Teams of cognitive agents with leader: how to let them some autonomy.

Damien Devigne
LIFL - CRNS UMR 8022
Université de Lille 1
Villeneuve d'Ascq, F-59655 cedex
`devigne@lifl.fr`

Philippe Mathieu
LIFL - CRNS UMR 8022
Université de Lille 1
Villeneuve d'Ascq, F-59655 cedex
`mathieu@lifl.fr`

Jean-Christophe Routier
LIFL - CRNS UMR 8022
Université de Lille 1
Villeneuve d'Ascq, F-59655 cedex
`routier@lifl.fr`

Abstract- [1] **Most often situated multi-agent simulations, of which platform-games are an example, uses reactive agents. This approach has limitations as soon as complex behaviours are desired. For these reasons we propose an approach using cognitive agents. They have knowledge, objectives and are able to build plans in order to achieve their goals and then execute them.**

In this paper we particularly address the problem of teams of cognitive agents. We chose to build teams directed by a leader. One major problem is the building of the team plan and in particular one difficulty is to find the means in order to let autonomy to the team members. This can be done if the leader builds abstract plans. We present in this article a solution to this problem.

1 Introduction

Our work aims at producing agent-based simulations where the agents, situated in a geographical environment, behave "rationally". An application of such a work can be simulation platforms of which computer games are an instance, movies is another like "*The Lord of the Rings*" and the MASSIVE application illustrate it[2].

The main characteristics that we consider for these simulations are: first, the environment is defined by a geography, then the notions of positions or coordinates have a meaning, this is a critical feature since relative positions must be considered before executing an action and consequently moves must be performed; second, the world is dynamic (agents can appear or disappear), and then heavily non monotonic (ie. values, once known, can change); third, agents are different: they can perform and suffer actions that are different from one to the other; and last, the agents are embodied: they are situated in the environment, and have a partial perception of it, from this it follows that the agent's knowledge is incomplete, moreover because of the non monotonic nature of the environment, this knowledge can be wrong.

According to J. Laird, computer games constitute the "killer application" for human-level AI [LvL00]. The characters involved in video games, like FPS or role-playing games, have indeed to be perceived as autonomous entities with increasing realistic behaviours. They have to be *convincing*, thus their behaviour must comply with the rational expectations of their partner or opponent human players. They also need to adapt to new situations, acquire additional abilities throughout the game, etc. In addition, team strategies are also often useful. Some research has been done concerning agents and games [Nar00], and most of them concern reactive agents [Nar98].

But reactive agents, while effective in several cases, offer limited behaviours. Indeed their behaviours are "short term directed" and not "goal oriented". Their ability to perform some tasks depends on the immediate surroundings and is not the result of wilful acts. In current commercial games, too often the character's behaviours are reactive ones, coded using scripts based on trigger/action sets. This approach has limitations [Toz02]. First the obtained behaviours are rather limited and it is difficult to get deliberate group behaviours unless they hard-coded them. This leads to a second major problem: the software design concern. It appears to be very difficult to reuse parts of AI from one game to another: scripts are too much tied to game design.

Our proposition aims at offering cognitive, driven by goals, proactive agents. To use cognitive agents allows to obtain more abstract reasoning. From one simulation/game to another the cognitive behavioural engine stays the same even if the context changes, and the behavioural components, the *interactions*, can be at least partially reused. Our approach uses declarative knowledge and thus favours the separation between the game logic and code. This promotes reusability and should ease the development.

In a first part we describe how we design simulated worlds or game environments: the geography, the laws that rule the world (the interactions) and the cognitive agents and their behavioural engine. Then we discuss teams and present our proposition to obtain team plans.

2 Simulations with cognitive agents

We define a simulation as follows:

$$Simulation = E \times I \times A$$

where E is the topologic *environment*, I is the set of *interactions* that rule the simulated world and A is the set of *situated agents* involved in the simulation. In the following subsections we will quickly define these three points.

2.1 Environment

The environment describes the geography of the simulated "world". We represent the environment by a graph where nodes are *places* and vertices denote *path* from one place to another (see Table 1). A place is an elementary geographical area. The granularity of a place depends on the simulation,

[1]This work is supported by the CPER TAC of the *région Nord-Pas de Calais* and the FEDER europeand fund.

