A Coordination Language for Collective Agent Based Systems: GroupLog

Fernanda Barbosa and José C. Cunha
Dep. de Informática - Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa - Portugal
(fb,jcc)＠di.fct.unl.pt

Abstract

The paper presents the GroupLog coordination language for collective agent based systems. It is a logic language based on the notion of group, a cooperating entity whose definition is important to model cooperation in a flexible and well-structured manner and to hide low-level management of coordination activities. In the paper we give an informal presentation of the GroupLog language and illustrate its use through examples.

1 Introduction

In multi-agent systems, agents cooperate to achieve some task that might not otherwise be achieved by each individual agent. Interaction between agents is absolutely essential in a multi-agent system. If agents must cooperate then they must be able to communicate and a model for coordinating their activities is required.

The coordination of a multi-agent system involves two main aspects. The first one relates to the organization of the multi-agent system and addresses the global structuring of the system and the interactions among agents. The second aspect relates to the modeling of the social behaviors of the agents, namely how each agent behaves in terms of its perception of the environment, and how global rules of behavior can be specified in terms of some coordination entity. These two levels of coordination are identified in [Schumacher, 1999] as objective and subjective coordination, respectively.

The focus of our work on the GroupLog model is to investigate the use of group based abstractions to support the structuring and the management of multiple cooperating entities. In the past decade, the group concept appeared at the operating system or at the middleware levels [Birman and Joseph, 1987], but only few proposals attempted its integration into programming languages [Pardyak and Bershad, 1994, Mafféis, 1996, Holzbacher, 1996, Fisher, 1995, Cruz and Ducasse, 1999, Buffo and Buchs, 1997].

As it has been widely recognized, logic based models allow a declarative programming style that may ease the specification of complex systems. Such models become more interesting when they are coupled with abstractions to support structuring capabilities. A logic based approach finds applications in multiple areas, where there is the need of some inference capability (modeled by logic based agents in our approach), coupled with a requirement for an adequate model for specifying the coordination between autonomous agents (modeled by groups in our approach).

GroupLog is based on a logic programming model which supports multi-agent systems through the specification of: (1) individual agents; (2) inter-agent communication patterns; and (3) cooperation behaviors.

GroupLog defines two main program structuring entities: agents and groups. These concepts aim at capturing respectively, the computation and the coordination aspects.

The agent notion aims at capturing the computation aspect and its encapsulation within an autonomous entity. In GroupLog, an agent is specified using the Extended Horn Clause model [Jacquet
and Monteiro, 1991], a logic programming model which allows an object based interpretation. It
defines an agent as a logic entity with well-defined interface, knowledge and behavior.

Through the group notion, GroupLog can be seen as supporting the above mentioned forms of
objective and subjective coordination. On one hand, groups specify the organization and structuring
of an agent space into distinct sets. Several inter-agent communication patterns are supported:
point-to-point, multicast, and multiple shared tuple spaces. Furthermore, it allows the modeling of
dynamic systems, where agents can dynamically enter and leave such cooperating groups. On the
other hand, each group can be seen as an agent, with well defined internal behavior and commu-
nication interface. So, the group notion can be used to model the social behaviors of its members,
which act as cooperating entities.

In the remainder of the paper, we first present a brief description of the GroupLog language. In 3,
we give two examples: "Two-Agent Meeting Scheduling" and "Dynamic Philosopher Table". Then
we briefly describe the current implementation approach. In 5, we discuss GroupLog in relation to
other languages and in 6 we present some conclusions.

2 The Language GroupLog

GroupLog extends the Extended Horn Clause language (EHC) [Jacquet and Monteiro, 1991], at two
levels: L1 defines agents and L2 defines groups of agents. A GroupLog system contains concurrently
executing agents able to: (1) communicate through interface predicates, and (2) join groups to
coordinate their activities. In the following, we first summarize EHC and then describe the two
mentioned levels.

2.1 Extended Horn Clauses

Extended Horn Clause Logic (EHC) is an extension to Horn Clause Logic (HC) with mechanisms
for concurrency and synchronous communication. EHC supports the parallel composition of goals
and provides clauses for modeling the interaction between two parallel processes through a joint
synchronization step.

Such EHC clauses allow an interpretation of message-based communication with an object, when
some process invokes method "mess(M)" of an object instance "obj(S)". In each object instance, a
perpetual process "obj(S)" is responsible for the handling of message "mess(M)" according to the
following cases:

(i) obj(S) :: mess(M) :: method(M) | obj(NewS) :: true.

