BAAC: A Prolog System for Action Description and Agents Coordination*

Agostino Dovier[1], Andrea Formisano[2], and Enrico Pontelli[3]

1 **Univ. di Udine, Dip. di Matematica e Informatica.** `agostino.dovier@uniud.it`
2 **Univ. di Perugia, Dip. di Matematica e Informatica.** `formis@dmi.unipg.it`
3 **New Mexico State University, Dept. Computer Science.** `epontell@cs.nmsu.edu`

Abstract

The paper presents a system for knowledge representation and coordination, where autonomous agents reason and act in a shared environment. Agents autonomously pursue individual goals, but can interact through a shared knowledge repository. In their interactions, agents deal with problems of synchronization and concurrency, and have to realize coordination by developing proper strategies in order to ensure a consistent global execution of their autonomously derived plans. This kind of knowledge is modeled using an extension of the action description language $\mathcal{B}$. A distributed planning problem is formalized by providing a number of declarative specifications of the portion of the problem pertaining a single agent. Each of these specifications is executable by a stand-alone CLP-based planner. The coordination platform, implemented in Prolog, is easily modifiable and extensible. New user-defined interaction protocols can be integrated.

1998 ACM Subject Classification I.2.3 Deduction and Theorem Proving, I.2.11 Distributed Artificial Intelligence

Keywords and phrases Knowledge Representation, Multi-Agent Systems, Planning, CLP

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.187

1 Introduction

Representing and reasoning in multi-agent domains are two of the most active research areas in *multi-agent system (MAS)* research. The literature in this area is extensive, and it provides a plethora of logics for representing and reasoning about various aspects of MAS, e.g., [13, 9, 17, 15, 7]. Several logics proposed in the literature have been designed to specifically focus on particular aspects of MAS, often justified by a specific application scenario. This makes them suitable to address specific subsets of the general features required to model real-world MAS domains. The task of generalizing these proposals to create a uniform and comprehensive framework for modeling different aspects of MAS domains is an open problem. We do not dispute the possibility of extending the existing proposals in various directions, but the task is not easy. Similarly, a variety of multi-agent programming platforms have been proposed, mostly in the style of multi-agent programming languages, e.g., Jason, ConGolog, 3APL, GOAL [1, 4, 3, 10], but with limited planning capabilities.

Our effort here is on developing a multi-agent system for knowledge representation based on a high-level action language. The starting point of this work is the action language $\mathcal{B}^{MV}$ [6]; this is a flexible single-agent action language, that generalizes the action language $\mathcal{B}$ [8], with support for multi-valued fluents, non-Markovian domains, and constraint-based

* This work is partially supported by INdAM-GNCS 2010, INdAM-GNCS 2011, and PRIN 20089M932N projects, and NSF grants 0812267, 0754525, and 0420407.

© Agostino Dovier, Andrea Formisano, and Enrico Pontelli;
licensed under Creative Commons License NC-ND

formulations (which enable, for example, the formulation of costs and preferences). In this work, we propose a further extension to support MAS scenarios. The perspective is that of a distributed environment, with agents pursuing individual goals but capable of interacting through shared knowledge and concurrent actions. A first step in this direction has been described in the $\mathcal{B}^{\mathsf{MAP}}$ language [5]; $\mathcal{B}^{\mathsf{MAP}}$ extends $\mathcal{B}^{MV}$ providing a multi-agent action language with capabilities for *centralized* planning. In this paper, we embed $\mathcal{B}^{\mathsf{MAP}}$ into a truly distributed multi-agent platform. The language is extended with $\mathcal{C}$ommunication primitives for modeling interactions among $\mathcal{A}$utonomous $\mathcal{A}$gents. We refer to this language as $\mathcal{B}^{\mathsf{AAC}}$. Differently from [5], agents can have private goals and are capable of developing independent plans. Agents' plans are developed in a distributed fashion, leading to replanning and/or to the introduction of coordination actions to enable a consistent global execution. The system is implemented in SICStus Prolog, using the libraries `clpfd` and `linda`.

2 Syntax of the Multi-agent Language $\mathcal{B}^{\mathsf{AAC}}$

The signature of the language $\mathcal{B}^{\mathsf{AAC}}$ consists of a set $\mathcal{G}$ of *agent* names, used to identify the agents in the system, a (unique) set $\mathcal{F}$ of *fluent* names, a set $\mathcal{A}$ of *action* names, and a set $\mathcal{V}$ of values for the fluents in $\mathcal{F}$—we assume $\mathcal{V} = \mathbb{Z}$. The behavior of each agent a is specified by an action description theory $\mathcal{D}_a$, i.e., a collection of axioms of the forms described next.

