Learning Behaviours of Autonomous Agents

Benoit Isaac
Lady Margaret Hall
`msc@helicos.com`

Oxford University
Computing Laboratory

SUBMITTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

August 2005

Abstract

This project is an investigation into the field of Machine Learning and Multi-Agent Systems. It aims at trying to learn the behaviour of an autonomous agent. The main idea is to assume that the movements of an agent can be described as a set of rules, in order to apply the Learning Classifier System paradigm, an extension of the Genetic Algorithms in Evolutionary Computation.

As a case study, the behaviour of a simulated duck has been transformed into a set of rules, and an object-oriented simulator with a learning system "plugged in" was built in Java. This thesis describes this experimentally-oriented project.

Contents

Acknowledgements

I would like to thank Dr Stephen Cameron, my thesis supervisor, for letting me choosing an interesting and challenging project. I am grateful to Dr Jeff Sanders, my college supervisor, for his advices and suggestions.

I would also like to thank the *Institut d'Informatique d'Entreprise*, my French Engineering School, for giving me the opportunity to follow this Master of Science in the University of Oxford.

Many thanks to all my friends, who were present when I needed help, either physically or *gmail*ly; and to my family for their patience and support.

Merci à la Basement Society.

Chapter 1

Introduction

The two main topics of this dissertation are *Machine Learning* and *Multi-Agent Systems*. We first give a brief introduction to these topics. Then we present the two main sources of inspiration for this MSc Project, the Robot Sheepdog Project and the field of Evolutionary Computation.

At the end of the chapter, an outline of the dissertation is given.

1.1 Topics

1.1.1 Machine Learning

Machine Learning is a an area of Artificial Intelligence which aims to develop techniques allowing computers to ”learn”. According to the *Oxford Dictionary*, learning is:

”gaining knowledge or skill by studying, from experience, from being taught.”

Thus Machine Learning is interested in finding possible methods to make computers using *experience* to act more rationally than if they didn't have any knowledge of the past. *Acting rationally* is acting in order to achieve the best outcome or the best expected outcome, if there is uncertainty.

In order to improve the performance of a system from experience, one might use several types of algorithms; the field of Machine Learning usually distinguishes three types (described, for example, in [Russell and Norvig, 1995]):

- **supervised learning** involves learning a function from a set of examples with inputs and outputs given. At each step, a *”teacher”* can tell the system whether the output it has chosen is correct or not. A typical example of supervised learning is a system trying to recognize hand-written numbers: a human can almost always tell if the guess of the system is good or not.

- **unsupervised learning** involves learning patterns in the input, without any information about the output. The system tries to learn which possible inputs can exist, from a set of examples. But since it doesn't know what is a ”good” or a ”correct” output, a purely unsupervised agent cannot learn what to do.

- **reinforcement learning** is the most general of the three types; it involves learning from reinforcement (or *reward*). At each step, the system might not get an information about whether or not its action was correct; but it sometimes receives *reward* or *punishement* when the state of the environment can be seen as good or bad. A typical example of reinforcement learning is a system trying to play chess: since it is very difficult to know whether a simple move of a pawn is useful, usually no reward is given for this kind of move. However, when the system manages to win against its opponent, a reward can be given. If the system looses, it can be punished.

In this project we developed a supervised learning system, although the framework we created can be easily adapted to a reinforcement learning system.

1.1.2 Multi-Agent Systems

In Computer Science, a *Multi-Agent System* is a system composed of several agents capable of mutual interaction. The agents can be autonomous (i.e. whose decisions are not controlled by another entity) or not; although for the system to be called *Multi-Agent*, at least two autonomous entities must interact, each of those having the possibility to control several sub-entities. The agents can be artificial (robots, software agents) or natural (human beings, animals).

Multi-Agent systems have become a very active field of research inside Artificial Intelligence, partly because :

1. the computational power available today allows researchers to simulate complex multi-agent scenarios which were impossible to be modelled some decades ago;

2. it is more and more believed that complex behaviours and self-organization can be manifested in a multi-agent system through the interaction of simple individual strategies, thus allowing the development of interesting complex applications by building only simple agents.

One shall not forget as well that another important factor of the development of studies in multi-agent environments is the military interest exhibited by some countries. Indeed, building teams of autonomous robots appears more reliable (if one entity is destroyed, the others can continue their job) and cheaper (most of the time the simple entities are identical, thus reducing production costs) than other solutions.

Research in Multi-Agent Systems is focused on several topics, among which we find:

- beliefs, desires, intentions and knowledge (which was the topic of the course *Logic Of Multi-Agent Information Flow)*,

- cooperation and coordination,

- multi-agent learning,

- distributed problem solving, etc.

Here we are interested in learning in a multi-agent context, with inspiration drawn from the Robot Sheepdog Project.

1.2 Inspiration

1.2.1 The Robot Sheepdog Project

The Initial Project

The Robot Sheepdog Project (RSP) (1995-1998) was a collaboration between SRI and the Universities of Bristol, Leeds and Oxford. This multi-disciplinary project covered robot building, machine vision, behavioural modelling; it was a first investigation into "**Animal-Interactive Robotics**" (AIR), i.e. the application of autonomous robots to control the behaviour of farmyard animals.

The aim of the project was to demonstrate the ability of a robotic system to exploit and control an animal's behaviour to achieve a useful task.

The initial idea was to construct a robot that would be able to replace a sheepdog, i.e. managing several tasks such as: herding the flock towards a specific goal, isolating an individual (for a special treatment), bringing back an isolated individual into the herd, etc.

Since dealing with sheep was thought to be a too big jump in one step, the RSP team developed a robot herding ducks instead (Figure 1.1c), because ducks are known to have a behaviour quite close to the behaviour of sheep. Indeed, sheepdogs are sometimes trained with ducks in the real-world.

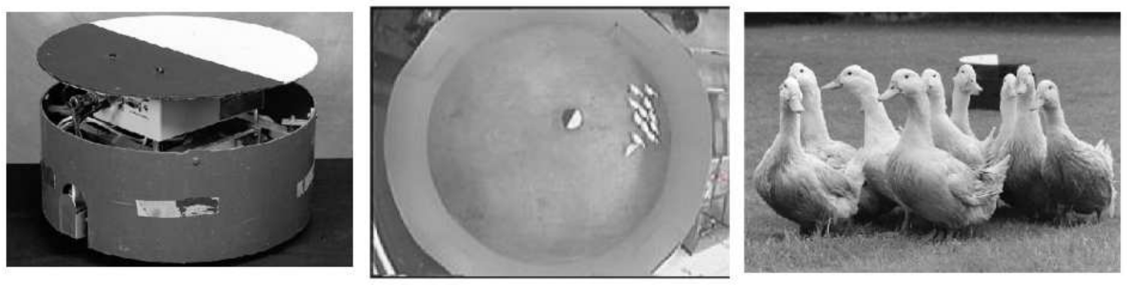


Figure 1.1: The RSP Project. From left to right : (a) The robot Rover - (b) Rover herding the flock (view from the overhead camera) - (c) Rover and the ducks. Adapted from [Vaughan, 1999].

After having designed the algorithm that could handle a flock of ducks in simulation (using a mathematical model of the ducks' behaviour), a robot (Figure 1.1a) was created to implement this algorithm in a real-world situation (Figure 1.1b), to check the validity of the model.

The RSP now

The Spatial Reasoning research group in the Computing Laboratory is continuing research with the use of robot ducks and robot sheepdogs, and the extensive use of simulation. Students projects within the RSP framework include the development of new control algorithms to deal with walls and corners (the initial arena was a circle) in [Kumar, 2001], enhancing the path planning capabilities of the robot in [Batterink, 2004] or using a neural network for the control of the robot in [Lurcock, 2001].

Extending the RSP

The RSP team successfully built a system that was capable of herding animals into a goal position by interacting with their natural behaviour. The model they designed was therefore proven to be a robust, general flock-control method. However, in the conclusion of his thesis, Richard Vaughan proposed the following enhancement to the system :

> *"While it is a stated requirement of an AIR system that it should not require manual optimization to work with any specific animal or group, the ability to self-optimize or adapt during run-time could be a very useful extension."*

Since the use of animals is far beyond the scope of a Master of Science project, in this dissertation we will try to find ways to implement a self-optimization technique in a simulated multi-agent system, using methods and algorithms from the field of Artificial Intelligence, especially Evolutionary Computation.

1.2.2 Evolutionary Computation

Evolutionary Computation is the field interested in metaheuristic optimization algorithms that are inspired by biological evolution. From the first ideas described by [Holland, 1975], a lot of research has been done to use these analogies of natural processes for the design of *learning* agents.

Genetic Algorithm

Holland proposed in [Holland, 1975] to use the notions of evolution described by Darwin as a way to create a kind of algorithm which was later called the *Genetic Algorithm*. Genetics Algorithms (GAs) are search algorithms based on the mechanics of natural selection and genetics. They involve describing the possible solutions of a problem as strings of bits. These strings can then be used as a sort of "D.N.A." and natural evolution processes are used to make the algorithm converge towards a global optimum.

The standard version of a GA uses a population of individuals describing solutions to the problem in terms of bits (their genotype), and simulates the evolution of this population through thousands of generations. A new generation is created from the previous one by **selecting the fittest individuals** (i.e. the best ones according to the fitness function, which describes how well the individuals performs for the optimization problem) and creating offsprings from previous individuals using genetic techniques such as **crossover** and **mutation** on their genotype. The Genetic Algorithm is expected to find solutions with increasing fitness throughout the generations; in the same sense that natural evolution is expected to "produce" the most adapted individuals in their environment.

Since randomness is introduced in the process (through mutation, crossover, and selection), this algorithm is only an *heuristic*, ie. it is not assured that this method will find an optimum. However, in several cases, GAs have been quite successfully applied to complex optimization problems like wire routing, scheduling, traveling salesman problems, etc. (see [Goldberg, 1989] for more details).

Learning Classifier System

Learning Classifier Systems (LCS), also designed by Holland in [Holland, 1986] are *"a kind of rule-based system with general mechanisms for processing rules in parallel, for adaptive generation of new rules, and for testing the effectiveness of new rules"* [Michalewicz, 1996] .

In a standard Classifier System (CS), the rules have the form :

$$[\text{Condition}] \rightarrow [\text{Action}]$$

where the condition part is encoded as strings of symbols in {0,1,#} (where # means "either 0 or 1") and the action is taken among a set of actions (therefore it can be encoded in different ways).

As an example (adapted from [Dorigo and Colombetti, 1994]), consider a simple task for a robot: chasing a light. If the condition describes where the light is (according to the sensors), then the action the robot should do is to go towards this light.

The figure 1.2 shows an example of classifier used in the Alecsys system created by Dorigo and Colombetti. We can see that, since the light can be in 16 different positions, there are 16 different rules that completely describe the optimal behaviour of the agent.

The task of the Learning Classifier System is to find these 16 rules, by evolving its set of rules (initialized randomly) using a **reinforcement process** (which provides reward when the action chosen by the system leads to a "good" outcome) and a **rule discovery system** (a Genetic Algorithm).

Within the Learning Classifier System framework, different extensions of the initial description by Holland have been proposed, for example the XCS (eXtended Classifier System, described in details in [Butz and Wilson, 2001]). We will give more details about the specific LCS we used and the adaptations we made in section 4.

