Being reactive by exchanging roles: an empirical study

Olivier Sigaud[1] and Pierre Gérard[1,2]

[1] Dassault Aviation, DGT/DPR/ESA
78, Quai Marcel Dassault, 92552 St-Cloud Cedex
[2] AnimatLab-LIP6, 8, rue du capitaine Scott, 75015 PARIS
olivier.sigaud@dassault-aviation.fr pierre.gerard@lip6.fr

Abstract. In the multi-agent community, the need for social deliberation appears contradictory with the need for reactivity. In this paper, we try to show that we can draw the benefits of both being reactive and being socially organized thanks to what we call "social reactivity".
In order to defend this claim, we describe a simulation experiment in which several sheepdog agents have to coordinate their effort to drive a flock of ducks towards a goal area. We implement reactive controllers for agents in the Classifier Systems formalism and we compare the performance of purely reactive, solipsistic agents which are coordinated implicitly with the performance of agents using roles. We show that our role-based agents perform better than the solipsistic ones, but because of constraints on the roles of the agents, the solipsistic controllers are more robust and more opportunistic. Then we show that, by exchanging reactively their roles, a process which can be seen as implementing a form of social deliberation, role-based agents finally outperform the solipsistic ones. Since designing by hand the rules for exchanging the roles proved difficult, we conclude by advocating the necessity of tackling the problem of letting the agents learn their own role exchange processes.

1 Introduction

Defining reactivity is difficult since the word has several meanings which are closely related but not exactly equivalent. According to [Kaelbing, 1990], reactivity is a matter of responsiveness in time. In order to be reactive, an agent must do the right thing at the right time. This view of reactivity has been one of the early leitmotives of the rising field of behavior-based artificial intelligence [Brooks, 1991,Matarić, 1994a] in reaction to the endless planning processes used in classical artificial intelligence robotics. Since the lack of reactivity of the planning robots prevented them from being used in dynamic environments, it was claimed that doing the right thing was pointless if it was not done in time.

Another definition of reactivity comes from a more formal background. In the framework of Markov Decision Processes (MDP), the *Markov hypothesis* holds when having any information about the past experience does not help an agent to adopt a better behavior at the current time step. If the Markov hypothesis holds, the problem faced by the agent is said to be a Markov problem. In this framework, an agent is said reactive if it selects its action according to its present situation without using any memory of the past. In a Markov problem, a reactive agent can act optimally. This view of reactivity, clearly presented in [Colombetti and Dorigo, 1993], is widespread in the reinforcement learning community. In this volume, [Bouzid et al., 2001] and [Riedmiller et al., 2001] present a framework relying on that formalism.

Both views of reactivity differ in the fact that the second one does not imply anything about the time that the agent spends choosing its current action. But they are closely related since an agent which makes no use of its memory cannot make any prediction about the future. Hence, a reactive agent in the second sense does not spend any time in planning, it only reacts to its current situation. Since this notion of reactivity is very restrictive, it should allow any agent which verify it to be reactive in the first sense, *i.e.* to decide what to do very fast, then to react in time to events in its environment.

Now, if an agent is reactive in the second sense, it can be proven formally that there are situations where it will not be able to adopt an optimal behavior. These problems are called *non-Markov problems*. Each time the Markov hypothesis does not hold, relying only on the current perception does not allow to select the best action. Hence it is clear that adopting a reactive behavior means selecting short-sighted actions, which may not be suitable when long-term strategies are necessary.

Collective tasks are full of such situations where looking for an immediate reward or pursuing an immediate goal is not the best thing that an agent may do. For instance, in the ROBOCUP domain [Asada and Kitano, 1999], if an agent has the ball and is close to the goal, it may shoot reactively even if an opponent is likely to catch the ball, whereas it might be more appropriate to give the ball to a teammate who is better located. The second behavior could be seen as deliberative rather than reactive because it seems to imply that the agent knows that after it passes the ball, its teammate will shoot and score. This example seems to support the view according to which deliberative social behaviors and reactivity are two opposite requirements which should be balanced in a multi-agent architecture.

In this paper we want to challenge this view. We will show through an empirical study that social behaviors can be as reactive as solipsistic behaviors. In our previous example, giving the ball to a teammate is a behavior which can be fired as reactively as shooting to the goal. The fact that giving the ball allows to score in the long term does not imply that social agents have to plan in order to find that such a behavior is more efficient than merely shooting. Our purpose is to show that giving roles to the agents and applying reinforcement learning schemes that take into account long-term rewards allows them to adopt some behaviors which an external observer would consider as exhibiting social deliberation abilities, whereas these behaviors are implemented reactively. In particular, we will show that giving to the agents the ability to exchange their roles is both something which helps finding better strategies and something which can be done straight-forwardly.

The paper is organized as follows. In the next section, we describe our simulated problem and the multi-agent strategy we designed to solve it. In section 3, we present the Learning Classifier Systems (LCS) [1] framework and how we used it in order to implement the controller of our agents. In section 4, we present the solipsistic controller which we designed and some obvious drawbacks of this design. In section 5, we show how our first hand-crafted controller was significantly improved by an explicit use of roles, resulting in a new architecture involving a set of behaviors devoted to the fulfillment of each role, and we present the benefits which can be drawn from such an architecture. In section 6, we compare the results obtained with both hand-crafted controllers through a first empirical study. In particular, we compare their robustness to changes in parameters of the simulation. In section 7, we discuss these results and show that the lack of robustness of the role-based controllers is due to the incapability of the agents to exchange their roles. Then we present further evidence for the necessity of letting the agents exchange their roles. In section 8, we present a new role-based controller which takes this necessity into account and show that the robustness problem is solved. In section 9, we discuss our architecture from a more multi-agent oriented stance, and highlight what would be necessary to apply it to more complicated problems. Since designing by hand the rules for exchanging the roles proved difficult, we conclude in section 10 by advocating the necessity of tackling the problem of letting the agents learn their own role exchange processes.

2 The problem and its representation

The necessity of having good benchmarks to test and compare algorithms and architectures is now central in the multi-agent research community. The ROBOCUP [Asada and Kitano, 1999] is such a benchmark seeming both general and complicated enough to act as a representative testbed for the entire field. In this volume, [Behnke and Rojas, 2001] and [Bredenfeld and Kobialka, 2001] illustrate their concepts in the ROBOCUP domain. But, if one uses machine learning techniques and adaptive capabilities to solve the complete task, the problem seems too difficult. In the particular case of reinforcement learning techniques, the agents do not get enough feedback to learn everything from scratch. The researchers may either use these techniques at one particular level of the game, or use them to solve particular subtasks (for instance, the pass to a teammate [Asada et al., 1999]).

