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

We describe a language providing concepts for modeling autonomous entities (agents) and co-operating entities (groups). Agents are the execution units of a GroupLog program. Each agent possesses a hidden internal behavior and a well-defined communications interface and its behavior is defined by a set of logical rules. Groups are important to model agent 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 several 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 are not able to interact with one another no global behaviour in multi-agent system is possible. So, an agent should be able to: (i) communicate with other agents and (ii) cooperate with other agents. And the need to communicate and cooperate leads to the need for coordinating the activities in a multi-agent system and to provide the ability to reuse descriptions of coordination mechanism. The coordination in a multi-agent system has two levels, related to: (1) the organization of multi-agent system and the interactions of agents and (2) the coordination functionality of the collective agents, which models the social behaviours. These two levels of coordination is presented in [7] as objective and subjective coordination, respectively. In GroupLog we have two main program structuring levels: agents (program entities) and groups (cooperating entities). The group notion is used to support the structuring and the management of multiple cooperating entities. Recently such abstractions appeared at the operating system or at the middleware levels [2], but only few proposals attempted their integration into programming languages [28, 27, 18, 7, 31, 30]. Groups support the organization of the agents space into distinct sets, each set acting like a meta-agent with well-defined internal behavior and communications interface. So, the group notion can be used to model the social behaviours of its members (cooperated entities). GroupLog is a logic programming model, that support the engineering of multi-agent systems concerning the specification of: (1) a single agent, (2) inter-agent communication and (3) a cooperating entity. Furthermore, the logic based approach finds applications in multiple areas, where there is the need of some inference capability (modeled by agents in GroupLog), coupled with a requirement for an adequate model for specifying the coordination between autonomous agents (modeled by groups).

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 "Philosopher Table". In 4, we discuss GroupLog in relation to other languages and in 5 we present some conclusions.
2 The Language GroupLog

GroupLog defines extensions to the Extended Horn Clause language (EHC) [3], that are supported at two levels: L1, defines agents as program units 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 (see [3]), 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 an 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). The execution of that object instance is modeled by a perpetual process and two cases are allowed to process message “mess(M)”: (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 “process(M)”. In both cases the object state changes from S to NewS.

EHC was used as the base language mainly because of this elegant interpretation of process interactions and because the language has rigorously defined semantics.

A EHC clause has the following form:

\[ H_1 :: \ldots :: H_n \rightarrow G \mid G_1 :: \ldots :: G_n \]

where \( H_i \) are atoms and \( G \) and \( G_i \) are goals. The goals are conjunctions of atoms built by using the following operators: \( \rightarrow \) denotes sequential composition, \( \mid \) denotes parallel composition and \& denotes simultaneous reduction of atoms.

A clause as follows can be used for the joint synchronization of two concurrent goals \( C_1 \) and \( C_2 \):

\[ (1) \ H_1 :: H_2 \rightarrow G \mid G_1 :: G_2 \]

On a concurrent execution, each goal invocation suspends and waits for the other and only then \( C_1 \) and \( C_2 \) may be simultaneously reduced to \( G_1 \) and \( G_2 \). This joint derivation step is only successful if there is a permutation of \( C_1 \) and \( C_2 \) that unifies with \( (H_1,H_2) \) with most general unifier \( \theta \) and if \( G\theta \) is true. In that case, each ith element of the permutation is reduced to \( G_i\theta \) (i=1,2).

Parallel goal composition, \( G_1 \mid G_2 \), may be achieved in two ways:

(a) Using two Horn Clauses

\[ H_i :: G_i \text{ and } H_j :: G_j \text{ where } G_1 \text{ unifies with } H_i \text{ and } G_2 \text{ unifies with } H_j \]

(b) Using one Extended Horn Clause

\[ H_1 :: H_2 \rightarrow G \mid G_1 :: G_2 \text{, where a permutation of } (G_1, G_2) \text{ unifies with } (H_1, H_2) \text{, as explained above.} \]

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 [10] and EHC clauses of form (1). Syntactically we use \( , \) instead of \( ; \) for sequential composition and the form below, where \( , \) stands for \( ; \):

\[ H_1 \rightarrow H_2 \rightarrow G \mid G_1 \rightarrow G_2 \]
2.2 $L_1$ - Dynamic structuring units of program entities - Agents

The need for structuring communication and synchronization in EHC motivated the definition of a dynamic entity, called agent. Syntactically, an agent is defined as follows:

