A Language Framework for Group Based Multi-Agent 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 two main program structuring levels: agents and 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 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. In the paper we give an informal presentation of the GroupLog language and illustrate its use through an example.

1 Introduction

In multi-agent systems several agents cooperate to achieve some task that might not otherwise be achieved by each individual agent. Group-oriented abstractions offer the possibility of joining related agents into logical units of cooperation that may help in the management of distributed computations. Nowadays, concurrent programming remains a complex task. This complexity could be reduced by a model supporting concepts for the structuring 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 [30, 29, 20, 7, 33, 32]. GroupLog provides concepts for modeling autonomous entities (agents) and cooperating entities (groups). The definition of groups in GroupLog is important to model agent cooperation in a flexible and well-structured manner and to hide low-level management of coordination activities. GroupLog relies on logic programming abstractions aiming at providing transparency to the user. 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 interactions between autonomous agents (modeled by groups). In the remainder of the paper, we first present a brief description of the GroupLog language and then, in 3 a new version of the Dining Philosophers Problem. In 4, we discuss GroupLog in relation to other languages and in 5 we give some conclusions.

2 The Language GroupLog

GroupLog defines extensions to the Extended Horn Clause language (EHC) [3], that are supported at two levels: L_1 defines agents as program units and L_2 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.

A EHC clause has the following form:

$$H_1 :: \ldots :: H_n : - G | G_1 :: \ldots :: G_m$$

where $H_i$ are atoms and $G$ and $G_i$ 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 "\::\:" is used to specify the joint unification of multiple conjuncts with the multiple atoms in a clause head and their simultaneous reduction to the goals in the clause body.

A clause as follows can be used for the joint synchronization of two concurrent goals $C_1$ and $C_2$

1. $H_1 :: H_2 : - G | G_1 :: G_2$

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, $G_1 || G_2$, may be achieved in two ways:

a) Using two Horn Clauses

$H_i : - G_i$ and $H_j : - G_j$ where $G_1$ unifies with $H_i$ and $G_2$ unifies with $H_j$

b) Using one Extended Horn Clause where "\::\:" is used to specify the synchronization

$H_1 :: H_2 : - G | G_1 :: G_2$, where a permutation of $(G_1,G_2)$ unifies with $(H_1,H_2)$, as explained above.

EHC also allows an interpretation of a message based communication with an object. Two cases are allowed to process message "mess(M)" by an instance of obj(S) whose execution is modeled by a perpetual process:

(i) $\text{obj(S)} :: \text{mess(M)} : - \text{method(M)} | \text{obj(NewS)} :: \text{true}$

(ii) $\text{obj(S)} :: \text{mess(M)} : - \text{method(M)} | \text{obj(NewS)} :: \text{process(M)}$

This is achieved by activating the clause with head $\text{obj(S)} : \text{mess(M)}$ and solving goal "method(M)". 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.

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 "\::\:".

$$H_1 : H_2 : - G | G_1 : G_2$$

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 in the following way:

```plaintext
agent <agent_name>{
  context(<list_modules>), /* clause context */
  interface(<list_interface_pred_names>), /* communication context*/

  <list of interface clauses> /* agent behavior */

  <creation clause> /* agent creation */
}
```
The following example defines a stack agent:

```prolog
agent stack{
  context()
  interface(push(X), pcp(X), tcp(X)).
  /* interface clauses*/
  stack(Id,L) : push(X) :- | stack(Id,[X|L]).
  stack(Id,[X|L]) : pcp(X) :- | stack(Id,L).
  stack(Id,[X|L]) : tcp(X) :- | stack(Id,[X|L]).
  /* creation clause*/
  stack(Id) :- | stack(Id,[]).
}
```

**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)`) 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, eg. for the `stack` agent:

```prolog
stack : create_instance(stack(Id)) :- | stack # stack(Id) : stack(Id)<<new. (2)
```

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 (2). 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)`.

The new instance `stack(1)` 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:

```prolog
a(Id) : new :- <initial_conditions> | <initial_configuration>.
```

When this clause is activated, the perpetual process `a(Id)` reduces to “<initial_configuration>”, if “<initial_conditions>” is true. At each point in a computation, the agent’s instance 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).