Therefore, the tendency in adaptive multi-agent simulations is to study much simpler application domains. The Prey/Predator pursuit domain involving several predators [Stone and Veloso, 1997] is such a benchmark and illustrates this trend. But in these latter cases, the problem is often oversimplified: the agents move in a grid-world, they have few possible actions. Hence, the problem lacks

[1] In order to make clear that we sometimes use the Classifier Systems formalism without applying learning algorithms, we will distinguish Classifier Systems (CS) as a formalism and Learning Classifier Systems (LCS) as a technique throughout this paper.

the continuous dynamics characterizing most industrial applications. Since our focus is on adaptive techniques and we have industrial applications in mind, we have chosen to design an original application which appears as a good compromise between the too complex ROBOCUP problem and the oversimplified Prey/Predator problem. We draw inspiration from [Vaughan et al., 1998], who have presented the Robot Sheepdog Project, involving a robot driving a flock of ducks towards a goal position. The algorithm controlling the robot was first tested in simulation and then implemented on a real robot driving a real flock of ducks.

In this paper, we present a simulated extension of the task to the case where several agents share the goal mentioned above. Since it is neither oversimplified nor too complex, we believe that this experiment is a good case-study to meet and tackle the difficulties arising when one tries to combine adaptive capabilities and multi-agent coordination schemes.

2.1 Description of the problem

Our simulated environment is shown in figure 1. It includes a circular arena, a flock of ducks and some *sheepdog agents* who must drive the flock towards a goal area. We tested all controllers in simulations involving at least three sheepdog agents and six ducks. The ducks and the sheepdog agents have the same maximum velocity. The goal is achieved as soon as all the ducks are inside the goal area.

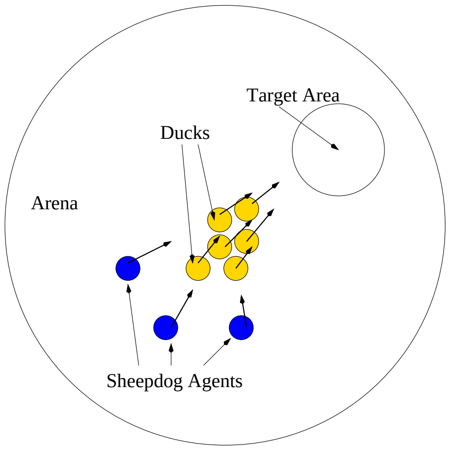

Fig. 1. The arena, ducks and sheepdogs

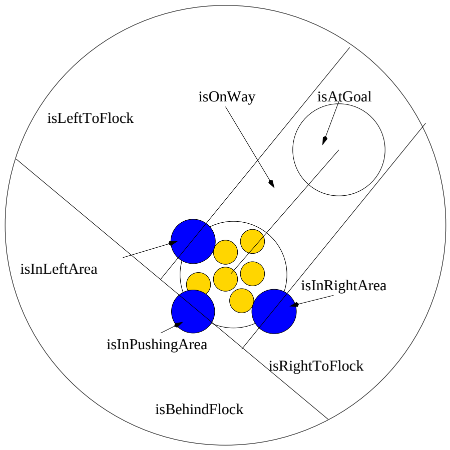

Fig. 2. Description of the situation

The behavior of the ducks results from a combination of three tendencies. They tend:

- to keep away from the walls of the arena [2];
- to join their mates when they see them, *i.e.* when they are within their visual range;
- to flee from the sheepdog agents which are within their visual range.

Once the behavior of the ducks is implemented, we must design the controllers of the sheepdog agents so that they drive the flock towards the goal area. A first step of this design process consists in finding which features of the simulated environment are relevant to achieve the goal of the sheepdog agents. This is what we present in the next section.

2.2 Description of the pre-conceived strategy

When one programs the sheepdog agents as simply being attracted by the center of the flock, it appears that, when a sheepdog agent is close to the flock and follows it, the flock tends to scatter because each duck goes away from the sheepdog along a radial straight line.

In order to solve this scattering problem, the strategy we adopted was to design the behavior of the agents so that at least one agent should push the flock towards the target area from behind, while at least one other agent should follow the flock on its left hand side and another one on its right hand side so that the flock would not scatter while being pushed.

2.3 Description of the inputs of the sheepdogs

As a result of this design, the description of the situation given to the agents consists of a set of tests on their position, as shown in figure 2. This gives us a first set of conditions:

• isAtGoal	• isOnWay
• isLeftToFlock	• isRightToFlock
• isInLeftArea	• isInRightArea
• isBehindFlock	• isInPushingArea

The important point is that all these position tests are relative to the position of the flock rather than absolute positions with coordinates. But the agents always know where they are with respect to the flock center, which would not be the case with an actual robot having a limited vision field. Thus these inputs might be thought of as delivered after treatments from a camera watching from above. Furthermore, there is no noise on them, which prevents us from drawing any conclusion on the applicability of our framework in the real world.

In order to coordinate the actions of the agents, we also added the following tests on the situation of other agents:

[2] Therefore, if they are left on their own, they tend to go to the center of the arena

- nobodyBehindFlock
- nobodyPushing
- nobodyInLeftArea
- nobodyInRightArea
- nobodyLeftToFlock
- nobodyRightToFlock
- nobodyOnWay
- isFlockFormed

Here again, the information is always perfectly accurate, while it would require a complicated communication protocol or a top-level manager to ensure this in an actual robotic experiment.

Our choices might appear surprising to multi-agent systems designers. But they are sound in our industrial context. Our focus is on adding adaptive behaviors capabilities in complex simulations where engineers do not want to take care about constraints on the availability of the information if this information is actually computed in the simulator [3]. Our choice would be different if we had to design a multi-robot system or to meet the constraining requirements of the ROBOCUP simulation league.

2.4 Description of the behaviors of the sheepdogs

All the behaviors of the sheepdog agents consist in going towards a certain point. In general, when the flock is formed, the sheepdog agents react to the center of the flock. But, when the flock is scattered, they can also react to the duck which is closest to them or the one which is the further from the center of the flock. The name of each behavior can be interpreted straight-forwardly. In the case of the "*driveXtoY*" behaviors, it consists in going behind X with respect to Y so as to push X towards Y. The overall behavior set is the following:

- doNothing
- goToGoalCenter
- goToFlockCenter
- followFlockToGoal
- goBehindFlock
- goToPushingPoint
- goToLeftGuidingPoint
- goToRightGuidingPoint
- goToRightOfFlock
- goToLeftOfFlock
- driveOutmostDuckToFlock
- driveClosestDuckToFlock
- driveClosestDuckToGoal
- goToClosestDuck
- goToOutmostDuck
- goAwayFromFlock

The controllers of our sheepdog agents involve 16 conditions and 16 basic behaviors. Designing the controller involving these sensori-motor capabilities consists in finding a good mapping between the conditions and the behaviors.

3 Implementing controllers as Classifier Systems

3.1 Elements of the Learning Classifier System framework

As we have some industrial applications in mind, we want to use a formalism into which we can put some expert control knowledge. But we also want to

[3] See [Sigaud and Gérard, 2001] for more information on the industrial side of this work

use adaptive techniques. In this context, the Learning Classifier Systems (LCS) formalism appears as a natural candidate.

