Autonomous Agents Coordination: Action Description Languages meet CLP($\mathcal{FD}$) and Linda *

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 knowledge representation formalism for multi-agent systems, where different autonomous agents reason and act in a shared environment. Agents are autonomously pursuing individual goals, but are capable of interacting through a shared knowledge repository and collaborative actions. In their interaction through shared portions of the world, agents deal with problems of synchronization and concurrency, and have to realize coordination by developing proper strategies and policies in order to ensure a consistent global execution of their autonomously derived plans.
To model this kind of knowledge, the paper proposes an high-level *Action Description Language (ADL)*. 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 among agents exploits a Linda-like infrastructure.
This is a working project and a concrete implementation of the system is being developed in SICStus Prolog.

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 domains, e.g., [12, 8, 15, 13, 6].

A large number of the logics proposed in the literature have been designed to specifically focus on particular aspects of the problem of modeling 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 some of these existing proposals to create a uniform and comprehensive framework for modeling several different aspects of MAS domains is an open problem. Although we do not dispute the possibility of

* Research partially funded by projects GNCS-INdAM:*Tecniche innovative per la programmazione con vincoli in applicazioni strategiche*; MUR-PRIN:*Innovative and multidisciplinary approaches for constraint and preference reasoning*; *Ricerca di base 2009–cod.2009.010.0336.*

extending several of these existing proposals in various directions, the task does not seem easy. Similarly, a variety of multi-agent programming platforms have been proposed, mostly in the style of multi-agent programming languages, like Jason [2], ConGolog [9], IMPACT [14], 3APL [11], GOAL [10], but with limited planning capabilities.

Our effort is on developing a knowledge representation formalism for multi-agent systems, in the form of a high-level action language. The foundations of this effort can be found in the action language B^{MV} [5]; this is a flexible single-agent action language, generalizing the action language B [7], with support for multi-valued fluents, non-Markovian domains, and constraint-based formulations—which enable, for example, the formulation of costs and preferences. B^{MV} was implemented in CLP($\mathcal{FD}$). In this work, we propose to extend B^{MV} 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 through collaborative actions.

A first step in this direction has been described in the B^{MAP} language [4]; B^{MAP} provides a multi-agent action language with capabilities for centralized planning. In this paper, we expand on this by moving B^{MAP} towards a truly distributed multi-agent platform. The language is extended with *C*ommunication primitives for modeling interactions among *A*utonomous *A*gents. We refer to this language simply as $\mathcal{B}^{\mathsf{AAC}}$. Differently from what described in [4], agents in the framework proposed in this paper can have private goals and are capable of developing independent plans. Agents' plans are composed in a distributed fashion, leading to replanning and/or introduction of coordination actions to enable a consistent global execution.

A first prototype of the resulting framework is being implemented, using CLP($\mathcal{FD}$) for the development of the individual plans of each agent, and Linda for the coordination and interaction among them.

2 Syntax of the Multiagent Language $\mathcal{B}^{\mathsf{AAC}}$

The signature of $\mathcal{B}^{\mathsf{AAC}}$ consists of a set $\mathcal{G}$ of *agent* names, used to identify the agents of the system, a set $\mathcal{F}$ of *fluent* names,[1] 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 means of an action description theory $\mathcal{D}_a$, namely a collection of axioms of the forms described in what follows.

Considering the action theory $\mathcal{D}_a$ of an agent a, name and priority of the agent are specified by *agent declarations*:

$$\texttt{agent } a \texttt{ [priority } n \texttt{].} \tag{1}$$

where $n \in \mathbb{N}$. We adopt the convention that 0 denotes the highest priority, which is also the default value, in absence of the priority declaration. As we will see,

[1] Intuitively, a fluent expresses a property of an object in a world, and forms part of the description of states of the world. Such properties might be affected by actions.

priorities might be used to resolve possible conflicts among actions of different agents.

It is possible to specify which agents are known to agent a, as follows:

$$\texttt{known\_agents}\ a_1, a_2, \ldots, a_k. \tag{2}$$

Consequently, agent a is able to explicitly query one of the a_is to start a communication phase (see below).