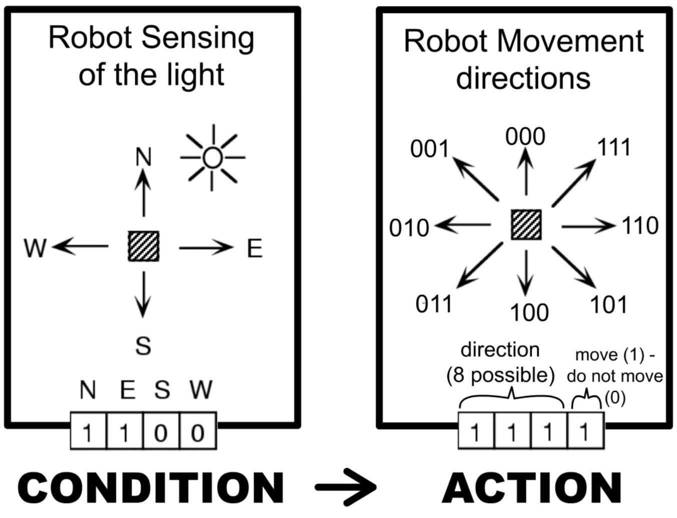


Figure 1.2: A classifier for the *Chase* behaviour. Adapted from [Dorigo and Colombetti, 1994].

1.3 Dissertation Outline

Chapter 2: Problem and Context Prior to the choice of the problem, a lot of relevant literature has been studied, in order to find an interesting problem to investigate. This section describes part of this literature, and explains in more details the global research question and the specific problem we have decided to focus on.

Chapter 3: Methodology In this section we describe the requirements of our system, and explain why Object Oriented Programming fits perfectly well to create such a system. We give details about how we are going to investigate our research problem.

Chapter 4: Learning Classifier System We give in this chapter the specific features of the Learning Classifier we designed, which is an adaptation of the Zeroth Level Classifier System.

Chapter 5: Implementation A program of about 5000 lines of code, with an advanced graphical user interface and an extensive modularity, has been developed; this chapter explains how we implemented our system using the design choices described in Chapter 3.

Chapter 6: Results and Discussion Here we give the results we obtained for different experiments with our system; we detail the successive changes we made to our system and why we made them.

Chapter 7: Conclusion A final overview of the project is given, and we explain how our results might be useful for further research.

Chapter 2

Problem and Context

In this chapter we present a summary of the relevant work in the different areas of research linked to the problem we chose, and show that the use of a Learning Classifier System within a Multi-Agent framework is quite unusual, as far as the author can see.

Then we describe precisely the objectives of this project, and explain why we think this topic is important.

2.1 Related Problems

We have presented previously the two domains related to this dissertation, *Machine Learning* and *Multi-Agent Systems*. Even if *Machine Learning* can be considered as a field in itself, most of the time researchers have a specific problem in mind, and they try a learning technique to tackle this problem. Below we give several problems (mostly multi-agent) where learning techniques have been applied.

2.1.1 The *Robocup* competition

In multi-agent systems, the main project leading most of the research in the field is the *Robocup* competition [Kitano et al., 1997], where teams of robots play soccer against each other. The competition started in 1997 with the following published goal:

> ”By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team.”[1]

This a great challenge for robotics and multi-agent systems. However, the most visible part of the project, competition of soccer played by robots, encouraged more research on *how to build* efficient small robots able to deal with a fast-changing environment. Most of the teams spend their time on vision systems, motors control, and other practical problems. To tackle this problem, and develop research in the

[1]From the official *Robocup* website : `http://www.robocup.org`

multi-agent part of the competition, the organisers of the competition created several different leagues, including one where the robots used (Sony Aïbo) are already well defined and available on the market, thus avoiding some of the practical problems. But this league is also dominated by the teams who actually manage to re-program most of the features of this robot, including the vision system and the driving system.

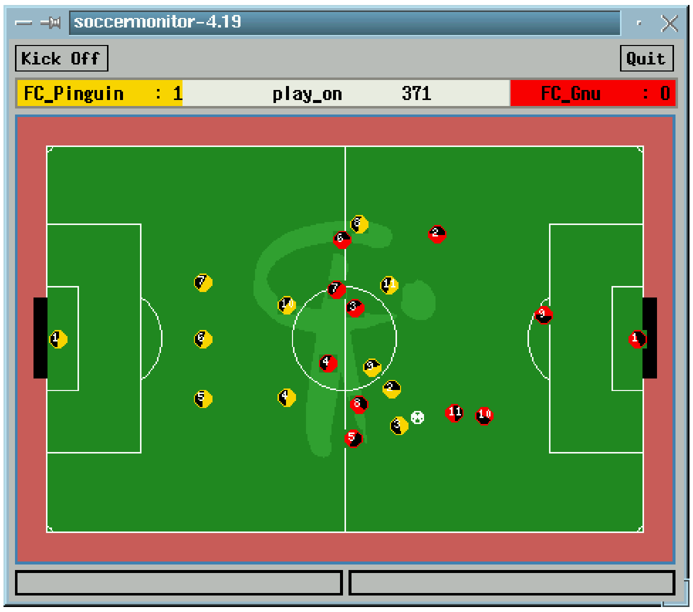


Figure 2.1: Screenshot of the Robocup Simulation league, a complex multi-agent environment.

The last league, the **simulation league**, where programs describing agents compete against other programs, has seen most of the work oriented towards a better understanding of a multi-agent system. Although learning has been present since the creation of this league (see for example the use of Genetic Programming in [Luke et al., 1998]), the focus has been more on *cooperation* of well-defined simulated agents (one's own team), rather than learning the behaviour of the opponent team. When learning is involved, it is most of the time to learn basic behaviours (controlling the ball, passing the ball) or learning how to combine pre-defined basic behaviours to form a team strategy (see [Salustowicz et al., 1998] or [Brusey, 2002] and, for an account of layered learning, [Stone and Veloso, 1998]).

This method (i.e. studying how to combine basic behaviours) is called the *Behaviour-Based* approach, which is another framework in the domain of Autonomous Agents.

2.1.2 The Behaviour-Based approach

This approach aims at decomposing an agent's behaviour into small behaviours. These primitive behaviours (for example, the reflex to avoid a wall) are arranged in different ways (e.g. in layers or in parallel) and the overall behaviour of the system emerges through the interaction of these primitive behaviours. If we consider each rule of a Classifier System (*Condition* → *Action*) as a primitive behaviour, then one might see the Classifier System as a Behaviour-Based approach.

Behaviour-Based robotics have been studied intensively by Arkin and Balch (see for example [Arkin, 1998], [Balch and Arkin, 1998], [Balch, 1998]) but here again learning techniques have been applied mostly to cooperative behaviours (e.g. formation control for a multi-robot team).

2.1.3 The *Animat* Problem

This dissertation could also be linked with the *Animat* approach [Wilson, 1985; Wilson, 1987], which aims at designing *animats*, i.e. simulated animals or real robots whose rules of behaviour are inspired by those of animals. Animals start with some hints about their environment at the beginning, and, through a complex process of learning, they manage to deal with new situations in a multi-agent environment, using similarities and inferences from their growing knowledge base.

Wilson was the first to link the Learning Classifier System to the Animat problem in [Wilson, 1987], but the LCS he created (the "***Boole***" system) was learning a simple Boolean function, i.e. a mapping from binary strings of length L to {0,1}. He chose the Boolean "multiplexer" function as a good benchmark for testing his ***Boole*** system. The basic function can be defined by thinking of each input string as having k "address" bits a_i and 2^k "data" bits d_i, with the string represented by

$$a_0a_1...a_{k-1}d_0d_1...d_{2^k-1}$$

The value of the function is given by the value (0 or 1) of the data bit that is indexed by the pattern on the address bits. Thus the smallest multiplexer ($k = 1$) can be fully described by the following four rules :

$$\begin{array}{ccccc} 0 & 0 & \# & \rightarrow & 0 \\ 0 & 1 & \# & \rightarrow & 1 \\ 1 & \# & 0 & \rightarrow & 0 \\ 1 & \# & 1 & \rightarrow & 1 \end{array}$$

Here $k = 1$, so there is only one address bit (the first bit):

- if its value is 0, then the value of the function is the value of the *second* bit (no matter what the third bit is, that's why we have the *"don't care"* symbol #);
- if its value is 1, then the value of the function is the value of the *third* bit (no matter what the second bit is).

The function can be written in a disjunctive form : $F_3 = \overline{a_0}d_0 + a_0d_1$.

In Wilson's experiments, the ***Boole*** system managed to learn quite accurately the rules of a multiplexer where $k = 2$ (ie. 6 bits for the inputs), by receiving payoff when its decision was good and using a Genetic Algorithm as a rule discovery system.

One of the most extensive study of a Learning Classifier System with an Animat is the work done by Dorigo and Colombetti called "*Robot Shaping*". Their book

[Dorigo and Colombetti, 1998] describes a framework to design small agents able to learn and perform relatively well for primitive behaviours like avoiding or approaching an object, escaping from a predator or chasing a prey (cf figure 1.2 page 16), and combinations of these behaviours.

However, most of this work does not include a multi-agent environment, apart from a basic *Prey/Predator* type of environment, where the Agent tries to learn how to escape from a moving predator, or how to chase a moving prey.

2.1.4 The Robot Sheepdog Project

Although learning was not present in the initial Robot Sheepdog Project (cf section 1.2.1 page 13), the multi-agent problem it provides was considered interesting and further studies were done. These include the application of Genetic Algorithms to the RSP problem by an undergraduate student ([Lurcock, 2001]), and an extensive study by Sigaud and Gérard: [Sigaud and Gérard, 2000; Sigaud and Gérard, 2001a; Sigaud and Gérard, 2001b].

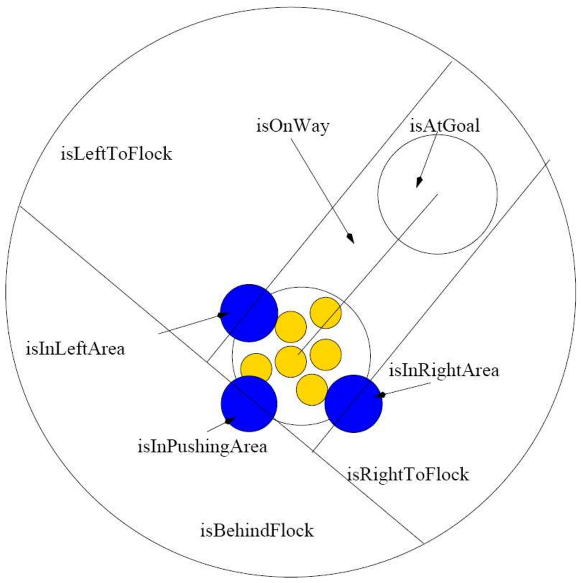


Figure 2.2: The situation and the tests describing the position of one sheepdog. Adapted from [Sigaud and Gérard, 2001a].

Sigaud and Gerard considered the case where several sheepdogs are used to herd the flock of ducks, and wanted to highlight the fact that using different roles for each sheepdog might be the best way to herd correctly the flock. Therefore, they implemented a Classifier System where each sheepdog would have a different role, described by a different set of rules.

