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.fc.ul.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 the agents 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 behavior in a multi-agent system is possible. So, an agent should be able to communicate and to cooperate with other agents. These needs require a model for coordinating the activities in a multi-agent system.

The coordination aspect 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 [35] 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 [2], but only few proposals attempted its integration into programming languages [27, 26, 18, 7, 30, 29].

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 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 [9], a logic programming model which is amenable to an object based interpretation. This allows the definition of 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 dy-
namically enter and leave such cooperating groups. On
the other hand, each group can be seen as acting like
a meta agent, with well defined internal behavior and
communications 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". 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 de-
dines groups of agents. A GroupLog system contains
concurrently executing agents able to: (1) communi-
cicate 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 concur-
rency and synchronous communication. EHC supports
the parallel composition of goals and provides clauses
for modeling an interaction between two parallel pro-
cesses through a joint synchronization step.

Such EHC clauses allow an interpretation of message
based communication with an object, when some pro-
cess invokes method "mess(M)" of an object instance
obj(S). The execution of that object instance is modi-
bled 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
case (i) the message is consumed. In case (ii) the
message is not consumed and is reduced to "process(M)"
In both cases the object state changes from S to NewS.

A EHC clause has the following form:
H1 :: ... :: Hn :| G | G1 :: ... :: Gn
where Hi are atoms and G and Gi are goals. The goals
are conjunctions of atoms built by using the following
directories: ":|" denotes sequential composition, ":|:" de-
notes parallel composition and ":&:" denotes simultane-
ous reduction of atoms.

A clause as follows can be used for the joint syn-
chronization of two concurrent goals C1 and C2

(1) H1 :: H2 :- G | G1 :: G2
On a concurrent execution, each goal invocation sus-
pends 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 per-
mutation of C1 and C2 that unifies with (H1,H2) 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, G1\|G2, may be achieved
in two ways:
(a) Using two Horn Clauses
H1 :: G1 and H2 :: G2 where G1 unifies with H1 and G2
unifies with H2
(b) Using one Extended Horn Clause
H1 :: H2 :- G | G1 :: G2, where a permutation of (G1,G2)
unifies with (H1, H2), as explained above.

Use of Extended Horn Clauses in GroupLog In Grou-
pLog we consider only Extended Horn Clauses with one
or two atoms in the clause head corresponding, respect-
ively, to Horn Clauses [10] and EHC clauses of form
(1). Syntactically we use ":|:" instead of ":|:" for sequen-
tial composition and the form below, where ":|:" stands
for ":|:
H1 :: H2 :- G | G1 :: G2

2.2 L1 - Dynamic structuring units of program en-
tities - Agents

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

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:

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) :-
}
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 predi-
cate create_instance(<agent_instance>) is invoked.Each agent type has an associated prototype that is re-
sponsible for the creation of its instances. The implicit
definition of the agent prototype is given by an EHC
clause, eg. for the stack agent:

(a) stack : create_instance(stack(Id)) :-
| stack # stack(Id) : stack(Id) <<new.

where: stack is the prototype associated with the type
of agent “stack”, # is the operator for parallel com-
position 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 ac-
vated, namely for agent type stack, in the above ex-
ample. 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)\(^1\);

(ii) the create_instance(stack(1)) goal invoked by
b(2) reduces to stack(1) <<new. This is the invo-
cation of interface predicate new in agent instance
stack(1).

The new instance stack(1) is represented by a per-
petual process, successively evaluating calls to a recur-
sively defined procedure with arguments representing
the successive states (or configurations) of the agent.
The creation clause of an agent is as follows:

a(Id) : new :- <initial_cond> | <initial_conf>.

When this clause is activated, the perpetual process
a(Id) reduces to <initial_conf>, if <initial_cond> is
ture. At each point in a computation, the agent in-
stance has a current configuration that is defined by its
perpetual process. In the example of stack(1), the
initial configuration is given by the empty list stack(1,[])
(see clause (3) in the stack agent).

\(^1\)In this way, we are able to model the creation and activation of
agent instances in the framework of the EHC model.

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

```plaintext
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
context(<list_modules>) where <list_modules> 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 de-
defines the interaction of the agent with its environment
and is specified by

```plaintext
interface(<list_intf_pred_names>)
```

where <list_intf_pred_names> is a list of interface predi-
cates and interface is a predefined predicate. The
interface predicates define the visible “entry points” of
an agent that can be invoked from the outside envi-
ronment. In the example of “stack” agent, these are:
push(X), pop(X) and top(X) (see clause (2) of stack
agent).

Communication forms The communication with an
agent uses operator “<<” and is synchronous. This can be
to one or one to many. In one to one commu-
nication, the destination agent and its interface predi-
cate are explicitly cited. For example