The LCS framework designed by [Holland, 1975] gave rise to popular adaptive algorithms. Since the work of [Wilson, 1994] who simplified this first framework, a classical LCS can be seen as composed of a population of rules, or *classifiers*, containing *conditions* as a set of observations and *actions*:

$$[Condition] \rightarrow [Action](Strength)$$

The different parts of the classifier are strings of symbols in $\{0, 1, \#\}$, where # means "either 0 or 1". The population of classifiers was generally evolved thanks to a *genetic algorithm* (GA) – see [Goldberg, 1989] – using the strength of the classifiers as a fitness measure. When several classifiers could be fired in the same state, the strength was also used to select the one which would be fired. In these early versions of LCSs, the quality of the classifiers was modified by the *Bucket Brigade* algorithm according to the estimated reward received by the agent for firing the classifier.

A major improvement of the LCS framework was acheived by [Wilson, 1995] in designing XCS, replacing a *strength-based* LCS by an *accuracy-based* one.

Recently, a new way of using the LCS framework has received a growing interest [Stolzmann, 1998]. Based on ideas of [Riolo, 1990], it consists in adding in the classifiers an $[Effect]$ part which allows the system to use the classifiers for anticipating rather than merely reacting to the environment. It uses direct experience in order to build new classifiers, instead of relying on a genetic algorithm. The classifiers of such LCSs contain the following components:

$$[Condition][Action] \rightarrow [Effect]\ (quality\, parameters)$$

The learning process of such LCSs can be decomposed into two complementary processes:

• *latent learning* consists in building a reliable model of the dynamics of the environment, by ensuring that the $[Effect]$ part of all classifiers are correct. This new part stores information about state transitions and allows lookahead planning. The latent learning process can take place at each time step without any reward, hence it is very efficient. In particular, as [Witkowski, 1997] has shown, the quality of anticipation of every classifier which can be fired at a time can be updated according to the subsequent input message, even if the classifier has not actually been fired;

• *reinforcement learning* consists in improving a policy using the experience of the system, so that it becomes able to choose the optimal action in every state. This process takes advantage of latent learning to converge faster.

These new approaches can be seen as replacing the blind search performed by the GA by an heuristic search which takes advantage of the previous experience to improve the classifiers. As a result, they are less general since, for instance, they are devoted to tackling multi-steps problems whereas GA-based LCS can also tackle single-step problems, but they are also more efficient in what they are designed for.

3.2 Our Algorithm

Our own classifiers contain the following components:

$$[Condition][Action] \rightarrow [Effect]\ R$$

where R estimates the immediate reward received by the system when the classifier is fired.

The latent learning process creates and deletes classifiers. The creation process can be split in two main parts:

- the effect covering mechanism adjusts the effect parts by comparing successive observations and correcting mistakes;

- the condition specialization process identifies the most general of relevant conditions.

A classifier which sometimes anticipates well and sometimes not is such that its $[Condition]$ part matches several distinct states. It is too general and must be replaced by new classifiers with more specialized $[Condition]$ parts.

These mechanisms allow the system to converge towards a set of accurate classifiers anticipating correctly. We use this information about the state transitions in order to improve the reinforcement learning process.

The first part of this process consists in estimating the immediate reward resulting from the firing of each classifier. At each time step, we use the received reward to update an estimation of the immediate reward (R) of every classifier involving the last action and the last state, even if it has not actually been fired. The state transition informations and the immediate reward estimations allow to use a Dynamic Programming algorithm [Bellman, 1957] to compute a policy. A more detailed description of this algorithm can be found in [Gérard and Sigaud, 2001].

Rather than initializing a LCS with random classifiers or completely general ones, we first tried to use the CS formalism for implementing expert classifiers without using its adaptive capabilities. The methodological issues of our work are discussed in detail in [Sigaud and Gérard, 2001].

4 A "basic" controller

In table 1, we present the first controller that we designed in order to implement the solution described in section 2.2. It can be seen that we only use 13 of the 16 available inputs. Each line in the table is a classifier telling to the agent what to do in a particular situation. For instance, the first line says that if the agent is in the pushing area and if there is nobody on the way of the flock towards the goal and if the flock is formed, then the agent should go towards the goal center.

From table 1, it can be seen that this representation of the controller is not very compact: there are a lot of "#", which means that each classifier uses very

isBehindFlock	isInPushingArea	isLeftToFlock	isRightToFlock	isInLeftArea	isInRightArea	isOnWay	nobodyBehindFlock	nobodyPushing	nobodyLeftToFlock	nobodyRightToFlock	nobodyOnWay	isFlockFormed	Action
#	1	#	#	#	#	#	#	#	#	#	1	1	goToGoalCenter
#	1	#	#	#	#	#	#	#	#	#	1	1	goToFlockCenter
1	#	#	#	#	#	#	#	1	#	#	#	#	goToPushingPoint
#	#	1	#	#	#	#	1	#	#	#	#	#	goBehindFlock
#	#	1	#	#	#	#	#	1	#	#	#	#	goBehindFlock
#	#	#	1	#	#	#	1	#	#	#	#	#	goBehindFlock
#	#	#	1	#	#	#	#	1	#	#	#	#	goBehindFlock
#	#	1	#	1	#	#	0	0	#	#	1	1	followFlockToGoal
#	#	#	1	#	1	#	0	0	#	#	1	1	followFlockToGoal
#	#	1	#	0	#	#	#	#	#	#	1	1	goToLeftPushingPoint
#	#	#	1	#	0	#	#	#	#	#	1	1	goToRightPushingPoint
#	#	#	#	#	#	1	#	#	#	1	#	#	goToRightPushingPoint
#	#	#	#	#	#	1	0	#	0	#	#	#	goToRightPushingPoint
#	#	#	#	#	#	1	#	#	#	1	#	#	goToLeftofFlock
#	#	#	#	#	#	1	0	#	0	#	#	#	goToLeftofFlock
#	#	#	#	#	#	1	#	#	1	#	#	#	goToLeftPushingPoint
#	#	#	#	#	#	1	0	#	#	0	#	#	goToLeftPushingPoint
#	#	#	#	#	#	1	#	#	1	#	#	#	goToRightofFlock
#	#	#	#	#	#	1	0	#	#	0	#	#	goToRightofFlock
#	#	#	#	#	#	#	#	#	#	#	#	0	driveClosestDuckToFlock
#	#	#	#	#	#	#	#	#	#	#	#	0	goToOutmostDuck
#	#	#	#	#	#	#	#	#	#	#	#	0	goToClosestDuck
#	#	#	#	#	#	#	#	#	#	#	#	0	driveOutmostDuckToFlock
#	#	#	#	#	#	#	#	#	#	#	#	0	goAwayFromFlock

Table 1. A hand-crafted controller

few of the available inputs. As a result, the controller is difficult to design, since any change in the input set involves reconsidering all the lines in the table. As it will appear in section 6, the controller could also be more efficient.

Of particular interest are the five last classifiers, which are devoted to the case when the flock is scattered. Since we had no idea of how to organize the behaviors in such a case, we only gave five possible behaviors to deal with that situation and let the controllers pick one of them at random at each time step. As we will show in section 6, this is not an efficient design, even though it still allows the sheepdogs to reach their goals. But this inefficient design also lets room for improvement by using adaptive algorithms. Though this is not the focus of this paper, in section 6 we will breifly mention that, by specializing these classifiers, *i.e.* by adding new conditions on them, and by giving them different probabilities of being selected, our learning algorithm was able to obtain very quickly a better controller than the one we designed by hand.