We assume the existence of a unique "global" set $\mathcal{F}$ of fluents, and any given agent a knows and can access only those fluents that are declared in $\mathcal{D}_a$ by axioms of the form (we refer to these fluents as the "local state" of the agent):

$$\texttt{fluent}\ f_1, \ldots, f_h\ \texttt{valued}\ dom_i. \tag{3}$$

with $f_i \in \mathcal{F}$, $h \geq 1$, and $dom_i \subset \mathcal{V}$ is a set of values representing the admissible values for each f_i (possibly represented as an interval $[v_1, v_2]$ if this is the case).

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

$$\texttt{FE} ::= n \mid f^t \mid f\texttt{@}r \mid \texttt{FE}_1 \oplus \texttt{FE}_2 \mid -(\texttt{FE}) \mid \texttt{abs(FE)} \mid \texttt{rei(C)} \tag{4}$$

where $n \in \mathcal{V}$, $f \in \mathcal{F}$, $t \in \{0, -1, -2, -3, \ldots\}$, $\oplus \in \{+, -, *, /, \texttt{mod}\}$, and $r \in \mathbb{N}$. FE is said 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 a relative time reference, indicating 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., it refers to an *absolutely* specified point in time). The last alternative in (4), a *reified expression*, requires the notion of constraint C. The semantics of rei(C) is a Boolean value depending on the truth of C.

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

An axiom of the form action x in $\mathcal{D}_a$, declares that the action $x \in \mathcal{A}$ is executable by the agent a.[2] A special action, nop (no operation) is always executable by every agent. It has no effect on fluents' values. Executability of actions is ruled by axioms of the form

$$\texttt{executable}\ x\ \texttt{if C}. \tag{5}$$

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

[2] Observe that the same action name x can be used for different actions executable by different agents. This does not cause ambiguity, because each agent knowledge is described by its own action theory.

conditions are considered in disjunction. The effects of an action execution are modeled through axioms (*dynamic causal laws*) of the form

$$x \texttt{ causes } \mathit{Eff} \texttt{ if } \mathit{Prec}. \tag{6}$$

where $x \in \mathcal{A}$, *Prec* is a constraint, and *Eff* is a conjunction of primitive constraints of the form $f = \texttt{FE}$, for $f \in \mathcal{F}$, and FE is a fluent expression. The axiom asserts that if *Prec* is `true` with respect to the current state, then *Eff* must hold after the execution of x.

Since agents share fluents, their actions may interfere and cause inconsistencies. A *conflict* happens when the effects of different concurrent actions are incompatible and would lead to an inconsistent state. A suitable procedure has to be applied to resolve a conflict and determine a consistent subset of the conflicting actions (see also Sect. 3.3). At least two perspectives can be followed, by assigning either a passive or an active role to the conflicting agents, during the conflict resolution phase. In the first case, a further actor is in charge of resolving the conflict, and all agents will adhere to its decision. Alternatively, agents themselves are in charge of reaching an agreement, possibly through negotiation. In case such last possibility is adopted, the following options allow one to specify in the action theories some basic reaction policies the agents might apply.

$$\texttt{action } x \ \mathit{OPT}. \tag{7}$$

where

$$\begin{aligned}
\mathit{OPT} ::= \ & \texttt{on_conflict } \mathit{OC} \ \mathit{OPT} \\
| \ & \texttt{on_failure } \mathit{OF} \ \mathit{OPT} \\
\mathit{OC} ::= \ & \texttt{retry_after } T \texttt{ [provided } C\texttt{]} \\
| \ & \texttt{forego [provided } C\texttt{]} \\
| \ & \texttt{arbitration} \\
\mathit{OF} ::= \ & \texttt{retry_after } T \texttt{ [if } C\texttt{]} \\
| \ & \texttt{replan [if } C\texttt{] [add_goal } C\texttt{]} \\
| \ & \texttt{fail [if } C\texttt{]}
\end{aligned}$$

Notice that in the same axiom one can specify policies to be adopted whenever a failure occurs in executing an action.