Name and priority of the agent a are specified in $\mathcal{D}_a$ by *agent declarations*:

$$\texttt{agent}\ a\ \texttt{[priority}\ n\ \texttt{]} \qquad (1)$$

where $n \in \mathbb{N}$. 0 (default value) denotes the highest priority. Priorities might be used to resolve conflicts among actions of different agents. Agent a can access only those fluents that are declared in $\mathcal{D}_a$ by axioms of the form:

$$\texttt{fluent}\ f_1, \ldots, f_h\ \texttt{valued}\ dom \qquad (2)$$

with $f_i \in \mathcal{F}$, $h \geq 1$, and $dom \subset \mathcal{V}$ is a set of values representing the admissible values for $f_1, \ldots, f_h$ (possibly represented as an interval $[v_1, v_2]$). We refer to these fluents as the "local state" of agent a. Fluents accessed by multiple agents are assumed to be defined consistently.

▶ **Example 1.** Let us specify a domain inspired by volleyball. There are two teams: *black* and *white*, with one player in each team; let us focus on the domain for the white team (Sect. 4 deals with the case that involves more players). We introduce fluents to model the positions of the players and of the ball, the possession of the ball, the score, and a numerical fluent `defense_time`. All players know the positions of all players. Since the teams are separated by the net, the `x`-coordinates of a black and white players must differ. This can be stated by:

```
agent player(white,X) :- num(X).
known_agents player(black,X) :- num(X).

fluent x(player(white,X)) valued [B,E] :- num(X), net(NET),B is NET+1, linex(E).
fluent x(player(black,X) valued [1,E] :- num(X), net(NET),E is NET-1.
fluent y(A) valued [1,MY] :- player(A), liney(MY).
fluent x(ball) valued [1,MX] :- linex(MX).
fluent y(ball) valued [1,MY] :- liney(MY).
fluent hasball(A) valued [0,1] :- agent(A).
fluent point(T) valued [0,1] :- team(T).
fluent defense_time valued [0,1].

team(black). team(white). num(1). linex(11). net(6). liney(5).
```

where `linex`, and `liney` are the field sizes, and `net` is the `x`-coordinate of the net. □

Fluents are used in *Fluent Expressions* (FE), which are defined as follows:

$$\texttt{FE} \quad ::= \quad n \mid f^t \mid f\texttt{@}r \mid \texttt{FE}_1 \oplus \texttt{FE}_2 \mid \; -(\texttt{FE}) \mid \texttt{abs}(\texttt{FE}) \mid \texttt{rei}(\texttt{C}) \tag{3}$$

where $n \in \mathcal{V}$, $f \in \mathcal{F}$, $t \in \{0, -1, -2, -3, \dots\}$, $\oplus \in \{+, -, *, /, \texttt{mod}\}$, and $r \in \mathbb{N}$. FE is referred to as a *timeless expression* if it contains no occurrences of f^t with $t \neq 0$ and no occurrences of $f\texttt{@}r$. f can be used as a shorthand of f^0. The notation f^t is an *annotated* fluent expression. The expression refers to the value f had $-t$ steps in the past. An expression of the form $f\texttt{@}r$ denotes the value f has at the r^{th} step in the evolution of the world (i.e., an *absolute* point in time). We use the expression $\texttt{pair}(\texttt{FE}_1,\texttt{FE}_2)$ to encode a pair, and the projection functions $\texttt{x}(\cdot)$ and $\texttt{y}(\cdot)$ such that $\texttt{x}(\texttt{pair}(a,b)) = a$ and $\texttt{y}(\texttt{pair}(a,b)) = b$. The reified expression $\texttt{rei}(\texttt{C})$ represents a Boolean value indicating the truth value of the constraint C.

A *Primitive Constraint* (PC) is formula $\texttt{FE}_1$ op $\texttt{FE}_2$, where $\texttt{FE}_1$ and $\texttt{FE}_2$ are fluent expressions, and op $\in \{=, \neq, \geq, \leq, >, <\}$. A *constraint* C is a propositional combination of PCs. As a syntactic sugar, f++ (f--) denotes the primitive constraint $f = f^{-1} + 1$ ($f = f^{-1} - 1$).

An axiom of the form action x in $\mathcal{D}_a$, declares that the action $x \in \mathcal{A}$ is available to the agent a. The same action name x can be used by different agents. A special action, nop is always executable by every agent, and it causes no changes to any of the fluents.

▶ **Example 2.** The actions for each player A of Example 1 are:

- A : $\texttt{move}(d)$ one step in direction d, where d is one of the eight directions: north, north-east, ..., west, north-west (i.e., analogous to the moves of a King on a chess-board).
- A : $\texttt{throw}(d, f)$ the ball in direction d (same eight directions as above) with a strength f varying from 1 to a maximum throw power (5 in our example).

