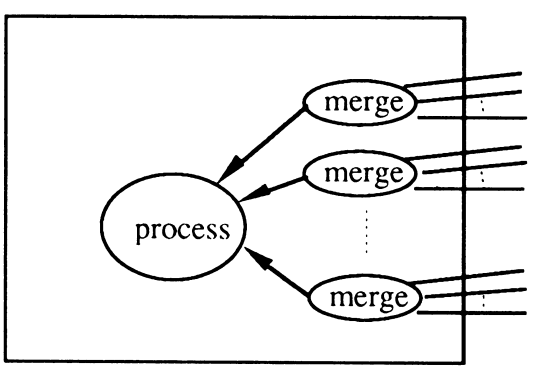

FIGURE 5. Server process.

Termination and deadlock detection are other important issues in the distributed implementation. In our model termination detection is achieved by the root processor, in the same way as in a uniprocessor implementation. An execution is finished when all the processes in the root processor have finished (including remote metacalls). A remote metacall finishes when the Status lists are closed.

Deadlock can easily be detected in a uniprocessor; if there is no process ready for execution and there are processes suspended in variables then a deadlock exists. Deadlock detection in distributed systems is more complicated. Because of the messages in transit we cannot immediately determine if a process is suspended because there is no remote producer for the variable or because there is a message still in transit. Deadlock detection in CLLs has been solved in different ways (Ichiyoshi *et al.*, 1987; Foster and Taylor, 1990). Usually, deadlock detection algorithms for distributed implementations of CLLs are based on distributed termination detection algorithms. A network of processes is deadlocked if all the processes have finished and some processes still remain suspended. We have implemented deadlock detection in a similar way to Ichiyoshi *et al.* (1987). Previous versions of the system used a technique based on diffusing computation with the remote metacall, as is explained in Díaz and Troya (1993), but this approach is less efficient.

4.2. Real-time extension

All the real time primitives are implemented in Parlog except the after primitive that is implemented as a built-in function. These primitives are implemented by using timed guards, i.e. clauses containing after primitives in the guards (Troya and Díaz, 1992). As an example, this is a naive implementation of timeout:

```
mode timeout (?, ?, ^).
timeout(Pred, Time, St) ← call(P, St, _) : true.
timeout(Pred, Time, St) ← after(Time) : St = timeout.
```

To implement real-time primitives it has been necessary to modify the usual CLL scheduling algorithm implementations. In a CLL implementation there is only one queue for all the processes and the scheduling algorithm is defined to make the execution more efficient (i.e. taking into account tail recursion).

In our implementation we have defined three priority levels and three execution queues. The scheduling algorithm is as follows:

- A process that can be reduced with a predicate with timed guards is considered a real-time process and it always has maximum priority. This allows the accurate measurement of the time expressed in the guards. If more than one of these processes is created at the same time the scheduling time is the same for all the processes. All these processes are queued in the maximum priority queue.
- The second queue (medium priority queue) is ordered by the dealiness of the processes. Once a real-time process is reduced, if a timed guard succeeds, the process deadline is computed and the process is inserted in this queue in order of deadlines.
- The third queue is for non-priority processes.
- A process is scheduled from a queue only if the higher priority queues are empty, or the deadlines of the second queue allow it.
- A process can be suspended at the same time in a variable and in timed guard. If the variable is instantiated the process must be deleted from the execution queue and if the time guard expires the process must be deleted from the suspension queue of the variable.

This algorithm works quite well if the charge of the processor is not very high and it succeeds in meeting time constraints in the order of 0.1 seconds. In any case, the system can lose a time constraint for different reasons (processor overload, garbage collection, ...) and in these cases the environment reports the loss of this deadline.

4.3. Garbage collection

One of the most important problems that arises in the distributed implementation of CLLs is garbage collection. Some algorithms for garbage collection in distributed systems do exist (Ichiyoshi *et al.*, 1987) but a common drawback of many of them is that the computation has to be stopped to achieve global garbage collection. There also exist algorithms that collect garbage incrementally, but they incur very high costs. In our model, there are no remote references to variables and garbage collection can be achieved locally, in the same way as in a monoprocessor implementation.

The efficiency of garbage collection algorithms is especially important in real-time systems because a garbage collection phase can cause real-time process deadline loss. Some proposals also exist for real-time garbage collection algorithms (Armstrong *et al.*, 1993).

In our environment we have used the *stop & copy* garbage collection algorithm (Baker, 1978) with some modifications. In the original algorithm a garbage collection phase starts when the heap or the arguments stack is full. In a real-time environment it is necessary to take into account process deadlines before starting a garbage collection phase. We have defined a critical threshold, that depends on the amount of free space in the stack and heap, and on the number of processes ready for execution. A garbage collection phase will start only if:

- There is no process with maximum priority.
- The deadlines of real-time processes allow a garbage collection phase without the loss of any deadline.