We remark here the difference between *conflict* and *failure*. The former, as mentioned above, occurs during the transition from a state to another, because of incoherent effects of concomitant actions. A failure occurs whenever an action x cannot be executed as planned by an agent a. (This might happen, for instance, because after the detection of a conflict involving x, the outcome of the conflict resolution phase requires x to be inhibited.) In this case the agent a might have to reconsider its plan. Hence reacting to a failure is a "local" activity the agent might perform after the state transition has been completed. In axioms of the form (7), one can specify different reactions to a conflict (resp. a failure) of the same action. Alternatives will be considered in their order of appearance.

The following example illustrates some specific cases of (7).

Example 1. Let us assume that the agents a and b have priority 0, while agent c has priority 2. Let us assume, moreover, that the current state is such that

actions `act_a`, `act_b`, and `act_c` are all executable (respectively, by agents a, b, and c), where their effects on fluent f are of setting it to 1, 2, and 3, respectively. Assume that the following options have been defined:

```
action act_a on_conflict retry_after 2
action act_b on_conflict forego
action act_c on_failure retry_after 3
```

and that the plan of agent a (resp., b, c) requires the execution of action `act_a` (resp., `act_b`, `act_c`) in the current state. Of course there is a conflict: the effects of concomitant execution of the three actions are incostintent. One possible conflict resolution procedure is that of focusing on higher priority agents. In the example at hand, this causes action `act_c` to be removed from execution list. Therefore agent c fails in executing its action and will react retrying to execute the same action after 3 steps.
Some policy must be now chosen to resolve the conflict between a and b. The first possibility is that agents have passive roles in conflict resolution, and a referee selects, according to some criteria, a (possibly maximal) consistent subset of the actions/agents. Assume a is selected (by simple lexicographical criteria)—then, it can set $f = 1$ and succeed, while b will get a failure message.
An alternative policy consists in not involving any referee and in making a and b in charge for resolving the conflict. In such a case, they will apply their `on_conflict` options. This causes a to retry the execution after 2 steps and b to forego. Both of them will get a failure message, because neither `act_a` nor `act_b` are executed. □

Apart from possible communication occurring among agents during the conflict resolution phase, other forms of "planned" communication can be modeled in an action theory. An axiom of this form

$$\texttt{request } C_1 \texttt{ if } C_2.$$

implicitly describes an action that allows an agent to broadcast a request to other agents. The action is executable if the precondition C_2 holds. By executing this action, an agents asks if there is another agent that can make the constraint C_1 true. Only an agent knowing all the fluents occurring in C_1 is allowed to answer to the request.

Instead of broadcasting an help request, an agent a can send such a message directly to another agent by providing its name:[3]

$$\texttt{request } C_1 \texttt{ to\_agent } a' \texttt{ if } C_2.$$

The following construct specifies a form of communication primitive that subsumes the previous ones:

$$\texttt{request } C_1 \texttt{[to\_agent } a' \texttt{] if } C_2 \texttt{ [offering } C_3 \texttt{].} \tag{8}$$

If the last option is used, the requesting agent also provides a "reward" by promising to ensure C_3 in case of acceptance of the proposal. Axioms of these types allow one to model bargains and transactions. Here is an example.

[3] Any request sent to a nonexistent agent will never receive an answer.

```
agent guitar_maker.       action make_guitar.
executable make_guitar if neck > 0 and strings >= 6 and
                          body > 0 and pickup > 0.

% actions for making two different kinds of guitars:
make_guitar causes guitars++ and neck-- and body-- and
                   strings = strings^-1 - 6 and pickup = pickup^-1 - 2
                   if pickup >= 2.
make_guitar causes guitars++ and neck-- and strings = strings^-1 - 6 and
                   body-- and pickup-- if pickup < 2.

% interaction with joiner:
request neck > 0 to_agent joiner if neck = 0.
request body > 0 to_agent joiner if body = 0.

% interaction with seller:
request strings > 5 to_agent seller if strings < 6
                    offering seller_account = seller_account^-1 + 8.
request pickup > 0 to_agent seller if pickup = 0
                   offering seller_account = seller_account^-1 + 60.

% the goal is to make 10 guitars:
goal guitars = 10.

% initially the maker owns some material:
initially guitars = 2 and body = 3 and
          neck = 5 and pickup = 6 and strings = 24.
```

