Using Classifier Systems as Adaptive Expert Systems for Control

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 complex simulations involving several interacting agents, the behavior of the overall program is difficult to predict and control. As a consequence, the designers have to adopt a trial-and-error strategy.
In this paper we want to show that helping experts to design simulation automata as classifier systems (CSs) by hand and using a semi-automated improvement functionality can be a very efficient engineering approach. Through the example of a simple multiagent simulation, we show how simulation automata can be implemented into the CS formalism. Then we explain how the obtained CS can be improved either by hand or thanks to adaptive algorithms. We first show how giving indications on the non-Markov character of the problems faced by the classifiers can help the experts to improve the controllers and we explain why adding modularity in the CS formalism is important. Then we show how the adaptive algorithms inherent to Learning Classifier Systems (LCSs) [1] can be used in such a context, we discuss our methodology and we present an experimental study of the efficiency of this approach. Finally, we point to difficulties raised by our perspective, we present directions for future research and conclude.

1 Introduction

In the domain of military operations, the simulations of fights between several aircrafts, whether at a tactical or a strategic scale, are becoming increasingly complex. The behaviors are more and more intertwined, there are more and more relationships between the actors in the air battle field, and the combinatorial of possible situations makes the evolution more and more difficult to foresee. In these domains, industrial and military studies make an intensive use of simulations. The core of these simulations are *simulation automata* [2], *i.e.* the parts of the programs which explicitly control the behavior of the agents. In

[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) throughout this paper.

[2] We call them automata whether these programs are explicitly implemented as finite state machines or not.

such a context, designing an automaton for a single agent in a simulator so that it manifests an appropriate behavior in any situation is becoming increasingly complex, too. As a consequence, the designers of automata which control the aircrafts tend to adopt a trial-and-error strategy and spend more and more time on this activity.

As a researcher in the industry of defense, our mission consists in helping these experts to automate this trial-and-error process by providing to them with the best of what adaptive techniques can do. Our challenge is to put into the hands of the experts a tool favoring the emergence of better solutions and minimizing the amount of work necessary to reconsider their design. Thus, in contrast with most researchers in adaptive behavior who tackle small scale problems or even toy-problems from scratch, we have to tackle very large scale problems where previously hand-crafted solutions exist.

There are many techniques and formalisms into which adaptive simulation automata can be designed. In our context, two key requirements for these techniques are that, as a starting point, the expert knowledge can be easily expressed in the formalism and that the result of the adaptation process is easily understandable by the experts. This is not the case, for instance, with recurrent neural networks [Beer and Gallagher, 1991], despite their efficiency. We have chosen the CS formalism because it meets these requirements and combines the adaptive power of both genetic algorithms [Goldberg, 1989] and reinforcement learning [Sutton and Barto, 1998].

In this paper we want to show that helping experts to design simulation automata as CSs by hand and using a semi-automated improvement functionality can be a very efficient engineering approach. We will present and discuss our methodology through an example. Since confidentiality concerns prevents us from publishing on the domain of military simulations, we have chosen to implement a simpler multiagent simulation for illustration purposes.

The rest of the paper is organized as follows.

In section 2, we present our illustrative simulation, and we show how we build a new simulation automaton. In section 3, we show how one can rephrase an existing simulation automaton written as a classical program into the CS formalism. Then we give in section 4 some methodological hints on how to improve these simulation automata by hand. In particular, we explain how giving indications on the non-Markov character of the problems faced by the classifiers can help the experts to improve the controllers. In section 4.4, we introduce the need for modularity and present how we can split a CS into modules.

In section 5, we turn towards the use of adaptive algorithms to improve these automata. We breifly present the LCS framework in general and the particular algorithm which we have developed. In section 6, we show through an empirical study how efficient these adaptive algorithms can be to improve the performance of the automata. We discuss the benefits of our methodology in section 7.

In section 8, at last, we present the problems which arise when adaptation results in the necessity of a global reorganization rather than in minor changes and discuss some areas which need further improvements. Finally, we conclude in section 9.

2 An illustrative experiment

In order to present in details some methodological aspects of our work, we will first present a simulator developed for illustration purposes.

We are inspired by the Robot Sheepdog Project from [Vaughan et al., 1998], involving a robot driving a flock of ducks towards a target position. The algorithm controlling the robot was first tested in simulation and then implemented on a real robot driving a real flock of ducks. As a testbed, we will use 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 multiagent coordination schemes, which corresponds to our industrial problem.