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

The following example defines a stack agent:

```java
agent stack{
    context().
    interface(push(X), pop(X), top(X)).
    /* interface clauses*/
    stack(1, L) : push(X) :- | stack(1, [X|L]).
    stack(1, [X|L]) : pop(X) :- | stack(1, L).
    stack(1, [X|L]) : top(X) :- | stack(1, L).
    /* creation clause */
    stack(1) : new :- | stack(1, []).
}
```

**Agent name and instances** Each agent has a name that defines its type. An instance of an agent has an unique identifier (e.g. stack(1) instance 1 of agent of type “stack”) and is created when the pre-defined predicate `create_instance(<agent_instance>)` is invoked. Each agent type has an associated prototype that is responsible for the creation of its instances. The implicit definition of the agent prototype is given by an EHC clause, e.g. for the `stack` agent:

(a) `stack : create_instance(stack(1)) :-`  
```plaintext
| stack # stack(1) : stack(1)<<new.
```

where: `stack` is the prototype associated with the type of agent “stack”, `#` is the operator for parallel composition of agent instances and `<<` is the operator for communication with an agent instance. 

When a program starts, the prototypes for all types of agents defined by the program are implicitly activated, namely for agent type `stack`, in the above example. When the `create_instance(stack(1))` goal is invoked by another agent’s instance, let us say `b(2)`, the EHC resolution mechanism tries to unify the pair `(stack,create_instance(stack(1)))` with the head of clause (a). As a result of this successful unification:

- (i) the goal `stack` in the pair reduces to `stack#stack(1)`. This is the parallel composition of prototype `stack` and the new instance `stack(1)`;
- (ii) the `create_instance(stack(1))` goal invoked by `b(2)` reduces to `stack(1)<<new`. This is the invocation of interface predicate `new` in agent instance `stack(1)`.

---

1In this way, we are able to model the creation and activation of agent instances in the framework of the EHC model.
The new instance \( stack(I) \) is represented by a perpetual process, successively evaluating calls to a recursively defined procedure with arguments representing the successive states (or configurations) of the agent. The creation clause of an agent is as follows:

\[
\text{a(\text{Id}) : new :- \langle \text{initial\_conditions} \rangle} \\
| \langle \text{initial\_configuration} \rangle .
\]

When this clause is activated, the perpetual process \( a(\text{Id}) \) reduces to \( \langle \text{initial\_configuration} \rangle \), if \( \langle \text{initial\_conditions} \rangle \) is true. At each point in a computation, the agent instance has a current configuration that is defined by its perpetual process. In the example of \( stack(I) \), the initial configuration is given by the empty list \( stack(I, []) \) (see clause (3) in the \( stack \) agent).

### Agent program

The clause context of a given agent is a set of clauses, defined by:

\[
\text{module <name\_module> \{ 
  clause 1
  ...
  clause n 
\}}
\]

where clause \( i \) is an EHC with one or two atoms in the head of the clause, as explained. The clause context is specified by the predefined predicate \( \text{context(\langle list\_modules \rangle)} \) where \( \langle list\_modules \rangle \) is a list of module names. Its meaning is the union of the clauses defined in the listed modules, in their textual ordering. In the example of "stack" agent, the clause context is empty (see clause (1) of \( stack \) agent).

### Agent interactions

The communication context defines the interaction of the agent with its environment and is specified by \( \text{interface(\langle list\_interf\_pred\_names \rangle)} \) where \( \langle list\_interf\_pred\_names \rangle \) is a list of interface predicate names and \( \text{interface} \) 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, these are: \( \text{push(X)} \), \( \text{pop(X)} \) and \( \text{top(X)} \) (see clause (2) of \( stack \) agent).

### Communication forms

The communication with an agent uses operator "\(<<\)" and is synchronous. This can be one-to-one or one-to-many. In one-to-one communication, the destination agent and its interface predicate are explicitly cited. For example \( stack(1)::<\text{pop(X)} \) denotes a communication with agent’s instance \( stack(1) \) through interface predicate “pop”. In one-to-many communication, we only need to specify the type of agent and the interface predicate, because this is a form of selective communication which addresses all instances of the specified type. For example \( stack(_):<\text{pop(X)} \) 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 \( \langle \text{process} \rangle \) is true:

(i) \( \langle \text{agent\_config} \rangle : <\text{interf\_pred\_name}> :- \\
    <\text{set\_cond}> | <\text{new\_agent\_config}> : \langle \text{process} \rangle .

(ii) \( \langle \text{agent\_config} \rangle : <\text{interf\_pred\_name}> :- \\
    <\text{set\_cond}> | <\text{new\_agent\_config}> .