```plaintext
stack(1) <<<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 ad-
dresses all instances of the specified type. For example

```plaintext
stack(_) <<<pop(X) denotes a communication addressing
all instances of an agent of type stack through inter-
faced 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 in-
terface predicate. Two forms are allowed, where form
(ii) is a simplification of (i) when <process> is true:

(i) <agent_config> : <interf_pred_name> :-
    <set_cond>
    | <new_agent_config> : <process>.

(ii)<agent_config> : <interf_pred_name> :-
    <set_cond> | <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 \(<agent\_config>\) of the addressed agent. This means that this agent instance must have a current configuration that unifies with \(<agent\_config>\). This current configuration is modeled through the agent’s perpetual process.

2. \(H_2\) 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.

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 “\(\text{stack}(\text{Id}, \{X,L]\); \text{pop}(X)\)” when stack agent is in its initial configuration (\(\text{stack}(1,\{\})\)). This means an invocation of “\(\text{stack}(1)\text{;pop}(C)\)” will suspend the invoker agent until a matching configuration (triggered by a change in to a nonempty stack) enables the cited clause. If there is a matching enabled interface clause, and if “\(\text{set\_cond}\)” 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 \(\text{stack}(\_); <\text{pop}(X)\), proceeds in two steps:

1. The communication suspends until there is an available instance of agent \(\text{stack}\) (i.e., that instance is currently waiting for interactions) with an enabled interface clause for this invocation of \(\text{pop}\);

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 \(\text{stack}\), or when the one to one communication fails in the selected instance of agent \(\text{stack}\).

**Agent termination** An instance of an agent can be canceled by the predefined interface predicate \(\text{kill}\), or implicitly canceled by an interface clause such as:

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

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:

\[
\text{<current\_config> : <interf\_pred\_name>}.\]

2.3 \(L_2\) - Dynamic grouping of agents - Group

The need for structuring the space of agents in \(L_1\) and supporting its cooperation motivated the definition of groups as dynamic and cooperating entities. Two forms of cooperation are allowed in \(L_2\): (1) access to a shared

\(^2\)The formal semantics of level \(L_1\) of GroupLog is already defined, but its presentation is beyond the scope of this paper.
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:

```
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 3.1):

```
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 `meet_sch(1)\(<\langle add(F)\rangle\)` adds agent $F$ to an instance of group `meet_sch(1)` and `meet_sch(1)\(<\langle delete(F)\rangle\)` removes agent $F$ from that instance `meet_sch(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 representative 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 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 `meet_sch(1)`, the goal `meet_sch(1)\(<\langle state(m1)\rangle\)` 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` `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. $\text{in}(G, \text{State}_\text{group})$: reads a subset from the shared state matching “State\text{group}” and removes it;
4. $\text{out}(G, \text{State}_\text{group})$: puts the subset “State\text{group}” in the shared state.

In 2 and 3 above, the invocations suspend until there is a subset of the shared state matching “State\text{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.

**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 L1. 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 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, $\text{meet}_\text{sch}$ and $\text{table}$, 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 3, as an agent society. Another typical example of a Philosophers’ Restaurant can be seen in [32], where the group notion is also used to model the dynamic evolution of the system.

3 In this example, this behavior is related to the arrivals and exits of the philosophers

### 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 [31]) is a simplified version which considers the meeting of two agents only - Two-Agent Meeting Scheduling. 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 $\text{meet}_\text{sch}$, as defined in section 2.3, is used to join the agents in such a way that they can cooperate through the shared group state. 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:

```plaintext
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, T)) :-
        found_time(S, meet(MeetId, T)),
        cut(meet_sch(I), tell(I, meet(MeetId, T))),
        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:

```plaintext
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)]),
        cut(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)),
```
The host and the guest are defined as agents in the following way:

```c
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(1)<add(host(Id))
        | host(Id,[[]]).
}
```

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. 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 goals:

- 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 exit.

The group `table` is defined as follows:

```c
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)).
}
```

```c
table(Id) : delete(Fil) :- | table(Id) :
    /* make the seat as free */
    in(table(Id),seat(free)),
    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)]) | 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 corresponding agent are described as follows:

```c
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).
}
```

```c
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)),
        /* exit */
        table(T)<delete(philosopher(Id)),
        write(Id,'-> leaves the table'),
        | philosopher(Id).
    philosopher(Id) : new :-
        | philosopher(Id).
}
```

Here we can see how the group notion is used to model the behavior of the cooperating entities, concern-
ing the arrival of the philosophers in the table.

4 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 logic language for concurrency, communication and non-determinism. The base language may be Horn Clause Logic [10], Temporal Logic, Linear Logic [15] 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 [14], CSO-Prolog [13], 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 with its environment, 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], ColaS [30], Electra [26] and Emerald [27];
Agents, as ConGolog [9], COOL [25], MetaTem [7], Agentspeak [8], 3APL [34], and Placa [6];
Actors, as Concurrent Agregate [16] and Synchronizers [20].

The interaction between dynamic entities can be modeled by:

Sending messages, as ShaDe [23], ConGolog [9], Concurrent Agregate [16], IAM [17], AgentSpeak [8], COOL [25], MetaTem [7], April [1], Placa [6] and Electra [26];
Shared tuples, as GammaLog [19], PoliS [24], Law-Governed Linda [22], MultiProlog [21], ESP [5] and LuCe [36].

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, ConGOLLog, 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 explicitly invocation of interface predicates. The notion of agent in L1 integrates the logic aspect with the object oriented model, as in [28]. 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 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. 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 [29]. 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 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 [26], Emerald [27], Synchronizers [20] and ColaS [30], 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 [5] and PoliS [24], the coordination between agents is done by allowing multi-tuple spaces. In the case of LuCe [36], the coordination between agents is based on the notion of a 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 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.
5 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 on top of the PVM system [33].

References


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


[34] http://www.students.cs.uu.nl/swp/1997/robo/3apl/