5 A role-based controller

isInPushingArea	isInLeftArea	isInRightArea	isFlockFormed	Former Role	New Role
1	#	#	1	F.Pusher	Pusher
#	1	#	1	F.LeftGuide	LeftGuide
#	#	1	1	F.RightGuide	RightGuide
1	#	#	0	F.Pusher	F.Pusher
#	1	#	0	F.LeftGuide	F.LeftGuide
#	#	1	0	F.RightGuide	F.RightGuide
1	#	#	0	Pusher	F.Pusher
#	1	#	0	LeftGuide	F.LeftGuide
#	#	1	0	RightGuide	F.RightGuide
0	#	#	#	Pusher	F.Pusher
#	0	#	#	LeftGuide	F.LeftGuide
#	#	0	#	RightGuide	F.RightGuide

Table 2. The role table (F. stands for Future)

The notion of role appears naturally in the strategy we presented in section 2.2. In our solution, at least one agent must push the flock from behind (playing a PUSHER role) and at least one agent must guide the flock on its left hand side and another one on its right hand side (playing LEFTGUIDE and RIGHTGUIDE roles respectively). Therefore we tried to modify the architecture of the controller used in section 3 so as to make an explicit use of roles. Our new architecture contains two kinds of components:

• The *role table* is a CS stating under which conditions on the situation a agent changes its role into another role. If no observation matches, the role remains the same. The roles are initialized so that each agent chooses between FUTUREPUSHER, FUTURELEFTGUIDE and FUTURERIGHTGUIDE randomly, but in such a way that each role is assigned to at least one agent. Then the role of the agent evolves between FUTUREX and X, where X is either PUSHER, LEFTGUIDE or RIGHTGUIDE. But with this controller, a pusher cannot become a lateral guide nor vice versa. Our *role table* is shown in table 2.

• The *behavior tables* are CSs which fire actions of the agent according to conditions on the situation. There is one table for each role. Hence, there is only one *behavior table* active at a time in the controller of each agent, the one which corresponds to the role played by the agent.

We have six behaviors, each one corresponding to the fulfillment of one particular role, *i.e.* FuturePusherBehavior, PusherBehavior, FutureLeftGuideBehavior, LeftGuideBehavior, FutureRightGuideBehavior and RightGuideBehavior. All these *behavior tables* are shown in tables from 3 to 8.

isInPushingArea	isFlockFormed	isBehindFlock	Action
0	1	#	goToPushingPoint
1	1	#	goAwayFromFlock
#	0	0	goBehindFlock
#	0	1	driveClosestDuckToFlock

Table 3. FuturePusherBehavior

isInPushingArea	isBehindFlock	nobodyOnWay	nobodyLeftToFlock	nobodyRightToFlock	isFlockFormed	Action
1	#	1	0	0	1	goToGoalCenter
0	1	#	#	#	1	goToPushingPoint
#	0	#	#	#	1	goBehindFlock
#	0	#	#	#	0	driveOutmostDuckToFlock
#	1	#	#	#	0	goToOutmostDuck

Table 4. PusherBehavior

isLefttoFlock	isInLeftArea	isFlockFormed	isBehindFlock	Action
1	0	1	#	goToLeftArea
0	#	1	#	goToLeftofFlock
#	#	0	0	goBehindFlock
#	#	0	1	driveClosestDuckToFlock

Table 5. FutureLeftGuideBehavior

isLefttoFlock	isInLeftArea	nobodyPushing	isFlockFormed	Action
1	#	0	1	followFlockToGoal
#	1	0	1	followFlockToGoal
0	0	#	1	goToLeftArea
#	#	#	0	goToOutmostDuck

Table 6. LeftGuideBehavior

Introducing roles in our architecture brings several benefits.

• It is easier to design a *behavior CS* devoted to fulfill one particular role, since a particular role corresponds to a specialized part of the global behavior. Hence, each *behavior table* is much smaller than the table 1 presented in section 3.

• It is easier to deal with the case where the flock is scattered. Since each agent can fire different actions according to its role, it is easier to find a good

isRighttoFlock	isInRightArea	isFlockFormed	isBehindFlock	Action
1	0	1	#	goToRightArea
0	#	1	#	goToRightofFlock
#	#	0	0	goBehindFlock
#	#	0	1	driveClosestDuckToFlock

Table 7. FutureRightGuideBehavior

isRighttoFlock	isInRightArea	nobodyPushing	isFlockFormed	Action
1	#	0	1	followFlockToGoal
#	1	0	1	followFlockToGoal
0	0	#	1	goToRightArea
#	#	#	0	goToClosestDuck

Table 8. RightGuideBehavior

coordination scheme between all actions, with respect to the case of the reactive controller where we had no control on which action would be fired by which agent.

• From a reinforcement learning research perspective, it is easier to design an internal reinforcement signal policy when we use roles. Generally, fulfilling a role corresponds to reaching a particular situation which can be detected by the agent, and/or to insure that some validity conditions hold. Then the agents can be rewarded or punished if the first condition holds or the second one is broken. In our flock control simulation, for example, playing a FutureLeftGuide role involves reaching the *leftArea* while playing a LeftGuide role involves keeping the flock formed. Hence, an agent in charge of the left side of the flock can be rewarded when it reaches the *leftArea*, becoming a LeftGuide, and punished if the flock is scattered, coming back to FutureLeftGuide. We think that this is a good way of introducing intermediate reinforcement signals, in a more natural framework than in [Matarić, 1994b], for instance.

6 Empirical Study

We first ran 2000 experiments to get a statistically significant view of the results obtained with these controllers. Although the hand-crafted role-based controllers appeared more efficient than the ones without roles with three sheepdogs, we did want to check whether it would be more or less robust with respect to the size of the population of agents, since the role-based controllers are designed for a group of three sheepdogs.

Therefore, we decided to test the robustness of both control policies by testing them with various sets of parameters, and particularly by changing the size of the population of sheepdogs.

6.1 Robustness to an increasing number of sheepdogs

We first tested the robustness of the controllers when the number of sheepdog agents was increased from three to twenty.

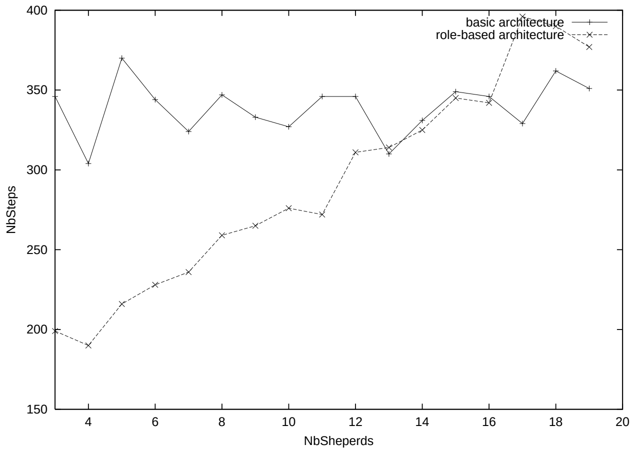

Fig. 3. Robustness of basic and role-based controllers to 3 to 20 sheepdogs

The results are shown on figure 3. Each point in the curves represents an average performance over 100 trials, and each set of 100 trials starts with the same 100 random initial positions. We must also mention that the goal is never reached in less than 95 time steps, which is the minimum number of time steps for the sheepdog agents to surround the flock and drive it to the goal from a lucky initial situation.