A garbage collection phase can also start if all the processes are of low priority. In order to implement this algorithm it is necessary to know a limit for the maximum time of garbage collection.

There are other approaches to improve garbage collection in real-time systems, as in that of Bekkers and Ungars (1992), which makes partial garbage collections. This is better in meeting time constraints, but is less efficient.

5. CONCLUSIONS

We have proposed a distributed declarative environment incorporating real time primitives and object-oriented mechanisms. The environment is based on CLLs, particularly on the Parlog language, which has been extended for distributed, real-time and object-oriented programming. The extensions have been made by putting forward an integrated view of each paradigm, by giving a global operational semantics. Transition systems have been used to define the behaviour of our environment, by taking into account the different mechanisms which have been incorporated into the language: inheritance, message passing, time guards, etc. An observability notion is introduced to model finite (success, deadlock and fail) and infinite computations. Although they have not been considered in the present paper, we are also interested in the development of other theoretical aspects. In this sense, we think the approach used by Pimentel and Troya (1992, 1993) to define a declarative and a compositional semantics for a concurrent object-oriented logic language could be extended, in order to obtain similar results when a real-time extension is included.

Implementation aspects have also been considered. In this sense, we have modified the communication mechanism between processes in different hosts, and we have implemented a remote metacall for distributed programming. The real-time extension has made the modification of the CLLs execution model necessary, including priorities and a new scheduling policy. Other implementation aspects have also been considered such as garbage collection, and termination and deadlock detection.

REFERENCES

Agha, G. (1989) *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA.

Ahuja, S., Carriero, N. and Gelernter, D. (1986) Linda and Friends. *IEEE Comp.*, **19**(8), 26–34.

Armstrong, J., Virding, R. and Williams, M. (1993) *Concurrent Programming in ERLANG.* Prentice Hall, Englewood Cliffs, NJ.

Baker, H. G. (1978) List processing in real time on a serial computer. *Commun. ACM*, **21**, 280–294.

Bal, H. E. *et al.* (1987) *Orca: A Language for Distributed Programming.* Technical Report IR 140, Department of Mathematics and CS, University of Vrije.

Bal, H. E., Steiner, J. G. and Tanenbaum, A. S. (1989) Programming languages for distributed computing systems. *ACM Comp. Surv.*, **21**, 261–322.

Banatre, J.-P. and Lemetayer, D. (1990) The GAMMA model and its discipline of programming. *Sci. Comp. Prog.*, **15**, 55–77.

Bekkers, Y. and Ungaro, L. (1992) REAL-time memory management for Prolog. In Voronkov, A. (ed.), *Lecture Notes in Artificial Intelligence*. Logic Programming.

Boer, F. S., Kok, J. N., Palamidessi, C. and Rutten, J. M. M. (1989) Semantic models for a version of PARLOG. In *Proc. 6th Int. Conf. on Logic Programming*, pp. 621–636, MIT Press, Cambridge, MA.

Boussinot, F. and de Simone, R. (1991) The Esterel language. *Proc. IEEE*, **79**, 1293–1304.

Brogi, A. and Gorrieri, R. (1989) Model theoretic, fixpoint and operational semantics for a distributed logic language. *Proc. 6th Int. Conf. on Logic Programming*, pp. 637–654, MIT Press, Cambridge, MA.

Clark, K. and Foster, I. T. (1987) A declarative environment for concurrent logic programming. In *Proc. TAPSOFT '87*, Pisa, Italy.

Clark, K. and Gregory, S. (1986) PARLOG: parallel programming in logic. *ACM Trans. Prog. Lang. Syst.*, **8**, 1–49.

Cohen, D., Huntbach, M. M. and Ringwood, G. A. (1991) *Logical Occam*. Technical Report, Department of Computer Science, Queen Mary and Westfield College, London.

Coleman, D., Hayes, F. and Bear, S. (1989) *Introducing Objectcharts or How to use Statecharts in Object Oriented Design*. Technical Report HPL-ISC-TM-89-167.

Crammond, J. (1990) *The Abstract Machine and Implementation of Parlog*. Technical Report, Department of Computing, Imperial College, London.

Davison, A. (1989). *Polka: A Parlog Object-Oriented Language*. PhD Thesis, University of London.

Díaz, M. and Troya, J. M. (1992) Extending concurrent logic languages for real-time and distributed systems. In *Proc. European Workshops on Parallel Computing*, Barcelona, Spain.

Díaz, M. and Troya, J. M. (1993) A Parlog based real-time distributed logic environment. *Future Generation Comp. Syst.*, **9**, 201–218.

Díaz, M., Pimentel, E. and Troya, J. M. (1992) A declarative environment for distributed programming. In *Proc. of 12th IFIP*, pp. 163–169.