Moreover, the player of each team is in charge of checking if a point has been scored (in such case, he whistles). We write the actions as act([A],action_name) and state these axioms:

```
action act([A],move(D)) :- whiteplayer(A),direction(D).
action act([A],throw(D,F)) :- whiteplayer(A),direction(D),power(F).
action act([player(white,1)],whistle).
```

where whiteplayer, power, and direction can be defined as follows:

```
whiteplayer(player(white,N)) :- agent(player(white,N)).
power(1). power(2). power(3). power(4). power(5).
direction(D) :- delta(D,_,_).    delta(nw,-1,1). delta(n,0,1). delta(ne,1,1).
delta(w,-1,0). delta(e,1,0). delta(sw,-1,-1). delta(s,0,-1). delta(se,1,-1).
```
□

The executability of the actions is described by axioms of the form:

$$\texttt{executable } x \texttt{ if C} \tag{4}$$

where $x \in \mathcal{A}$ and C is a constraint, stating that C has to be entailed by the current state in order for x to be executable. We assume that at least one executability axiom is present for each action x. Multiple executability axioms are treated as a disjunction.

▶ **Example 3.** In our working example, we can state executability as follows:

```
executable act([player(white,1)],whistle) if [S eq 0] :- build_sum(S).
executable act([A],move(D)) if [hasball(A) eq 0, defense_time gt 0,
      Net lt x(A)+DX, x(A)+DX leq MX, 1 leq y(A)+DY, y(A)+DY leq MY] :-
        action(act([A],move(D))), delta(D,DX,DY), net(Net), linex(MX), liney(MY).
executable act([A],throw(D,F)) if
      [hasball(A) gt 0,defense_time eq 0, 1 leq x(A)+DX*F, x(A)+DX*F leq MX,
      1 leq y(A)+DY*F, y(A)+DY*F leq MY] :-
            action(act([A],throw(D,F))), delta(D,DX,DY), linex(MX), liney(MY).
```

These axioms state that neither a player nor the ball can leave the field. build_sum is recursively defined to return the expression: $\texttt{defense\_time} + \texttt{hasball}(A_1) + \cdots + \texttt{hasball}(A_n)$

where $A_1, \ldots, A_n$ are the players (i.e., `player(white,1)` and `player(black,1)`). Let us observe that $=, \neq, \leq, <$, etc. are concretely represented by `eq`,`neq`,`leq`,`lt`, respectively. ☐

The effects of an action are described by axioms (*dynamic causal laws*) of the form:

x `causes` *Eff* `if` *Prec* (5)

where $x \in \mathcal{A}$, *Prec* is a constraint, and *Eff* is a conjunction of primitive constraints of the form $f = FE$, where $f \in \mathcal{F}$. The axiom asserts that if *Prec* is `true` w.r.t. the current state, then *Eff* must hold after the execution of x. Since agents share fluents, their actions may cause inconsistencies. A *conflict* happens when the effects of different actions lead to an inconsistent state; a procedure has to be applied to resolve conflicts and determine a consistent subset of the conflicting actions (Sect. 3.2). A *failure* occurs whenever an action x cannot be executed as planned by an agent a as a consequence of the above procedure.

▶ **Example 4.** Let us state the effects of the actions in the volleyball domain. When the ball is thrown, with force f, in direction d, it reaches a destination cell whose distance is as follows: a) if d is either north or south then $\Delta X = 0, \Delta Y = f$; b) if d is east or west then $\Delta X = f, \Delta Y = 0$; c) if d is any other direction, $\Delta X = f, \Delta Y = f$. As a further effect of throw, the fluent `defense_time` is set (to 1 in our example).

```
actocc([A],throw(D,F)) causes hasball(A) eq 0 :- action(act([A],throw(D,F))).
actocc([A],throw(D,F)) causes defense_time eq 1 :- action(act([A],throw(D,F))).
actocc([A],throw(D,F)) causes pair(x(ball),y(ball)) eq
      pair(x(A)^-1+ F*DX,y(A)^-1+ F*DY) :-
      action(act([A],throw(D,F))), delta(D,DX,DY).
actocc([A],throw(D,F), causes hasball(B) eq 1
      if [pair(x(B),y(B)) eq pair(x(A)+F*DX, y(A)+F*DY)] :-
      action(act([A],throw(D,F))), player(B), neq(A,B),delta(D,DX,DY).
actocc([A],throw(D,F)) causes point(black) eq 1 if [x(A)+F*DX eq Net] :-
      action(act([A],throw(D,F))), delta(D,DX,_), net(Net).
```