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 target 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 target area.

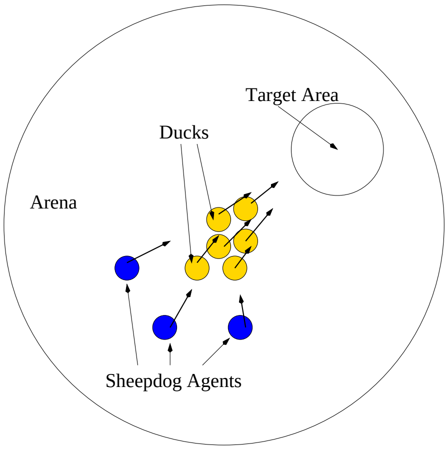

Fig. 1. The arena, ducks and sheepdogs

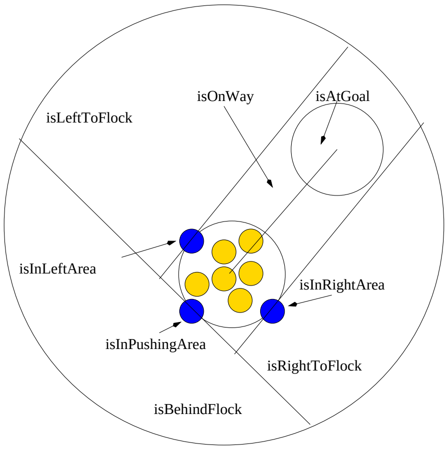

Fig. 2. Description of the situation

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

- to keep away from the walls of the arena [3];
- 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 target 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.

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:

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

The important point is that we defined pushing and guiding areas relative to current position of 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
- nobodyInLeftArea
- nobodyInRightArea
- 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

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

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
- goToFlockCenter
- goBehindFlock
- goToLeftArea
- goToRightOfFlock
- driveOutmostDuckToFlock
- driveClosestDuckToGoal
- goToOutmostDuck
- goToGoalCenter
- followFlockToGoal
- goToPushingPoint
- goToRightArea
- goToLeftOfFlock
- driveClosestDuckToFlock
- goToClosestDuck
- 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 Rephrasing an existing program in the CS formalism

As mentioned above, 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 CS formalism [Holland, 1975] appears as a natural candidate.

Since the work of [Wilson, 1994], a classical CS can be seen as composed of a population of rules, or *classifiers*, containing observations, consisting in a set of conditions, and actions:

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

The different parts of the classifier are strings of symbols in $\{0, 1, \#\}$, where $\#$, the "*don't care symbol*", means "either 0 or 1". This formalism can be extended from boolean numbers to integers without difficulty. The $[Condition]$ parts of the classifier are compared with an *input message* assigning its truth value to each condition at a given time step of the simulation. If the input message matches the $[Condition]$ part of one particular classifier, it can fire the corresponding action.

The complex simulation automata implemented by experts in conventional programs can be formalized very straight-forwardly into the CS formalism. A programmer who designs a simulation automaton must provide with two things:

- methods instantiating *conditions* in which the behaviors can be fired;
- methods (or functions, in the case of functional programming) implementing *basic behaviors* of the automaton or combinations of basic behaviors into more complex ones, and computing *parameters* to these behaviors (for instance, the relative position of the location where one agent must go).

In all the industrial simulators which we have examined, the conditions are expressed in terms of parameters describing the situation of the agent, either intrinsic to their own state or relative to relevant objects in the simulation.

It is easy to rephrase such a sequence of tests into a set of classifiers. The output of each *condition method* makes an entry in the input message, and one different action message must be mapped to each possible behavior, taking into account the fact that a method implementing a parameterized behavior makes as many behaviors as there are possible values for the parameters. This is what we have described in section 2.2 in the case of our ducks experiments.

Adding new conditions and parameters generally consists in considering new relevant objects, which may eventually be virtual (for instance, a position behind another object). As it will be emphasized later, this process of adding new conditions cannot be reduced to merely considering a logical combination of other available conditions.

Then the programmer must design a higher level function or method combining the conditions, the behaviors and their parameters. This function can take the form of a long sequence of successive tests stating under which conditions which behavior can be fired. Building the automaton in the CS formalism merely consists in designing as many classifiers as necessary to tell under which conditions each behavior should be fired. The CS in charge of controlling our sheepdog agents is shown on table 1.