[2]see `http://www.massivesoftware.com/news.html`

the only constraint is that inside a place there is no restriction neither for moves, nor for perception (restrictions due to the other agents, like collision problems, are exempt). A place can represent a room, a town or any other part of the environment, and inside a place the position of an agent can be handled discretely or continuously depending on needs.

$$\begin{aligned} \textit{Environment} &= \{\textit{place*},\textit{path*}\} \\ \textit{path} &= (\textit{place}_{origin}, \textit{place}_{dest}, \textit{condition}) \end{aligned}$$

Table 1: Definition of environment

A path denotes an oriented transition between two places. It is defined by the places that it links, and a condition that must be satisfied if an agent wants to use this path (usually most of the conditions are simply *true*). The paths are oriented and the condition to go from some place a to a place b is not necessarily the same than the one to go from b to a. This formalism allows to describe, for example, that a door between two rooms must be opened if we want to go from one room to the other, or that an agent must be able to swim to cross a river between two fields. In this last case, our approach allows, depending on needs, to choose to represent or not the river with a place. It depends on whether ot not the crossing of the river has a meaning in the simulation (see figure 1).

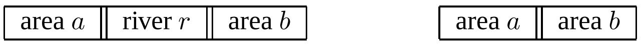

Figure 1: Left: river is modelled. Paths are: (a,r,*"agent can swim"*), (r,a,*true*), (b,r,*"agent can swim"*), (r,b,*true*) Right: river is not modelled. The paths are: (a,b,*"agent can swim"*),(b,a,*"agent can swim"*).

2.2 Interactions

Knowledge representation is based on the notion of what we call "*interactions*"[MPR03]. Since the objective is to achieve simulations (like games are), it is necessary to represent the laws that rule the simulated world and to allow the agents to manipulate these as knowledge elements. We introduce our interactions in this goal.

Interactions are the backbone of our simulation model. They are at the basis of the knowledge representation in the simulation. Some agents (the actors) can perform interactions and others (possibly the same) can suffer from them (the targets).

An interaction is defined by a name and three parts:

- a *condition*, it tests the current context of execution of the interaction and consists mainly of tests on values of target or actor properties.

- a *guard*, it checks general conditions for the interaction applicability, typically it defines that to be fired an interaction requires that the distance between the target and the actor must be less than some given value.
  The guard is separated from the condition since it corresponds to the knowledge due to the geographically situated feature of the simulations. In a non situated context, one would have only the condition and action parts. The guard will be at the origin of the moves in the plan, moves that are indeed specific to situated problems. We do not express explicitly in an interaction that the agent has a move to do in order to fire it, we want the agent to plan it when required.

- an *action*, it describes the consequence of the interaction, it can be a change in the state of the actor and/or of the target (ie. a change of the value of a property), and/or the activation of an environment action (like the creation of an agent).

By example, to *open* an object (door, chest, window, etc.) makes it changing from *closed* state to *opened* one. The nature of the target is of no importance here (insofar as it can suffer *open*), this knowledge can then be represented in a "universal" way by the interaction (see below). In this sense, interactions are declarative knowledge: they describe an action and not how to solve/use it. Let us remark that there is no mention of moves to be performed to fire the interaction, the knowledge due to the situated property is mentioned in the guard. An interaction is a piece of *abstract* knowledge where the situated point of view is taken into account in the guard.

$$\texttt{open:} \left\{ \begin{array}{rcl} \texttt{condition} &=& \text{“}\texttt{target.opened} = \texttt{false}\text{”} \\ \texttt{guard} &=& \text{“}\texttt{distance}(\texttt{actor}, \texttt{target}) < 1\text{”} \\ \texttt{action} &=& \text{“}\texttt{target.opened} = \texttt{true}\text{”} \end{array} \right.$$

One advantage in using such interactions is that this approach favours a good software engineering design. Since interactions are not tied to a particular agent nor to a simulated world, they can be reused from one to another. This is clearly the case with the above *open* interaction. Reusability is of course an important concern in software engineering design and in particular in AI game design where it is reputed to be not applied although wished.