The effects of the other two actions `move` and `whistle` can be stated by:

```
actocc([player(white,1)],whistle) causes point(white) eq 1
      if [x(ball) lt NET] :- net(NET).
actocc([player(white,1)],whistle) causes point(black) eq 1
      if [NET lt x(ball)] :- net(NET).
actocc([A],move(D)) causes pair(x(A),y(A)) eq pair(x(A)^-1+DX,y(A)^-1+DY) :-
      action(act([A],move(D))), delta(D,DX,DY).
actocc([A],move(D)) causes defense_time-- :- action(act([A],move(D))).
actocc([A],move(D)) causes hasball(A) eq 1
      if [pair(x(ball),y(ball)) eq pair(x(A)+DX,y(A)+DY)] :-
      action(act([A],move(D))), delta(D,DX,DY).
```

Let us observe here that we concretely used `actocc` instead of `act`. This has been introduced to give the idea of the occurrence of an action. ☐

At least two perspectives can be followed, by assigning either a passive or an active role to the conflicting agents during conflict resolution. In the first case, a supervising entity is in charge of resolving the conflicts, and all the agents will adhere to the supervisor's decisions. Alternatively, the agents are in charge of reaching an agreement, possibly through negotiation. The following declarations describe basic reaction policies the agents can use:

`action` x $[OPT]$ (6)

where:

OPT ::= `on_conflict` OC $[OPT]$ | `on_failure` OF $[OPT]$
OC ::= `retry_after` T `[provided` C`]` | `forego [provided` C`]` | `arbitration`
OF ::= `retry_after` T `[if` C`]` | `replan [if` C`] [add_goal` C`]` | `fail [if` C`]`

In these axioms one can also specify policies to be adopted when a failure occurs during action execution. Reacting to a failure is a "local" activity the agent performs after the state transition has been completed. In the axioms (6), one can specify different reactions to a conflict (resp. a failure) of the same action, to be considered in their order of appearance.

Apart from the communications occurring among agents during conflict resolution, other forms of "planned" communication can be modeled in an action theory. An agent might seek help from other agents to make a constraint true. The request can be broadcast to all known agents or sent to some specific agents. The agent can optionally offer a "reward" in case of acceptance of the proposal. This allows us to model negotiations and bargaining.

$$\texttt{request}\ C_1\texttt{[to\_agent}\ a'\texttt{] if}\ C_2\ \texttt{[offering}\ C_3\ \texttt{]} \tag{7}$$

Various *global constraints* can be exploited to impose control knowledge and maintenance goals representing properties that must persist. For example:

- FC `holds_at` n: the fluent constraint FC holds at the n^{th} time step.
- `always` FC: the fluent constraint FC holds in all states of the evolution of the world.

A detailed description of these constraints and their semantics can be found in [6].

An *action domain description* consists of a collection $\mathcal{D}_a$ of axioms of the forms described so far, for each agent $a \in \mathcal{G}$. Moreover, it includes a collection $\mathcal{O}_a$ of goal axioms (objectives), of the form `goal C`, where `C` is a constraint, and a collection $\mathcal{I}_a$ of initial state axioms of the form: `initially C` where `C` is a constraint involving only timeless expressions. We assume that all the sets $\mathcal{I}_a$ are drawn from a consistent global initial state description $\mathcal{I}$, i.e., $\mathcal{I}_a \subseteq \mathcal{I}$. A specific instance of a planning problem is a triple $\left\langle \langle \mathcal{D}_a \rangle_{a\in\mathcal{G}}, \langle \mathcal{I}_a \rangle_{a\in\mathcal{G}}, \langle \mathcal{O}_a \rangle_{a\in\mathcal{G}} \right\rangle$. The problem has a solution only if $\langle \mathcal{O}_a \rangle_{a\in\mathcal{G}}$ characterizes a consistent state, i.e., there exists a consistent assignment of values to the fluents that satisfies the constraint $\bigwedge_{a\in\mathcal{G}} \bigwedge_{\texttt{goal}\,C\in\mathcal{O}_a} C$.

3 System Behavior

The behavior of $\mathcal{B}^{\mathsf{AAC}}$ can be split in two parts: the semantics of the action description languages used *locally* by each agent, ignoring the axioms (6) and (7), and the behavior of the *overall* system that deals with agents' interactions. Let us assume that there is an overall planning horizon length N. Due to space restrictions, we don't enter into the details of the "local" semantics which is given in terms of transition systems (as in [8]). The formal semantics of the language $\mathcal{B}^{MV}$, upon which $\mathcal{B}^{\mathsf{AAC}}$ is defined, is given in detail in [6].

3.1 Concurrent Plan Execution

The agents are autonomous and plan their activities independently. In executing their plans, the agents must take into account the effects of concurrent actions. We developed a basic communication mechanism among agents by exploiting a *tuple space*, realized using the Linda system [2]. Moreover, most of the interactions among concurrent agents, especially those aimed at resolving conflicts, are managed by a specific process, the *supervisor*, that also provides a global time to all agents, enabling them to execute their actions synchronously.