4 Improving the CS automaton by hand

4.1 Reasons for not using fully automatized learning algorithms

Since CSs are well known to be a kind of adaptive tool, why should we not apply their learning algorithms as such to the classifiers written by experts ? There are three main reasons.

The reinforcement signal may not be available To apply learning algorithms to CSs, one must provide to the automaton with some reinforcement signals. In particular application domains, some reinforcement signals may be obvious: if our agents are aircrafts, every behavior which leads to their destruction must be avoided, so it must be punished. Depending on the purpose of the simulation, there may also be a goal, and agents could be rewarded when they reach it.

But, though a simulation driven only by these efficiency criteria could reveal interesting solutions, it is also necessary to take into account the necessity to design realistic behaviors with respect to behaviors found in real-world situations. For instance, if we want to test the potentialities of one aircraft against standard opponents, it is important that the simulated opponents act in a way as similar as possible to what a real opponent would do.

Eliciting what makes a behavior realistic is well known to be very hard. This is generally achieved through programming a first version of the automaton, observing the resulting behavior, then adding constraints and re-iterating until satisfaction. In this process, the control of the expert is necessary to check whether a proposed behavior matches the realism requirements or not. Hence this cannot be completely automated.

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

The evolved CSs need validation One strong reason for not using a fully-automated LCS in operational context is that the experts want to keep control of what the system is doing. It may be important that they can formally approve or reject the solution built by the system. Particularly, in the military domain, people are very reluctant to let the system take unforeseen decisions (and we won't blame them for that, will we?).

There are two complementary strategies to tackle this concern. The first one consists in giving to the experts a control on the evolution of the classifiers. The second one consists in applying adaptive algorithms off-line and validating the obtained controllers before using them in operational situations.

Experts learn from improving the system Another strong argument for letting the experts monitor the adaptation of their classifiers is that they may also learn a lot from the evolution of the classifiers and of the behavior of the simulation. The LCS may propose new solutions where the automaton needs improvement, and thus reveals weaknesses or misconceptions. This is perhaps the main justification for using adaptive programs in that context nowadays. The more the experts are involved in the adaptation process, the more they learn about the dynamics of the system.

Furthermore, observing the external behavior of the system can reveal some solutions which are both unforeseen by the expert and more efficient than the one they have designed. We describe such a case in an experimental framework in [Sigaud and Gérard, 2000]. In these favorable cases of emergence, it is important that the experts have a perfect knowledge of the underlying automaton, so that they can interpret what is happening.

4.2 Adjusting the CS automaton

As explained in section 3, once rephrased into the CS formalism, the classifiers controlling the behavior of the automaton take the form of a module whose inputs are conditions on the environment and whose outputs are the basic behaviors of the automaton. This module contains a list of classifiers which appears as a table. Most of the time, only a few conditions are relevant for firing one particular basic behavior, so there are many more don't care symbols ("#") than 0s and 1s in the table.

In table 1, we present the controller that we designed in order to implement the solution of our flock control simulator described in section 2.2. It can be seen that we only use 13 of the 16 available inputs.

There are two ways of improving a simulation automaton. The first one is to optimize the set of classifiers. As we will present in section 5, this is what fully automated LCS are good at. Indeed, once the CS description is settled, the only thing learning algorithms may do is to optimize the mapping between conditions and actions with respect to the fitness criteria. But the way experts improve their automata cannot be reduced to this optimizing process. From our experience, it is clear that the key difficulty for adjusting efficient simulation automata consists more in finding good inputs and basic behaviors than in optimizing the mapping between them.

Thus it seems necessary to help the experts to find where new conditions and actions should be added before trying to optimize the mapping between them. We address this point in the next section.

4.3 Non-Markov Indicators

In the *Markov Decision Process* (MDP) framework, an agent moves from *state* to *state* thanks to *actions*, and the distribution of probability of reaching the next state only depends on the state of the agent and on its action. When this condition, called the *Markov hypothesis*, is not verified, there is a *hidden state*, the agent must solve a *non-Markov* problem.

Trying to tackle non-Markov problems is hard. The algorithms which are intended to solve the hidden state planning problem from the formal framework – see [Kaelbing et al., 1998] for a very clear presentation – face a problem of combinatorial explosion and are restricted to solving very small-size *non-Markov* problems.