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

1. \( H_1 \) corresponds to the \( <\text{agent\_config}> \) of the addressed agent. This means that this agent instance must have a current configuration that unifies with \( <\text{agent\_config}> \). This current configuration is modeled through the agent’s perpetual process.
2. $H_2$ corresponds to the `<interface predicate>` in the interface predicate that must unify with the interface predicate that is cited by the invoker agent.

3. $G$ corresponds to `<set condition>` that must be true so that the communication can be successful.

4. $G_1$ corresponds to the `<new agent config>` for the addressed agent.

5. In form (i) $G_2$ corresponds to `<process>` and has a similar interpretation as "process(M)" in the communication with an object in EHC (see section on EHC).

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 atomic step, i.e. is available again for further interactions. If the first three conditions are satisfied, the agent changes to a new configuration, otherwise it keeps its current configuration and the communication fails in the invoker.

Given the actual configuration of an agent and an invoked interface predicate there is a (possibly empty) set of clauses which are enabled by that configuration. These are the clauses of that interface predicate such that `<agent config>` unifies with the current agent configuration of the agent. All other clauses are disabled for that configuration. An example is given by the clause with head "stack([Id, X|L]; pop(X))" when stack agent is in its initial configuration (stack(1,[])). This means an invocation of "stack(1); pop(C)" will suspend the invoker agent until a matching configuration (triggered by a change to a nonempty stack) enables the cited clause. If there is a matching enabled interface clause, and if "set condition" is true, the communication is accepted by the invoked agent so it changes its current configuration to `<new agent config>`. This state 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 goal `<process>` locally to the invoked agent i.e. within this agent context (this communication is said to be ongoing). The evaluation of `<process>` cannot involve communication with other agents and 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 `<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. The behavior of an agent is modeled in a logic framework given by EHC, except the meaning of the interactions\(^2\). Otherwise we would have to manage the complexity of distributed backtracking which would affect the practical feasibility of the model.

One-to-many communication, as in `stack(\_); pop(X)`, proceeds in two steps:

- 1. The communication suspends until there is an available instance of agent `stack` (i.e. that instance is currently waiting for interactions) with an enabled interface clause for this invocation of `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 such instances of "stack" are found, or there is no interface predicate "pop" in agent `stack`, or when the one-to-one communication fails in the selected instance of agent `stack`.

\(^2\)The formal semantics of level L1 of GroupLog is already defined, but its presentation is beyond the scope of this paper.
**Agent termination** An instance of an agent can be cancelled by the predefined interface predicate *kill*, or implicitly cancelled by an interface clause such as:

\[
\langle \text{current_config} \rangle : \langle \text{interf_pred_name} \rangle := \\
\langle \text{set_cond} \rangle \mid \text{true} : \text{true}.
\]

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

\[
\langle \text{current_config} \rangle : \langle \text{interf_pred_name} \rangle.
\]

### 2.3 L₂ - Dynamic grouping of agents - Group

The need for structuring the space of agents in L₁ and supporting its cooperation motivated the definition of groups as dynamic entities. Two forms of cooperation are allowed in L₂: (1) access to a shared group state, based on the Linda model [4]; and (2) direct communication through interface predicates. The group, like a meta-agent, has well defined clause, communication and behavior contexts. Its creation is done by the predicate *create_instance* and there is a perpetual process, called the group representative, that models the successive configurations of the group and the associated transitions. Syntactically, a group is defined as follows:

```prolog
group <name_group>{
    context(<list_modules>).
    interface(<list_interface_pred_names>).
    <list of interface clauses>
    <the creation clause>
}
```

The following example defines a group “meet_sch”, which has an interface predicate (*begin*), to simulate the beginning of the Meeting Scheduling (in section 4):

```prolog
group meet_sch{
    context().
    interface(begin).
    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).
    meet_sch(Id) : new :- | meet_sch(Id).
}
```

**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”. For example

- **friends(1)**<<**add**(F) adds agent F to an instance of group **friends**(1) and
- **friends(1)**<<**delete**(F) removes agent F from that instance **friends**(1). 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:

  - **my_group(<group_list>)**
    - where <group_list> returns the list of groups containing the invoking agent;
• see(<a-group>)
  is true if <a-group> contains the invoking agent;

• members(<a-group>,<members_list>)
  where <members_list> returns the list of members in <a-group>;

• see_member(<a-group>,<a-member>)
  is true if <a-member> belongs to <a-group>.

Structuring the space of agents  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-many 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.

Group communication  The interface clauses have the following form:

(i)  <group_config> : <interf_pred_name> :-
     <set_cond> | <new_group_config> : <process>.
(ii) <group_config> : <interf_pred_name> :-
     <set_cond> | <new_group_config>.

