The use of roles in a multiagent adaptive simulation

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

[1] Dassault Aviation, DGT/DPR/DESA
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. This paper is about the use of roles in a multiagent adaptive context. 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 use the Classifier System formalism to control the agents. We show that using a notion of role is natural in such a context. We compare the performance of an expert controller with and without roles. Then we show how applying adaptive techniques to that "bootstrap" controller can improve the performance with respect to expert rules.
From this empirical study, it appears that an emergent strategy gets better results than the conception of the roles we designed by hand. Thus we advocate the necessity of tackling the problem of evolving the roles of the agents.

1 Introduction

The necessity of having good benchmarks to test and compare algorithms and architectures is now central in the multiagent 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. 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 another player [Asada et al., 1999]).

Therefore, the tendency in adaptive multiagent 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 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 work on an application which appears as a good compromise between the too complex Robocup problem and the oversimplified prey-predator problems. 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 governing the behavior of 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 robots 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 multiagent coordination schemes.

The purpose of the paper is to show that explicitly using roles in a multiagent domain is an efficient design technique improving both the resulting controllers and their adaptive capabilities. In such a context, providing the system with the ability to evolve roles is both something which helps finding better strategies and something which can be done straight-forwardly with adaptive algorithms.

In a wider perspective, the general purpose of our approach is to show that providing some expert control knowledge as a starting point to a Classifier System (CS) is both an efficient way to improve the expert solution through adaptive algorithms and a good strategy to make the use of CSs feasible even for complicated industrial problems– see [Sigaud, 2000] for a more complete discussion.

The paper is organized as follows. In the next section, we describe our simulator and the multiagent strategy we used to solve the task. In section 3, we present how we implemented this strategy in the formalism of CSs. In section 4, we give the results obtained with a hand-crafted controller. In section 5, we show how our first hand-crafted controller was significantly improved with an explicit use of roles, resulting in a new architecture involving a set of behaviors devoted to the fulfillment of each role. We present the benefits which can be drawn from such an architecture. Then, we compare the results with those obtained by applying adaptive algorithms to each behavior. In section 7, we show in an empirical study that an emergent strategy obtains better results than the conception of the roles that we designed in section 2. Therefore we advocate in section 8 the necessity of tackling the problem of evolving the roles of the agents. We show how roles can be formalized as internal states and conclude that our adaptive algorithm will provide a first step towards this functionality.

2 The problem and its representation

2.1 Description of the problem

Our simulation 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 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.

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

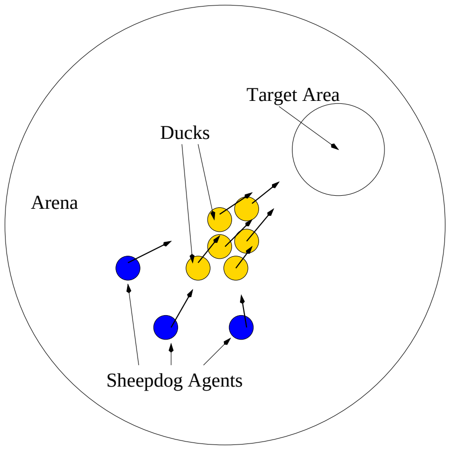

Fig. 1. The arena, ducks and sheepdogs

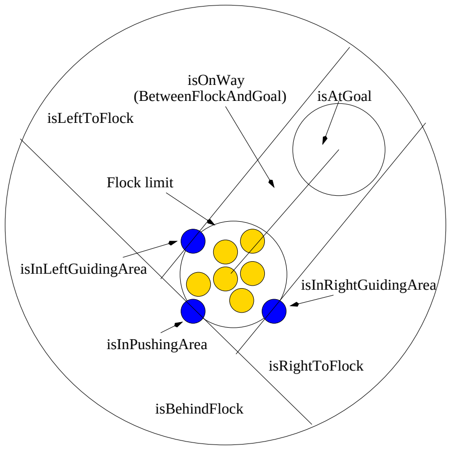

Fig. 2. Description of the situation

- to keep away from the walls of the arena [1];
- 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, our task is to 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 simulation 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 does not scatter while being pushed.