To increase the generic nature of our interactions we propose a way to specialize them. This is not the object of this paper to detail it but let us say that the aim is to keep the declarative and abstract nature while taking into account the fact that to solve a given abstract action can require different conditions. To solve that we use something like inheritance of interactions. Using an example should help to present it shortly: again consider the *open* interaction, we said that it can be applied to different types of targets and used in a plan such that "*to fetch an apple in the next room I must* open *this door*". Now let us assume that this door is a lockable one (and is indeed locked). From an abstract point of view the plan is still valid, but the *open* interaction must be understood as "make the door change from *closed* to *opened* state *when it is unlocked*". The problem is then how the same abstract plan (open the door to fetch the apple) can receive different solutions (just "open" or "unlock and then open") depending on the target (whether it is lockable or not). Our solution is to allow to specialize interactions by adding extra conditions, thus you create another *open* interaction that "inherits" the previous one and to which you add the condition `target.isLocked=true`. This is this version that is given as *can-suffer* interaction to the lockable agents.

2.3 Agents

Our agents are embodied agents that are situated in their environment, they are influenced by it and more precisely by where they are in it.

We distinguish two kinds of agents: inanimate and animate agents (not to mistake them with mobile/non-mobile agents). Both are defined by properties and can suffer interactions but the latter can also perform interactions and have a behavioural engine (see Table 2). The animate agents are cognitive and proactive agents. They are responsible for the dynamics of the simulations. The interactions that an agent can perform correspond to its abilities.

Agents	=	*Animate* | *Inanimate*
Inanimate	=	{*Properties**, can-suffer}
Properties	=	*(name, value)*
can-suffer	=	*Interaction-name**
Animate	=	*Inanimate* ∪ can-perform ∪ *brain* ∪ *goals*
can-perform	=	*Interaction-name**
brain	=	*planning engine* ∪ *memory*
goals	=	*interaction-goal* | *condition-goal*

Table 2: Agent's definition

The cognitive agents The structure of the animate agent's "mind" is presented in Figure 2. Agents have a memory that can be seen like a "degraded environment". This one represents the knowledge base for all the information gathered by the agent concerning the environment: the topology of the environment, the other agents (their position and state). This information is used by the planning engine to determine the action that the agent must try to execute in the environment to fulfil its goal. A perception module is used to pick up information in the environment and to update the memory, this perception is local. Updates are performed by a separate module that has an influence on the planning engine in order to adapt the currently computed plan to the new perceived situation. This last module is a kind of "short term memory". From this it results that the knowledge of an agent is not complete, then an agent may have to search for unknown information, and can be wrong, but the agent is supposed to behave with respect to its knowledge. This is due to the dynamic and non monotonic nature of the environment. Knowing that its environment is non monotonic must be taken into account by the agent.

Goals Animate agents have goals. The satisfaction of these goals leads the agents to behave according to a computed plan. There exist two kind of goals. First, the *interaction-goals*, they correspond to an (inter)action that the agent has to execute. The target of this interaction can be less or more precisely given: from a named agent to any agent that can be the target of the interaction, as shown in the next table:

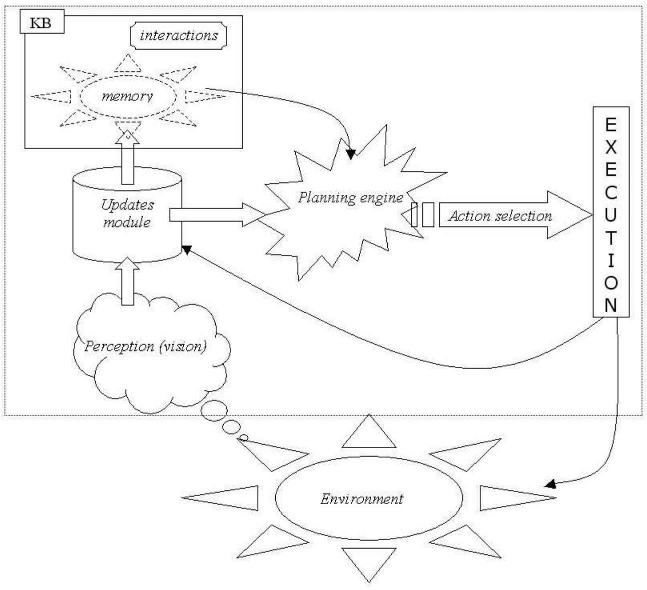

Figure 2: Different elements of agent's mind.