If a trial lasts more than 4000 time steps, it is stopped and counted as a failure. Failures are not taken into account in the computation of the average, since their duration is arbitrary. Since there are very few failures, we do not devote a figure to show them. Indeed, the worst case was four failures over the 100 trials that give one point on the figures. It appears that the role-based controller failed ten times on the $18 \times 100 = 1800$ trials, while the basic and learned basic controllers only failed respectively five times and two times over the 1800 trials. The failures happen more often with more than twelve sheepdogs in the role-based case, which supports our diagnosis of a lack of robustness of this solution.

It can be seen in figure 3 that the role-based architecture performs better with three sheepdogs than the basic one, but that the basic architecture is more robust to an increasing number of sheepdogs. It can also be seen in figure 4 that the controller obtained from applying adaptive algorithms to the basic architecture during two trials performs better than the hand-crafted basic one, and is still robust.

6.2 Robustness to a change in the behavior of the ducks

In order to understand better the phenomena observed in section 6.1, we also tried to modify the behavior of the ducks so as to modify the dynamics of the

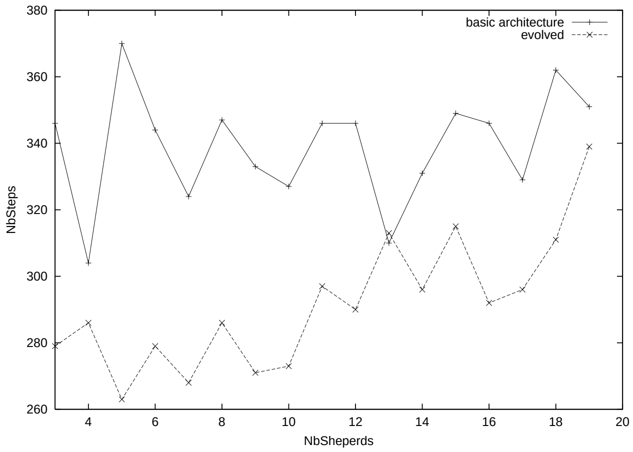

Fig. 4. Robustness of basic and evolved controllers to 3 to 20 sheepdogs

environment of the sheepdogs. We tuned the sensitivity of the ducks with respect to the walls of the arena so that they would keep away from these walls only when getting too close to them. As a result, the flock tends to form anywhere in the arena rather than only in the center as in the previous case. However, the repulsiveness of the walls is sensed far before the ducks reach the target area. As a result, it is not easier for the sheepdogs to drive the flock to the target area. We also lowered the tendency of the ducks to go towards each other so that the flock would scatter more often. These two modifications makes the job harder for the sheepdogs.

The relative performance of the basic and role-based controllers with both kinds of ducks can be seen on figure 5. It can be seen that, as expected, the performance of the basic controller is very sensitive to the change of the behavior of the ducks. The performance is much worse with the new ducks, and tends to be much less robust to an increase of the number of sheepdogs. On the contrary, the performance of the role-based controller is nearly unaffected by the change of ducks, both curves are nearly identical.

7 Discussion of the results

7.1 Explaining the results

The reduced performance of the basic architecture when applied to the new ducks can be explained by the fact that the flock is scattered more often. We have shown that it was more difficult to design an efficient strategy with the basic controller to deal with the case where the flock was scattered, since the behavior of the different agents could not be specialized.

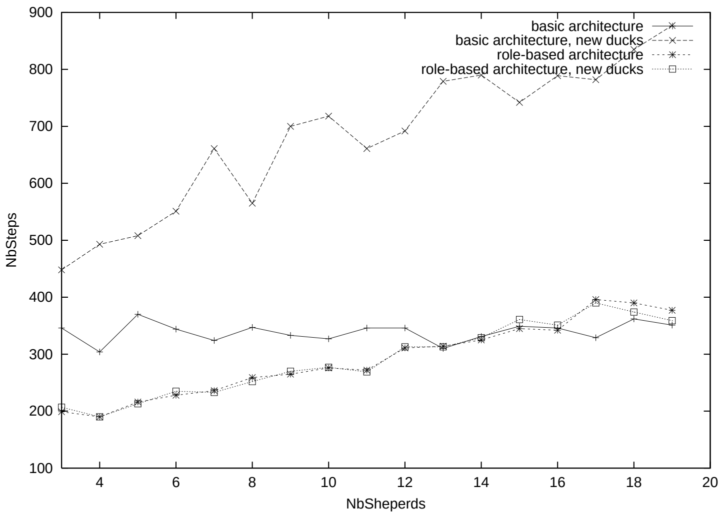

Fig. 5. Robustness of both controllers to a new behavior of ducks

In contrast, the performance of the role-based controllers is not degraded, their efficiency is not affected by the increasing tendency of the flock to scatter.

But why is this that the role-based controllers are less robust to an increasing number of sheepdogs than the basic one? From a closer look at a lot of simulation runs, it appeared that this comes from a longer time spent in the initial messy situation before the flock can get formed. At the beginning of each trial, indeed, all the sheepdogs and ducks are scattered at random in the arena. Hence, the more sheepdogs there are among the ducks, the longer it takes to the ducks to form a flock.

This is particularly true for the role-based agents. Since each agent has its own role at the beginning, it must reach its pushing or guiding area, even if it is by the wrong side of the flock. As a result, it may cross the flock and scatter it or at least delay the movement of the ducks towards each other. Therefore, the more role-based agents there are, the more they tend to prevent the ducks from forming a flock.

The basic agents, on the contrary, organize themselves more opportunistically with respect to their initial positions. Each agent goes to the closest pushing or driving area. Since there are more agents, these areas are reached faster and this compensates for the longer time spent in forming the flock.

We can summarize this finding in asserting that the basic controllers are less tightly designed, but result in more opportunistic behaviors than the one obtained with the role-based controllers.

7.2 Good reasons for exchanging the roles

We have shown in section 6 that our role-based controller was less robust than the basic one because the role of the agents were assigned from the start and the agents were not allowed to re-organize themselves opportunistically.

The lack of opportunism of the role-based architecture comes from the fact that our hand-crafted role table specifies too narrowly the situations into which one role should be exchanged with another one. More precisely, as we have said, one agent which has started with a FutureX role can only switch to a X role and back, where X stands for Pusher, LeftGuide and RightGuide.

Three different considerations convinced us that the agents should be able to exchange their roles in order to solve their task more efficiently.

• The first one is that a good way of improving the performance of our role-based solution would be to let the agents choose their initial role according to their initial position: they would choose the role driving them to the closest guiding or pushing area. But if we do so, nothing guarantees that there will be at least one agent to play each role. Then it is necessary that they exchange their roles in order to coordinate their efforts.

• The second evidence in favor of letting the agents exchange their roles has been found by examining some particular trials. To our surprise, we discovered that the controllers without roles were often manifesting an unexpected strategy more efficient that the one we had in mind. This strategy is shown in figure 6.