Alternatively, some people who work in reinforcement learning – for instance, [Whitehead and Lin, 1995,McCallum, 1996,Donnart, 1998,Lanzi, 1998] – try to adapt their algorithms to the non-Markov case, since it is well-known that the proof of convergence underlying the reinforcement learning algorithms are restricted to the Markov case [Lanzi, 2000].

In the case of multiagent simulations, designing the input and output of the agents so that the problem they solve is Markov is very hard, since agents cannot have a perfect knowledge of every relevant feature of all other agents and of the behavior of these agents. Hence we are doomed to solving non-Markov problems.

Some researchers [McCallum, 1993,Witkowski, 1997,Dorigo, 1994] have proposed algorithms which pick the relevant inputs from a set given beforehand. In these approaches, adding a new input strongly relies on the non-Markov character of the problem: if the system receives different rewards for the same action in what it considers as the same situation, this means that in fact the situation is not the same, then the problem is non-Markov and one input must be added in order to distinguish better. Thus, checking whether a problem is Markov might be a key factor in the improvement process.

If the problem solved by an agent is non-Markov, then there must be at least one module of the agent which solves a non-Markov problem. In such a module, there must be at least one classifier facing a hidden state. The fewer hidden states there are in the problem which an agent must solve, the easier it is to find a good policy, and the more efficient the agent will be. Furthermore, improving it with a learning algorithm will be easier.

This has lead us to the intuition that giving to the experts indications that some elements in their solution are facing a non-Markov problem would be very informative to them. And it is!

Classifiers facing a hidden state can be identified by recording, for each classifier and each state into which it has been fired, a list of all the subsequent states. If there are several subsequent states for the same initial state, the problem is non-Markov. We can define the *non-Markov rate* NMR of the classifiers as the number of subsequent states divided by the number of initial states into which it has been fired.

$$NMR = \frac{|non - Markov\, cases|}{|cases\, when\, fired|}$$

Modifying by hand the classifiers which have the highest non-Markov rate has proven to result in very efficient improvements of the automata. As long as the classifier contains some "#" in its [*Condition*] part, it can be further specialized.

Interestingly, adding inputs to classifiers is what most fully automated LCS do when they rely on specialization mechanisms, but they do so by picking a new condition in a set of inputs already available. But when the [*Condition*] part is completely specialized, the only possible improvement consists in adding new inputs to the global automaton. The easiest solution, here, rather than trying to learn to solve the non-Markov problem as such, is that the expert adds new inputs when necessary.

4.4 Modular Classifier Systems

From table 1, it can be seen that the representation using a basic controller could be more compact: there are a lot of "#", which means that each expert classifier uses very 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.

Furthermore, the adaptive algorithms of the LCS are slow to converge on such a representation, since the search space is very large. Here we have only used 13 inputs, but simulation automata designed for industrial purpose can be huge. In the case of simulations of military operations, some automata designed by experts to control a single agent involve around 200 inputs.

Representing such automata as tables with 200 columns would make them unreadable for experts. Hence, it seems necessary to split the automata into smaller modules, building *modular classifier systems* (MCS).

One good way of splitting a CS devoted to fulfilling a behavior into modules is to identify several lower level behaviors whose conjunction acheives the global behavior. The lower level behaviors can in turn be split into even lower level behaviors, or they can be represented directly as CSs. This gives rise to a hierarchical decomposition of the global behavior. Then, if they act on independent actuators, some of the basic level behaviors could be run independently. If this is not the case, one must find a way to synchronize them.

Among many architectures which may result from these considerations, we chose to develop a simple one where a high level CS, or *decision CS*, is devoted to monitoring the execution of several basic behavior CSs, choosing one among them at each time step. The decision CS shares their inputs with the behavior CSs, but it also needs an information on previous decisions. Its output tells which basic behavior must be fired at a given time step. Interestingly, our architecture is very similar to those of [Wiering and Schmidhüber, 1997] and [Sun and Sessions, 2000], how both presented learning algorithms devoted to applying adaptive algorithms to them.

4.5 Modularity in our experimental set-up

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 controller presented in table 1 so as to make an explicit use of roles. Our new architecture contains two kinds of components:

• The *role CS* 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. Our *role CS* is shown in table 2.