An example of behaviours is given in figure 2.3. The condition part of the classifier is a set of bits which represent:

- some tests describing the current situation of the agent relatively to the position of the flock and the goal (*isAtGoal, isLeftToFlock, isBehindFlock, etc.* - cf

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

Classifiers for the **Pusher** Behaviour.

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

Classifiers for the **LeftGuide** Behaviour.

Figure 2.3: Classifiers for different roles for sheepdogs. Adapted from [Sigaud and Gérard, 2001a].

figure 2.2) ;

- some tests describing the situation of the other sheepdogs (*nobodyBehindFlock*, *nobodyPushing*, *etc.*) and the status of the flock (*isFlockFormed*).

Depending on these conditions, the sheepdogs have the choice between 16 different actions (*goToGoalCenter*, *doNothing*, *goToPushingPoint*, *etc.*) and the action part of the classifier is therefore one of these actions.

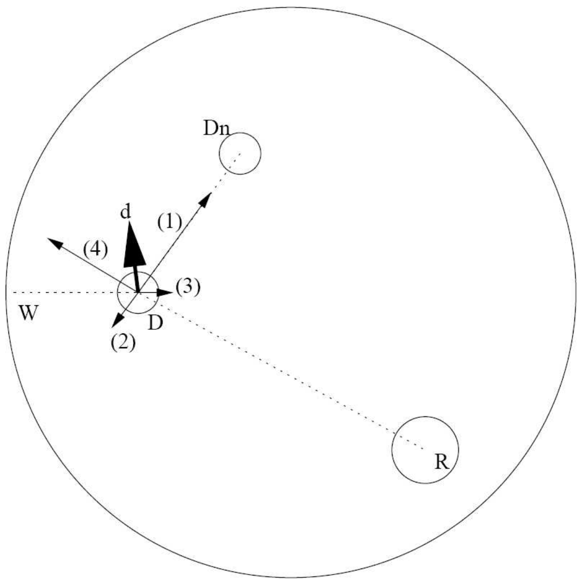
The learning was done using a special enhancement of the standard Classifier System, called the Anticipatory Classifier System.

2.2 Our problem

The previous review of relevant literature shows that studies including Learning within a Multi-Agent system have mostly attempted to find ways to combine basic behaviours in order to achieve a useful cooperation of several agents. Here we are interested in describing and learning the behaviours of *external* agents, i.e. agents whose "mind" and possibilities are unknown to our system. We describe in this section what is exactly our research problem and why we are interested in it.

2.2.1 Research Question

In the Robot Sheepdog Project, a model (Fig. 2.4) was used to describe the flock behaviour, and the system was built using this model in simulation. Thus, the system has been developed under the assumption that the model is a good representation of the duck behaviour.



$$\vec{d} = \sum_{n=1}^{N} \left[\underbrace{\left(\frac{K_1}{(|\overrightarrow{DD_n}| + L)^2} \right) \widehat{DD_n}}_{(1)} - \underbrace{\left(\frac{K_2}{|\overrightarrow{DD_n}|^2} \right) \widehat{DD_n}}_{(2)} \right] - \underbrace{\left(\frac{K_3}{|\overrightarrow{DW}|^2} \right) \widehat{DW}}_{(3)} - \underbrace{\left(\frac{K_4}{|\overrightarrow{DR}|^2} \right) \widehat{DR}}_{(4)}$$

Figure 2.4: Flock model. Key: gain parameters $K_{1\to 4}$; repulsion bias parameter L (ensures repulsion $>$ attraction at small distances, preventing collisions); duck position D, other duck D_n; Robot position R; Nearest point on wall W; algorithm terms $(1 \to 4)$ and resultant $\vec{d}$ (where $\widehat{a}$ is the unit vector of $\vec{a}$). Adapted from [Vaughan, 1999].

This dissertation aims at finding ways to develop a system that would **not** know in advance this model. The system we want to build would have some clues about the generic behaviour of the agents, but no particular model in mind. Thus, only observation of the reactions of the external agents could allow the system to *learn* their internal model.

As an example, consider the model in Fig. 2.4 which describes the duck behaviour used in the RSP project. According to this model which gives the movement vector $\vec{d}$ of each duck, ducks react to the environment with respect (roughly) to these "rules":

(1) Ducks are attracted to each other, aggregating the flock;

(2) Ducks are repelled from each other, preventing collisions and maintaining inter-ducks spacing;

(3) Ducks are repelled from obstacles (walls), preventing collisions;

(4) Ducks are repelled from the robot, modelling the aversive response of the ducks to the robot.

We can see here four different rules, and some parameters K_1, K_2, K_3 and K_4 describe to what extent each rule affect the overall behaviour of the duck. One can easily imagine other rules: attraction to food, attraction to the leader of the group (if there is one), etc.

The question we want to focus on is the following : **could a system, through experiments, *build* a model of the behaviour of unknown agents, in order to interact successfully with them later?** Is it possible to build a model of the external agents' behaviour, using just observations (and prior basic knowledge about animal behaviour)?

Depending on what they know from the environment, and what they know about other agents (and possibly other agents' knowledge), the agents take decisions at any time, which impacts on their environment. The question is therefore : are we able to find methods to infer (or *learn*) their "internal rules" from their actions ?

If a system is able to find these rules *during run-time*, it could improve its interaction with these external agents, having information about how they behave.

The RSP was focused on specific animals, ducks, but here we want to avoid building a system that would describe only the behaviour of ducks. Moreover, a (long-term !) goal would be to find a generic system that could be able to deal with any animal, or even better with any other agent (including machines).

Indeed, even if one might argue that no definition of what is inside the "mind" of the animals has been outlined yet, from a computer scientist point of view, we can still imagine that these animals are "unknown external agents" and see their internal state as a "black box". Since the internal state of an *artificial* agent could also be studied as if it was a "black box", we believe that there is no theoretical problem to try to describe the behaviour of any agent in a discrete way (with rules, algorithms and specifications).

This is the point of view shared by many researchers trying to develop *Animats*, which we have described previously.

2.2.2 Why the RSP problem ?

As we have seen, in a too complex situation like the *Robocup* simulation league, it seems that learning can only be applied to solve some particular subtasks, or to combine different primitive behaviours. Although studies did show some success training a neural network to learn how to play soccer by imitating a selection of hand-coded teams (see [McMillan, 2004]), most of the systems didn't manage to learn a team strategy from scratch, nor tried to adapt themselves *during run-time* to the opponent's strategy.

Therefore, most of the adaptive multi-agent simulations involve much simpler situations, namely the *Predator/Prey* environment. However, this environment might be seen as too simple, since there are only two kinds of agents, and one reacts only to the other.

As stated by Sigaud and Gérard in [Sigaud and Gérard, 2000],

"[the Robot Sheepdog environment] appears as a good compromise between the too complex Robocup problem and the oversimplified prey-predator problems."

Thus we chose to model this environment, but as we will see, since our task is to learn the behaviour of the *"external agents"* (we prefer this name rather than *"ducks"* because their behaviour might be modified, and might no longer look like the behaviour of ducks), we will only keep the basic features of this problem :

- A circular arena, thus simplifying the reaction of the agents to this entity (they react according to the nearest point on the wall, see Fig. 2.4 page 22).
- A set of agents evolving in that arena, reacting to each other (attracted, repelled).

2.2.3 Motivation

We believe that trying to understand how other agents behave is another interesting domain in multi-agent systems, that has not yet been studied deeply. So far most of the studies have concentrated on how pre-defined agents can cooperate; but a study of how unknown agents behave might be useful for several reasons.

On a practical point of view, learning processes are very important if we are to create more robust autonomous robots; they should be able to deal with new situations and new agents they encounter. Furthermore, the number of robots used in our daily life will probably increase over the next decades, and therefore it seems important that robots should be able to deal with other machines, especially if they are not able to communicate with each other (for example, an "advanced" robot dealing with basic limited vacuum cleaner robots in the same room).

On a more theoretical point a view, finding ways and algorithms to describe unknown agents' behaviour could be a useful step towards a better understanding of multi-agent systems. If we think of external agents as *artificial agents*, we *know* that their behaviour can be described by an algorithm (with probabilities involved if we are using randomness at some point), since they are machines. But if we see their internal state as a "black box", then from an external point of view, they might look like real animals ; therefore our model of these artificial agents might also be used to deal with animals. Thus such a learning system might then be useful in real situations like the Robot Sheepdog, where the Robot could self-optimize during run-time, as it was proposed by R. Vaughan.

We see several other applications for a system that can learn the behaviours of external agents, such as:

- improving video games (e.g. in a race game, the computer could learn which shortcuts you are taking, and use your strategy),
- learning the behaviour of guards patrolling, from a satellite view (what they are doing when something block their way, how often they meet, where).

Of course these are very long-term goals, and we shall give now the specific scope of this project, keeping this motivation in mind.

2.3 Scope of the project

2.3.1 Hypotheses

As we have seen for the model of the ducks in the RSP, a first approximation of the behaviour of external agents is to describe it as **a set of rules**. Here, we are not interested in the reasons why these rules exist in the external agent: for a robot, it could be the programmer's design choice, whereas for an animal, it could be an instinct[2]. But we decide for this project to assume that a set of rules is a good approximation to describe an agent behaviour.

Furthermore, we assume that the agent obeys to a set of rules *which does not change over time*. Of course, this assumption is quite strong, since it implies that the agent's "mind" does not *evolve* over time, nor learn any new rules.

But as we want to try to use techniques outlined in the courses *Intelligent Systems I & II*, especially **reinforcement learning** and **supervised learning**, we prefer to fix the set of rules and create a learning system whose aim is to find this set of rules.

Therefore we will describe our external agents' behaviour as a set of rules, each of them explaining part of the movement vector of the agent for a given time step t. The overall behaviour will be described by the sum of the vectors "activated" at each time step.

In that respect, we can describe the multi-agent system we chose to study as set of **simple reflex agents** (cf. 2.5, with the terminology used in the *Intelligent Systems I* course and Russell and Norvig in their book ([Russell and Norvig, 1995]).

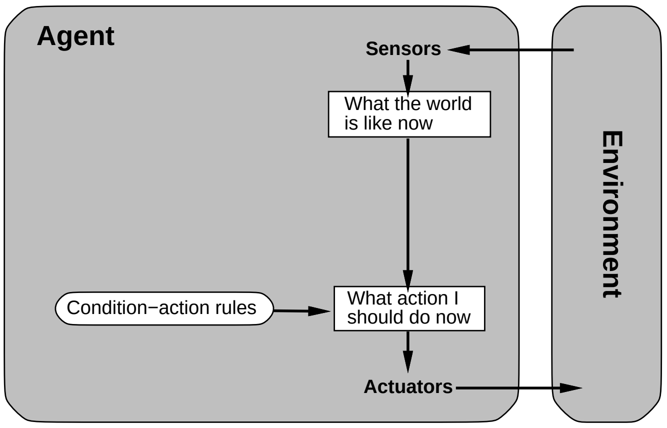


Figure 2.5: Simple Reflex Agent. Figure from [Russell and Norvig, 1995].