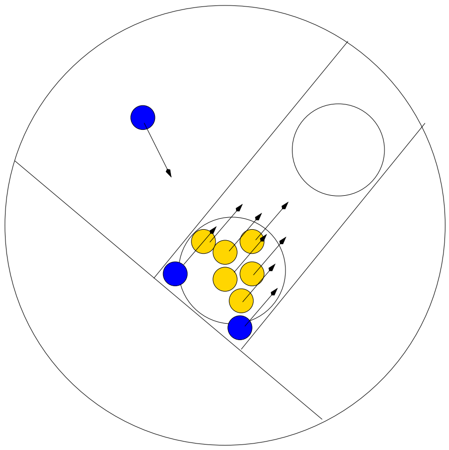

Fig. 6. An emergent strategy

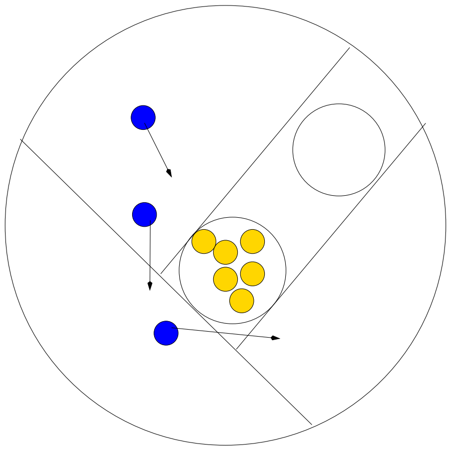

Fig. 7. Two guides by the same side

It happens that two sheepdog agents are able to drive the flock to the target area. This strategy seems very robust since the ducks seldom escape from the chase of the sheepdog agents. It can be seen as a different distribution of the roles, where two agents play new roles between pusher and guide, and the other ones may help to form the flock again when necessary.

• The last one was also revealed by a closer examination of the behavior of the agents. In the situation depicted in figure 7, both guides are by the same side of the flock while the pusher is ready to push. If the agents cannot exchange their roles, the agent which is behind the flock will start pushing it and the flock will scatter, since there is no guide on one side. But if the agents can exchange their roles, the best solution here is that the PUSHER becomes a RIGHTGUIDE while one of the LEFTGUIDES becomes PUSHER and comes behind the flock in order to push. This is the kind of social reorganization which we will present in the next section.

8 A further inquiry

8.1 The new role table

In order to check that exchanging the roles would allow our controllers to be both more efficient than the basic ones and more robust than the first role-based ones, we designed by hand the new role table shown in table 9. The corresponding behavior tables are the same as in section 5.

The task was more difficult than what we expected. The difficulty comes from the fact that nothing guarantees anymore that there will be at least one agent to play each role, while this condition is necessary for success. Thus, the classifiers must be designed in such a way that each change of role from one agent is balanced quickly by another change of role from another agent which will play the dropped role. In order to do this, it appeared necessary to add new inputs to coordinate more efficiently the roles. These input state respectively whether there is already an agent which plays a PUSHER, a LEFTGUIDE and a RIGHTGUIDE role or not, without taking into account whether it is a FUTURE one or not. This gives an argument in favor of distinguishing only three roles and two behaviors per role, as we will discuss in section 9.

Once again, these informations about the role played by other agents are considered as available through the simulation platform while it would require complicated communication mechanisms to be maintained among a team of robots. We didn't tackle any team state maintenance, as [Stone and Veloso, 1999] or [Tambe et al., 1999] do, for instance.

The classifiers shown in table 9 can be split into four groups.

• The first group, up to classifier 8, deals with the starting situation. Each agent is initialized with the START role, and will only play it during one time step. The classifiers tell which role the agent should choose according to their initial location with respect to the pushing and guiding areas. In the case when the agent is within the flock of ducks or on its way to the goal, it chooses at random to become either FUTURELEFTGUIDE or FUTURERIGHTGUIDE. Since

	isBehindFlock	isLefttoFlock	isRighttoFlock	isInPushingArea	isInLeftArea	isInRightArea	isFlockFormed	isTherePusher	isThereLeftGuide	isThereRightGuide	Former Role	New Role
1	0	0	0	#	#	#	#	#	#	#	Start	F.RightGuide
2	0	0	0	#	#	#	#	#	#	#	Start	F.LeftGuide
3	1	0	0	#	#	#	#	#	#	#	Start	F.Pusher
4	#	1	#	#	#	#	#	#	#	#	Start	F.LeftGuide
5	#	#	1	#	#	#	#	#	#	#	Start	F.RightGuide
6	#	#	#	1	#	#	1	#	#	#	Start	Pusher
7	#	#	#	#	1	#	1	#	#	#	Start	LeftGuide
8	#	#	#	#	#	1	1	#	#	#	Start	RightGuide
9	#	#	#	1	#	#	1	#	1	1	F.Pusher	Pusher
10	#	#	#	#	1	#	1	1	#	1	F.LeftGuide	LeftGuide
11	#	#	#	#	#	1	1	1	1	#	F.RightGuide	RightGuide
12	#	#	#	1	#	#	0	#	1	1	Pusher	F.Pusher
13	#	#	#	0	#	#	#	#	1	1	Pusher	F.Pusher
14	#	#	#	#	1	#	0	1	#	#	LeftGuide	F.LeftGuide
15	#	#	#	#	0	#	#	1	#	#	LeftGuide	F.LeftGuide
16	#	#	#	#	#	1	0	1	#	#	RightGuide	F.RightGuide
17	#	#	#	#	#	0	#	1	#	#	RightGuide	F.RightGuide
18	#	1	#	#	#	#	#	#	#	#	F.RightGuide	F.LeftGuide
19	#	#	1	#	#	#	#	#	#	#	F.LeftGuide	F.RightGuide
20	#	#	1	#	#	#	#	#	0	#	RightGuide	F.Pusher
21	#	1	#	#	#	#	#	#	#	0	LeftGuide	F.Pusher
22	#	#	1	#	#	#	1	#	0	#	F.RightGuide	F.Pusher
23	#	1	#	#	#	#	1	#	#	0	F.LeftGuide	F.Pusher
24	1	#	0	#	#	#	1	0	#	#	F.LeftGuide	F.Pusher
25	1	0	#	#	#	#	1	0	#	#	F.RightGuide	F.Pusher
26	#	#	1	#	#	#	#	0	#	#	RightGuide	F.Pusher
27	#	1	#	#	#	#	#	0	#	#	LeftGuide	F.Pusher
28	#	#	#	#	#	#	#	#	0	1	Pusher	F.LeftGuide
29	#	#	#	#	#	#	#	#	0	1	F.Pusher	F.LeftGuide
30	#	#	#	#	#	#	#	#	1	0	F.Pusher	F.RightGuide
31	#	#	#	#	#	#	#	#	1	0	Pusher	F.RightGuide

Table 9. The new role table (F. stands for Future)

the roles are chosen according to the initial position and these positions are random, nothing guarantees that the roles will be equally distributed between the agents.