(ii) obj(S) :: mess(M) :: method(M) | obj(NewS) :: process(M).

This is achieved by activating one of these clauses and solving goal "method(M)", as explained below.
In clause (i) the message is consumed. In clause (ii) the message is not consumed and is reduced to
the goal "process(M)". In both cases the object instance "obj(S)" reduces to "obj(NewS)" which
can be interpreted as a change in the configuration of the object instance.

A EHC clause has the following form:
(A) H1 :: ... :: Hn :: G | G1 :: ... :: Gn

where Hi are atoms and G and G1 are goals. The goals are conjunctions of atoms built by using the
following operators: ";;" denotes sequential composition, "||" denotes parallel composition and "&"
denotes simultaneous reduction of atoms.

The above example of an object instance corresponds to clause (A) with n=2. This is:
(1) H1 :: H2 :: G | G1 :: G2

A clause as (1) can be used for the joint synchronization of two concurrent goals C1 and C2. On a
concurrent execution, each goal invocation suspends and waits for the other and only then C1 and
C2 may be simultaneously reduced to G1 and G2. This joint derivation step is only successful if
there is a permutation of C1 and C2 that unifies with (H1,H2) with most general unifier θ and if Gθ
is true. In that case, each ith element of the permutation is reduced to G1θ (i=1,2).
Parallel goal composition, $C_1|C_2$, may be achieved in two ways:

(a) Using two Horn Clauses
$H_i :: G_i$ and $H_j :: G_j$ where $C_1$ unifies with $H_i$ and $C_2$ unifies with $H_j$

(b) Using one Extended Horn Clause
$H_1 :: H_2 :: G | G_1 :: G_2$, where a permutation of $(C_1, C_2)$ unifies with $(H_1, H_2)$, as explained above.

Simultaneous reduction of goals $C_1\&C_2$ can only use the (b) clause. In the following, this is used to model the communication between two agents.

**Use of Extended Horn Clauses in GroupLog** In GroupLog we consider only Extended Horn Clauses with one or two atoms in the clause head corresponding, respectively, to Horn Clauses [Lloyd, 1987] and EHC clauses of form (1). In section 2.2, we present an operational interpretation of these clauses in GroupLog. Syntactically we use "\" instead of ";" for sequential composition, we use "<<" instead of "\&" for communication, and ";" instead of ":\".

### 2.2 L₁ - Dynamic structuring units of program entities - Agents

The specification of an agent consists of giving:

- The name of the agent;
- The list of interface predicates which can be invoked by other agents;
- The agent program.

Syntactically, an agent is defined as follows:

```agent <agent_name>
  /* agent context */
  context(<list_modules>).
  /* communication context*/
  interface(<list_interface_pred_names>).
  /* agent behavior */
  <list of interface clauses>
  /* agent creation */
  <creation clause>
</agent>
```

In the following we illustrate the definition of a stack agent.

```agent stack{
  context().
  interface(push(X), pop(X), top(X)).
  /* interface clauses*/
  stack(Id, L) : push(X) :- | stack(Id, [X|L]).
  stack(Id, [X|L]) : pop(X) :- | stack(Id, L).
  stack(Id, [X|L]) : top(X) :- | stack(Id, [X|L]).
  /* creation clause */
  stack(Id) : new :- | stack(Id, []).
}</agent>
```

**Agent name and instances** Each agent has a name that defines its type. An instance of an agent of a given type has a unique identifier (e.g. stack(1) instance 1 of agent of type “stack”). Such instance is created by invoking the pre-defined predicate `create_instance(<agent_instance>)`, which creates a perpetual process that represents the agent instance and activates the creation clause.
a(Id) : new :- <initial_conf> | <initial_conf>.

As a result, the top goal (initial_conf) is launched which corresponds to the initial configuration of the agent, if the goal initial_cond evaluates to true. In the example of stack(1), the agent starts with a current configuration defined as stack(1,[2]) (see clause (3) in the stack agent), which corresponds to the empty stack, i.e. the initial configuration.