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

In that particular case, each lower level module is seen as devoted to achieve one particular basic behavior, and the higher level module is seen as deciding which basic behavior should be fired according to the role of the agent. The role itself depends on the situation of the agent.

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)

We have six behaviors, each one corresponding to the fulfillment of one particular role, *i.e.* FUTUREPUSHERBEHAVIOR, PUSHERBEHAVIOR, FUTURELEFTGUIDEBEHAVIOR, LEFTGUIDEBEHAVIOR, FUTURERIGHTGUIDEBEHAVIOR and RIGHTGUIDEBEHAVIOR.

As an example, the initial PusherBehavior CS is shown in table 3. The complete set of behavior tables can be found in [Sigaud and Gérard, 2000].

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

Table 3. The PusherBehavior table

5 Applying adaptive algorithms

Up to that point, we have shown how to rephrase an expertise into the CS formalism. This formalism is convenient for improving the automata since adjusting by hand the global behavior of one automaton merely consists in changing some values into others in the classifiers. Using CSs in such a way that the knowledge of an expert is coded into a set of classifiers would make the reader feel that we have re-invented a sort of expert system devoted to control. But a CS is more than an expert system. Since adaptive algorithms can sometimes be applied to it, our framework is rather what we call an *Adaptive Expert System*.

5.1 Learning Classifier Systems

Up to that point, we have presented CSs as a formalism to code an hand-crafted automaton. Rather than being done by hand, the adjustment process can be more or less automated, using LCS algorithms. The first LCSs were designed by [Holland, 1975]. In these initial versions, the strength of classifiers was modified by the *Bucket Brigade* algorithm according to the estimated reward received by the agent for firing the classifier. The population of classifiers was 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.

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 GA. 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 much more efficient in what they are designed for.

5.2 Our Algorithm

Our algorithm, YACS, is an instance of heuristic search LCS based on anticipation. Its 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, 2000], in this volume.

6 Empirical study

In order to check whether our role-based controller was more efficient than the basic one, and whether it could be improved by our learning algorithm, we conducted the following experimental 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 agents population, since the role-based controllers are designed for a group of three sheepdogs.

Therefore, we decided to test the robustness of both control policies when the number of sheepdog agents was increased from three to twenty.

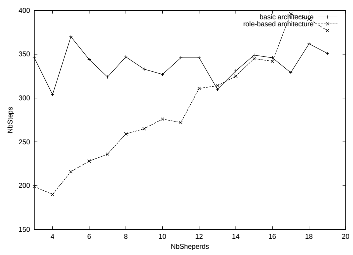

Fig. 3. Robustness of basic and role-based controllers to an increasing number of sheepdogs

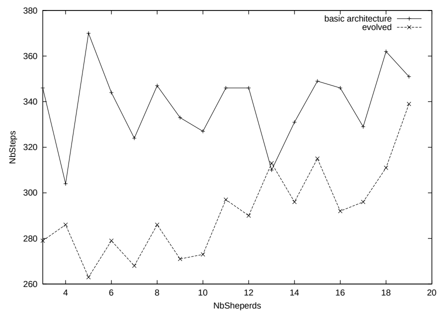

Fig. 4. Robustness of basic and evolved controllers to an increasing number of sheepdogs

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 time for the sheepdog agents to surround the flock and drive it to the target area from a lucky initial situation.

If a trial lasts more the 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. But, since there are very few failures, their impact on the results is not very significant. More precisely, 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 much better with three sheepdogs than the basic one, but that the basic architecture is more robust to an increasing number of sheepdogs. The complete presentation of this robustness study and further results are given in [Sigaud and Gérard, 2000].

It can also be seen in figure 4 that the controller obtained form applying adaptive algorithms to the basic architecture during two trials performs better than the hand-crafted one, and is still robust.

Hence, it must be highlighted that applying adaptive algorithms to our hand-crafted controllers results in a significant improvement of the performance. From an engineering perspective, it means that the expert who has to design a controller can write a first draft of this controller before applying adaptive algorithms to optimize it. Hence he spends much less time in this design, which is very appealing in an industrial context. This finding shows that the CS formalism is adapted for coding controllers both because the knowledge of the expert can be easily represented in it and because applying optimization algorithms is straight-forward in the formalism.

7 Discussion

7.1 Controlling the adaptation

If we want experts to control the adaptation of classifiers, our tool should highlight all the modifications of the initial classifiers and let them approve or reject the changes. Another way to tackle the validation concern could be to let the expert constrain the exploration of the state space. Actually, what we do is a combination of the two.