1. At the beginning, the supervisor acquires the initial state description $\mathcal{I} = \bigcup_{a\in\mathcal{G}} \mathcal{I}_a$.
2. At each time step the supervisor starts a new state transition:
   - Each active agent sends to the supervisor a request to perform an action specifying its effects on the (local) state.
   - The supervisor collects these requests and determines whether subsets of actions/agents are conflicting. A conflict occurs whenever agents require incompatible assignments of values to the same fluents. The transition takes place once all conflicts have

been resolved and a subset of compatible actions has been identified using one or more policies (see below). These actions are enabled while the remaining ones are inhibited.
- All the enabled actions are executed, yielding changes that define a new (global) state.
- These changes are then sent back to all agents, to update their local states. Those agents whose actions have been inhibited receive a failure message.

3. The computation stops when the time N is reached.

After each step of the local plan execution, the agents need to check if the reached state still supports their successive planned actions. If not, each agent has to reason locally and revise its plan, i.e., initiate a replanning phase. This may occur in two cases: **(a)** The proposed action was inhibited, so the agent actually executed a `nop`; this case occurs when the agent receives a failure message from the supervisor. **(b)** The interaction was successful, i.e., the planned action was executed, but the effects of the actions performed by other agents affected fluents in its local state, preventing the successful continuation of the rest of the plan—e.g., the agent a may have assumed that the fluent g maintained its value by inertia, but another agent changed such value. This might affect the executability of the next action of a's plan.

3.2 Conflicts Resolution

A conflict resolution procedure is performed by the supervisor whenever it identifies a set of incompatible actions. Different policies can be adopted in this phase and different roles can be played by the supervisor. First, the supervisor exploits the priorities of the agents to attempt a resolution of the conflict, by inhibiting the actions of low priority agents. If this does not suffice, further options are applied. Two simple options have been implemented in our prototype, assigning the active role either to the supervisor or to the conflicting agents. The architecture is modular, and can be extended with more complex policies.

- The supervisor has the *active role*—it decides which actions to inhibit. In the current prototype, the arbitration strategy is limited to **(1)** A random selection of a single action to be executed or **(2)** The computation of a maximal set of compatible actions to be executed. This computation is done by solving a dynamically generated CSP. In this strategy, the `on_conflict` policies assigned to actions by axioms (6) are ignored.
- The supervisor simply notifies the set of conflicting agents about the inconsistency of their actions. The agents involved in the conflict are completely in charge of resolving it via negotiation. The supervisor waits for a solution from the agents. In solving the conflict, each agent a makes use of one of the `on_conflict` directives (6) specified for its conflicting action x. The semantics of these directives are as follows (`[provided` C`]` is an optional qualifier; if omitted it will be interpreted as `provided true`):
  - The option `on_conflict arbitration` causes the execution of the supervisor which performs an arbitration phase to resolve the conflict, as previously described.
  - The option `on_conflict forego provided` C causes the agent a to "search" among the other conflicting agent for someone, say b, that can guarantee the condition C. In this case, b performs its action while the execution of a's action fails, and a executes a `nop` in place of its action x. Different strategies can be implemented in order to perform such a "search for help", e.g., a round-robin policy described below, but other alternatives are possible and should be considered in completing the prototype.
  - The option `on_conflict retry_after` T `provided` C, differs from the preceding one because a will execute `nop` during the following T time steps and then will try again to execute its action (provided that the preconditions of the action still hold).
  - If there is no applicable option (e.g., no option is defined or none of the agents accept to, or is able to, guarantee C), the action is inhibited and its execution fails.

The way in which agents exploit the `on_conflict` options can rely on several policies. In the current prototype we implemented a round-robin policy. Let us assume that the agents $a_1, \ldots, a_m$ aim at executing actions $z_1, \ldots, z_m$, respectively, and these actions are conflicting. The agents are sorted by the supervisor, and the agents take turn in resolving the conflict. Suppose that at a certain round j of the procedure the agent a_i is selected. It determines the j-th option for its action and tries to apply it. If the option is directly applicable or an agreement is reached with another agent on a condition C, then the two agents exit the procedure. If no arbitration is invoked, then the remaining agents will complete the procedure. If the option does not lead to an agreement, then the next agent in the sequence will start its active role in the round, while a_i will wait its next turn in round $j + 1$. This procedure always ends with a solution to the conflict, since a finite number of `on_conflict` options are defined for each action. This a rigid policy, and it represents a simple example of how to realize a terminating protocol for conflict resolution. Alternative solutions can be added to the prototype thanks to its modularity.

Once all conflicts have been addressed, the supervisor applies the enabled actions, and obtains the new global state. Each agent receives a communication containing the outcome of its action execution and the changes to its local state. Moreover, further information might be sent to the participating agents, depending on the outcome of the coordination procedure. For instance, when two agents agree on an `on_conflict` option, they "promise" to execute specific actions (e.g., one agent may have to execute T consequent `nop`). This information has to be sent back to the interested agents to guide their replanning phases.