**Agent interactions**  The communication context defines the interaction of the agent with its environment and is specified by interface(<list_interf_pred_names>) where <list_interf_pred_names> is a list of interface predicate names and interface is a predefined predicate. The interface predicates define the visible "entry points" of an agent that can be invoked from the outside environment. In the example of "stack" agent, the interface predicates are: push(X), pop(X) and top(X) (see clause (2) of the stack agent).

**Agent program**  The program of an agent is a non empty set of clauses, defined by the union of the agent context and the set of interface predicates and the creation clause. The agent context is specified by a list of module names using the predefined predicate context(<list_modules>). Its meaning is the union of the clauses contained in the listed modules in their textual ordering. Each module is an ordered set of EHC clauses as described in section 2.1, where n<=2. In the example of a "stack" agent, the agent context is empty (see clause (1) of the stack agent).

The interface clauses specify the way an agent interacts with its outside environment. The communication with an agent uses operator "<<" and is synchronous. This can be one to one or one to any. In one to one communication, the destination agent and its interface predicate are explicitly cited. For example stack(1)<<pop(X) denotes a communication with agent’s instance stack(1) through interface predicate “pop”. In one to any communication, we only need to specify the type of agent and the interface predicate, because this is a form of selective communication targeted at all instances of the specified type. For example stack(_)<<pop(X) denotes a communication addressing all instances of an agent of type stack through interface predicate “pop”.

The agent exhibits a well defined behavior, depending on its current configuration and on the invoked interface predicate. There is a non empty set of Extended Horn Clauses associated with each interface predicate. Two forms are allowed, where form (ii) is a simplification of (i) when <process> is true:

(i) <agent_conf> : <interf_pred_name> :-
   <set_cond> | <new_agent_conf> : <process>.

(ii) <agent_conf> : <interf_pred_name> :-
   <set_cond> | <new_agent_conf>.

The above clauses have the same interpretation as an EHC clause (see (1) in section 2.1) where:

1. H1 corresponds to the <agent_conf> of the addressed agent. This means that this agent instance must have a current configuration that unifies with <agent_conf>. This current configuration is managed by the agent’s perpetual process.

2. H2 corresponds to the <interf_pred_name> in the interface predicate that must unify with the interface predicate that is cited by the invoker agent.

3. G corresponds to <set_cond> that must be true so that the communication can be successful. The goal <set_cond> may include communication with other agents.

4. G1 corresponds to the <new_agent_conf> for the addressed agent.

5. In form (i) G2 corresponds to <process> and has a similar interpretation as "process(M)" in the communication with an object in EHC (see section 2.1).
The evaluation of the first three conditions (1,2,3) as well as the transition to a new configuration (4) correspond to an atomic step in the invoked agent. This means any other invocations made to this agent remain suspended until this agent has completed such an atomic step, i.e. is available again for further interactions. If the first three conditions are satisfied, the agent changes to a new configuration.

From the point view of the invoker agent, the invocation fails if:

- The invoked agent does not exist, or
- The invoked predicate does not unify with \(<\text{interface\_name}>\) (condition 2), or
- The goal \(<\text{set\_cond}>\) fails (condition 3), or
- The goal \(<\text{process}>\) fails (condition 5).

An invocation of an interface predicate suspends if:

- The invoked agent is busy evaluating a previous invocation, or
- The invoked agent has not a current configuration that unifies with \(<\text{agent\_config}>\) (condition 1).

Given the current configuration of an agent and an invoked interface predicate there is a (possibly empty) set of clauses which are \textit{enabled} by that configuration. These are the clauses of that interface predicate such that \(<\text{agent\_config}>\) unifies with the current agent configuration of the agent. All other clauses are \textit{disabled} for that configuration. This notion of disabled clauses represents the clauses that can not be used to unify with an invoked interface predicate because of the current agent configuration. An example of a disabled clause is given by the clause with head "\text{\textit{stack}([1],[X],[L]):\textit{pop}(X)}" when stack agent is in its initial configuration (\textit{stack}([1],[])). This means an invocation of "\textit{stack}([1]):\textit{pop}([C])" will suspend the invoker until a matching configuration (triggered by a change to a nonempty stack) enables the cited clause. If there is a matching enabled interface clause, the communication is accepted by the invoked agent, so if \textit{"set\_cond"} is true, it changes the configuration to \textit{<new\_agent\_config>}. This configuration transition is non reversible and the agent becomes available for further interactions with other agents. This means that in clause (ii) the interaction is completed and the invoker proceeds with its computation. However in clauses like (i), the invoker agent must wait until this interaction is completed by the evaluation of \textit{g\:<\text{process}>} locally to the invoked agent i.e. within this agent context (this communication is said to be \textit{ongoing}). The evaluation of \textit{<process>} is intended to perform local management actions internal to the invoked agent so we restrict its evaluation not to involve communication with other agents. It is performed concurrently with further interactions between the invoked agent and other invoker agents. This corresponds to an implicit spawning of a concurrent thread, internal to the invoked agent. If \textit{<process>} fails, the communication fails for the invoker, although the invoked agent has already committed to a new configuration. Whenever an agent changes to a new configuration, there is a commitment. This decision relates to our view of an agent as a reactive entity that evolves to new configurations as a result of its interactions with the environment. Otherwise we would have to manage the complexity of distributed backtracking which would affect the practical feasibility of the model.