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

```prolog
module <name_module>{
  clause 1
  ...
  clause n
}
```

---

1. In this way, we are able to model the creation and activation of agent’s instance in the framework of the EHC model.
where \( \text{clause } i \) 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` by

\[
\text{context}(<\text{list_modules}>)
\]

where \(<\text{list_modules}>\) 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 defines the interaction of the agent with its environment and is specified by `interface(<list_interface_pred_names>)` where \(<\text{list_interface_pred_names}>\) 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, these are: `push(X)`, `pop(X)` and `top(X)` (see clause (2) of stack agent).

### Communication forms

The communication with an agent uses operator “\(<\text{\textless\text\textless}\)” 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/1”. 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/1”. 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 \(<\text{process}>\) is true:

\[
\begin{align*}
\text{(i) } \text{<agent\_configuration>} : & \text{<interface\_pred\_name>} : \neg \text{<set\_conditions>} \\
& \text{<new\_agent\_configuration>} : \text{<process>}. \\
\text{(ii) } \text{<agent\_configuration>} : & \text{<interface\_pred\_name>} : \neg \text{<set\_conditions>} \\
& \text{<new\_agent\_configuration>}. \\
\end{align*}
\]

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\_configuration>` of the addressed agent. This means that this agent instance must have a current configuration that unifies with `<agent\_configuration>`. This current configuration is modeled through the agent’s perpetual process.

2. \(H_2\) corresponds to the `<interface\_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\_conditions>` that must be true so that the communication can be successful.

4. \(G_1\) corresponds to `<new\_agent\_configuration>` 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 \text{enabled} by that configuration. These are the clauses of that interface predicate such that `<agent\_configuration>` 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_conditions" is true, the communication is accepted by the invoked agent 
so it changes its current configuration to <new_agent_configuration>. This state transition is non 
reversible and the agent becomes available for further interactions with other agents. This means that 
in clauses like (ii), 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 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. 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(X)<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 more than one of such instances are found, then 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/1" in agent stack, or the communication fails in the selected instance of agent stack.

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

<current_configuration> : <interface_pred_name> :- <set_conditions> | true : true.

This implies the forced failure of all communications involving this agent (both suspended and ongo- 
ing). If <set_condition> is true, this clause simplifies to:

<current_configuration> : <interface_pred_name>.

2.3 L2 - Dynamic grouping of agents - Group

The need for structuring the space of agents in L1 and supporting their 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 is a meta-agent and a composition of agents sharing the group state and a 
consistent view of the group history. The group, like an 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 in the following way:

2The formal semantics of level L1 of GroupLog is already defined, but its presentation is beyond the scope of this paper.
The following example defines the meeting and the farewell of N friends that joined in a restaurant:

group friends{
  context().
  interface(begin(N,R)).
    /* List of Interface Clauses */
    /* the group begins the meeting phase */
    friends(Id) : begin(N,R) :- out(friends(Id),restaurant(R,N)),
                 friends(Id,wait_go(N)).
    /* the last friend joins the group (begins the farewell phase) */
    friends(Id,wait_go(1)) : add(F) :- in(friends(Id),restaurant(R,N)),
                          R<go_eat(Id,N,T), /* ask for a table T */
                          out(friends(Id),go_eat(R,T)) /* inform the group members */
                          | friends(Id,wait_exit(N)).
    /* a friend joins the group */
    friends(Id,wait_go(N)) : add(F) :- N1 is N-1 | friends(Id,wait_go(N1)).
    /* the last friend leaves the group and the group is destroyed */
    friends(Id,wait_exit(1)) : delete(F).
    /* a friend leaves the group */
    friends(Id,wait_exit(N)) : delete(F) :- N1 is N-1 | friends(Id,wait_exit(N1)).
    /* Creation Clause */
    friends(Id) : new :- | friends(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 group instance friends(1) and friends(1)<delete(F) removes agent F from group 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>.

The structuring of the space of agents In L_1, all agents belong to the universe of agents (U) that models a flat space. The notion of groups in L_2 allows to control the visibility and access to the agents. In L_2, an agent only belongs to U if it is not a member of any group. All agents in U can directly communicate with each 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 he had previously joined. In L_2, 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)  \(<\text{group\_configuration}>\) :  \(<\text{interface\_pred\_name}>\) :- \(<\text{set\_conditions}>\)
    |  \(<\text{new\_group\_configuration}>\) : \(<\text{process}>\).
(ii) \(<\text{group\_configuration}>\) :  \(<\text{interface\_pred\_name}>\) :- \(<\text{set\_conditions}>\)
    |  \(<\text{new\_group\_configuration}>\).