Indeed we assume that the behaviour of the agents depends only, at each time step, on the *current* environment, i.e. on the position of the other entities, but not on information from the situation in the previous time steps. This type of behaviour is called a *stimulus-response behaviour*, as opposed to a *dynamic behaviour*, which requires some kind of internal state (eg. past experience).

[2]However, this level of abstraction describing an animal's reactions only based on several instincts might appear quite insufficient for the reader interested in animal behaviour and philosophy of mind: we still don't know *why* these instincts exist !

Finally, even if the final aim would be to build autonomous agents able to deal with these external agents (like the Robot Sheepdog in the RSP), we will focus here only on **how to *learn* the behaviour of external agents**. We hope that once these behaviours have been described, they could be used later by an autonomous system that would interact with them, for example.

2.3.2 Objectives

Since we decided to use a set of rules as a description of an agent's behaviour, we have chosen the **Learning Classifier Systems** framework, as it is a rule-based system, for our Learning System. Although before starting this project we didn't know how well this system could perform in our context comparatively to other learning techniques, it was decided to focus only on this Machine Learning technique mostly because an extensive comparative study of several techniques would not have been possible due to time limitations.

Therefore the objectives of this project are :

1. **Building a system** that allows us to use a Learning Classifier System in a Multi-Agent Environment, where the aim is to learn which internal rules make the agents moving. This system should be *extendable*, so that different behaviours and different Classifier Systems can be used, and *interactive*, so that different tests can be made easily.
2. **Defining ways to describe the behaviours** of external agents in a form that can be applied to a Learning Classifier System. As a case study, we shall try to define the behaviours of the ducks in the Robot Sheepdog Project.
3. **Experimenting** the Learning Classifier System to try to learn the behaviours previously defined in **2** as if they were unknown to the system, just by *observing* the actions of these agents.

2.4 Summary

After a literature review, we have chosen to try the Learning Classifier System technique within a Multi-Agent problem for an interesting task: trying to discover which "rules" could describe the behaviour of external agents.

The project aims at developing a system where several behaviours can be tried and learned; this system should have the possibility to be extended easily in order to try different Learning Classifier Systems for different behaviours.

Chapter 3

Methodology

In this chapter we describe the requirements our system shall satisfy in order to meet our objectives, and we explain which methods are going to be used to fulfil these requirements.

3.1 Requirements

We have three important objectives: **(1)** building a system, **(2)** defining descriptions of the behaviours of the agent that could fit into a Learning Classifier System, **(3)** experimenting with our descriptions and testing our LCS. We give now more details about these objectives.

3.1.1 Overview

In order to meet the first objective, we need to develop a tool that allows us to simulate the behaviours of *several* agents interacting in an arena, and this tool should have the ability to learn the behaviour of each agent independently.

Thus we shall construct a program that can be both a *simulator* of behaviours, and a *learning system* at the same time, since our objective is to learn **during run-time**, that is to say, to learn the behaviours of the agents while they are moving inside the arena.

3.1.2 An extendable framework

From the second objective, we can infer that our system must be extendable, in (at least) two directions.

Changing the Behaviour of an Agent

Several behaviours need to be tested, and the simulator should show us how the movements of the agents are modified when we modify their behaviour. The system we are going to build should be able to deal with new behaviours as follows:

- since the behaviour of an agent is (by assumption) considered to be a set of rules, these rules should be easily mutable to see the impact on the reactions of the agent and the other agents *during the simulation*. As long as we are only changing rules that are based on the same definitions (ie., for a LCS rule, same number of bits and same meaning for each bit), this shall not require to start again the simulator. Therefore a Graphical User Interface (GUI) seems a good solution to allow these changes of rules.

- if we are to change the behaviour in a more complex way (eg. changing the meaning of the rules), this might require recompilation of the program; however, extending the program should be easy and quite straightforward.

For example, consider the Classifier System used by the Animat chasing a light described on page 16, it should be made possible to change the set of rules so that the Animat is avoiding the light (this would just require to change the [Action] part of the rules). This change would be made in order to see how the learning system is working for that kind of behaviour.

But if we want the Animat to react to another stimulus as well, eg. some food, then the *structure* of the rules are modified (the [Condition] part of the rules must be expanded): this might require to change the source code to implement this new "environment". However, the program should be written in such a way that adding a new environment is easy and does not affect all the source code. This requirement ensures that the framework can be used for several environments quite easily, and could be adapted for different experiments.

Changing the Learning Classifier System

It should also be possible to change as well the Learning Classifier System used, in order to try different versions or enhancements that have been proposed by the LCS community since the standard version proposed by Holland.

3.1.3 An interactive approach

Finally, for the third objective (experimentations), we need to build a system which is easy to use (with a useful graphical interface) and that provides some idea of the performance of the system for each test. A first obvious way to see how the test is going is to watch the simulation, and the learning system if possible. But some precise data should be recorded as well so that we can plot the results and evaluate the performance of the system.

3.2 Design Choices

From these requirements, several design choices were made, and we describe them below.

3.2.1 Object-Oriented Programming

It appeared quite soon that Object-Oriented Programming (described in the courses *Object-Oriented Programming I & II*) would be a relevant paradigm to use to build our system, since it brings very useful features: abstraction, modularity, and reusability.

First, thanks to **abstraction**, we can elaborate some generic rules about how to define an **Agent**, and by using abstract classes and inheritance we can design several agents that could fit into our system. This abstraction process could also be applied for the Learning System, allowing us to design a unified Classifier System with different implementations depending on the specific enhancements we want to use and test.

The **modularity** feature would be useful, for example, to separate the simulation and the display, which is one of the requirements we have.

The **reusability** feature is a requirement for our project, and we should ensure that we design our system in such a way that further tests and research can be made in different environments. Object-Oriented Programming usually makes it easier to have reusable programs.

Finally, the multi-agent problem is well suited for an Object-Oriented approach: every agent can be represented as an object, and several agents can be handled easily, by creating several instances of a super class Agent.

3.2.2 Graphical User Interface

Since most of this research is going to be done with a "trial and error" approach (cf section 3.4 below), a useful Graphical User Interface (GUI) should be provided, in order to modify parameters and do extensive tests. Indeed, the use of an heuristic (a Genetic Algorithm) implies that it's not possible to find a "solution" by calculation, and an empirical study (including tuning the parameters) has to be carried out.

Most of the recent Object-Oriented programming languages have extensive libraries to create a GUI, for example GTK for the language C++, Swing and Awt for Java, etc.

3.2.3 Use of Design Patterns

Within the Object-Oriented paradigm, there are several **Design Patterns** that already seem relevant for our system. Design Patterns are described as: *"reusable solutions to recurring problems that we encounter during software development"* [Grand, 1998]. Some of them have been described in the courses *Object-Oriented Programming I & II*, and we will use as well the two well-known books [Gamma et al., 1994; Grand, 1998] as references.

The first Design Pattern we can see is the **Template** pattern, sometimes called the **Template Method** ([Gamma et al., 1994]). This pattern allows the programmer to design only the skeleton of an algorithm in an abstract class, and let the subclasses implement some inner parts of the algorithm, by implementing abstract methods called in the main algorithm. We shall see at least two interesting applications of this pattern in our system:

- for the Learning Classifier System, where some parts of the algorithm (eg. how to do the crossover, the selection, etc.) could be implemented in different ways, but within the same main procedure.
- for the definition of the meaning of the rules. They all share some common features (they have an [Action] part, a [Condition] part, they are coded as bits of strings), but their meaning could be different depending on the behaviour we want to model.

Another useful pattern might be the **Observer** pattern, which defines the relation between the simulation (data model), and the GUI (observer) and ensures that data model and observer remain separated while still allowing interactions. However the original pattern allows several displays for a unique data model, whereas here we want to build only one window showing the simulation and receiving inputs from the user; therefore we might have to adapt the original pattern.

3.3 Further specifications

3.3.1 Overview of the system

It was decided to use a *state machine* for the simulator: at each time step, the simulator calculates the new position of each agent, depending on the positions of the other agents and the internal rules this agent has.

In order to have a more precise view of how the system is expected to work, we give in Procedure 1 an outline of the main procedure.

This procedure "contains" a lot of the requirements described previously, and we shall now explain this sketch of the main procedure.

lines 3,16: Our system should allow the user to choose whether or not a graphical view of the simulation is displayed. Thus the simulation must be able to run even if the graphical interface is not displayed. Furthermore, if simulation and graphics are separated, some set tests could be run more efficiently without the display.

line 6: Customization of the behaviours should be possible for each agent, if the GUI has been launched.

lines 9 to 21: The system should work as a state machine, updating the positions (l. 19) of each agent at each time step (once for each **repeat** loop).

line 12: If we want only to test the behaviours within the simulator, it should be possible to switch off the learning system. If the learning system is switched on, it works also as a state machine: at each time step, it tries to "guess" where the agent is going to be at the next time step, by using the set of rules it has

Procedure 1 Sketch of the system

```
 1: Initialize the Arena
 2: Create the first $m$ Agents with a *default* behaviour
 3: if isGUIused then
 4:    Start window
 5:    (Add more agents $a_{m+1}, a_{m+2}, ..., a_{m+n}$ to the arena)
 6:    Customize the real behaviours of each agent $a_i$
 7:    Paint the Arena and the Agents
 8: end if
 9: repeat
10:    for each agent $a_i$ do
11:       DEFINE NEW POSITION $\vec{P}_i^{t+1}$ FROM($\vec{P}_i^t$,other entities $e_{j \neq i}$,behaviour$_{real}$)
12:       if isLearningActive then
13:          GUESS NEW POSITION $\vec{P'}_i^{t+1}$ FROM($\vec{P}_i^t$,$e_{j \neq i}$,behaviour$_{expected}$)
14:          COMPARE $\vec{P}_i^{t+1}$ and $\vec{P'}_i^{t+1}$ $\Rightarrow$UPDATE behaviour$_{expected}$   {*see chapter 4*}
15:       end if
16:       if isGUIused then
17:          Repaint the agent
18:       end if
19:       $\vec{P}_i^t \leftarrow \vec{P}_i^{t+1}$
20:       Record some data in a log file {*to measure the performance of the system*}
21:    end for
22: until termination criteria are not met
```

constructed so far. After that, the "guess" is compared to the real movement of the agent, and the learning system should evolve to take into account the result of the comparison.

line 20: In order to measure the performance of a given LCS with some given parameters, the system should record some data useful to compare different sets of test.

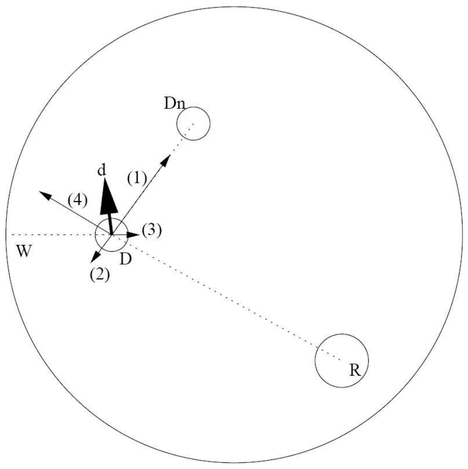
line 22: The termination criteria might be a given number of time steps, the user stopping the simulation, a condition telling that the system has become stable, etc.