In these clauses the “set_cond” and “process” goals are evaluated by a thread of the group representa-
tive in the group clause context. In these goals we allow the invocation of: (1) predicates defined
by the group clause context, (2) predicates to access the shared group state or (3) interface predicates
of group members or other agents. Unlike in an agent, communication is allowed in goal <process>.
Communication within the group may involve only group members, or the group representative and
the 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 friends(1), the goal
friends(1)<state(m1) creates the shared group state, based on the contents of module 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 - rd, ts, in e out allow access to the state of a group G:

1. ts(G,State_group) : is true if there is a subset of the shared state that matches “State_group”;
2. rd(G,State_group) : reads a subset from the shared state that matches “State_group”;
3. in(G,State_group) : reads a subset from the shared state matching “State_group” and removes
   it;
4. 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 “S-
tate_group”. The predicates ts and out are non-blocking. Namely, ts fails if no matching is possible.
The modifications made to the shared state, due to these predicates, are not reversible.
Group termination The elimination of a group is achieved through the interface predicate “kill”, or implicitly through the activation of a clause like previously presented in \( L_1 \). As soon as the termination of a group is activated, its interface predicates disappear except for the “delete” predicate. This is so that the current group members may leave the group. When there are no more members, all pending and ongoing communications are forced to fail, and the group is removed from all its enclosing groups, and from the universe \( U \).

3 Examples

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

- (1) communication and coordination among related agents;
- (2) dynamic configuration of the system.

In both examples presented in this section, “Two-Agent Meeting Scheduling” and "The Dynamic Philosophers’ Table", the need to coordinate a set of related agents motivate the definition of a group, 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, "The Dynamic Philosophers’ Table", we can see how the group notion is more than a composition of agents. In this case, the group has a behavior itself, that define the coordination rules of its members \(^3\), as an agent society. Other typical example of a Philosophers' Restaurant can be seen in [33], where the group notion is 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 of N agents. This meeting time must be accepted by all agents. The basic idea is to allow agents coordinate themselves in order to reach an agreement. The version below (see also in [32]), for simplicity, considers the meeting of two agents only - Two-Agents Meeting Scheduling. In this version, one of the agents acts as the host (it is the agent who is trying to arrange the meeting) and the other agent is the guest. To guarantee that the cooperation will reach a conclusion, we assume that the agents always share a free slot of the appropriate length in their agenda where the meeting can be scheduled. The group “meet\_sch”, as defined in section 2.3, is used to join the agents in such way that they can cooperate through the shared group state. In our example, the two agents have some common knowledge related to: (1) finding a possible time for the meeting in accordance with his agenda and (2) telling a possible time to the cooperative agents. 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)),
        out(meet\_sch(1),tell(I,meet(MeetId,T))),
        ts(meet\_sch(1),stop\_sch(I)).
    tell_time(I,\_,\_) :- in(meet\_sch(1),stop\_sch(I)).
)\(^3\)The behavior related to the arrival and the exit of philosophers from the table
```

\(^3\)The behavior related to the arrival and the exit of philosophers from the table
The cooperative 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 the confirmation of the time (in the case of the guest). This knowledge is defined in the following modules:

```prolog
module sched_host{
  ex_host(I, Guest, S, meet(MeetId, T)) :-
    rd(meet_sch(1), inf_meet(MeetId, T)),
    tell_time(I, S, meet(MeetId))
    || match_time(I, Guest, MeetId, T).

  match_time(I, Guest, MeetId, T) :-
    in(meet_sch(1), [tell(I, meet(MeetId, T)),
                     tell(Guest, meet(MeetId, T))]),
    out(meet_sch(1), [stop_sch(I),
                     confirm(I, meet(MeetId, T))]).
}

module sched_guest{
  ex_guest(I, Host, S, meet(MeetId, T)) :-
    rd(meet_sch(1), inf_meet(MeetId, T)),
    tell_time(I, S, meet(MeetId))
    || conf_inv(I, Host, MeetId, T).

  conf_inv(I, Host, MeetId, T) :-
    in(meet_sch(1), confirm(Host, meet(MeetId, T))),
    out(meet_sch(1), stop_sch(I)).
}
```

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

```prolog
agent host{
  context(schedule, sched_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(1)<>add(host(Id))
    | host(Id, []).
}

agent guest{
  context(schedule, sched_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(1)<>add(guest(Id))
    | guest(Id, []).
}
```

In this example, we can see how GroupLog concepts are used to coordinate a set of related agents.
3.2 The Dynamic Philosophers’ Table

In the classical version of this example there are five philosophers; each philosopher is seating in the table and has two forks by its sides. In our versions, we assume that forks are in the middle of the table, i.e. they are not associated to the seats.

To model the coordination of the philosophers we have a group - table. Here the concept of group is used to:

- join the philosophers and allows to share a state (forks and seats);
- model the behavior of the table, concern the arrival and the exit of each philosopher.

The group table is defined as follows:

```plaintext
group table{
    context().
    interface().