• The second and third groups of classifiers do the job which was done in table 2 by our former role table. The second group, from classifier 9 to classifier 11, deals with the case when a FutureX has reached its intermediate goal and becomes an X, while the third group, from classifier 12 to classifier 17, deals with the case when an X has failed playing its role and comes back to the FutureX role.

• The last group of classifiers is devoted to the exchanges of roles. The classifiers 18 and 19 tell that if a guiding agent is by the wrong side of the flock with respect to its role, it should change its role rather than try to cross the flock and scatter it. The classifiers from 20 to 27 describe what the guides should do in the situation described in figure 7. Classifiers 20 to 23 are fired if there is no guide to deal with the other side, while classifiers 24 to 27 are fired if there is no pusher. This last situation can occur either if the pusher went to the other side as described in figure 7, or in the initial situation if there was no agent choosing the FuturePusher or the Pusher role at the beginning. At last, the four last classifiers tell what the pusher should do in the situation described in figure 7.

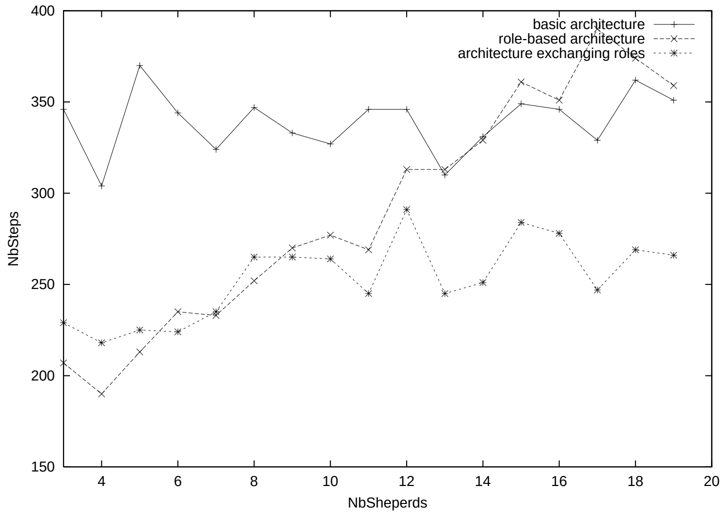

Fig. 8. Robustness of the new controller to an increasing number of sheepdogs

The empirical study of the robustness of this new controller gave the results shown in figure 8. We used the first kind of ducks under the conditions described in section 6.1. Two curves were already given in figure 3, we present them again for comparison with the new one.

It can be seen that we have acheived what we were trying to. Even if there is still a slight rising slope and if the performance with three to five robots is not as good as the one of the former role-based controller, the new controller is both more robust than this former controller and more efficient than the basic one. There are only three failures over 1800 trials with this controller, one with three sheepdogs, one with five and one with eighteen of them.

We also checked the number of times when each classifier of table 9 was fired. It appears that the classifiers for exchanging from LeftGuide to RightGuide and vice versa are fired 23 times in average on 100 trials, while all the other classifiers for exchanging the roles are fired less than 5 times. This shows that, while these classifiers are used very seldom, much less than once per trial, they result in a very significant improvement of the controllers.

Now, we can claim that the case depicted on figure 7 is particularly representative of the discussion we raised in the introduction. The results we obtained show that being reactive and solipsistic is inefficient in that particular situation. It is the kind of situation where the agents must reorganize with each other in order to be more efficient. We have shown that this reorganization can be dealt with in our reactive, CS-based framework, just by designing roles and by letting the agents exchange their roles. Our point was that this seemingly deliberative social behavior can be written as classifiers in the role table of our agents in such a way that they *react socially* to the situation depicted in figure 7 just by exchanging their roles. This does not take more time than any other reactive behavior. Here, our agents are clearly reactive in the first sense given in the introduction, *i.e.* they are responsive in time, but not in the second sense, since they use a memory of their former role.

9 Discussion from a multi-agent perspective

We have already said that our research goals are directed towards adaptive behaviors more than towards multi-agent architecture. But having adaptive multi-agent systems also implies to design general architecture providing flexibility and abstraction. Thus, we must improve our work with that respect too. Henceforth, we discuss here some obvious limitations of our architecture from a multi-agent perspective.

First, another way to look at our role-based architecture would be to consider that there are only three roles (Pusher, LeftGuide and RightGuide), and that the fulfillment of each role involves two behaviors (FutureX and X). In the case of our example, implementing this way to articulate roles with several behaviors would give rise to an unnecessary complication of the architecture. But in more complicated examples, if more behaviors are necessary to fulfil one role or if the fulfillment of two different roles involves some common behaviors, distinguishing roles and behaviors by binding to each role a set of behaviors and a way to sequentialize their activation would provide a higher degree of flexibility and abstraction.

Such a mechanism can be found in architectures devoted to solve ROBOCUP problems both in [Tambe et al., 1999] and in [Stone and Veloso, 1999]. The first shares closer goals with our work since some of the behaviors are learned. But the second introduces a higher level of organization, namely the articulation between roles and formations, which might also help improve our work.

Indeed, the fact that having more sheepdogs to drive the flock results in poorer performance unless we design a very robust controller is rather counter-intuitive. The key point here is that our agents use the same strategy whatever their number. This strategy relies on the assumption that the flock will get formed fast, which is no more valid when the number of agents increases. Thus, a major way for improvement would be to let the agents change their strategy when their number increases. For instance, as soon as they are as numerous as ducks, each agent could take care of one particular duck, rather than wait for the flock to be formed. Now, using different formations according to the number of agents would be a good way to implement different strategies.

There is no technical nor theoretical obstacle to improve our architecture in that way. But the reader must keep in mind that our research goal is the bottom-up building of a control architecture thanks to learning processes, which is more difficult than just hand-crafting correct and flexible multi-agent controllers.

10 Future Work and Conclusion

Even if we have shown in a preliminary study that applying adaptive algorithms to our hand-crafted controllers results in a significant improvement of the performance, we have not defended yet our claim that agents can find by themselves the coordination schemes presented in table 9. Our claim that our system is still reactive can be refuted because all the anticipation necessary to find such a good coordination scheme has been given by the expert, rather than learned by the system.

Our first focus was on the improvement of hand-crafted solutions because, from an engineering perspective, an automated improvement of an expert controller means that the expert who designed the controller can rely on adaptive algorithms to optimize it. Hence he spends less time in this design, which is very appealing in an industrial context. Our first results have shown that the classifier systems formalism is good for coding controllers both because some knowledge of the expert can be easily represented in it and because applying optimization algorithms is straight-forward in the formalism.

But now we will have to start studying whether our algorithms are able to learn similar role-based controllers from scratch. This is not the case yet with the algorithm used here and presented in detail in [Gérard et al., 2001]. Obtaining such a result would be all the more interesting that designing by hand the role exchange strategy presented in section 8.1 proved difficult and time consuming.

It is clear that the performance of the role exchange architecture heavily depends on the definition of the controller, and that this controller was difficult to design by hand. The point is that the behavior tables were designed first and the role exchange table afterwards, whereas they are highly interdependent. Maybe, a different set of behavior tables would have resulted in a simpler role exchange table. This fact supports the claim that both the behavior tables and the role exchange table should be built by an automated learning process in a unified framework.