One to any communication, as in \textit{\textit{stack}([X]):\textit{pop}(X)}), proceeds in two steps:

1. The communication suspends until there is an available instance of agent \textit{stack} (i.e. that instance is currently waiting for interactions) with an enabled interface clause for this invocation of \textit{pop};
2. If multiple instances are found in step 1, one is selected in a non deterministic way and the communication with the selected instance proceeds as in one to one communication.

The communication fails when no instances of "\textit{stack}" are found, or there is no interface predicate "\textit{pop}" in agent \textit{stack}, or when the one to one communication fails in the selected instance of agent \textit{stack}. 

5
Agent termination  An instance of an agent can be canceled by the predefined interface predicate kill, or implicitly canceled by an interface clause such as:

\(<\text{current\_config}> : \text{<interf\_pred\_name>} :- \text{<set\_cond>} | \text{true} : \text{true}\).

This implies the forced failure of all communications involving this agent (both suspended and ongoing). If \(<\text{set\_cond}>\) is true, this clause simplifies to:

\(<\text{current\_config}> : \text{<interf\_pred\_name>}.\)

2.3  L2 - Dynamic grouping of agents - Group

The need for structuring the space of agents in L1 and supporting agent coordination motivated the definition of groups as dynamic entities. Cooperation relies on two basic forms of communication: (1) access to a shared group state, based on the Linda model [Gelernter, 1985]; and (2) direct communication through interface predicates. Each group has a name that defines its type. Associated with each group we define an agent whose name is the same as the group name. This agent is the group representative and is defined by:

- a list of interface predicates, which defines the list of group interface predicates, i.e. those predicates that can be invoked by external agents (see (2) below);
- a program, which is defined in the same way as for any agent (see (1), (3), (4) below).

Syntactically, a group is defined as follows:

\[
\begin{align*}
\text{group} & \ <\text{name\_group}> \\
& \{ \\
& \quad \text{context}\ (<\text{list\_modules}>). \quad (1) \\
& \quad \text{interface}\ (<\text{group\_interface\_pred\_names}>). \quad (2) \\
& \quad <\text{list\ of\ interface\ clauses}> \quad (3) \\
& \quad <\text{the\ creation\ clause}> \quad (4) \\
& \}
\end{align*}
\]

So we are using the notion of group representative to handle the interface of a group. A group instance of a certain type is created by the predicate create\_instance, which indeed creates an instance of the agent corresponding to the associated group representative. In order to understand the behavior of a group we must discuss three main aspects:

- the group membership;
- the group communication;
- the shared group state.

Group membership The group is a composition of agents, called its members. The group membership changes dynamically as a result of predefined interface predicates “add” and “delete”.

A group hides its members from the outside but allows the redirection of communication to them through the group interface predicates. The internal concurrency to a group is explicitly defined by its members that have access to a shared group state, for internal coordination. An agent can belong to one or more groups and inspect their membership through the following predefined predicates:

- \(\text{my\_group}(<\text{group\_list}>)\) where \(<\text{group\_list}>\) returns the list of groups containing the invoking agent;
- \(\text{members}(<\text{a\_group}>, <\text{members\_list}>)\) where \(<\text{members\_list}>\) returns the list of members in \(<\text{a\_group}>; it can only be invoked by the members of a group.\)
In L1, all agents belong to the universe of agents (U) that models a flat space. In L2, an agent only belongs to U if it is not a member of any group. All agents in U can directly communicate with one other using one to one or one to any communication. When an agent in U joins a group, it is removed from U so it becomes inaccessible to all other agents in U. An agent A can only be accessed by the following classes of agents: (1) directly by members of the same group as A, through the interface predicates of A, and (2) indirectly, by other agents, if they have access to the group interface predicates of one of the groups containing A. An agent is put back in U only when it leaves all the groups it had previously joined. In L2, the termination of an agent implies its removal from all its groups as well as from U. The elimination of a group is done as described for an agent but it is only completed when no members are left in the group.