3.3 Failure Policies

Agents receiving a failure message from the supervisor need to revise their original plans to detect if the local goals can still be achieved. Different approaches can be used. For instance, one agent could avoid developing an entire plan at each step, but only produce a partial plan for the very next step. Alternatively, an agent could determine the "minimal" modifications to the existing plan in order to make it valid with respect to the new encountered state. At this time, the prototype includes only replanning from scratch at each step.

While replanning, the agent might exploit the `on_failure` options associated to the inhibited action. The intuitive semantics of these options is as follows. (The options declared for the inhibited action are considered in the given order, executing the first applicable one).

- `retry_after` T `[if` C`]`: the agent evaluates the constraint C; if C holds, then it will execute `nop` T times and then try again the failed action (provided it is executable).
- `replan [if` C_1`] [add_goal` C_2`]`: the agent evaluates C_1; if it holds, then in the replanning phase the goal C_2 will be added to the local goal. The `add_goal` C_2 is optional; if it is not present then nothing will be added to the goal, i.e., it is the same as `add_goal true`.
- `fail [if` C_1`]`: this is analogous to `replan [if` C_1`] add_goal false`. In this case the agent declares that it is impossible to reach its goal.
- If none of the above options is applicable, then the agent will proceed as if the option `replan if true` is present.

It might be the case that some global constraints (such as `holds_at` and `always`) involve fluents that are not known by any of the agents. Therefore, none of the agents is able to consider such constraints while developing his plan. These constraints have to be enforced when merging the individual plans. In doing so, the supervisor adopts the same strategies introduced to deal with conflicts and failures among actions, as described earlier.

3.4 Broadcasting and Direct Requests

Let us describe a simple protocol for implementing communications among agents, following an explicit request of the form (7). We assume that the current state is the i-th one of the plan execution. The handling of requests is interleaved with the agent-supervisor interactions; nevertheless, requests and offers are directly exchanged among agents. The main steps involved in a state transition, from the point of view of an agent a, are:

1. Agent a tries to execute its action and sends this information to the supervisor (Sect. 3.1).
2. Possibly after a coordination phase, a receives from the supervisor the outcome of its attempt to execute the action (namely, failure or success, the changes in the state, etc.)
3. If the action execution is successful, before declaring the current transition completed, the agent a starts an interaction with the other agents to handle pending requests. All the communications associated to such interactions are realized using Linda's tuple-space.

   3.a. Agent a fetches the collection H of all pending requests. For each request $h \in H$, e.g., originating from agent b, a decides whether to accept h. Such a decision might involve exploitation of the planning facilities, in order to determine if the requested condition can be achieved by a, possibly by modifying its original plan. If possible, a posts its offer into the tuple-space and waits for a rendezvous with b.

   3.b. Agent a checks whether there are replies to the requests it previously posted. For each request, a collects the set of offers/agents that expressed their willingness to help a and, by using some strategy, selects one of them, say b. The policy for choosing the responding agent can be programmed (e.g., by exploiting priorities, etc.). Once the choice has been made, a communicates with the selected agent, declares its availability to b, and communicates the fulfillment of the request to the other agents. The request is also removed from the tuple space, along with all the obsolete offers.
4. At that point, the transition is completed for agent a. By taking into account the information about the outcome of the coordination phase in solving conflicts (point (2)), the agreement reached in handling requests (point (3)), a might need to modify its plan. If the replanning phase succeeds, then a will execute the next action in its local plan.

3.5 Implementation Issues

A prototype of the system has been implemented in SICStus Prolog, using the libraries `clpfd` for reasoning (and the planners described in [5, 6]), and the libraries `system`, `linda/server`, and `linda/client` for process communication. The system is organized in modules and it is available, together with some sample domains at `www.dimi.uniud.it/dovier/BAAC`.

Each autonomous agent corresponds to an instance of the module `plan_executor`, which, in turn, relies on a planner for planning/replanning activities, and on `client` for interacting with other actors in the system. As previously explained, a large part of the coordination is guided by the module `supervisor`. Notice that both the `supervisor` and `client` act as Linda-clients. Conflict resolution functionalities are provided to the modules `client` and `supervisor` by the modules `ConflictSolver_client` and `ConflictSolver_super`, respectively. Finally, the `arbitration_opt` module implements the arbitration protocol(s).