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

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

- isAtGoal
- isOnWay
- isLeftToFlock
- isRightToFlock
- isInLeftGuidingArea
- isInRightGuidingArea
- isBehindFlock
- isInPushingArea

The important point is that we defined pushing and guiding areas relative to the flock in order to implement the pushing and guiding behaviors. In order to coordinate the actions of the agents, we also added the following tests on the situation of other agents:

- nobodyBehindFlock
- nobodyPushing
- nobodyInLeftGuidingArea
- nobodyInRightGuidingArea
- nobodyLeftToFlock
- nobodyRightToFlock
- nobodyOnWay
- isFlockFormed

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 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 use adaptive techniques. In this context, the Classifier System (CS) formalism appears as a natural candidate.

The CS framework [Holland, 1975] gave rise to popular adaptive algorithms. A classical CS is composed of a population of rules, or *classifiers*, containing observations composed of conditions and actions:

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

The different parts of the classifier are strings of symbols in $\{0, 1, \#\}$, where # means "either 0 or 1". The strength of the classifier can be modified by the *Bucket Brigade* algorithm [Holland, 1975] according to the estimated reward given to the agent for firing the classifier. The population of classifiers is generally evolved thanks to a *genetic algorithm* – see [Goldberg, 1989] – using the strength of the classifiers as a fitness measure. When several classifiers can be fired in the same state, the strength is also used to select the one which will be fired.

Recently, a new way of using the CS 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 rules 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 CSs contain the following components:

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

The learning process of such CSs 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 provides information about state transitions and allows planning. The latent learning process can take place at each time step regardless of a goal, 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.

3.2 Our Algorithm

Our own classifiers contain the following components:

$$[Observation][Action] \rightarrow [Effect]\ (Q_e,\, R,\, Q_a)$$

- Q_e is the quality of the effect part of the classifier, also known as the quality of anticipation. It estimates the transition probability between the observation and the effect when the action is chosen.
- R estimates the immediate reward received by the system when the classifier is fired.
- Q_a is the quality of the action represented by the classifier. It corresponds in classical CSs to the *Strength* parameter used in the action selection process.

The latent learning process creates and deletes classifiers, and adjusts the qualities of anticipation. Let us consider a classifier with an [*Observation*] part matching the previous state and whose action is the one which has been chosen at the previous time step.

If the classifier was able to anticipate the current state, we increase its Q_e, even if it was not fired. If it was not able to, we decrease its Q_e and a new classifier may be created, which would anticipate well. This process allows to discover new [*Effect*] parts.

New [*Observation*] parts are discovered by a specialization process. A classifier whose Q_e has been sometimes increased and sometimes decreased is such that its [*Observation*] part matches several distinct states. It is too general, therefore its anticipation is sometimes correct and sometimes not. Such classifiers are replaced by new classifiers with more specialized [*Observation*] 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 the quality of the action (Q_a) for each classifier. These qualities define the policy.

A more precise description of this algorithm can be found in [Gérard, 2000].

4 Empirical Study without roles

4.1 Using a hand-crafted controller

In this section, we present results obtained by hand-crafted controllers without turning the adaptive capabilities on. Rather than initializing CSs with random classifiers or completely general ones, we first try to use the CS formalism for implementing expert rules without using its adaptive capabilities.

In table 1, we present the 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.

We ran 2000 experiments to get a statistically significant view of the results obtained with this controller. During a trial, if the goal is not reached after 4000 time steps, we consider that the controller results in a loop behavior and will never succeed, hence we stop the trial. We must also mention that the goal is never reached in less than 115 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.

The average number of time steps to reach the goal is 1759.45. The controller is stopped after 4000 time steps in 26.65% of the trials.