**Group communication** The interface clauses have the following form:

(i) \[ \text{<group_config> : <interf_pred_name> :-} \]
\[ \text{<set_cond> | <new_group_config> : <process>}. \]

(ii) \[ \text{<group_config> : <interf_pred_name> :-} \]
\[ \text{<set_cond> | <new_group_config>}. \]

In these clauses the “set_cond” and “process” goals are evaluated by the group representative in the group context (see (1) in the group definition). In these goals we allow the invocation of: (1) predicates defined by the group context, (2) predicates to access the shared group state or (3) interface predicates of group members or other agents. Unlike an agent, communication is allowed in goal <process> which is evaluated as a thread of the group representative. Besides using the interface predicate of other agents, each member can communicate with other members in the same group through the following predicate, which provides a form of broadcast:

\[ \text{all(<a_group>, <int_pred_name>, <type>, <ans>)} \]

where \text{<a_group>} is an instance of a group and \text{<int_pred_name>} is an interface predicate. All group members are addressed such that they have defined \text{<int_pred_name>} as an interface predicate. The third argument \text{<type>} is one of (none, one, all) and it means the invoker requires, respectively, no answer, one answer, or answers from all the addressed group members. The fourth argument \text{<ans>} is a list of pairs \text{(instance, reply)} such that \text{<instance>} identifies the invoked agent, and \text{<reply>} is the complete solution to \text{<int_pred_name>}. The group representative can also use the \text{all} predicate to address its group members.

**Shared group state** The shared group state is a multi set of atoms, defined in a module by the predefined interface predicate “state”. For example, given an instance of group meet_sch(1), the goal \text{meet_sch(1)<state(m1)} creates the shared group state, based on the contents of module \text{m1}. The members of the group may interact by accessing the shared group state, i.e. by reading and writing state elements. The predefined predicates - \text{rd, ts, in e out} allow access to the state of a group G:

1. \text{ts(G, State_group)} : is true if there is a subset of the shared state that matches “State_group”;
2. \text{rd(G, State_group)} : reads a subset from the shared state that matches “State_group”;
3. \text{in(G, State_group)} : reads a subset from the shared state matching “State_group” and removes it;
4. \text{out(G, State_group)} : puts the subset “State_group” in the shared state.

In 2 and 3 above, the invocations suspend until there is a subset of the shared state matching “State_group”. The predicates \text{ts} and \text{out} are non blocking. Namely, \text{ts} fails if no matching is possible. The modifications made to the shared state, due to these predicates, are not reversible. Note that the evaluation of these predicates is atomic with respect to the subset of the shared group state.
3 Examples

In this section, we illustrate the use of GroupLog through two typical examples. GroupLog concepts can be used to define the coordination of a set of related agents, as it allows:

- communication and coordination among related agents;
- dynamic configuration of the system.

In the examples presented in this section, “Two-Agent Meeting Scheduling” and “Dynamic Philosophers’ Table”, the need to coordinate a set of related agents motivates the definition of two groups, $meet_{sch}$ and $table$, respectively.

In the first example, “Two-Agent Meeting Scheduling”, we can see how the members of a group can coordinate through the shared group state. In the second example, ”Dynamic Philosophers’ Table”, we can see also how the group notion provides more than a composition of agents. In this case, the group has a behavior itself, that defines the coordination rules of its members as an agent society. In this example, this behavior is related to the arrivals and departures of the philosophers.

Another typical example of a Philosophers’ Restaurant can be seen in [Barbosa and Cunha, 1999], where the group notion is also used to model the dynamic evolution of the system.