These clauses have the same form as in an agent except concerning the “set\_conditions” and “process” goals, that 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 in an agent, communication is allowed in goal \(<\text{process}>\). 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(ml) creates the shared group state, based on the contents of module ml. 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. 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 “State\_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.

Consistent view of the group membership  The members of a group observe a consistent view of the shared group state and of the group membership by imposing an ordering upon the events occurring in the group thus defines in the group history3.

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 the groups where it belongs, and from the universe U.

3 Dining Philosophers

In the classical version of this example with five philosophers, each one is seating in the table and has two forks by its sides. We assume that forks are in the middle of the table, i.e. they are not associated with the seats. This example extends the problem to a restaurant of philosophers with multiples tables – The Philosophers Restaurant (see also in [27]). We model a group of philosopher (“friends”) that meet in a restaurant. There are the notions of philosopher, table, restaurant and

\footnote{The detailed definition of the group semantics in terms of these events is beyond the scope of this paper.}
friends. The **philosopher** models the behavior of a philosopher. The **table** models the coordination of the philosopher meal. The **restaurant** models the set of tables and the behavior of the waiter. The **friends** models the meeting and farewell of n philosophers.

The behavior of a **philosopher** agent is: (1) meets his friends, (2) waits for all his friends, (3) enters the restaurant and joins a table, (4) takes his meal, (5) says good-bye to his friends and (5) exits the table and restaurant. This behavior is modeled by the interface predicate `go_friend/(F,N_eatings)`, where `F` is the friends identification and `N_eatings` the number of times of each philosopher eats. The **philosopher** definition is the following:

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

    philosopher(Id) : go_friend(F,N_eatings) :-
        F<<add(philosopher(Id)), (1)
        rd(F,go_eat(R,T)), (2)
        R<<add(philosopher(Id)),T<<add(philosopher(Id)), (3)
        simulate(N_eatings), (4)
        F<<delete(philosopher(Id)), (5)
        T<<delete(philosopher(Id)),R<<delete(philosopher(Id)) (6)
        | philosopher(Id).

    philosopher(Id) :- | philosopher(Id).
}
```

In the predicate `simulate(N)`, the table T allows to model the shared resource (forks).

The **table** group is used to gather the philosophers and has three interfaces predicates: **buy(R)** - models a new table in the restaurant R, **make(N)** - models the set of the table for N philosophers and **clean** - models the cleaning of the table. The **table** group is defined in the following way:

```plaintext
module table_knowledge{
    make(Id,0).
    make(Id,N) :- N1 is N-1, make(Id,N1) || out(table(Id),fork).
    clean(Id,0).
    clean(Id,N) :- N1 is N-1, clean(Id,N1) || in(table(Id),fork).
}
group table{
    context(table_knowledge).
    interface(buy(R),make(N),clean).

    table(Id) : buy(R) :- R<<add(table(Id)), out(R, table(Id,free)) | table(Id).
    table(Id) : make(N) :- N > 1, out(table(Id),count(N)), make(Id,N) | table(Id).
    table(Id) : clean :- in(table(Id),count(N)), clean(Id,N) | table(Id).