Fig. 1. An action description in $\mathcal{B}^{\mathsf{AAC}}$ for a guitar maker agent

Example 2. Consider a situation where three agents exist: a guitar maker, a joiner that provides wooden parts of guitars (bodies and necks), and a seller that sells strings and pickups. For simplicity, we assume that the maker has plenty of money (so we do not take into account what he spends), that the seller wants to be paid for his materials, and that necks and bodies can be obtained for free (e.g., the joiner has a fixed salary paid by the maker). The money income of the seller is modeled by changes in the value of the fluent `seller_account`. In Fig. 1 we report an action description theory that models the agent `guitar_maker` (analogous theories can be formulated for the other two agents). Observe that two point-to-point interactions are modeled—namely, the one between the `guitar_maker` and the `joiner`, to obtain necks and bodies, and the one between the `guitar_maker` and the `seller`, to buy strings and pickups. Two kind of guitars can be made, differing in the number of pickups. □

We can inherit from [5] the capability of dealing with static causal laws and with cost constraints for actions and plans. Moreover, it is possible to allow fluent references of the form $f^t = \mathtt{FE}$, with $t > 0$ (a future value for the fluent f is "booked"). For simplicity, we do not consider these features in this paper.

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, for each agent

a, 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. For simplicity, we assume all the collections $\mathcal{I}_a$ as 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

$$\left\langle \bigwedge_{a\in\mathcal{G}} \mathcal{D}_a, \bigwedge_{a\in\mathcal{G}} \mathcal{I}_a, \bigwedge_{a\in\mathcal{G}} \mathcal{O}_a \right\rangle$$

The problem has a solution only if $\bigwedge_{a\in\mathcal{G}} \mathcal{O}_a$ characterizes a consistent state.

3 Semantics

The semantics of $\mathcal{B}^{\mathsf{AAC}}$ can be split into two parts: the semantics of the action description languages used locally by each agent, that do not consider the axioms (7) and (8), and the semantics of the overall system that deals with agents' interactions. Let us assume that there is an overall time limit `N`, within which the planning activities of all agents have to be completed.

3.1 Local semantics

As far as the local view is concerned, following [7], the semantics is given in terms of transition systems. Nodes (states) of the transition systems are uniquely characterized by assigning a value to each fluent. Two states u, v are linked if and only if there is an action applicable to u and leading to v. The formal semantics of the language B^{MV}, upon which $\mathcal{B}^{\mathsf{AAC}}$ is defined, is given in detail in [5].

A plan of an agent is a sequence of states $s_0, \ldots, s_{\mathsf{N}}$ such that s_i, s_{i+1} are linked in the transition system. Evaluation of a fluent expression f^t in a state is the value of the fluent in state s_{i+t}. Evaluation of a fluent expression f`@`i is the value of the fluent in state s_i. Evaluation of a constraint in a state s_i can be inductively defined in the natural way.

Each agent a looks for a sequence of states $s_0, \ldots, s_k$, with s_0 determined by $\mathcal{I}_a$ and $k \leq \mathsf{N}$ such that $\mathcal{O}_a$ are all satisfied in state k. As soon as the goal for agent a is satisfied, the agent succeeds and "exits" the system. In practice, we might assume that it always executes `nop` from time $k+1$ to time `N`. Observe, that the length of each agent's local-plan might change as steps are executed, because of the replanning phases the agent may perform, as a consequence of failures. Observe, moreover, that it suffices for the agent to reach the goal within `N` steps, i.e., its goal should hold at a time $k \leq \mathsf{N}$, but it does not need to hold at time $k+1$.