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

    table(Id) : delete(Fil) :- | table(Id):
        /* make the seat as free */
        in(table(Id), seat(Fil)),
        out(table(Id), seat(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: he takes the forks, eats and releases 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 agent philosopher are described in the following way:

```plaintext
module Knowledge_philosopher{
    simulate(0).
    simulate(N) :- see(table(T)),
    /* try to take the forks */
    in(table(T), [fork, fork]),
    write(Id, ’’-> begins to eat’’),
    nl, N1 is N-1,
    /* free the forks */
    out(table(T), [fork, fork]),
    write(Id, ’’-> finished to eat’’), nl,
    simulate(N1).
}
```
agent philosopher{
  context(Knowledge_philosopher).
  interface(eat(X)).

  philosopher(Id) : eat(N_eatings) :-
    /* entry in one table */
    table(_,_)<<add(philosopher(Id)),
    /* eat */
    simulate(N_eatings),
    /* look for his table */
    see(table(T)),
    /* exit the table */
    table(T)<<delete(philosopher(Id)),
    write(Id,"' -> leaves the table'"
         | philosopher(Id).
}

4 Related Work

Recently, models have been proposed based on coordination concepts aiming 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 logic language for concurrency, communication and non-determinism. The base language may be Horn Clause Logic [10], Temporal Logic, Linear Logic [26] or Situation Calculus. In the first case, we have Rose [11], DeltaProlog [12], MultiProlog [21]. In the second, MetaTem [7]. In the third, COOL [25] and IAM [17] and in the last case ConGolog [9]. Specification of concurrency has also been introduced jointly with an object-oriented model such as in DLP [13], CSo-Prolog [14], ShaDE [23], IAM [17] and COOL [25].

The motivation to use EHC as the base language for GroupLog is due to its elegant interpretation of a process interaction and its rigorously defined semantics. The dynamic entities of a program can be modeled by:

Processes, as April [1] and MultiProlog [21];

Objects, as ShaDe [23], Law-Governed Linda [22], IAM [17], ColoS [31], Electra [27] and Emerald [28];

Agents, as ConGolog [9], COOL [25], MetaTem [7], Agentspeak [8], 3APL [36] and Placa [6];

Actors, as Concurrent Aggregate [16] and Synchronizers [20].

The interaction between dynamic entities can be modeled by:

Sending messages, as ShaDe [23], ConGolog [9], Concurrent Aggregate [16], IAM [17], AgentSpeak [8], COOL [25], MetaTem [7], April [1], Placa [6] and Electra [27];

Shared tuples, as GammaLog [19], PoliS [24], Law-Governed Linda [22], MultiProlog [21], ESP [5] and LuCe []

L₁ vs others models In L₁, we structured the concurrency and communication with the agent notion, but this language does not aim 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 APL. In L₁, one simple form of communication is allowed: the explicitly invocation of interface predicates. The notion of agent in L₁ integrates the logic aspect with the object-oriented model, as in [29]. In this model it is possible to model blackboard-based systems, which is only allowed in GroupLog in L₂.

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 [2]. Groups allow the modeling of a cooperative 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. Because a group is a meta-agent, 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 [30]. 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 [7], COOL [25] and Concurrent Aggregates [16], 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 [27], EmeraK [28], Synchronizers [20] and CooS [31], the group is seen as a logical unit that manipulates and restricts the invocation of the members group interface. In Synronisers, the notion of coordination is modeled by a special object (synchroniser) 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 CooS it is possible to dynamically change the coordination behavior, which is not possible in GroupLog. In GroupLog, as in Electra and EmeraK, 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 group members individual interface. In languages where the communication is modeled by shared-memory, like ESP [5] and Polis [24], the coordination between agents is done by allowing multi-tuple spaces. In the case of LuCe [4], the coordination between agents is based on the notion of logic tuple centre, 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 tuples 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.

5 Conclusions and Future Work

GroupLog is a programming language allowing clear specification of systems supporting the concept 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 to: (1) enable communication and (2) define the coordination rules of related agents (its members). Ongoing work is related to the formal definition of the semantics and a distributed implementation of GroupLog on top of the PVM system [35].

References


[18] A. A. Holzbacher. A Software Environment for Concurrent Coordinated Programming In Coordination Languages and Models, First Int. Conf. COORDINATION96, Italy, April 1996.