    table(Id) :- | table(Id).
}
```

The **restaurant** group is keeps the information about the availability of the tables and accepts groups of philosophers. This group has two interface predicates: **go_eat/(F,N,T)** - models the asking for a table T to a group F of N philosophers and **end_eat/(F)** - models the exit from group F. In this group a
new table T is created to model the joining of two tables (see clause (1) below). The definition of the restaurant group is the following:

module waiter{
  waiter_entry(F,N,table(T)) :- see(restaurant(Id)),  
      ( N <= 5, /* the friends F need only one table */
        in(restaurant(Id),table(T,free)),
        table(T)<make(N), /* inform to set the table(T) */
        out(restaurant(Id),cliente(F,T,single))); /* the friends F need two tables */
  in(restaurant(Id),table(T1,free),table(T2,free),count(C)),
  T is C+1, out(restaurant(Id),count(T)),
  /* create the table T, that models the tables T1 and T2 */
  create_instance(table(T)), restaurant(Id) <-add(table(T), (1)
  table(T)<make(N), out(restaurant(Id),cliente(F,[T1,T2])))
},

waiter_exit(F) :- see(restaurant(Id)),
     in(restaurant(Id),cliente(F,T,L)),
     ( L is 'single', /* if is a single table, free the table T */
       free_table(Id,[T]); /* if is not single, free the tables in T */
       free_table(L),
       restaurant(Id) <-delete(table(T)),
       table(T)<kill /* destroys the group table(T)*/
).

free_table(Id,[ ]).
free_table(Id,[T,L]) :- table(T)<clean, /* clean the table */
      out(restaurant(Id),table(T,free)), free_table(Id,L).
}

module friends{
  interface(go_eat(F,N,T), end_eat(F)).

restaurant(Id) : go_eat(F,N,T) :- N>1, N<11 | restaurant(Id) : waiter_entry(F,N,T).
restaurant(Id) : end_eat(F) :- | restaurant(Id) : waiter_exit(F).

restaurant(Id) :- | restaurant(Id).
}

The meeting and farewell actions are modeled by the friends group defined in section 2.3. The coordination of the N philosophers is modeled through the configuration and the shared state of the group. There are two phases: one represents the meeting and another one the farewell, as modeled in the group configuration (wait goto(X) and wait exit(X)). The shared state is used to coordinate the arrival of all the friends.

This example illustrates how groups are important to specify: (1) a group of agents, which must keep a shared knowledge (as in table and restaurant), (2) the dynamic evolution of the system (as in restaurant, table and friends) and (3) the synchronization of the agents in a group (as in table and friends).

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 [28] or Situation Calculus. In the first case, we have Rose [11], DeltaProlog [12], MultiProlog [23]. In the second, MetaTem [7]. In the third, COOL [27] and IAM [19] 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 [25], IAM [19] and COOL [27]. Concerning communication, several distinct approaches are possible: Shared-memory models are based on the sharing of logical variables between the goals in a clause body [17], or on the definition of a globally shared communication space [15, 5, 16, 25, 19, 21, 23, 24, 26, 27]; Distributed-memory models are based on message-passing where messages are interpreted as logical terms, or on remote predicate invocations where well-defined process interfaces are provided [13, 14].

The motivation to use EHC as the base language for GroupLog is because its interpretation of a message-based communication with an object is an elegant form to model interaction between two concurrent entities.

The dynamic entities of a program can be modeled by: Processes, as April [1] and MultiProlog [23]; Objects, as ShaDe [25], Law-Governed Linda [24], IAM [19], ColaS [33], Electra [29] and Emerald [30]; Agents, as ConGolog [9], COOL [27], MetaTem [7], Agentspeak [8], 3APL [34] and Placa [6]; Actors, as Concurrent Aggregate [18] and Synchronizers [22].

The interaction between dynamic entities can be modeled by: Sending messages, as ShaDe [25], ConGolog [9], Concurrent Aggregate [18], IAM [19], AgentSpeak [8], COOL [27], MetaTem [7], April [1], Placa [6] and Electra [29]; Shared tuples, as GammaLog [21], PoliS [26], Law-Governed Linda [24], MultiProlog [23] and ESP [5].

L₁ vs others models In L₁, we structured the concurrency and communication through 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 an interpretation very similar 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 [31]. In this model it is possible to model blackboard-based systems, which is only allowed in GroupLog in L₂.

L₂ vs others models The notion of a group of agents allows to structure the communication and the organization of an agents space, in order to allow the cooperation among them. 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]. With groups, one can model 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 [32]. In the 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 [27] and Concurrent Aggregates [18], 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 [29], Emerald [30], Synchronizers [22] and ColaS [33], the group is seen as a logical unit that manipulate and restricts the invocation of the members group interface. In Synchronisers, 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 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 group members individual interface. In languages where the communication is modeled by shared-memory, like ESP [5] and PoliS [26], the communication structuring is done by allowing multi-tuple spaces. In case of PoliS there are hierarchical tuple spaces. The $L_2$ 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. Ongoing work is related to the formal definition of the semantics and the distributed implementation of GroupLog.

References