3.1 Two-Agent Meeting Scheduling

The Meeting Scheduling is a complex problem which consists of finding a time schedule for a meeting involving N agents. The meeting time must be accepted by all agents. The basic idea is to allow the agents to coordinate themselves in order to reach an agreement. The version below (see also in [Hindriks et al., 1999]) is a simplified version which considers the meeting of two agents only - Two-Agents Meeting Scheduling. The solution can be generalized to N agents by having one of them as the host and the others as guests. In this version, one of the agents acts as the host, that is the agent who is trying to arrange the meeting. The other agent is the guest. We assume that the agents always share a free time slot of the appropriate length in their agenda such that the meeting can be scheduled. The group $meet_{sch}$, as defined in below, is used to join the agents in such a way that they can cooperate through the shared group state.

```prolog
group meet_sch{
    context().
    interface(begin).
        /* interface clauses */
    meet_sch(Id) : begin :-
        members(meet_sch(Id),[H,I]),
        rd(meet_sch(Id),meet(MeetId)),
        H<begin(I,MeetId) || I<begin(H,MeetId) | meet_sch(Id).
        /* creation clause */
    meet_sch(Id) : new :- | meet_sch(Id).
}
```

In this example, both agents have some common knowledge related to: (1) finding a possible time for the meeting in accordance with their agenda, and (2) telling about a possible time slot to the other partner. This knowledge is defined as:

```prolog
module schedule{
    found_time(S,meet(MeetId,T)) :-
        /* returns in T a possible time to meet MeetId in accordance with the schedule S */
    tell_time(I,S,meet(MeetId)) :-
        found_time(S,meet(MeetId,T)),
        cut(meet_sch(I),tell(I,meet(MeetId,T))),
```

8
ts(meet_sch(I), stop_sch(I)).

tell_time(I, _, _) :- in(meet_sch(I), stop_sch(I)).
}

The agents are modeled by two concurrent goals: (1) find and inform about the possible times, and (2) find a common time (in the case of the host) or wait for the confirmation of the time (in the case of the guest). This knowledge is defined in the following modules:

module schd_host{
   ex_host(I, Guest, S, meet(MeetId, T)) :-
      tell_time(I, S, meet(MeetId)) || match_time(I, Guest, meet(MeetId, T)).
   match_time(I, Guest, Meet) :-
      in(meet_sch(I), [tell(I, Meet), tell(Guest, Meet)]),
      out(meet_sch(I), [stop_sch(I), confirm(I, Meet)]).
}

module schd_guest{
   ex_guest(I, Host, S, meet(MeetId, T)) :-
      tell_time(I, S, meet(MeetId)) || conf_guest(I, Host, meet(MeetId, T)).
   conf_guest(I, Host, Meet) :-
      in(meet_sch(I), confirm(Host, Meet)),
      out(meet_sch(I), stop_sch(I)).
}

The host and the guest are defined as agents in the following way:

agent host{
   context(schedule, schd_host).
   interface(begin(MeetId, Guest)).

   host(Id, S) : begin(MeetId, Guest) :-
      ex_host(host(Id), Guest, S, meet(MeetId, T))
      | host(Id, [meet(MeetId, T) | S]).
   host(Id) : new :-
      meet_sch(I) <<add(host(Id)) | host(Id, []).
}

agent guest{
   context(schedule, schd_guest).
   interface(begin(MeetId, Host)).

   guest(Id, S) : begin(MeetId, Host) :-
      ex_guest(guest(Id), Host, S, meet(MeetId, T))
      | guest(Id, [meet(MeetId, T) | S]).
   guest(Id) : new :-
      meet_sch(I) <<add(guest(Id)) | guest(Id, []).
}

This program is started by creating an instance of the group “meet_sch(I)”, and the instances of agents “host(I)” and “guest(I)”. In this example, we can see how GroupLog concepts are used to coordinate a set of cooperating agents through a shared group state.

### 3.2 The Dynamic Philosophers’ Table

In the classical version of this example there are five philosophers [Sibertin-Blan, 1994]. Each philosopher is seating at the table and has two forks by its sides. In this version, we assume that forks are in the middle of the table, i.e. they are not associated to the seats.

A group, called table, is used to coordinate the philosophers. This group has two objectives:
• To join the philosophers together and allow them to share the state of the table (forks and seats);

• To model the behavior of the table, concerning the philosophers' arrival and departure.