This is only a broad view of the system, at this point we don't have any precise requirement on how the learning system should work (procedures GUESS NEW POSITION, COMPARE, UPDATE), nor how we measure the performance of the system. We will give more details about the Learning Classifier System in next chapter.

However, the method we are going to use for DEFINE NEW POSITION can be described already, since it was decided to re-use part of the work of Richard Vaughan.

3.3.2 Design of the simulation model

The model used by R. Vaughan (we reproduce the figure here for convenience) determines the positions of the agents by an adapted **potential field algorithm**.



$$\vec{d} = \sum_{n=1}^{N} \left[\underbrace{\left(\frac{K_1}{(|\overrightarrow{DD_n}| + L)^2}\right) \widehat{DD_n}}_{(1)} - \underbrace{\left(\frac{K_2}{|\overrightarrow{DD_n}|^2}\right) \widehat{DD_n}}_{(2)} \right] - \underbrace{\left(\frac{K_3}{|\overrightarrow{DW}|^2}\right) \widehat{DW}}_{(3)} - \underbrace{\left(\frac{K_4}{|\overrightarrow{DR}|^2}\right) \widehat{DR}}_{(4)}$$

Figure 3.1: Flock model. Key: gain parameters $K_{1\to 4}$; repulsion bias parameter L (ensures repulsion > attraction at small distances, preventing collisions); duck position D, other duck D_n; Robot position R; Nearest point on wall W; algorithm terms $(1 \to 4)$ and resultant $\vec{d}$ (where $\widehat{a}$ is the unit vector of $\vec{a}$). Adapted from [Vaughan, 1999].

This method uses an analogue of a charged particle in an electrical field. The motion of the particle is determined by its current position and the different forces acting on it. However, in the simplified model used by R. Vaughan, the assumption is that these forces are in fact directly modelling a movement vector. Thus we don't model the effect of the mass here, and we don't use forces to determine acceleration. We simplify by saying that every other entity $e_{j\neq i}$ (including walls) around the Agent a_i can affect its movement, and this is represented by a vector $\vec{v}_{e_j \Rightarrow a_i}$ which determines part of the movement vector $\vec{M}_i^t$ of the Agent calculated at a given time t.

$$\vec{M}_i^t \overset{def}{=} \left[\sum_{j\neq i} \vec{v}_{e_j \Rightarrow a_i}(e_j, behaviour_i, \vec{P}_i^t)\right] + 0.1 \times \vec{M}_i^{t-1} \tag{3.1}$$

$$\vec{P}_i^{t+1} \overset{def}{=} \vec{P}_i^t + \vec{M}_i^t \tag{3.2}$$

In our model (equations 3.1 and 3.2), this vector $\vec{v}_{e_j \Rightarrow a_i}$ depends on the behaviour

$behaviour_i$ of the Agent a_i, which encodes the reaction of the Agent a_i to this entity e_j, ie.:

- whether the Agent is attracted or repelled by the other entity (this corresponds to the sign of the term in Vaughan's model, multiplied by the unit vector between the Agent position and the position of the other entity; see fig. 3.1);
- how strong is the response of the Agent to this entity (this corresponds to the constants K_1, K_2, K_3, K_4 in Vaughan's model);
- etc.

With these informations (contained in the behaviour of the Agent, in a form that still remains to be specified), the vector $\vec{v}_{e_j \Rightarrow a_i}$ can be calculated, scaling it according to the inverse square of the distance between the Agent and the other entity (previous models reviewed by Vaughan et al. have showed that a real animal would indeed have such a nonlinear response).

Finally, in order to obtain the movement vector $\vec{M}_i^t$ of the Agent, we add all the vectors representing all the influences of the other entities (some might not affect its movement, this depends on its behaviour), and add to this sum 10 % of the previous movement vector $\vec{M}_i^{t-1}$ (in order to roughly model the *inertia* of the Agent). This gives us the movement vector, that we apply to the current position $\vec{P}_i^t$ in order to know the new position $\vec{P}_i^{t+1}$ at the next time step $t+1$ (equation 3.2).

3.4 Method

3.4.1 Overview

We have chosen to use Evolutionary Computation, and more specifically Learning Classifier Systems, to learn the behaviours of the external agents. Although some have managed to put mathematical foundations to explain the success of Genetic Algorithms (work by [Holland, 1975] and [Goldberg, 1989]), Evolutionary Computation is still a field where most of the studies are **empirical**.

Indeed, it is necessary in Evolutionary Computation to spend a lot of time **tuning** the different parameters, and making some little changes to the algorithm so that it performs better.

Therefore we want to emphasise that our study should use a "trial and error" approach, that can be outlined as follows:

1. Find generic rules to put into the "brain" of the agents in order to describe their behaviour.
2. Once the meaning of the rules is fixed, describe the agents' behaviours as a set of such rules.
3. Launch the system, expecting it to **learn** this set of rules as if they were unknown, *only* by observing their movements.

4. If the results are bad (learning doesn't succeed), try to simplify the set of rules, or the meaning of the rules (eventually change the features of the Learning System) and start again at **(1)**.

5. If the results seem good, increase the complexity of the agents' behaviour and start again at **(3)**.

It seems necessary to have an idea of how to **measure** the performance of the system, in order to be able to say "the results are good/bad".

3.4.2 Performance

In this project, the assumption is that the system **doesn't know** what is inside the "mind" of the agents, and tries to guess their movements from past experience, by inferring from their movements a set of rules that could describe their behaviour.

A first problem appears here: since the final aim is to use our system with *unknown* agents, how are we going to assess that the system has "correctly guessed" their internal rules ?

Of course, in this project we are using a simulation, and we *choose* what is inside the "minds" of the simulated agents, so an obvious way to measure the performance would be to compare the **chosen** set of rules with the **expected** one (ie. the one that the system inferred from its observations). But this is not an appealing solution, since it involves *knowing something we assume unknown.* However, this solution might be useful for the tests, in order to have some idea of the performance of the system.

Another solution was then designed, which doesn't imply breaking our initial assumption. The idea is to use the only thing we can observe, **the actual movement of the agent**, to assess the performance of the system. At each time step, the system makes a *guess* about the next position of the agent, and this guess is immediately confirmed or denied by the actual movement of the agent. We can then compare the **actual** movement vector and the **expected** movement vector, and calculate for example the distance between the actual position and the expected position. If we add these distances over all the time steps of the experiment, we obtain an "objective" measure for the performance: the closer to the real positions the guesses are, the better the system performs.

3.5 Summary

Several general requirements for the system have been outlined. Some of them have led to design choices and further specifications. In particular, the need for abstraction and reusability, and the multi-agent context, make the Object-Oriented paradigm a good choice for building our system.

A sketch of the system has been given, and we have explained our approach to the problem and how we are going to assess our results. We need to describe now in further details the actual Learning Classifier System we have used.

Chapter 4

The Learning Classifier System

In this chapter we detail the Learning Classifier System framework, and explain the specific features of our LCS, which is an adaptation of Wilson's Zeroth Level Classifier [Wilson, 1994].

4.1 Overview of the standard LCS

A Learning Classifier System can be described as:

> "a Machine Learning System that learns syntactically simple string rules (if-then rules) to guide its performance in an arbitrary environment." [Goldberg, 1989]

We have explained roughly what the purpose of an LCS is in Chapter 1 (page 15), and we need now to detail the mechanisms.

A LCS usually consists of three main components (cf Figure 4.1):

1. A RULE AND MESSAGE SYSTEM
2. An APPORTIONMENT OF CREDIT SYSTEM
3. A RULE DISCOVERY SYSTEM

The RULE AND MESSAGE SYSTEM is a *production system* where the rules are generally of the form [**Condition**] $\rightarrow$ [**Action**], which means the action may be taken (the rule is activated) when the condition is satisfied.

The LCS has a pool of rules (generated randomly at the beginning), and a **strength** is associated with each rule, representing their ability to get good rewards from the environment.

At each time step, some information comes to the detectors from the environment, and goes to the message list. Depending on the messages in the message list, some rules are activated (if one message in the system satisfies their [Condition] part). A selection among the activated rules is made, and the selected rules deliver their message (their [Action] part) back to the message list. In the standard form of the LCS, the message list can therefore contain information from the detectors *mixed*

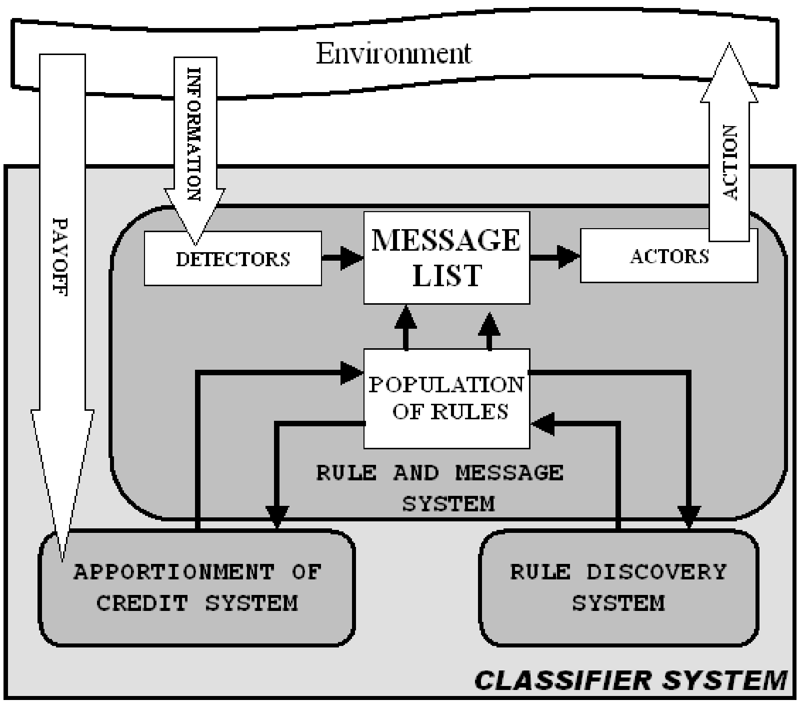


Figure 4.1: The Architecture of a standard Learning Classifier System. Adapted from [Geyer-Schulz, 1995].

with messages from the rules. This allows the system to create "chains of rules": a rule sends a message to the message list, activating another rule on the next time step, which in turn sends a message to the message list, and so on.

At the end of the selection step, if some messages are meaningful for the effectors, an action is carried out in the environment. In a **reinforcement learning** problem, this action might not always get a payoff (reward) from the environment; but when the action is judged to have a positive impact on the environment, a reward is given to the LCS. Thanks to this reward, the LCS will update the strength of the rules, using its APPORTIONMENT OF CREDIT SYSTEM . But since the action chosen by the LCS might be the result of the sequential activation of *several* rules (as explained above), it needs to give a reward to all the rules which belonged to the "chain" which lead to the useful action. This can be done using several techniques, the original one being Holland's *bucket brigade* (see [Goldberg, 1989] for further details).