A remark here is needed on actions of the form `request` C_1 `if` C_2 in $\mathcal{D}_a$. The constraint C_2 is evaluated by the agent a in the state s_i. If C_2 holds then a is allowed to send a request for help in achieving C_1 (see details in Sect. 3.3). Let us focus here on the evaluation of the constraint C_1. If in a future instant,

say $j > i$, some other agent b accepts to fulfill the required condition, then b guarantees the satisfiability of C_1, as evaluated with respect to the j-th state s_j. Thus, fluents of the form f^t with $t < 0$, that are allowed in C_1, have to be considered by agent b w.r.t. the j-th state.

3.2 Concurrent plan execution

Agents are autonomous and develop their activities independently, except for the execution of the actions/plans. In executing their plans, the agents must take into account the effects of concurrent actions. A basic communication mechanism among agents is realized by exploiting a tuple space, and the accesses to the tuple space occur through Linda-like primitives [3]. Moreover, most of the interactions among concurrent agents (especially those aimed at resolving conflicts) are ruled by a specific process, a *supervisor*, that also provides a global timing for all agents enabling them to execute their actions synchronously.

More in general, the supervisor process stores the initial state and the changes caused by the successful executions of actions. It synchronizes the action executions and controls the coordination and the arbitration in case of conflicts. It also sends a success or a failure signal to each agent at each action execution attempt, together with the list of changes to its local state.

Let us describe how the execution of concurrent plans proceeds. As mentioned, each action description includes a collection of constraints describing a portion of the initial state.

Supervisor process

- At the very beginning the supervisor acquires the specification $\mathcal{I} = \bigcup_{a \in \mathcal{G}} \mathcal{I}_a$ of the initial state.
- At each time step the supervisor starts a new state transition:
  - Each active agent sends to the supervisor a request to perform an action (typically, next action of its locally computed plan), by specifying its effects on the (local) state.
  - The supervisor collects all these requests and starts an analysis, aimed at determining those subsets of actions/agents that conflicts (if any). There is a conflict whenever agents require incompatible assignments of values to fluents. The transition takes place once all conflicts have been resolved and a sub-collection of compatible actions has been identified by means of some fixed policy (see below). These actions are enabled while the remaining ones are inhibited.
  - Enabled actions are executed. This changes the current (global) state.
  - These changes are then sent back to all agents to make them update their local states. All agents are also notified about the outcome of the procedure. In particular, those agents requiring an inhibited action receive a failure message.
- The computation stops when time N is reached.

Notice that after each step of the local plan execution, each agent needs to check if the reached state still supports its subsequent planned actions. If not, the

agent has to reason locally and revise its plan (replan phase). The replanning is due to the fact that the reached state might be different from the expected one. This may occur in two cases:

1. The proposed action was inhibited, so the agent actually executed a `nop` (in this case it has received a failure notice from the supervisor).
2. Its interaction was successful, i.e., the planned action was executed, but the effects of actions of other agents affected fluents in its local state—for instance, an agent a assumed that fluent g held its value by inertia, but another agent b changed such value. There is no direct conflict between the actions of a and b, but agent a has to verify that the rest of its plan is still applicable (e.g., the next action in a's plan may have lost its executability condition).

3.3 Conflicts resolution

A conflict resolution procedure is invoked by the supervisor whenever it determines a subset of incompatible actions.

Different policies can be adopted in this phase and different roles can be played by the supervisor. First of all, the supervisor exploits priorities of agents to attempt a solution of the conflict, by inhibiting actions of lower priority agents. If this does not suffice, further options are applied. We describe here some of the easiest viable possibilities, that we have already implemented in our prototype. The architecture of the system is highly modular (cf. Sect. 3.6), and it can be easily extended by adding more complex policies and protocols.

The two approaches we implemented so far, differ by assigning the active role either to the supervisor or to the conflicting agents, in resolving the conflict.

1. The supervisor has the *active role*—it acts as a referee and decides, without any further interaction with the agents, which actions have to be inhibited. In the current prototype, the arbitration strategy is limited to
   - a random selection of a single action to be executed; or
   - the computation of a maximal set of compatible actions to be executed. This computation is done by solving a CSP (by generating at run-time a suitable CLP($\mathcal{FD}$) encoding).