goal	type of target
`eat(apple_12)`	a given named apple
`eat(an apple)`	any apple
`eat(*)`	any eatable (ie. "who can-suffer eat") agent

Second, the *premiss-goals* (or *condition-goals*), they correspond to a condition that the agent wants to become true. For example:

`actor.energy > 100`:
"having his energy being greater than 100"

Planning engine In order to achieve their behaviour the animate agents have an engine that computes plans in order to satisfy the goals given to them. The plan is produced by a backward chaining on the *can-perform* interactions of the agents. The plan building depends on the information stated in the memory (ie. the beliefs base) of the agent. In a rather classical way, the plan can be viewed as a tree (see Figure 3). The nodes are made of the different goals and subgoals encountered during the resolution. Some are *interaction-goals*, others are *premiss-goals*. Thus the tree is an alternation of condition and interaction nodes and corresponds to an AND-OR tree. AND-nodes correspond to condition-nodes (for condition-goals) and OR-nodes to interaction-nodes (for interaction-goals).

The condition and interaction nodes are classical case. The sons of a condition node are interaction-nodes built from the interactions whose action part offers a way to satisfy the condition (or help to satisfy it). The interaction-node's sons are built from the conditions that can be found in the condition and guard parts of the interaction: from these, condition-nodes are built. This is classical in backward chaining.

We want to underline a point that introduces differences in comparison with planning in non situated context. Indeed, since we consider embodied agents situated in a geographical environment and since we want to simulate their behaviour in such an environment, agents must perform moves. Typically an agent must move next to a target to interact with it. It is here that the guards that we have introduced in our interactions play their roles. To be allowed to fire the action part of the interaction, the agent must satisfy the conditions and the guards. But guards, since they imply moves that are a crucial side-effect in situated simulations, require specific consideration. We have discussed this problem in [DMR04] where we show in which ways the situated context has an influence on “planning while executing”. Indeed, considering only conditions and actions leads to *abstract plans* that are valid independently of any situated context: “to open a door only requires that it is closed and makes it opened”. However, simulations in situated environment implies that in the *execution plan* moves must be done, and then “*to open a door*” requires also the actor being next to the door as expressed in the guard of the *open* interaction. This guard must be considered while building the plan. The backward chaining on the guard produces the moves and these moves can require specific planning in order to be achieved. The conditions that must be satisfied in order to be able to perform a move are the conditions that exist between the places in the computed path. Then the same *abstract plan* can lead to several *execution plans* each depending on the execution context where the agent is situated.

Let us just add that the plan is not rebuild at each step but partial replanning is done according to the new perceived information given by the updates module.

A small example To illustrate the different points described in the previous paragraph we will consider a very small and simple example. We consider a world with two places/rooms separated by a door d, the path between these two rooms has the condition “d*.locked=false*”. Four interactions define the laws: *unlock*, *take*, *move*, *push* (see Table 3). In the world are an animate agent a that can perform these 4 interactions, and three inanimate agents, the door d that can suffer *open*, a key k that can suffer *take* (and can be used to unlock d) and a button b that can suffer *push*. The goal of the agent is to push on b. The figure 3 presents the planning in two different situations.

unlock:	*condition*	=	“`target.locked` = `true`”
	guard	=	“`distance(actor, target)` < 1”
	action	=	“`target.locked` = `false`”
take:	*condition*	=	`true`
	guard	=	“`distance(actor, target)` < 1”
	action	=	“`actor.own(target)` = `true`”
push:	*condition*	=	`true`
	guard	=	“`distance(actor, target)` < 1”
	action	=	“`target.pushed` = `true`”
move:	*condition*	=	`conditionsfoundinpath`
	guard	=	
	action	=	“`distance(actor, target)` < 1”