Finally the third component, the RULE DISCOVERY SYSTEM ensures that new rules are created, in order to explore the space of the possible different rules. Most of the time, the rule discovery system is a Genetic Algorithm, which combines the "best" rules (with the highest strengths) and modifies them in order to obtain new offsprings, with the hope that recombination of useful rules creates rules which are even more useful (this is the basic principle of natural evolution and selection).

Thus, we obtain a system able to **explore** the search space (thanks to the Genetic Algorithm which creates new rules) and **exploit** the possible solutions (by testing the classifiers and updating their strengths).

A first simplification of this system is to remove the message list: rules give messages directly to the effectors. This is the simplification chosen by Wilson for his Zeroth-Level Classifier System (ZCS, [Wilson, 1994]) that we describe now.

4.2 The Zeroth-Level Classifier System

In order to make the explanations more understandable, we consider the following example: a robot trying to reproduce the behaviour of a dog. The robot is moving in its environment and can encounter four different entities: a toy (big and red), an apple (small and red), a cat (big and grey) and a mouse (small and grey).

In our example, the robot has:

- two different **sensors**, which give information about the *color* (we suppose here that its vision system is bad, and can only detect whether something is red or not) and the *size* (small or big) of the entity;
- a number of **effectors**, which allow four different actions: *don't touch*, *destroy* (or *kill* for a living entity), *play* and *eat* (might be in fact *collect* for a robot!).

The aim of the robot is to learn how to behave like a dog, ie. to correctly react to the four different entities in the following way: *don't touch the cat*, *kill the mouse*, *play with the toy* and *eat the apple*.

Now, in order to use a LCS, we define a *Template* for the rules, which describes the meaning of the bits in the [Condition] and in the [Action] part. For the Condition part, we use two bits: the first one giving information from the first sensor (ie. 1 when it's red, 0 otherwise), and the second one giving information from the second sensor (ie. 1 when it's big, 0 when it's small). For the Action part, since there are four actions possible, we also use two bits: 00 for *don't touch* , 01 for *destroy*, 10 for *play* and 11 for *eat*.

Therefore the behaviour the robot should learn can be fully described by the four rules given in table 4.1.

	Big?	Red?				
Cat	1	0	→	0	0	Don't touch!
Mouse	0	0	→	0	1	Destroy (kill)
Toy	1	1	→	1	0	Play
Apple	0	1	→	1	1	Eat
	CONDITION			ACTION		

Table 4.1: Behaviour of a simulated Dog.

The process used by the ZCS to learn these rules can be described by the Figure 4.2.

In the ZCS, the Rule and Message system is reduced to the population of rules inside the system, together with a selection process.

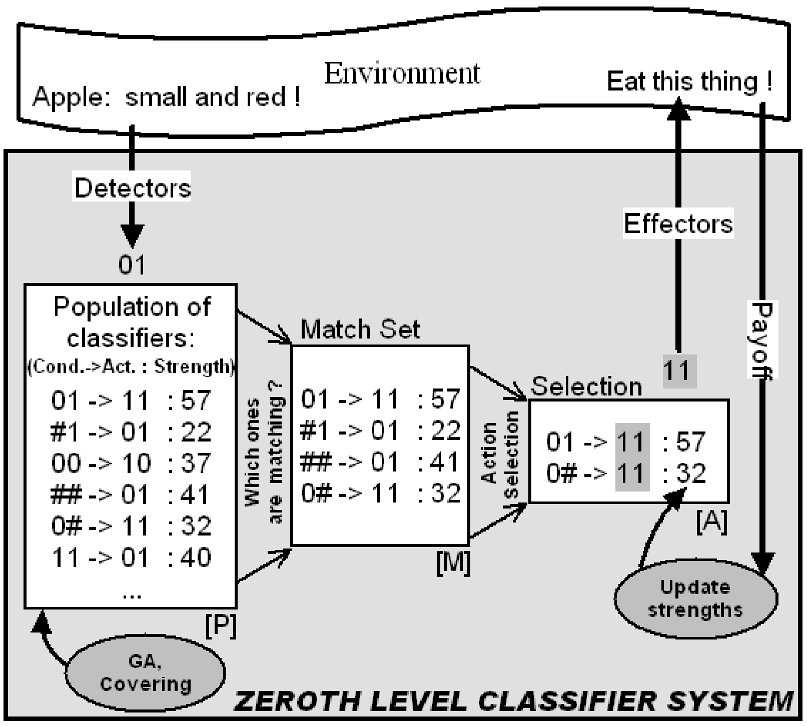


Figure 4.2: A Zeroth-Level Classifier System. Adapted from [Wilson, 1994].

Suppose our robot meets an apple on its way. The detectors send the information **01** (small and red).

From this information, the selection process will first create the **Match Set** $[M]$ from the population $[P]$. The Match Set is the set of classifiers which have a [Condition] part satisfied by the information given by the detectors. For the example of the apple, all the classifiers having **01** as a condition will be selected, but also the more generic classifiers containing a "don't care" (#) symbol. For example, a classifier whose [Condition] is **#1** will also be selected, since it means *"a red object (no matter how big)"*.

From this Match Set, a particular action a (in the figure, **11**) is selected with probability equal to the sum of the strengths of the classifiers in $[M]$ which advocate that action, divided by the sum of the strengths of the classifiers in [M]. Thus a "good" action (associated with rules having a high strength) will be more likely to be selected. With this action, an **Action Set** $[A]$ is created, by choosing all the classifiers in $[M]$ which advocated a.

Finally, this action is sent to the effectors, and the corresponding action is carried out in the environment.

Then, if a reward is given by the environment, the REINFORCEMENT SYSTEM updates the strengths of the rules by distributing the reward among the classifiers in $[A]_{-1}$ (ie. the action set of the previous time step) in order to obtain an *implicit bucket brigade*. We don't detail this process since we are not going to use it (see **Specific features** below for the process we used).

Every n time steps (n is to be chosen), the Genetic Algorithm (GA) is used as a RULE DISCOVERY SYSTEM. Two rules are randomly chosen among $[A]$, and a **crossover** might be applied on their [Condition] part to obtain two new offsprings (see fig. 4.3); then a **mutation** might change some of the bits of the offsprings (see fig. 4.4).

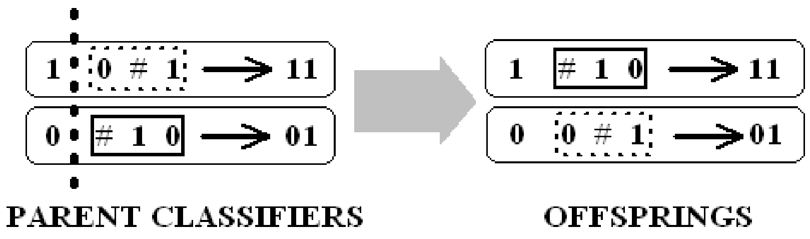


Figure 4.3: Crossover. Notice that the crossover is applied only to the [Condition] part.

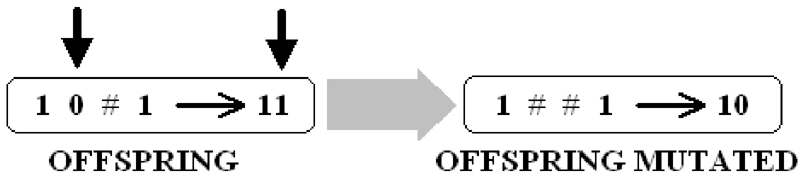


Figure 4.4: Mutation. Notice that actions can change and that ”#” can appear.

Then the two offsprings are inserted in the population, and in order to stay below the maximum size of the population, some classifiers might be deleted (the weakest ones having more chances to be deleted).

The last feature of the ZCS to explain is *covering*. When the Match Set $[M]$ is created, if there are not enough different actions in $[M]$ (”enough” is to be chosen), then the system generates new classifiers for the Match Set (ie. whose [Condition] is satisfied by the information from the detectors) with a random action among those not already in the Match Set.

The idea is that over the time, the classifiers leading to the maximum payoff from the environment will become stronger, and the system will select them more often, thus trying to maximize the reward it can receive.

In our example, if the action chosen for the apple is ”eat !”, a reward will be given to the robot, and it will increase the strengths of the correct rules, thus ”recording” in its population of rules this important instinct for a dog: whenever you see an apple, eat it!

It is interesting to notice that Classifiers Systems can also contains generic rules which can be activated for a new situation. For example, a rule (#**0** $\rightarrow$ **00**) which says ”*don't touch what is not red*” might be a good way to get payoff (although not the maximum payoff) when the robot encounters a cat or a mouse and doesn't have yet a specific rule to deal with a situation **10** (cat) or **00** (mouse).

4.3 An adaptation of the ZCS

For our problem, we need to adapt the ZCS because of two major reasons:

1. Our problem is multi-agent. Thus at each time step, an agent's detectors should be activated for each other entity around (in our dog example, the robot does not react to several stimuli coming at the same time). Indeed, for the duck model in the RSP, the duck reacts to all the entities around *in parallel* (its movement is determined by the position of *all* the other entities).

2. We do not need a bucket brigade, nor an implicit bucket brigade, because we do not use **reinforcement learning**. Indeed, at each time step we are able to calculate the payoff to give to the system (this is **supervised learning)**, since we can always compare the expected position of the agent with the real position of the agent and give a payoff proportional to the distance between the two positions.

Therefore we have designed new algorithms to address these problems, they are explained below.

4.3.1 Main Algorithm

We follow for our description the conventions from [Butz and Wilson, 2001], who have given a very clear description of the XCS, another classifier system developed by Wilson. In particular:

- functions in SMALL CAPITALS are procedures which are described later (or in Appendix A).
- we named the population set $[P]$, the match set $[M]$, the action set $[A]$.
- a classifier cl has a [Condition] part $cl.C$, an [Action] part $cl.A$ and a strength $cl.s$.

The following constants should be determined before launching the system:

- `GAperiodicity` (was n page 38): the periodicity of the use of the Genetic Algorithm. Typical values are for example *every 50 time steps*,*every 200 time steps*, etc.
- `sizeActionSet`: the number of classifiers to activate in order to calculate the response of the agent for *one* entity.
- `maxError`: the maximum error between the expected position and the actual position. If the error is greater than `maxError`, then no reward is given (see section 4.3.2).
- β, γ: constants which are used in the Apportionment of Credit System (see section 4.3.2).

- `minNbActionsForMatchSet`: miminum number of different actions to have in a Match Set; below this number, covering occurs.

- different probabilities (probability for crossover, mutation, for the apparition of the `#` symbol in a classifier created by covering, etc.).

We give in procedure 2 the main steps of our ZCS. It details the outline of the system given in procedure 1 page 31.

Procedure 2 Main Procedure of our ZCS

```
1:  timeSteps ← 0
2:  repeat
3:     timeSteps ← timeSteps +1
4:     for each agent $a_i$ do
5:        DEFINE NEW POSITION $\vec{P}_i^{t+1}$ FROM($\vec{P}_i^t$, $a_{j\neq i}$, behaviour$_{real}$)
6:        if isLearningActive then
7:           $[A]$ ← GUESS NEW POSITION $\vec{P'}_i^t$ FROM($\vec{P}_i^t$, $a_{j\neq i}$, behaviour$_{expected}$)
8:           $r$ ← COMPARE $\vec{P}_i^{t+1}$ and $\vec{P'}_i^{t+1}$                {reward}
9:           UPDATE $[A]$ CONSIDERING THE REWARD $r$
10:          if (timeSteps % GAperiodicity) $= 0$ then
11:             RUN GENETIC ALGORITHM on $[A]$
12:          end if
13:       end if
14:       $\vec{P}_i^t$ ← $\vec{P}_i^{t+1}$
15:    end for
16: until termination criteria are not met
```

Although we have put a "**for**" loop on line 4, it comes only from procedure 1, and is just here to explain that each agent's behaviour is learned by a separate ZCS.