   Note that, in this strategy, `on_conflict` policies assigned to actions by axioms (7) are ignored.
   Such a centralized way of resolving the conflicts might represent a critical point of the system, since all conflicting agents must wait for supervisor's decision. We describe, in what follows, a second approach that reduces such a dependence between agents and supervisor.
2. The supervisor just notifies the set of conflicting agents about the joint inconsistency of their actions. The set of agents involved in the conflict is completely in charge for resolving it by means of a negotiation phase. 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 (7) specified for its conflicting action x. The semantics of these directives are as follows (in all the cases `[provided` C`]` is an optional qualifier; if it is omitted it is interpreted as `provided true`):

- The option `on_conflict arbitration` causes the explicit invocation of the supervisor which performs an arbitration phase (involving all the conflicting agents) 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 (in other words we could say that a executes a `nop` in place of its action). Different strategies can be implemented in order to perform such a "search for help". A simple one is the round-robin policy described below, but, clearly, many other alternatives are possible and should be considered in completing the prototype.
- Similarly, 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.

Also the manner in which agents negotiate and exploit the `on_conflict` options can rely on several policies an protocols, of different complexity. For instance, one possibility moght be the election of a "leader" within each of the conflicting set S of agents. This agent is then in charge for coordinating the agents in S so to resolve the conflict without interacting with the supervisor; another possibility would consist in not identifying a privileged agent and in leaving each agent of S free to proceed and to find an agreement by sending proposals to other agents (possibly by adopting some order of execution, some priorities, etc.) and receiving theis proposals/answers. In the current prototype we implemented a round-robin policy. Such a rather rigid policy is just a simple example of how to realize an alwais terminating protocol for conflict resolution. Different solutions can be easily added to the prototype thanks to its modularity. The round-robin policy proceeds as follows. 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 they 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 the remaining agents complete the procedure. If the option does not yield success (e.g., the agents do not agree), then the next agent in the sequence will start its active role in the round, while a_i waits its next turn in round $j + 1$.
Notice that this procedure always ends with a solution to the conflict, since a finite number of `on_conflict` options are defined for each action.

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.[4] Moreover, further information might be sent to 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., the fact that one agent has to execute T consequent `nop`, etc.). This information has to be sent back to the interested agents to guide their replanning phases.

3.4 Failure policies

Agents receive a failure message from the supervisor whenever their requested actions have been inhibited. In such case, the original plan of the agent has to be revised to detect if the local goal can still be reached, possibly by replanning. Also in this case different approaches can be applied. For instance, one agent could avoid developing an entire plan at each step, but limit itself to produce a partial plan for the very next step. Alternatively, an agent could attempt to determine the "minimal" modifications to the existing plan in order to make it valid with respect to the new encountered state.[5]

In this replanning phase, the agent might exploit the `on_failure` options corresponding to the inhibited action. The intuitive semantics of these options can be described as follows.

- `retry_after` T `[if` C`]`: the agent first evaluates the constraint C; if C holds, then it executes T times the action `nop` and then tries again the failed action (provided that its executability and its preconditions still hold).
- `replan [if` C_1`] [add_goal` C_2`]`: the agent first evaluates C_1; if it holds, then in the following replanning phase the goal C_2 is added to the current local goal. The option `add_goal` C_2 is optional; if it is not present then nothing is 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. It quits and does not participate to the subsequent steps of the concurrent plan execution.
- If none of the above options is applicable, then the agent will proceed as if the option `replan if true` is present.

All the options declared for the inhibited action are considered in the given order, executing the first applicable one.

3.5 Broadcasting and direct requests

Let us describe a simple protocol for implementing point-to-point and broadcast communication between agents following an explicit request of the form (8). In particular, let us assume that the current state is the i-th one of the plan execution—hence, the supervisor is coordinating the transition to the $i + 1$-th

[4] Actually, the supervisor might provide some other information that might be useful for the agent. For instance, it detects if the effects of an action has been subsumed by other executed actions.