Table 3: Definitions of the interactions (adapted - but without distortion - to shorten the example)

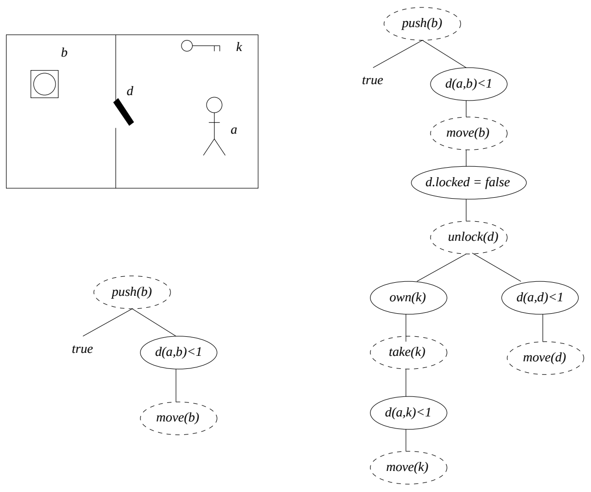

Figure 3: An animate agent a is situated in an environment where are also 3 inanimate agents, a door d, a button b and a key k. The goal of a is to push b. a must build a plan to achieve it. A plan can be drawn as a tree, nodes due interaction-goals are drawn with dashed lines and nodes due to condition-goals with solid lines. Depending on the execution context, different plans can be obtained. Left is the tree obtained when d is not locked and right is the case where d is locked. a must adapt its plan to the context.

3 Teams

In the previous section we have briefly described our approach to model the simulated world and the agents. In particular we present how we obtain individual agent behaviour that allows agents to achieve tasks. However individual behaviours are not enough, sometimes tasks must be done by groups, or teams, of agents. In computer games the need of teams is important: teams of fighters in FSP games, groups of units in strategy games, teams of characters in role-player games, etc.

Several situations can require the use of teams. First, getting a number of agents to do a job can speed up its achievement, this corresponds to task parallelization when several agents have the same abilities and perform similar tasks simultaneously, one agent could have done it alone but it would have taken more time. Second, in some cases, one agent is not sufficient to perform a task, and several must cooperate *simultaneously* to do it, by example this is the case when a heavy load must be carried and two or more agents are required in order to lift it, of course they must do this simultaneously. Third, complex tasks require a lot of different abilities and it is not often the case that one agent alone has all of them, then several “specialist agents” that together gather these abilities must cooperate to achieve the task. Of course, these three cases can mix. In this paper we mainly address this last case.

3.1 Teams of cognitive agents with leader

Making teams of agents work has been the subject of several approaches. Emergence is a solution to obtain a team behaviour [CGGG03]. But in this case we think that the notion of team work is only "apparent" since the team behaviour is a collateral effect of the sum of the individual behaviours and is not deliberate. We mean that the agents are not conscious that they work in a team and no team strategy is explicitly planned.

Our objective is to make our cognitive agents work in a team and being aware of it. To perform this we make some choices in this paper:

1. the team is assumed to be already created, that means that we are not concerned here with the problem of recruiting an able agent or constituting the team before doing the job.
2. the team is directed by a leader which is an agent that plays a particular role in the team. It is known at the beginning.
3. the leader is in charge of the team strategy and coordination, therefore our work does not consider the cases where agents negotiates to cooperate and to find an agreement on a plan. For example, plan merging [KMS98, AFH$^+$97] is not our interest here.

Having a leader that builds the team plan can be seen as a restriction since this implies that control is partially centralized. However one can see by oneself that this is often the case in real life: firemen in a squadron or workers in a building site obey to the orders of their leader. The point is that in such teams, even if the leader gives orders, team members still have their autonomy. They must behave according to the leader plan but have to use their knowledge and abilities to achieve these orders.