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

isBehindFlock	isInPushingArea	isLeftToFlock	isRightToFlock	isInLeftGuidingArea	isInRightGuidingArea	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

uses very few of the available inputs. As a result, the controller is difficult to design and is not very efficient.

Furthermore, the adaptive algorithms of the CS would be slow to converge on such a representation, since the search space is too big. Therefore, rather than trying to adapt it, we first tried to add further structure in our controller, as explained in the next section.

5 An architecture to deal with roles

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 his 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.

• 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, the one which corresponds to the role played by the agent.

isInPushingArea	isInLeftGuidingArea	isInRightGuidingArea	isFlockFormed	Former Role	New Role
1	#	#	1	FuturePusher	Pusher
#	1	#	1	FutureLeftGuide	LeftGuide
#	#	1	1	FutureRightGuide	RightGuide
1	#	#	0	FuturePusher	FuturePusher
#	1	#	0	FutureLeftGuide	FutureLeftGuide
#	#	1	0	FutureRightGuide	FutureRightGuide
1	#	#	0	Pusher	FuturePusher
#	1	#	0	LeftGuide	FutureLeftGuide
#	#	1	0	RightGuide	FutureRightGuide
0	#	#	#	Pusher	FuturePusher
#	0	#	#	LeftGuide	FutureLeftGuide
#	#	0	#	RightGuide	FutureRightGuide

Table 2. The role table

Our *role table* is shown in table 2. We have 6 behaviors, each one corresponding to the fulfillment of one particular role, *i.e.* PusherBehavior, LeftGuideBehavior, RightGuideBehavior, FuturePusherBehavior, FutureLeftGuideBehavior, FutureRightGuideBehavior. As an example, the initial PusherBehavior table is shown in table 3.

Introducing roles in our architecture brings several benefits.

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

Table 3. The PusherBehavior table

• 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 presented in section 3.

• 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 example, for instance, playing a FutureLeftGuide role involves both reaching the *leftGuidingArea* and keeping the flock formed. Hence, an agent holding the FutureLeftGuide role can be rewarded when he reaches the *leftGuidingArea* and punished if the flock is scattered. We think that this way of providing intermediate reinforcement signals is less *ad hoc* than it was in [Matarić, 1994], for instance.

6 Empirical Study with roles

As in section 3, we studied the performance of this new architecture first with hand-crafted classifiers, and then with the adaptive algorithms turned on.

6.1 Using a hand-crafted controller

First of all, it appears that the results of the expert role-based controller are much better than in the case without roles. Over 2000 trials, we only had 32 failures (versus 533 in the previous case), and the average number of time steps is 626.05 (versus 1759.45 in the previous case).

The controller involving roles proved to be both more efficient and easier to design since the coupling between roles and actions was often straight-forward. Of course, this might come from the fact that our design is less accurate in the first case than in the second. Actually, we devoted much less time to the design of the role-based controller, but since we had designed the other one beforehand and learned from it, we cannot draw too many conclusions out of that.

6.2 Turning on the adaptive algorithms

We used the adaptive capabilities of our algorithm by turning on the adaptive algorithms during 9 trials only, then turning them off again and measuring the performance of the evolved system on 2000 more trials. Before adaptation, the initial controller of each agent included 60 rules. The adaptation mainly consisted in specializing these controllers, hence the number of rules reached 139, 149 and 153 respectively after the adaptation trials.

As a result, we further improved the performance of our controllers. Over 2000 trials, we only had 24 failures (versus 32 without adaptation), and the average number of time steps dropped to 568.55 (versus 626.05 without adaptation).

7 A further inquiry

Although the hand-crafted role-based controllers appeared more efficient than the ones without roles, we did want to check whether it would be more or less robust with respect to the size of the agents population. Therefore, we decided to test the robustness of both control policies when the number of sheepdog agents was increased from three to nine. We discovered that the performance of the controller without roles increases as the number of agents is augmented but, quite surprisingly, the performance of the controller involving roles rather tends to decrease. For each number of agents, we ran more than 500 experiments, so we are positive about the fact that this phenomenon is not a statistical artifact. In order to explain it, we examined more closely a lot of simulation runs displayed by a graphical interface. 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 3.