[5] At this time, the prototype includes only replanning from scratch at each step.

state by executing the $i+1$-th action of each local plan. The handling of requests is interleaved with the agent-supervisor interactions that realize plan execution—though, the supervisor does not intervene on it and the requests and offers are directly exchanged among agents. We can sketch the main steps involved in a state transition, from the point of view of an agent a, as follows:

(1) The agent a tries to execute its action and sends this information to the supervisor (as explained in Sect. 3.2).

(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 succeeded, before declaring accomplished the current transition, a starts an interaction with other agents to handle pending requests. During such interaction, the communication among agents relies on the Linda tuple-space (requests and offers are posted and retrieved by agents).

  (3.a) Agent a fetches the collection H of all the requests still pending and emitted until step i. For each request $h \in H$, a decides whether to accept the request for help from the agent b that sent the request 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. In the positive case, a posts its offer into the tuple-space and waits for a rendezvous with b.

  (3.b) Agent a checks whether there are answers to the requests it previously posted. For each request for which there are answers, a collects the set of offers/agents that expressed their willingness to help a. By using some strategy, a selects one of the responding agents, say b. The policy for choosing the responding agent can be programmed (e.g., by exploiting priorities, agent's knowledge on other agents, random selection, trust criteria, utility and optimality considerations, etc.). Once the choice has been made, a establishes a rendezvous with each responding agent and **(a)** declares its availability to b, **(b)** 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) The transition can then be considered completed for the 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 proceed with the execution of the next action in its local plan.

Note that we provided separated descriptions for steps (3.a) and (3.b). In a concrete implementation, these two steps have to be executed in an interleaved manner, to avoid that a fixed order in sending requests and offers causes deadlocks or starvation.

3.6 Implementation issues

A first prototype of the system has been implemented in SICStus Prolog, using the libraries `clpfd` for agents reasoning (by exploiting the interpreter for Action Description Languages described in [5]), `system`, `linda/server`, and `linda/client` for handling process communication. A server process is launched, generating the connection address that must be used by the client processes. This piece of information is stored in a text file, where a launching script (`runner`, available for both Linux and Windows) can also find the number of agents and the bound `N` on the maximum number of steps.

The system is structured in modules. Fig. 2 displays the modules composing the Prolog prototype and their dependencies. As far as the reasoning/planning module is concerned, we slightly modified the interpreter of [5] to accept the new syntax presented here (module `sicsplan` in Fig. 2). The modules `spaceServer` (through `lindaServer`) and `lindaClient` implement the interfaces with the Linda tuple-space. These modules support all the communications among agents. Each autonomous agent corresponds to an instance of the module `plan_executor`, which in turn relies on `sicsplan` 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`s act as linda-clients. Conflict resolution functionalities are provided to the modules `client`s 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 simply 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`.

4 Conclusions and future work

In this paper, we illustrate a preliminary design of an high-level action description language for the description of multi-agent domains. The language enables the description of agents with individual goals operating in a shared environments. The agents can explicitly interact (by requesting help from other agents in achieving their own goals) and implicitly cooperate in resolving conflicts that may arise during execution of their individual plans. The main features of the framework we described in this paper have been realized into an implementation, based on SICStus Prolog. The implementation is fully distributed, and uses Linda to enable communication among agents. Such a prototype is currenlty being refined and extended with further features.

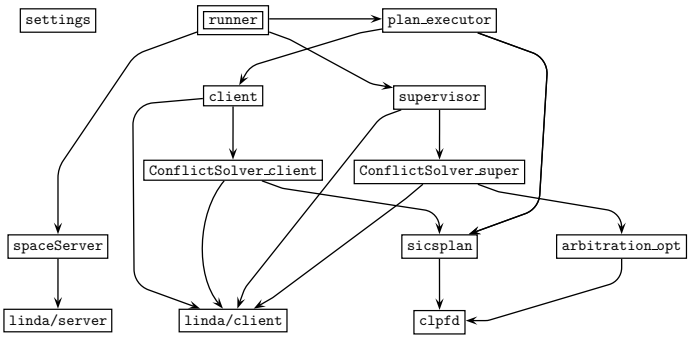

Fig. 2. The dependencies between modules in the system. The modules' names recall the corresponding Prolog-files names. The module `runner` is the starter of the application. The module `settings` specifies user options (policies, strategies, etc.) and the sources files containing the action descriptions, it is imported by all the others (we omitted drawing the corresponding arcs, as well as the nodes relative to less relevant SICStus libraries).