Indeed, an important feature is the granularity of the leader orders. A site foreman does not order a bricklayer "*take this red brick, bring it there and put it on the foundation, then take this second one, bring it there and put it next to the first one, then take this third one...*". His order is simply "*build this brick wall here*". How the wall is built is the responsibility and the competence field of the bricklayer. As we see here, the leader gives rather high level orders and is not concerned with details. Moreover these orders can be abstract insofar as they are not necessarily tied to a particular situated context: the foreman can use the plan of construction in his office to show the bricklayer the walls to be built, this one is in charge to do it according to the site constraints and situation. The worker is autonomous once the order has been given, probably he only must report when he succeeds or even informs his leader when a problem he cannot solve occurs.

Therefore, the leader is in charge to build the plan that solves the team goal but this plan does not describe the solution in full details.

In the following (see paragraph 3.3) we propose a solution to reproduce this behaviour: the leader has the knowledge about its team members abilities, it builds an abstract plan to solve the team goal and it distributes orders to the members. The members autonomously resolve their tasks and report to the leader.

3.2 Description of a team

Describing a team simply as a group of agents is not sufficient. Our proposition consists in describing the team structure independently of any concrete agent and then to instantiate it with the members.

Since we are interested in teams that gather several complementary specialists, we design the team structure in term of roles. A role corresponds to a set of abilities (ie. interactions) required to play it (see Table 4). We add a cardinality to each role, this allows to precise when several agents of the same type are required in a team. This information is for example useful to handle dynamic reorganization of the team, but we will no more use it in the following of this paper.

To instantiate a team consists in selecting existing able agents and to attribute them some role in the team. Of course to be able to play a role an agent must have all the interactions that describe it in its *can-perform* property. Then a team is given by a team-structure and a mapping from the role in the structure and the agents members of the team.

TeamStructure	=	*(Role, Cardinality)**
Role	=	*Interaction**
Cardinality	=	*Natural..Natural*
Team	=	*TeamStructure* $\times$ *(Role,Agent)**

Table 4: Definition of team

The knowledge concerning the team is given to the team leader. In this paper we are not interesting in how the leader recruits its team-mates.

3.3 Our proposition

Giving autonomy to the team members is an important point. First, as said before, this is more realistic and simulates what happens in real life. Second, this avoids to obtain stupid behaviours, in particular because we consider situated agents in dynamic environments like game worlds are. One must not forget that, as we say earlier, agent's knowledge, and by way of consequence the leader's knowledge, is not necessarily correct. Then, in the case where the leader builds the plan in every detail and gives very precise orders to the team members, those having no right to modify them, it is more than probable that members will be confronted with unexpected situations and will not be able (nor authorized actually) to handle them. As an example, such situations can be due to objects that are not where they are supposed to be. Then a precise order like "*go to a given precise location and take the brick*" can not be solved by a non autonomous agent if the brick has been moved. This is not the case with the more abstract order "*take the brick*" given to an autonomous agent that can decide and plan how to find the brick according to the environment it is confronted with. As a third advantage this provides an easy way to consider team of teams, we will discuss this later in the paper.

As it has been described earlier our agents are cognitive ones and are able to build plan according to their knowledge. Insofar as individual and team plans are of the same nature, there is no reason that the individual planning strategy could not be applied to team planning.

So, the problem that arises is: *How to adapt the individual planning to team work in order to let some team members autonomy*. Here follows our proposition.

As we have seen the plan can be viewed as a tree where root is the goal and leaves are actions to be executed in order to solve the goal. In a team plan, leaves are then the orders that the leader gives to the members. To be able to build this plan, the leader must have some knowledge concerning the members abilities, that is about their *can-perform* interactions.

Let us consider that the leader knows all the *can-perform* interactions of the members. Then exactly like in the individual planning, he could build a plan to solve the goal. But in this case he would build a full plan and team members will no more have any planning autonomy! So the solution is to not let the leader plans "until the end", but to limit the tree depth. But this can not be done arbitrarily. There is no reason to decide that the leader unfolds the tree until it has a depth of 3 rather than 5 or 10. The appropriate depth will depend on the goal and the members abilities, it will be different at every time. In order to compute the depth's limit the leader would have to compute the full plan before to cut it at a relevant depth. It is a nonsense to compute the full plan, then to forget it and ask the members to re-build it!