One could imagine our system as a little device attached to a duck, for example: at each time step, it knows the exact coordinates of the duck (eg. thanks to a little GPS device), and tries to guess where the duck is going to be in the next time step, without knowing the motivations of the duck. The system detects the entities around the duck (wall, sheepdog, other ducks, etc.) and makes its guess according to the pool of rules it "thinks" might describe the motivations of the duck. Then it compares its guess to the next position of the duck and updates its rules.

Keeping this analogy, in our system we provide a little "device" for *each* agent. Although they don't run in parallel[1], they are completely separated.

Therefore in the following we consider only how *one* ZCS works, and in the main system several ZCS will work in parallel, as sketched by the **for** loop.

[1]This would involve dealing with complicated concurrency issues such as: "how and when should we get the *current* position of the other entities, since they are updated continuously?" and was decided to be beyond the scope of this project.

4.3.2 Specific features

Generating the "Action Set"

The procedure GUESS NEW POSITION is the crucial part of our multi-agent system: it generates the expected movement vector of the agent, by adding all the movement vectors generated by the [Action] parts of the selected classifiers. It is important to notice that an Agent is influenced by several entities around him, therefore several classifiers (which might have different [Action] parts) are selected and added to the "*Action Set*". Thus the name "Action Set" doesn't really correspond to the reality of our system (since it is *not* a set of classifiers having the same [Action] part) but we decided to keep the name for convenience.

Procedure 3 GUESS NEW POSITION $\vec{P'}^{t}_{i}$ FROM($\vec{P}^t_i$, $e_{j\neq i}$, behaviour$_{expected}$ $[P]$) **returns** $[A]$

```
 1: $[A] \leftarrow [\,]$                                          {empty action set}
 2: $\vec{V} \leftarrow \vec{0}$                                  {will receive $\sum_{j\neq i} \vec{v}_{e_j \Rightarrow a_i}$}
 3: for each entity $e_j$ around agent $a_i$ do
 4:     $d_j \leftarrow$ GET DEFINITION OF $e_j$ FROM TemplateBehaviour
 5:     $[M] \leftarrow$ GENERATE MATCH SET OUT OF $[P]$ SATISFYING $d_j$
 6:     $[A_i] \leftarrow$ GENERATE ACTION SET FROM $[M]$
 7:     $\vec{v}_{e_j \Rightarrow a_i} \leftarrow$ GET VECTOR FROM $[A_i]$ USING TemplateBehaviour
 8:     $\vec{V} \leftarrow \vec{V} + \vec{v}_{e_j \Rightarrow a_i}$
 9:     $[A] \leftarrow [A] \cup [A_i]$
10: end for
11: $\vec{M}^t_i \leftarrow 0.1 \times \vec{M}^{t-1}_i + \vec{V}$
12: $\vec{P'}^{t+1}_{i} \leftarrow \vec{P}^t_i + \vec{M}^t_i$
13: return $[A]$
```

This algorithm is to be linked to the equation 3.1 page 32 which defines the movement vector of the Agent. Here, the *TemplateBehaviour* should be understood in the same way as the *Template* for our dog example: it defines the meaning of the bits for the [Condition] and [Action] part. However, as we explained previously, the [Action] part is not a physical action but a vector (the meaning of each bit is defined in the *TemplateBehaviour*, therefore the procedure GET VECTOR FROM $[A_i]$ depends on *TemplateBehaviour* and will be explained later). The sum of all the vectors will determine the expected movement vector for the Agent.

Since we have already explained how we generate the Match Set (page 38), we don't put its algorithmic definition here (see Appendix A for further details).

It remains to explain how we decided to generate *each* Action Set for each entity around the Agent. This is done by selecting `sizeActionSet` classifiers using a **Roulette Wheel** (RW) selection with slots of the roulette sized according to the strength of the classifiers. The idea is to select the "best" classifiers, while still giving a chance to the weaker ones, in order to keep diversity in the population (ie. to test different solutions). The Roulette Wheel selects a classifier randomly, but the probability of a classifier to be selected is proportional to its strength.

The full algorithm is given in Appendix A.

Apportionment of credit system

Here we explain how we update the strengths of the rules. The procedure COMPARE just returns a reward proportional to the distance (called `error`) between the *expected* position and the *actual* position. Since the reward must be positive, we give a reward proportional to (`maxError` − `error`) when the error is not too big, 0 otherwise (no reward). We took the value 10 for `maxError` and a factor of 200, so the maximum reward the LCS can get at a given time step is $(10 - 0) \times 200 = 2000$.

Procedure 4 COMPARE $\vec{P}_i^{t+1}$ and $\vec{P'}_i^{t+1}$ **returns** r

```
1: error ← distance between $\vec{P}_i^{t+1}$ and $\vec{P'}_i^{t+1}$
2: if error < maxError then
3:     return (maxError − error) ×200      {to have a stronger impact even with small errors}
4: else
5:     return 0                             {no reward, the error is too big}
6: end if
```

Therefore the rules receive reward only when the position they generated (through their vectors) is close to the actual position of the agent. When the error is too big, no reward is given, but then the strengths of the activated rules wouldn't change. We would rather prefer that the rules are actually *punished*. So we decrease their strengths with the following method adapted from the standard ZCS when the environment is single-step (ie. when the payoff is known at each time step).

Each activated rule gives a percentage β of its strength to a common bucket[2] and receives back only its share of the reward plus its share of a *fraction* γ of the bucket, with $\gamma < 1$. Therefore if the rules receive no reward (error too big), they will receive back only part of their contribution to the bucket, and their strength will decrease.

The full algorithm is given in Appendix A.

4.3.3 Genetic Algorithm

The Genetic Algorithm used is classic.

We don't specify here how to do a crossover or a mutation; for further details see figures 4.3 and 4.4, or refer to [Goldberg, 1989].

Once crossover and mutation have been applied, the two offsprings are inserted in the population to be tested on the next time steps.

[2]Although we use the name "bucket" here, the method we describe is very different from Holland's *bucket brigade*, fully described in [Goldberg, 1989].

Procedure 5 RUN GENETIC ALGORITHM on $[A]$

```
1: $cl_1, cl_2 \leftarrow$ SELECT CLASSIFIERS USING RW FROM $[A]$   {see procedure 10 in Appendix A}
2: $cl'_1 \leftarrow$ COPY $cl_1$
3: $cl'_2 \leftarrow$ COPY $cl_2$
4: $cl'_1.s \leftarrow cl_1.s/2$
5: $cl'_2.s \leftarrow cl_2.s/2$
6: $cl'_1, cl'_2 \leftarrow$ APPLY ONE POINT CROSSOVER TO $cl'_1$ AND $cl'_2$
7: APPLY MUTATION TO $cl'_1$
8: APPLY MUTATION TO $cl'_2$
9: ADD CLASSIFIER $cl'_1$ WHILE MAINTAINING SIZE IN $[P]$   {see Appendix A}
10: ADD CLASSIFIER $cl'_2$ WHILE MAINTAINING SIZE IN $[P]$
```

4.4 Summary

The standard Learning Classifier System has been explained. We have introduced what we call a *Template*, which defines the meaning of the bits in the Classifiers. On the other hand, a *Behaviour* is an arbitrary set of rules whose meaning is given by the *Template*. A small example of a possible *Template* and a possible *Behaviour* has been produced, to give an idea of how we are going to describe the behaviour of the ducks in the RSP model (our second objective).

Finally, we have detailed our Learning Classifier System, which is an adaptation of Wilson's ZCS for a single-step environment because we know the reward we can give to the system at each time step. We also designed our ZCS so that it can work in a multi-agent environment, with several rules activated to produce one single movement (which is the combination of all the responses to the other entities around the Agent).

Chapter 5

Implementation

In this chapter we describe the main features of our implementation. The program can be tested by following the instructions given in Appendix B.

5.1 Use of Java and Eclipse

The choice of **Java** for the Object-Oriented programming language was dicted by pragmatic reasons: at the beginning of the project it was decided to try to re-use an open source Java environment called **TeamBots** for the simulator, as the purpose of the project was more focused on the Learning System. TeamBots[1] was developed by researchers at the Carnegie Mellon University, especially Tucker Balch, for his dissertation about multi-robot problems (cf [Balch, 1998] for an account of the work). However, after some experiments with this platform, it appeared that the program was too big to be adapted and modified for our project, and that we should start again "from scratch". But it was decided to continue to use Java for convenience. Java also provides two main features which we considered: (1) it compiles to bytecode and can therefore be executed on several platforms easily; (2) it is fully integrated within the **Eclipse** IDE (Integrated Development Environment). Eclipse has been used for this project and has proved to be very time-saving, mainly because of:

- its ability to compile the Java code *on the fly* (called *incremental compilation*), showing the errors as annotations in the margin of the source code;
- its refactoring tools (safe rename);
- its code completion tool which gives directly the legal completions of methods, names, etc.

Finally, Eclipse allows third-party plug-ins, and we have used Omondo's *Eclipse UML* to generate the UML diagrams that we present in this chapter and in Appendix C.

[1] `http://www.teambots.org`

5.2 Overview of the system

The main organization of the system can be described by the diagram in Figure 5.1.

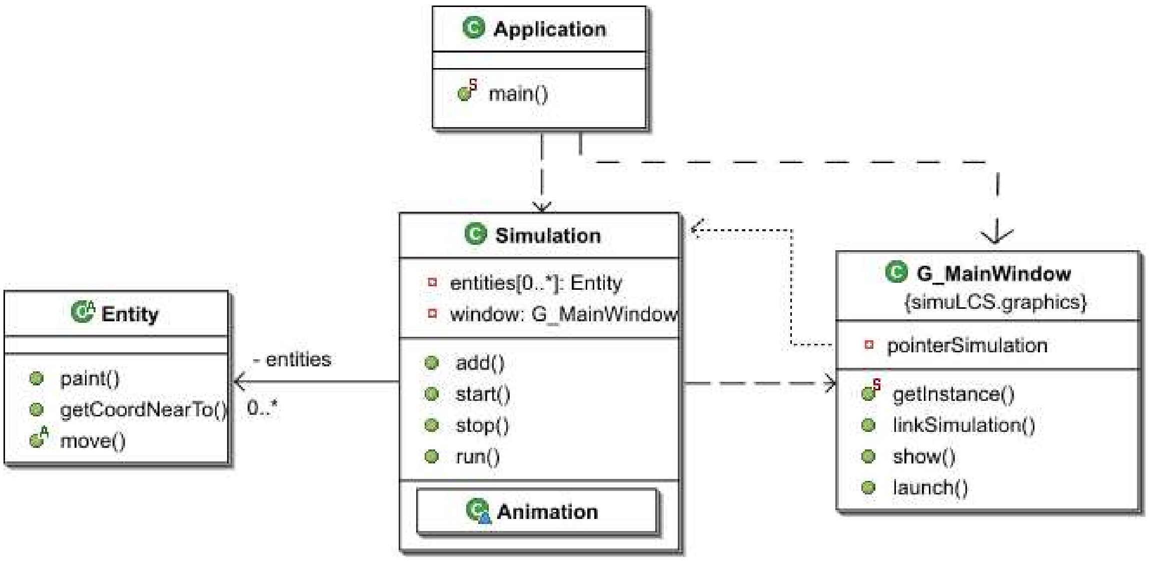


Figure 5.1: The main components of the system.