The YACS system presented in [Gérard and Sigaud, 2000] is a combination of different functionalities used to improve the classifiers. One of them, the anticipation learning mechanism, improves the $[Effect]$ part, another one specializes the [*Condition*] part, and one is devoted to combining conditions with new actions. Each of these functionalities can be switched on or off, depending on the use we want to make of the LCS. Typically, in a fully automated experiment, all the switches are set on. In the case where we use expert classifiers as a starting point, we set some of them off in order to insure that the classifiers only get adjusted thanks to specialization. This guarantees that the system only explores the domain of the state space the expert want it to explore. This is an indirect way to constrain the adaptation process. But in our view, this does not eliminate the necessity of a validation of the resulting classifiers by the expert.

7.2 Benefits of our methodology

Letting the experts elicit their knowledge to set the initial CS is beneficial with respect to using a LCS from scratch with a random CS. It is well known that, when we use a LCS from scratch, the initial exploration phase before the system starts to converge can be very long even for simple problems. Using the expert knowledge that way allows us to cut through that phase which may be prohibitively long for our real-world applications.

Furthermore, 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 CS* is much smaller than the CS presented in section 4.2.

• 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, 1994], for instance.

8 Future work

8.1 Global reorganizations

We have shown that an expert must have a pre-conception of the way by which his agents will solve the problem at hand in order to define their inputs and behavior sets. This is only once he has implemented the agents and observed their behavior that he will be able to refine his initial conception, making changes in the behavior modules.

But it sometimes happens that observing the behavior of the system during the adaptation process reveals a completely different strategy to solve the problem. In these cases, implementing the new solution may require new inputs, new behaviors and new mappings between the two.

These cases where a global reorganization seems necessary give a strong argument for adopting a modular approach, since some modules may remain unchanged despite the new perspective. But the design of a fully-automated LCS which would be able to tackle such global reorganizations is a very challenging research goal. Therefore these cases also give a strong argument for adopting a semi-automated approach, since the eye of the expert is necessary to identify when such global redescriptions are necessary.

8.2 Improving the tool

In order to get more convenient, our tool should include a graphical interface, devoted both to the design of adaptive automata and to their improvement.

In order to help to improve adaptive automata, the interface must highlight the classifiers which need to be improved according to the non-Markov indicators. It must also let the users validate or reject new classifiers.

We have presented a way to use CSs as a particular case of expert systems. Rather than expressing the knowledge of the expert directly into the CS formalism, which makes it hard to read, we should also use a higher level input/output language to translate strings of 0, 1 and # into readable assertions like "if ($param_1 > 50.0$) and ($condition2$ holds) and (...) then $actThisWay$". For instance, the SAMUEL system [Grefenstette et al., 1990] uses such a language for interface concerns.

8.3 Solving non-Markov problems

More interestingly for the scientific community, we also want to automate more processes. We have shown that we can indicate to the users the classifiers which have a high non-Markov rate. If a correlation can be found between the variability in subsequent situations and an input which does not appear in the condition part of the classifier, then adding this input to the classifier might solve the missing information problem. We have to go into further investigation in this area before presenting results, but solving non-Markov problems is the next stage in the agenda of YACS.

8.4 Structural Modularity

Structural modularity is not the kind of temporal modularity presented in section 4.4. High level conditions can be a logical combination of lower level conditions. In such cases, the computation of the truth value of these conditions can be rephrased into a small CS whose conditions are the lower level ones and the action is the assertion of the truth value of the higher level one. These modules are connected one to another so that the output of one module makes an entry in the input message of another module. This can be a good way to break the complexity of the conditions of the automata.

It is interesting to use this form of modularity when one lower level module can be re-used by several higher level modules. Our attempt to use this form of modularity in our flock driving experiment was not successful, but the need for structural modularity is not obvious in such a problem, since the automaton is not very complicated. In our industrial applications, on the contrary, there should be a lot of reusable modules.

Combining learning algorithms with structural modularity raises interesting questions. When a higher level module is reinforced, how should it share its reinforcement with lower level modules which give its inputs ?

Though we did not tackle this question yet, the learning algorithm presented in [Gérard and Sigaud, 2000] already measures how relevant is a particular input to a particular classifier. We will use this information to design a structural reinforcement sharing algorithm.