Therefore this approach is not correct. We must not forget that we want the plan built by the leader to be abstract. And then the leader does not need to have explicitly all the knowledge about the *can-perform* of its team members. Actually, the leader only has to know what high-level tasks the members are able to do. It even does not need to know by itself how to perform these tasks, like a foreman does not necessarily have to know how to build the wall, it suffices that he knows that the bricklayer is able to do it. Indeed, the leader has to know what things must be done but not necessarily how to do them.

To achieve this we propose to hide some knowledge to the leader: the guards and conditions in the *can-perform* of the members are hidden to the leader. Thus the leader can know which of the member's interactions can be used in its plan since, knowing their action parts, it can use them during its backward chaining. However, since the leader knows no condition (nor guard) for these, it considers they are satisfied and stops the chaining. Then these interactions become necessarily leaves of the leader tree plan and can be distributed to the members as goals. Those members, having full knowledge, are able to make the appropriate plan.

Let us take an example. We have one agent named a_0 who can perform some interaction I_0 (see Table 5). Two other agents, named a_1 and a_2 can respectively perform interactions $\{I_1, I_3, I_4\}$ and $\{I_2, I_5\}$ (see Table 6).

a_0 will be the leader of the team. The team structure is made of two roles r_1 and r_2 that are defined respectively by interactions $\{I_1, I_3\}$ and $\{I_2\}$. This implies that the "high level" tasks the leader can ask to the members are to satisfy p_1, p_2 or p_3. The conditions and guards of these interactions are hidden to a_0 (see Table 5).

	leader.can-perform	team.can-perform		
name	I_0	I_1	I_2	I_3
conditions	p_1, p_2	–	–	–
guard	true	–	–	–
actions	p_0	p_1	p_2	p_3

Table 5: Leader's knowledge, conditions and guards are hidden for the interactions of the team roles. They are considered as *true*.

As we can see, agents a_1 and a_2 can play roles r_1 and r_2 respectively, they are chosen as team members.

name	I_1	I_2	I_3	I_4	I_5
conditions	p_3, p_4	p_5	true	true	true
guard	G_1	G_2	G_3	G_4	G_5
actions	p_1	p_2	p_3	p_4	p_5

Table 6: Definitions of *can-perform* interactions of team members.

Now, the team is given the goal p_0. The leader, a_0 builds the plan for it. According to its knowledge, the backward chaining leads to use interaction I_1[3] that requires p_1 and p_2 to be solved. These lead respectively to the use of I_1 and I_2 (in this plan I_3 does not interfere from the leader point of view). For the leader, I_1 and I_2 have their conditions and guards satisfied (since they are hidden and seem to have none), then it stops its planning here (see Figure 4).

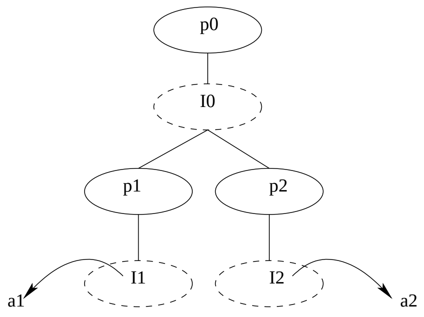

Figure 4: Tree representing the plan built by the leader. It is developed until the abstract team's interactions are reached, goals can then be given to the team members.

Now the leader can distribute the tasks to its team members according to their role in the team, indeed the leader is not able to perform the task itself since I_1 and I_2 are not in its *can-perform*. Then it gives p_1 to a_1 as goal and p_2 to a_2. Now each agent autonomously solves its goal according to its knowledge and builds the appropriate plan (see Figure 5). Then it can determine which action to fire to solve its goal and can inform its leader when it succeeds or if it fails.

The importance of the autonomy of the agent is increased while considering the influence of the surrounding environment and specially with the situated aspect. This is expressed within the guards whose resolution has not been de-

[3]In the following we assume condition p_i not to be satisfied.