The group table is defined as follows:

group table{
  context().
  interface().

  table(Id) : add(Fil) :- | table(Id) : in(table(Id),seat(free)),
  /* after finding a free seat */
  out(table(Id),seat(Fil)).
  /* write the seat as taken */

  table(Id) : delete(Fil) :- | table(Id) : in(table(Id),seat(Fil)),
  out(table(Id),seat(free)).
  /* make the seat as free */
  /* creation of the table: the free seats and forks are kept in the table state */
  table(Id) : new :- out(table(Id),[fork,fork,fork,fork,fork,
    seat(free),seat(free),
    seat(free),seat(free),
    seat(free)]) | table(Id).
}

The simulation of each philosopher is described as usual: to take the forks, to eat and to release the forks. The philosopher's simulation is described for a finite number of eating actions, which must be specified by the interface predicate "eat". The module which defines the philosopher's knowledge, and the corresponding agent is described as follows:

module Knowledge_philosopher{
  simulate(0).
  simulate(N) :- see(table(T)),
    /* try to take the forks */
    in(table(T),[fork,fork]),
    write(Id,'-> begins eating'),
    nl, N1 is N-1,
    /* free the forks */
    out(table(T),[fork,fork]),
    write(Id,'-> finished eating'),
    nl,simulate(N1).
}

agent philosopher{
  context(Knowledge_philosopher).
  interface(eat(N)).

  philosopher(Id) : eat(N) :-
    /* arrival */
    table(_)<<add(philosopher(Id)),
    /* eat */
    simulate(N),
    /* lock for his table */
    see(table(T)),

  /* remaining actions */
}
4 Implementation

The implementation of GroupLog relies upon a distributed extension to Prolog, called PVM-Prolog [Cunha and Marques, 1997]. PVM-Prolog provides the following types of functionalities, which are accessed through a set of system predicates:

- Support of an interface to the PVM [Geist, 1994] system from a Prolog process. This allows process creation, grouping and destruction, and interprocess communication based on message passing where messages are interpreted as Prolog terms.

- Support of a multithreaded model within each Prolog process. Each thread executes within a PVM-Prolog process which is a PVM task running an instance of a modified Prolog abstract machine. All the threads within a process have access to the same logic program but they can evaluate independent concurrent goals. Thread scheduling is handled by the modified Prolog machine. Threads in the same process communicate using term queues, an efficient mechanism which exploits the shared address space of the enclosing process. Threads in distinct processes can communicate using the PVM-Prolog interface predicates for message passing.

The implementation of GroupLog is responsible for the mapping of the language constructs and concepts (agent and groups) onto the mentioned PVM-Prolog mechanisms. Such mapping is written in PVM-Prolog. PVM-Prolog runs on any parallel and distributed platform that supports the PVM system. The Prolog interface has been implemented on several Prolog systems but the multithreaded component of the model requires a specific abstract machine that was modified to handle internal concurrency to each process.

5 Related Work

Recently, several models have been proposed based on coordination concepts. They aim at integrating a number of components together such that the collective set forms a single application that can take advantage of distributed systems.

Many proposals extend a base language for concurrency, communication and nondeterminism. The base language may be Horn Clause Logic [Lloyd, 1987], Temporal Logic, Linear Logic [Girard, 1987] or Situation Calculus. In the first case, we have Rose [Brogl, 1990], Delta Prolog [Cunha et al., 1989], MultiProlog [Boschere, 1989]. In the second, MetaTem [Fisher, 1995]. In the third, COOL [Castellani and Ciancarini, 1994] and IAM [Andreoli et al., 1993] and in the last case ConGolog [Lesprance et al., 1996]. Specification of concurrency has also been introduced jointly with an object-oriented model such as in DLP [Eliens, 1992], ShaDE [Castellani et al., 1996], IAM [Andreoli et al., 1993] and COOL [Castellani and Ciancarini, 1994]. The main motivation to use EHC as the base language for GroupLog is its rigorously defined semantics for the interaction of a process with its environment.

The dynamic entities of a program can be modeled by: Processes, as April [MCabe and Clark, 1995] and MultiProlog [Boschere, 1989]; Objects, as ShaDe [Castellani et al., 1996], Law-Governed

The interaction between dynamic entities can be modeled by: Sending messages, as ShaDe [Castellani et al., 1996], ConGolog [Lesprance et al., 1996], Concurrent Aggregate [Chien, 1993], IAM [Andreoli et al., 1993], AgentSpeak [Weerasooriya et al., 1995], COOL [Castellani and Ciancarini, 1994], MetaTem [Fisher, 1995], April [MCabe and Clark, 1995], Placa [Thomas, 1993] and Electra [Maffeis, 1996]; Shared tuples, as GammaLog [Ciancarini et al., 1996], PoliS [Ciancarini and Mascolo, 1996], Law-Governed Linda [Minsky and Leichter, 1995], MultiProlog [Boschere, 1989], ESP [Gancarini, 1994] and LuCe [Denti and Omicini, 1999].