9 Conclusion

In this paper, we have presented an application of CSs to a moderately complicated multiagent problem. We have drawn some lessons out of this experiments on how could CSs be used in real world domains.

The main message of that paper is that using the CS formalism can be a very efficient way to implement or rephrase simulation automata written by experts. It provides with both a clear and concise representation of the "intelligence" of the system, which can be easily handled through a graphical interface, and a convenient way to adapt this intelligence through a trial-and-error process, either by hand, automatically, or even through a mixture of the two.

The second message is that pointing out classifiers dealing with the non-Markov property of the problem is very helpful and gives an efficient indication of what must be improved in the design of the automaton. This is not a surprise since this is at the core of the algorithms of XCS [Wilson, 1995], but we proposed a new methodology to involve the experts in the improvement process when it is necessary.

10 Acknowledgements

The author wants to thank the reviewers of an early version of this paper and all the IWLCS2000 workshop attendees for valuable comments on this work.

References

[Beer and Gallagher, 1991] Beer, R. D. and Gallagher, J. C. (1991). Evolving dynamic neural networks for adaptive behavior. *Adaptive behavior*, 1(1):91–122.

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

[Donnart, 1998] Donnart, J.-Y. (1998). *Architecture cognitive et propriétés adaptatives d'un animat motivationnellement autonome.* PhD thesis, Université Pierre et Marie Curie, Paris, France.

[Dorigo, 1994] Dorigo, M. (1994). Genetic and non-genetic operators in ALECSYS. *Evolutionary Computation*, 1(2):151–164.

[Gérard and Sigaud, 2000] Gérard, P. and Sigaud, O. (to appear, 2000). Yacs: Combining dynamic programming with generalization in classifier systems. In Stolzmann, W., Lanzi, P.-L., and Wilson, S. W., (Eds.), *LNCS: Proceedings of the Third International Workshop on Learning Classifier Systems.* Springer-Verlag.

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

[Grefenstette et al., 1990] Grefenstette, J. J., Ramsey, C. L., and Schultz, A. C. (1990). Learning sequential decision rules using simulation models and competition. *Machine Learning*, 5(4):355–381.

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

[Kaelbing et al., 1998] Kaelbing, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101.

[Lanzi, 1998] Lanzi, P. L. (1998). Adding memory to XCS. In *Proceedings of the IEEE Conference on Evolutionary Computation (ICEC98).* IEEE Press.

[Lanzi, 2000] Lanzi, P. L. (2000). Adaptive agents with reinforcement and internal memory. In Meyer, J.-A., Wilson, S. W., Berthoz, A., Roitblat, H., and Floreano, D., (Eds.), *From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior*, pages 333–342, Paris. MIT Press.

[Matarić, 1994] Matarić, M. J. (1994). 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.

[McCallum, 1993] McCallum, R. A. (1993). Overcoming incomplete perception with utile distinction memory. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 190–196, Amherst, MA. Morgan Kaufmann.

[McCallum, 1996] McCallum, R. A. (1996). Learning to use selective attention and short-term memory. In Maes, P., Mataric, M., Meyer, J.-A., Pollack, J., and Wilson, S. W., (Eds.), *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, pages 315–324, Cambridge, MA. MIT Press.

[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, 2000] Sigaud, O. and Gérard, P. (to appear, 2000). Being reactive by exchanging roles: an empirical study. In Hannebauer, M., Wendler, J., and Pagello, E., (Eds.), *LNCS : Balancing reactivity and Social Deliberation in Multiagent 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.

[Sun and Sessions, 2000] Sun, R. and Sessions, C. (2000). Multi-agent reinforcement learning with bidding for segmenting action sequences. In Meyer, J.-A., Wilson, S. W., Berthoz, A., Roitblat, H., and Floreano, D., (Eds.), *From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior*, pages 317–324, Paris. MIT Press.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning, an introduction.* MIT Press, Cambridge, MA.

[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.

[Whitehead and Lin, 1995] Whitehead, S. D. and Lin, L.-J. (1995). Reinforcement learning of non-Markov decision processes. *Artificial Intelligence*, 73(1-2):271–306.

[Wiering and Schmidhüber, 1997] Wiering, M. and Schmidhüber, J. (1997). HQ-learning. *Adaptive Behavior*, 6(2):219–246.

[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.