The work is preliminary but already shows strong potential and several avenues of research. The immediate goal in the improvement of the system consists in adding refined strategies and coordination mechanisms, involving for instance, payoff, trust, etc. Then, we intend to evaluate the performance and quality of the system in several multi-agent domains (e.g., game playing scenarios, modeling of auctions, and other domains requiring distributed planning). We also plan to investigate strategies to enhance performance by exploiting features provided by the constraint solving libraries of SICStus (e.g., the use of the table constraint [1]).

We will investigate the use of future references in the fluent constraints (as supported in 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 forms of negotiation with temporal references).

We will also explore the implementation of different strategies associated to conflict resolution; in particular, 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 (e.g., accepting to provide a property but failing to make it happen).

Also concerning trust evaluation, different approaches can be integrated in the system. For instance, a "controlling entity" (e.g., either the supervisor or a privileged/elected agent) could be in charge for assigning the "degree of trust" of each agents. Alternatively, each single agent could develop its own opinion on other agents' reliability, depending on the behaviour they manifested in past

interactions. Finally, work is needed to expand the framework to enable greater flexibility in several aspects, such as:

- in handling deadlines for requests—e.g., by allowing axioms of the form
$$\texttt{request}\ \ C_1\ \ \texttt{if}\ \ C_2\ \texttt{until}\ \ T$$
indicating that the request is valid only if accomplished within T time steps.
- in admitting dynamic changes in the knowledge the agents have on other agents (e.g., an action might make an agent aware of the existance of other agents; so, modifying the knowledge specified by axioms (2)), or on the world (e.g., an action might change the rights another agent has to access/modify some fluents; so, modifying the knowledge specified by axioms (3)).

References

[1] R. Barták and D. Toropila. Reformulating constraint models for classical planning. In D. Wilson and H. C. Lane, editors, *FLAIRS'08: Twenty-First International Florida Artificial Intelligence Research Society Conference*, pages 525–530. AAAI Press, 2008.

[2] R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak using Jason*. J. Wiley and Sons, 2007.

[3] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[4] A. Dovier, A. Formisano, and E. Pontelli. Representing Multi-Agent Planning in CLP. In E. Erdem, F. Lin, and T. Schaub, editors, *LPNMR 2009*, volume 5753 of *LNCS*, pages 423–429. Springer, 2009.

[5] A. Dovier, A. Formisano, and E. Pontelli. Multivalued action languages with constraints in CLP(FD). *Theory and Practice of Logic Programming*, 10(2):167–235, 2010.

[6] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about knowledge*. The MIT Press, 1995.

[7] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 2:193–210, 1998.

[8] J. Gerbrandy. Logics of propositional control. In *AAMAS*, pages 193–200, 2006.

[9] G. D. Giacomo, Y. Lespèrance, and H. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1–2), 2000.

[10] K. Hindriks and T. Roberti. GOAL as a Planning Formalism. In *MATES*, 2009.

[11] J. M. M. Dastani, F. Dignum. 3APL: A Programming Language for Cognitive Agents. *ERCIM News, European Research Consortium for Informatics and Mathematics*, 53, 2003.

[12] L. Sauro, J. Gerbrandy, W. van der Hoek, and M. Wooldridge. Reasoning about Action and Cooperation. In H. Nakashima, M. P. Wellman, G. Weiss, and P. Stone, editors, *AAMAS'06: Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 185–192. ACM, 2006.

[13] M. Spaan, G. Gordon, and N. Vlassis. Decentralized planning under uncertainty for teams of communicating agents. In *AAMAS*, pages 249–256, 2006.

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

[15] W. van der Hoek, W. Jamroga, and M. Wooldridge. A logic for strategic reasoning. In *AAMAS*, pages 157–164, 2005.