It happens that two sheepdog agents are able to drive the flock to the target area. Furthermore, this strategy seems more robust than the pre-conceived one, the ducks escape less often from the sheepdog agents chase. What happened is a typical case of favorable emergence.

This phenomenon can be explained quite straight-forwardly. In the case of controllers using roles, the behavior of the agents is very tightly determined. In the case of controllers without roles, on the contrary, the more sheepdog agents there are, the more likely is the situation where at least two agents are pushing the flock from behind as shown in figure 3 to take place.

First, this explains that, in the role-based case, if there are three agents, the performance are better, since the agents do exactly what they are told to, so the system less often goes to unforeseen situations resulting in degraded performance. But this also explains that the performance does not increase as the

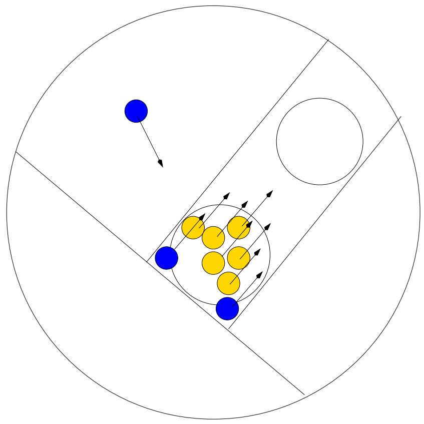

Fig. 3. An emergent strategy

number of agents is augmented, since the additional agents act exactly as the three original ones.

Moreover, the performance slightly decreases because, as long as the agents have not reached the area to which they have been assigned, they may be on the route of the flock already pushed by others agents. Then, the more agents there are, the more they tend to divert the flock away from its route to the goal area.

8 Future Work and conclusion

Rather than showing that using roles might be detrimental to the design of adaptive controllers, our experimental study has shown that a too tight preconception of the roles can result in a degraded performance if a better solution exists. This finding makes a strong argument for the design of an adaptive system which would be able to modify the organization of a society of agents when a better strategy is found by chance.

Therefore, we plan to extend the scope of our adaptive algorithms towards an architecture reflecting the one we designed 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 them and conditions on the situation. Implementing roles as internal states will give us a control system for an agent able to create and evolve its own roles. Furthermore, it should be able to globally reorganize its behaviors thanks to the adaptive processes.

To summarize, we presented a simulation experiment into which several agents had to solve a common task. In this particular case, it appeared that giving roles to the agents was an efficient way to design a control strategy. But it also appeared that a misconception of the roles could result in degraded performance and robustness of the strategy. Thus we advocated for the necessity to let the roles evolve as well as the behavior of the agents.

9 Acknowledgements

The authors want to thank the anonymous reviewers who provided very valuable advices to improve this paper.

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.

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

[Gérard, 2000] Gérard, P. (to appear, 2000). Combining anticipation and dynamic programming in classifier systems. In Stolzmann, W., Lanzi, P.-L., and Wilson, S. W., (Eds.), *Third International Workshop on Learning Classifier Systems*, Paris, France.

[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.

[Matarić, 1994] Matarić, M. J. (1994). Rewards functions for accelerated learning. In Cohen, W. W. and Hirsch, H., (Eds.), *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, San Francisco, CA. Morgan Kaufmann Publishers.

[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, 2000] Sigaud, O. (to appear, 2000). On the usefulness of a semi-automatized classifier system: the engineering perspective. In Stolzmann, W., Lanzi, P.-L., and Wilson, S. W., (Eds.), *Proceedings of the third International Workshop on Learning Classifier Systems*, Paris, France.

[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.

[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.

[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.