Let us remark that all the policies exploited in coordination, arbitration, and conflict handling can be customized by providing a different implementation of individual predicates exported by the corresponding modules. For instance, to implement a conflict resolution strategy different from the round-robin described earlier, it suffices to add to the system a new implementation of the module `ConflictSolver_super` (and for `ConflictSolver_client`, if the specific strategy requires an active role of the conflicting agents). Similar extensions can be done for `arbitration_opt`. A `settings.pl` file is available to enable specification of various parameters, e.g., the names of the files containing the action descriptions, the number of planning steps allowed, the selected conflict resolution strategies, etc.

As far as the planning module is concerned, we modified the interpreters of the $\mathcal{B}^{MV}$ and the $\mathcal{B}^{\mathsf{MAP}}$ languages [5, 6] to accept the coordination constructs described in this paper. The two planners, `sicsplan` and `bmap`, have been integrated in the system to process $\mathcal{B}^{MV}$ and $\mathcal{B}^{\mathsf{MAP}}$ theories. However, the system is open to further extensions and different planners (even not necessarily based on Prolog technology) can be easily integrated thanks to the simple interface with the module `plan_executor`, which consists of few Prolog predicates.

4 The Volleyball Domain

Let us describe a specification in $\mathcal{B}^{\mathsf{AAC}}$ of a coordination problem between two multi-agent systems. Let us extend the domains described in Examples 1–4. There are two teams: *black* and *white* whose objective is to score a point, i.e., to throw the ball in the field of the other team (passing over the net) in such a way that no player of the other team can reach the ball before it touches the ground. Each team is modeled as a multi-agent system that elaborates its own plan in a centralized manner (thus, each step in the plan consists of a set of actions).

The playing field is discretized by fixing a `linex`$\times$`liney` rectangular grid that determines the positions where the players (and the ball) can move (see Fig. 1). The leftmost (rightmost) cells are those of the black (white) team, while the net ($x = 6$) separates the two subfields. There are p players per team ($p = 2$ in Fig. 1). The allowed actions are: `move`(d), `throw`(d, f), and `whistle`. During the defense time, the players can move to catch the ball and/or to re-position themselves on the court. When a player reaches the ball (s)he will have the ball and will throw the ball again. A team scores a point either if it throws the ball to a cell in the opposite subfield that is not reached by any player of the other team in the defense time, or if the opposite team throws the ball in the net. The captain (first player) of each team is in charge of checking if a point has been scored. In this case, (s)he `whistle`s.

Each team is modeled as a centralized multi-agent system, which acts as a singe agent in the interaction with the other team. Alternative options in modeling are also possible—for instance, one could model each single player as an independent agent that develops its own plan and interacts with all other players. The two teams have the goal of scoring a point: `goal(point(black) eq 1).` for blacks and `goal(point(white) eq 1).` for whites.

At the beginning of the execution every team has a winning strategy, developed as a local plan; these are possibly revised after each play to accommodate for the new state of the worlds reached. An execution (as printed by the system) is reported in Fig. 1, for a plan length of 9. The symbol `O` (respectively, `Y`) denotes the white (respectively, black) players, `Q` (resp. `X`) denotes a white player with the ball. The throw moves applied are:

`[player(black,1)]:throw(ne,3)`	(time 1)	`[player(black,2)]:throw(se,3)`	(time 3)
`[player(white,1)]:throw(w,5)`	(time 5)	`[player(black,1)]:throw(e,5)`	(time 7)

Let us observe that, although it would be in principle possible for the white team to reach the ball and throw it within the time allowed, it would be impossible to score a point. Therefore, players prefer to avoid to perform any move.

The complete description of the encoding of this domain is available at `http://www.dimi.uniud.it/dovier/BAAC`. The repository includes also additional domains—e.g., a domain inspired by games involving one ball and two-goals, as found in soccer. Although the encoding might seem similar to that of volleyball, the possibility of contact between two players makes this encoding more complex. Indeed, thanks to the fact that the net separates the two teams,

```
Time 0:           Time 1:           Time 2:           Time 3:           Time 4:
******|******     ******|******     ******|******     ******|******     ******|******
*     |     *     *     |     *     *     |     *     *     |     *     *     |     *
*  Y  |    O*     *  Yo |    O*     *   X |   O *     *   Y |   O *     *  Y  |  O  *
*     |     *     *     |     *     *     |     *     *     |     *     *     |     *
*     |  O  *     *     |  O  *     * Y   | O   *     * Y   | O   *     *Y    |     *
*X    |     *     *Y    |     *     *     |     *     *     |o    *     *     |Q    *
******|******     ******|******     ******|******     ******|******     ******|******
 Time 5:           Time 6:           Time 7:           Time 8:           Time 9:
******|******     ******|******     ******|******     ******|******     ******|******
*     |     *     *     |     *     *     |     *     *     |     *     *     |     *
*  Y  |  O  *     * Y   | O   *     * Y   | O   *     *Y    | O   *     *Y    | O   *
*     |     *     *     |     *     *     |     *     *     |     *     *     |     *
*Y    |     *     *     | O   *     *     | O   *     *     | O   *     *     | O   *
* o   |O    *     * X   |     *     * Y   |o    *     *Y    |o    *     *Y    |o    *
******|******     ******|******     ******|******     ******|******     ******|******
```