Elshiewy, M. A. (1990). *Robust Coordinated Reactive Computing in Sandra*. PhD Thesis, SICS.

Foster, I. (1990). *System Programming in Parallel Logic Languages*. Prentice-Hall, Englewood Cliffs, NJ.

Foster, I. and Taylor, S. (1988) Flat Parlog: a basic for comparison. *Int. J. Parallel Prog.*, 1–16.

Foster, I. and Taylor, S. (1990) *STRAND: New Concepts in Parallel Programming*. Prentice-Hall, Englewood Cliffs, NJ.

Halbwachs, N., Caspi, P., Raymond, P. and Pilaud, D. (1991) The synchronous dataflow programming language LUSTRE. *Proc. IEEE*, **79**, 1305–1320.

Harel, D. (1987) Statecharts: a visual approach to complex systems. *Sci. Comp. Prog.*, **8**, 231–274.

Hoare, C. A. R. (1985) *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ.

Ichiyoshi, N., Miyazaki, T. and Taki, K. (1987) A distributed implementation of flat GHC on multi-PSI. In *Proc. 4th Int. Conf. on Logic Programming*. MIT Press, Cambridge, MA.

INMOS Limited (1988) *Occam 2 Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ.

Kahn, K. and Miller, M. (1988) Language design and open systems. In *The Ecology of Computation*, pp. 291–313. North-Holland, Amsterdam.

Kahn, K., Tribble, E. D., Miller, M. S. and Bobrow, D. G. (1987) Vulcan: logical concurrent objects. In Shriver and Wegner (eds), *Research Directions in Object-Oriented Programming*, pp. 75–112, MIT Press, Cambridge, MA.

Koymans, R., Shyamasundar, R. K., Roever, W. P., Gerth, R. and Arun-Kumar, S. (1985) Compositional semantics for real time distributed computing. In *Proc. Logics of Programs (LNCS 193)*, Springer-Verlag, New York.

Leguernic, P., Gautier, T., Leborgne, M. and Lemaire, C. (1991) Programming real time applications with Signal. *Proc. IEEE*, **79**, 1321–1336.

Liskov, B. and Scheiffler, R. (1983) Guardians and actions linguistic support for robust, distributed programs. In *ACM Trans. Prog. Lang. Syst.*, **5**.

Nicollin, X. and Sifakis, J. (1991) An overview and synthesis on timed process algebras. In *Real-time Theory in Practice*, pp. 526–548, Springer-Verlag, New York.

Ohki, M., Takeuchi, A. and Furukawa, K. (1987) An object-oriented programming language based on the parallel logic programming language KL1. In *Proc. 4th Int. Conf. on Logic Programming*, pp. 894–909, MIT Press, Cambridge, MA.

Pimentel, E. and Troya, J. M. (1991) A concurrent object oriented logic language. In *Atti del Sesto Convegno sulla Prog. Log. GULP'91*.

Pimentel, E. and Troya, J. M. (1992) $L^2 \| O^2$: operational and declarative semantics. In *Proc. ALPUK'92, Workshops in Computer Science*. Springer-Verlag, New York.

Pimentel, E. and Troya, J. M. (1993) Compositionality issues of concurrent object-oriented logic languages. *LNCS 694*, pp. 529–540, Springer-Verlag, New York.

Saraswat, V. A., Kahn, K. and Levi, J. (1990) Janus: A step towards distributed constraint programming. In *Proc. ICLP*.

Shapiro, E. (1983) *A Subset of Concurrent Prolog and its Interpreter*. Technical Report, The Weizmann Institute of Science.

Shapiro, E. (1989) The family of concurrent logic programming languages. *ACM Comp. Surv.*, **21**, 413–510.

Shapiro, E. and Takeuchi, A. (1983) Object oriented programming in concurrent prolog. *New Generation Comp.*, **1**, 25–48.

Taylor, S. (1989). *Parallel Logic Programming Techniques*. Prentice-Hall, Englewood Cliffs, NJ.

Troya, J. M. and Díaz, M. (1992) RDLE: a real-time distributed logic environment. In *Proc. 11th IEEE Int. Phoenix Conf. on Computing and Communications*, Phoenix, AZ.

Ueda, K. (1985) *Guarded Horn Clauses*. ICOT Technical Report TR-103, Institute for New Generation Computer Technology, Tokyo.

US Department of Defense (1983) *The Ada Programming Language*. US Government Printing Office, Washington, DC.

Warren, D. H. D. (1982) Perpetual processes—an unexploited prolog technique (short communication). *Logic Prog. Newslett.*, **3**.

Yoshida, K. and Chikayama, T. (1988) A'UM: a stream-based concurrent object-oriented language. In *Proc. Int. Conf. on FGCS*, pp. 638–649, ICOT, Tokyo.