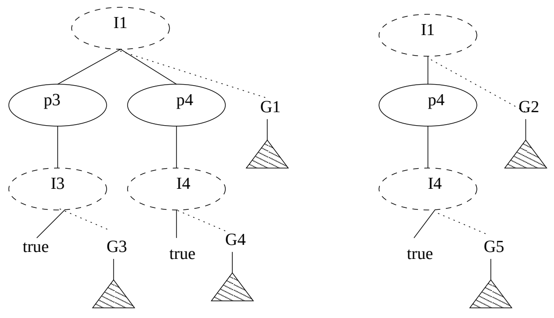

Figure 5: Trees representing the individual plans built autonomously by the 2 team members (guard's plan are not detailed, they depend on the context).

tailed in above trees. Indeed, as we have seen since guards express that the actor must be near the target in order to fire the interaction, they requires planning to be solved. But this is highly context dependant. And it would have been particularly irrelevant for the leader to plan it instead of the members.

3.4 Team of teams

With our above described approach it is easy to build "team of teams", or team built as a hierarchy of agents, where one "big leader" orders to subleaders that order to and so on, until "basic members". In fact it applies immediately to such cases without change!

Indeed, each level of the hierarchy corresponds to a level of decision with its type of orders. The higher in the hierarchy an agent is, the more high-level or abstract its orders are. For example, a works foreman can order a bricklayer leader to have walls built and the carpenter leader to have windows installed. Each of them orders to his team-mates to do the appropriate work.

With our approach the leader's planning stops with the abstract orders that can be given to team members and become a goal for them. If a team member agent is a leader itself, it applies the same procedure: starting from the goal that he has received, he builds a plan that stops when the abstract knowledge of its team (ie. interactions where conditions are hidden) is used. Then it can give orders to its team members. As we note nothing particular has to be done in order to consider hierarchies of teams: building the teams with their knowledge suffices.

4 Conclusion

In this paper we propose a mean to handle teams of cognitive situated agents directed by a leader. The presented solution let the team members some autonomy in the way they contribute to the team plan achievement. Indeed, the leader uses abstract knowledge on team's abilities to build an abstract plan and then distributes high-level orders to its team-mates.

Several problems have not been addressed in this paper and require complementary works, among them let us cite:

- how is the team built, that is how the leader recruits its team-mates?
- how to dynamically reorganize a team when an agent leave it (the cardinality information that are just mentioned in the paper can be used here)?
- how the information are exchanged inside the team?
- how to proceed in the case of teams with no leader?

They are, with others, the subjects of future works.

Thanks Many thanks to the anonymous referee that corrects so many of our language mistakes. Thanks to all referees for their remarks too.

Bibliography

[AFH+97] R. Alami, S. Fleury, M. Herrb, F. Ingrand, and S. Qutub. Operating a large fleet of mobile robots using the plan-merging paradigm. In *Proceedings of IEEE ICRA 97*, 1997.

[CGGG03] D. Capera, J-P. Georgé, M-P. Gleizes, and P. Glize. The amas theory for complex problem solving based onself-organizing cooperative agents. In *Proceedings of WETICE'03*, pages 383–388, 2003.

[DMR04] D. Devigne, P. Mathieu, and J-C. Routier. Planning for spatially situated agents in simulations. In *Proceedings of the 2004 IEEE/WIC/ACM International Joint Conference IAT'04*, 2004.

[KMS98] Subbarao Kambhampati, Amol Mali, and Biplav Srivastava. Hybrid planning for partially hierarchical domains. In *Proceedings of the 15th national/10th conference on Artificial Intelligence/Innovative applications of artificial intelligence*, pages 882–888. AAAI, 1998.

[LvL00] John E. Laird and Michael van Lent. Human-level AI's Killer Application: Interactive Computer Games. 2000.

[MPR03] P. Mathieu, S. Picault, and J-C. Routier. Simulation de comportements pour agents rationnels situés. In *Actes des Secondes Journées Francophones MFI'03*, pages 277–282, mai 2003.

[Nar98] A. Nareyek. Specification and development of reactive systems. In *1998 AIPS Workshop*, pages 7–14, Menlo Park California, 1998. AAAI Press.

[Nar00] A. Nareyek. Intelligent agents for computer games. In *Computers and Games, Second International Conference, CG 2000. LNCS 2063.*, pages 414–422, 2000.

[Toz02] Paul Tozour. The perils of ai scripting. In *AI Game Programming Wisdom*, pages 541–547, 2002.