Figure 1 A representation of an execution of the volleyball domain

in the volleyball domain rules like the following one suffice to avoid collisions:

```
always(pair(x(A),y(A)) neq pair(x(B),y(B))) :-
       A=player(black,N),B=player(black,M), num(N), num(M), N<M.
```

In a soccer world this is not true because only the supervisor can be aware, in advance, of possible contacts between different team players originating from concurrent actions. This generates interesting concurrency problems, e.g., concerning the ball possession after a contact. A simple way to address this problem consists in assigning a fluent to each field cell, whose value can be -1 (free), 0 (resp., 1) if a white (resp. black) player is in the cell. The supervisor identifies a conflict when two opponent players move to the same cell, thus assigning to that fluent a different value. In this case, the supervisor arbitrarily enables one action, the other agent waits a turn to retry the action:

```
action act([A],move(D)) on_failure retry_after 1 on_conflict arbitrate :-
      agent(A), direction(D).
```

5 Conclusions and future work

In this paper, we designed a system for reasoning with action description languages in multi-agent domains. The language enables the description of agents with individual goals operating in shared environments. The agents can interact (by requesting help from other agents in achieving goals) and implicitly cooperate in resolving conflicts that arise during execution of their plans. The implementation is distributed, and uses Linda to enable communication.

The work is preliminary but shows strong potential and several directions of research. The immediate goal is to refine strategies and coordination mechanisms, involving, for instance, payoff, trust, etc. We intend to evaluate the performance and quality of the system in several multi-agent domains (e.g., game playing scenarios, auctions, and other domains requiring distributed planning). We will investigate the use of future references in the fluent constraints (as supported in $\mathcal{B}^{MV}$)—we believe this feature may provide a more elegant approach to handle the requests among agents, and it is necessary to enable the expression of complex interactions among agents (e.g., to model commitments). In particular, we view this platform as ideal to experiment with models of *negotiation* (e.g., as discussed in [14]) and to deal with *commitments* [11]. We will also explore the implementation of different strategies associated to conflict resolution; we are interested in investigating how to capture the notion of "trust" among agents, as a dynamic property that changes depending on how reliable agents have been in providing services to other agents.

References

1 R. H. Bordini, J. F. Hübner and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak using Jason.* J. Wiley and Sons, 2007.

2 N. Carriero and D. Gelernter. Linda in context. *CACM*, 32(4):444–458, 1989.

3 M. Dastani, F. Dignum, and J.-J. Meyer. 3APL: A programming language for cognitive agents. *ERCIM News*, 53:28–29, 2003.

4 G. De Giacomo, Y. Lespérance, and H. J. Levesque ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.

5 A. Dovier, A. Formisano, and E. Pontelli. Representing multi-agent planning in CLP. *LPNMR*, LNCS 5753, pp. 423–429, Springer, 2009.

6 A. Dovier, A. Formisano, and E. Pontelli. Multivalued action languages with constraints in CLP(FD). *TPLP*, 10(2):167–235, 2010.

7 R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about knowledge.* MIT Press, 1995.

8 M. Gelfond and V. Lifschitz. Action languages. *ETAI*, 2:193–210, 1998.

9 J. Gerbrandy. Logics of propositional control. In [12], pages 193–200.

10 K. V. Hindriks and T. Roberti. GOAL as a planning formalism. *MATES*, pp. 29–40. 2009.

11 A. U. Mallya and M. N. Huhns. Commitments among agents. *IEEE Internet Computing*, 7(4):90–93, 2003.

12 H. Nakashima, M. Wellman, G. Weiss and P. Stone editors. *Proc. of AAMAS.* ACM, 2006.

13 L. Sauro, J. Gerbrandy, W. van der Hoek, and M. Wooldridge. Reasoning about action and cooperation. In [12], pages 185–192.

14 T. Son, E. Pontelli, and C. Sakama. Logic programming for multiagent planning with negotiation. *ICLP*, pp. 99–114. Springer, 2009.

15 M. T. J. Spaan, G.J. Gordon, and N.A. Vlassis. Decentralized planning under uncertainty for teams of communicating agents. In [12], pages 249–256.

16 V. S. Subrahmanian, P. Bonatti, J. Dix. T. Eiter, S. Kraus and F. Ozcan, and R. Ross *Heterogeneous Agent Systems: Theory and Implementation.* MIT Press, 2000.

17 W. van der Hoek et al. A logic for strategic reasoning. *AAMAS*, pp. 157–164. ACM, 2005.