Therefore, we are now extending the scope of our algorithms towards the ability to build an architecture reflecting the one we designed in order to implement the use of roles in our flock control experiment. Our algorithm will be able to create internal states when necessary and to let evolve the mapping between these internal states and some conditions on the situation. Implementing roles as internal states should give us a control system for an agent able to create and evolve its own roles. Furthermore, the agents team should be able to globally reorganize their behaviors thanks to the adaptive processes.

To summarize, we presented a simulation testbed into which several agents had to solve a common task and we have shown how giving roles to the agents was an efficient way to design a control strategy. We have shown how these roles could be represented in the CS framework, and that such a way of using them gives an ability to react socially to multi-agent situations.

At last, we believe that the experimental testbed presented in this paper, though it is quite simple, is rich enough to raise most of the issues that we will meet in our industrial applications. As a conclusion of our study, it appears that the framework exposed here can be reused for more complicated multi-agent tasks, but it would require improvements by the multi-agent side, for instance if we would want to use it in the ROBOCUP domain. We did not try to do it because it would be too much time consuming while we are expected to work on our industrial problems. But we can already infer that obstacles to apply our framework to the design of a ROBOCUP team are that an organizational level would be necessary both to ensure the correct computation of all team information that we considered as directly available in our work, and to bring all the necessary flexibility and abstraction capabilities which are not present today in our architecture.

11 Acknowledgements

The authors want to thank the anonymous reviewers of an early version of this paper who gave valuable advices to improve it and all the attendees of the ECAI2000 workshop on "Balancing Reactivity and Social Deliberation in Multi-Agent Systems" who raised interesting points which have been beneficial to the continuation of this work.

References

[Asada and Kitano, 1999] Asada, M. and Kitano, H., (Eds.) (1999). *Robocup-98: Robot Soccer World Cup II.* Lectures Notes in Artificial Intelligence 1604, Springer-Verlag.

[Asada et al., 1999] Asada, M., Uchibe, E., and Hosoda, K. (1999). Cooperative behavior acquisition for mobile robots in dynamically changing real-worlds via vision-based reinforcement learning and development. *Artificial Intelligence*, 110(2):275–292.

[Behnke and Rojas, 2001] Behnke, S. and Rojas, R. (2001). A hierarchy of reactive behaviors handles complexity. In Hannebauer, M., Wendler, J., and Pagello, E., (Eds.), *this issue*. Springer-Verlag.

[Bellman, 1957] Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ.

[Bouzid et al., 2001] Bouzid, M., Hanna, H., and Mouaddib, A.-I. (2001). Theoretic-Decision Approach for Task Allocation in Resource-Bounded Agents. In Hannebauer, M., Wendler, J., and Pagello, E., (Eds.), *this issue*. Springer-Verlag.

[Bredenfeld and Kobialka, 2001] Bredenfeld, A. and Kobialka, H.-U. (2001). Team Cooperation using Dual Dynamics. In Hannebauer, M., Wendler, J., and Pagello, E., (Eds.), *this issue*. Springer-Verlag.

[Brooks, 1991] Brooks, R. A. (1991). Intelligence without reason. A.I. Memo 1293, Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

[Colombetti and Dorigo, 1993] Colombetti, M. and Dorigo, M. (1993). Training agents to perform sequential behavior. Technical Report TR-93-023, International Computer Science Institute, Berkeley.

[Gérard and Sigaud, 2001] Gérard, P. and Sigaud, O. (to appear, 2001). YACS: Combining dynamic programming with generalization in classifier systems. In Stolzmann, W., Lanzi, P.-L., and Wilson, S. W., (Eds.), *LNAI 1996 : Advances in Classifier Systems*. Springer-Verlag.

[Gérard et al., 2001] Gérard, P., Stolzmann, W., and Sigaud, O. (to appear, 2001). YACS: a new learning classifier system with anticipation. *Journal of Soft Computing*.

[Goldberg, 1989] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley.

[Holland, 1975] Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. The University of Michigan Press.

[Kaelbing, 1990] Kaelbing, L. P. (1990). An architecture for intelligent reactive systems. In Allen, J., Hendler, J., and Tate, A., (Eds.), *Readings in Planning*, chapter 11, pages 713–728. Morgan Kaufmann Publishers, Inc.

[Matarić, 1994a] Matarić, M. J. (1994a). *Interaction and Intelligent Behavior*. PhD thesis, MIT AI Mobot Lab.

[Matarić, 1994b] Matarić, M. J. (1994b). Rewards functions for accelerated learning. In Cohen, W. W. and Hirsch, H., (Eds.), *Proceedings of the Eleventh International Conference on Machine Learning*, San Francisco, CA. Morgan Kaufmann Publishers.

[Riedmiller et al., 2001] Riedmiller, M., Moore, A., and Schneider, J. (2001). Reinforcement Learning for Cooperating and Communicating Reactive Agents in Electrical Power Grid. In Hannebauer, M., Wendler, J., and Pagello, E., (Eds.), *this issue*. Springer-Verlag.

[Riolo, 1990] Riolo, R. L. (1990). Lookahead planning and latent learning in a classifier system. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 316–326, Cambridge, MA. MIT Press.

[Sigaud and Gérard, 2001] Sigaud, O. and Gérard, P. (to appear, 2001). Using classifier systems as adaptive expert systems for control. In Stolzmann, W., Lanzi, P.-L., and Wilson, S. W., (Eds.), *LNAI 1996 : Advances in Classifier Systems*. Springer-Verlag.

[Stolzmann, 1998] Stolzmann, W. (1998). Anticipatory Classifier Systems. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Golberg, D. E., Iba, H., and Riolo, R., (Eds.), *Genetic Programming*. Morgan Kaufmann Publishers, Inc., San Francisco, CA.

[Stone and Veloso, 1997] Stone, P. and Veloso, M. (1997). Multiagent systems: A survey from a machine learning perspective. Technical Report CMU-CS-97-193, School of Computer Science, Carnegie Mellon University, Pittsburg, PA 15213.

[Stone and Veloso, 1999] Stone, P. and Veloso, M. (1999). Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence*, 110(2):241–273.

[Tambe et al., 1999] Tambe, M., Adibi, J., al Onaizan, Y., Erdem, A., Kaminka, G. A., Marsella, S. C., and Muslea, I. (1999). Building agent teams using an explicit teamwork model and learning. *Artificial Intelligence*, 110(2):215–239.

[Vaughan et al., 1998] Vaughan, R., Stumpter, N., Frost, A., and Cameron, S. (1998). Robot sheepdog project achieves automatic flock control. In Pfeifer, R., Blumberg, B., Meyer, J.-A., and Wilson, S. W., (Eds.), *From Animals to Animats 5: roceedings of the Fifth International Conference on Simulation of Adaptive Behavior*, pages 489–493, Cambridge, MA. MIT Press.

[Wilson, 1994] Wilson, S. W. (1994). ZCS, a zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18.

[Wilson, 1995] Wilson, S. W. (1995). Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175.

[Witkowski, 1997] Witkowski, C. M. (1997). *Schemes for Learning and behaviour: A New Expectancy Model*. PhD thesis, Department of Computer Science, University of London, England.