L1 vs others models In L1, we structured the concurrency and communication with the agent notion, but this language does not allow to provide a theory to model the mental state of an agent, as in MetaTem, ConGOLOG, AgentSpeak, 3APL and Placa. The agent behavior is only dependent on the interface predicates and its configuration, i.e. the entities are reactive and act in accordance with the interaction and its configuration, like in the actor model. This behavior is modeled by EHC clauses, with a very similar interpretation to the rule based one in AgentSpeak and 3APL. In L1, one simple form of communication is allowed: the explicit invocation of interface predicates. The notion of agent in L1 integrates the logic aspect with the object oriented model, as in Ciampolini et al., 1997.

L2 vs others models The definition of groups in GroupLog was the result of an incremental development process which started with our early experimentation with the ISIS system [Birman and Joseph, 1987]. Groups allow the modeling of a cooperating entity and the dynamic evolution of a system. A group can be created or destroyed, as its members can join or leave the group at any time. The group members are hidden from the outside environment. It is possible to have a group as a member of another group, so this allows the composition of the group notion, as the context notion defined in Buffo and Buchs, 1997. In a group we allow two forms of communication: by invocation of interface predicates or through the shared group state. So, L2 is also an experiment towards unifying distributed-memory (remote predicate call) and shared-memory models (shared data). In some of programming languages, like MetaTem [Fisher, 1995], COOL [Castellani and Ciancarini, 1994] and Concurrent Aggregates [Chien, 1993], the group notion is used to restrict the communication to a certain group of agents, which may alleviate some of the inefficiencies that occur in full broadcasting. In other languages, like Electra [Maffeis, 1996], Emerald [Pardyak and Bershad, 1994], Synchronizers [Papadopoulos and Ardeb, 1998] and ColaS [Cruz and Ducasse, 1999], the group is seen as a logical unit that manipulates and restricts the invocation of the members group interface. In Synchronizers, the notion of coordination is modeled by a special object (synchronizer) that restricts the invocation of the group of objects using constraints. In most of these programming languages, as in GroupLog, the group is a dynamic entity. But in Synchronizers and ColaS it is possible to dynamically change the coordination behavior, which is not possible in GroupLog. In GroupLog, as in Electra and Emerald, the members of the group perceive a consistent view of: (1) the other agents who are also part of the group and (2) the shared state. The main difference between GroupLog and these languages is the group interface predicates, that may be distinct from the individual interface of the group members. In languages where the communication is modeled by shared memory, like ESP [Gancarini, 1994] and PoliS [Ciancarini and Mascolo, 1996], the coordination between agents is done by allowing multi-tuple spaces. In the case of LuCe [Denti and Omicini, 1999], the coordination between agents is based on the notion of a logic tuple center, which is very similar to the group notion in the sense that it is possible to model the coordination laws of its members. The L2 level of GroupLog supports the structuring of the tuple space into multiple parts, as each is naturally encapsulated within a specific group such that only its members are allowed to access the group state. This is a good approach to address both modularity, information hiding, and scalability concerns in large scale real applications.
6 Conclusions and Future Work

GroupLog is a programming language allowing clear specification of systems consisting of multiple cooperating entities. It can be used to implement multi-agent and open systems, as it allows: (1) the logical specification of the agent rules, (2) communication and coordination among agents, and (3) a system to be completely dynamic and open, in that new groups and agents may appear/disappear at any time. The main contribution of GroupLog is the notion of group: (1) to enable communication in a structured agent space, and (2) to define the coordination rules of cooperating agents (the group members). Ongoing work is related to the formal definition of the semantics and a distributed implementation of GroupLog.

Acknowledgments

Sincere thanks are due to Andrea Omicini and George Papadopoulos for their support and comments on various versions of this paper. We also would like to thank the comments and suggestions for improvement made by the anonymous referees. This work was partially supported by the Portuguese SETNA project (PRAXIS Contract 2-2.1-TIT-1557-95).

References