The system works as follows:

- the Application class (containing the main method) creates a Simulation;
- if the GUI is used, the Application launches the display as well (provided by G_MainWindow) and links the display and the simulation to each other, so that they can interact;
- the arena and a set of other entities (class Entity) are created and handled by the Simulation;
- the Simulation, through the method start(), launches a new Thread (with the inner class Animation) which will run the main procedure already described in chapter 3 (page 31);
- if the GUI is used, the user can modify the settings of the simulation;
- the simulation can then start by a call to the method run() (if there is a GUI, the system waits for the user to click on the button "Start").

In order to differentiate the classes used for the graphical display, we have prefixed their name by "G_" and created two different packages: a package for the simulator and the learning system (simuLCS), and a sub-package (simuLCS.graphics) for the components of the GUI.

5.3 Simulator and Learning System

A quite detailed overview of the class diagram is given in Figure 5.2 page 48. This diagram summarizes the main features of the most important classes in our system (GUI classes are not included). We explain briefly this diagram below, and the following sections explain in more details each important set of classes. We provide for each of them their main responsibilities, and a more precise UML diagram can be found in Appendix C.

The class Application creates a Simulation. The Simulation contains a set of objects from the class Entity and makes them moving with a thread Animation. An AgentClassifier is a special kind of Entity having a *behaviour* described as a ClassifierSet, ie. a set of instances of the class Classifier. These classifiers are rules whose meaning is described by a subclass of the abstract class Template. If the Simulation contains an AgentClassifierLearning, then the system tries to learn its behaviour by using the Learning Classifier System previously described, and implemented in the classes ZClassifierSet and ZClassifier.

5.3.1 Simulation

The Simulation class is the main class of the system. It is responsible for keeping a record of all the entities (arena, agents, etc) which interact in the simulation, and managing the simulation itself, through a Thread Animation.

The use of a thread is quite usual for this kind of program, where the simulation requires a lot of calculations: if only one thread was used, the simulation could remain busy calculating the positions of the agents and the inputs from the user (through the GUI) would be most of the time ignored by the program. Launching another thread allows the program to switch very often between the thread handling the GUI and the thread calculating the positions, thus being able to react to user inputs.

The following important methods belong to the Simulation class:

- add(Entity) which add an entity to the simulation. This method can be called by clicking on a button of the GUI or at the beginning of the program, by the application itself.
- start(), which starts the simulation.
- stop(), which pauses the simulation (it can be resumed by calling start() again).
- finish(), which ends the simulation, closes the log files and allows the user to start another simulation (using reset()).
- step(), which runs the simulation *only for one time step*. This is very useful for debugging: if we want a lot of informations to be sent to the console (to understand what the program is doing), we cannot have these informations dispatched to the console at *each* time step, because this would cause the program to be very slow, since it needs to output a lot of data to the console

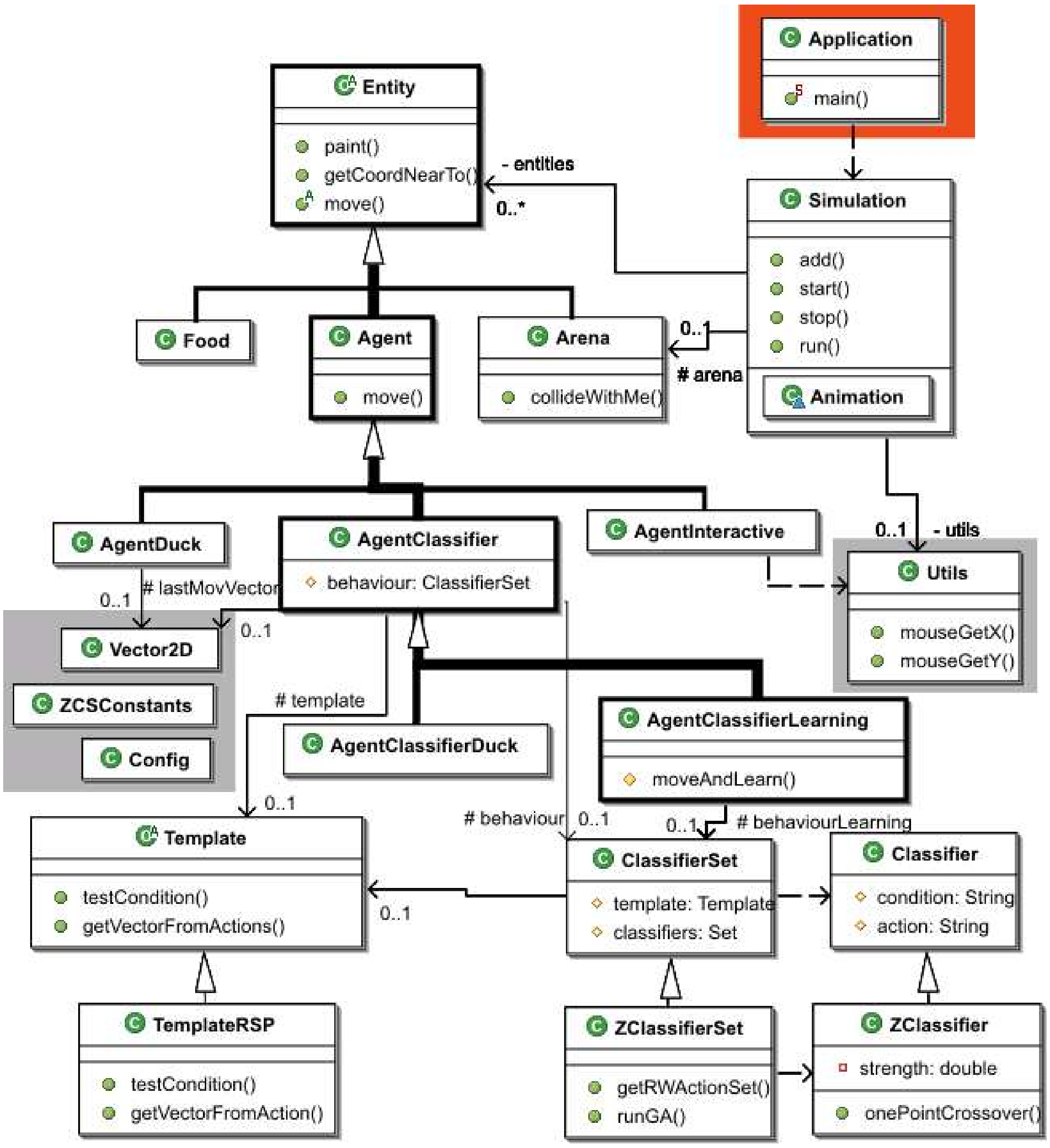


Figure 5.2: Class diagram overview. The GUI classes are not shown. Classes with a grey background are utility classes. Inheritance is shown with an arrow subclass ▷ superclass.

continuously. Thus the possibility to execute only one step was designed, with the ability to switch the program from a "low information mode" to a "verbose mode" to see the details of the computation.

Animation

The Animation is responsible for executing the main procedure described page 31. It implements the interface Runnable so that it can be used for creating a Thread. At each time step, it calls the function move for each entity belonging to the simulation, and repaint the entity if the GUI has been launched.

A summary of the Simulation class (UML diagram) is given in Appendix C.

5.3.2 Entity

Entity is an abstract class that provides a unified definition of all the things that could interact with an Agent. It was decided to use it as a superclass for Agent *and* for other kinds of entities which cause the Agent to react.

The design of this class can be related to the *Abstract Superclass Pattern* (page 67, [Grand, 1998]): the idea is to ensure that an Agent can consider all the entities around in the same way. Indeed, in the duck model designed for the RSP Project, a duck reacts not only to other ducks but also to the wall, and its reaction to the wall (repelled) is very similar to its reaction to the robot sheepdog: in both cases, its reaction depends on the distance (and the unit vector) between him and the other entity (and a constant).

Therefore it was decided that all the entities would implement a common abstract class Entity, which provides the basic attributes:

- name and id: entities can have a common name which might be displayed in the simulation, but they have a **unique** id (identifier), used for example when we need to debug and know quickly which entity is currently used;
- color and size, used to draw the circle in the display;
- coord: the current position (a Point2D) of the entity in the simulation.

and the following methods:

- paint(), which by default draws a circle of radius size at position coord with color color;
- getCoordNearTo(Entity other), which gives the distance between this entity and the Entity other;
- move(), which is an abstract method and has to be implemented by subclasses.

Here it is important to notice that we voluntarily allow subclasses to reimplement those functions. An obvious example is the getCoordNearTo method: although for most of the entities, we will consider their current position (coord) as the position used to calculate the distance from this entity to an other entity, the Arena should reimplement this function. Indeed, the Arena is a big circle and the closest point of the Arena to a given Agent is *not* its position (arbitrarily fixed at $(0, 0)$), but should be calculated as the closest point of the wall, which depends on the position of the Agent and the size of the Arena.

Finally, the move() function, called for each entity by the Animation thread, should be implemented by subclasses: the Arena is not moving, whereas an Agent can move.

The following hierarchy inherits from the superclass Entity (this hierarchy is reproduced in Appendix C as a UML diagram):

- Arena, an Entity which doesn't move and represents the circular wall around the agents.
- Food, another non-moving Entity used to test a new stimuli for the Agents.
- **Agent**, the superclass of all the different Agents. We used inheritance to ensure that our system was working at each stage of the development, by expanding the possibilities of an Agent using subclasses. Thus a simple Agent is just placed randomly inside the Arena and moves randomly. Agent has several subclasses more elaborated:
  - AgentDuck, an Agent that moves according to the initial flock model (equation) of the RSP. This was used to try our system and see how the ducks are flocking with this behaviour.
  - AgentInteractive, an Agent that moves according to the position of the user's cursor (mouse). This allows us to have a "manual" robot sheepdog, which is useful for testing the response of the ducks to a simulated robot.
  - **AgentClassifier**, an Agent that moves according to **a set of rules**, called the behaviour. Two subclasses were developed:
    * AgentClassifierDuck, an Agent having a duck behaviour; this Agent's behaviour is not learned by the system.
    * AgentClassifierAutomatic, an automatic Agent that can change its behaviour over time to test several situations (it will be described in section 6.2).
    * **AgentClassifierLearning**, the last class of our hierarchy. It has a *real* behaviour (since it is a subclass of AgentClassifier) but also an *expected* behaviour, which is the behaviour the system tries to *infer* from the movements of the agent.

The class AgentClassifierLearning implements the learning method described in Procedure 2 page 41. At each time step, a call to its move() method defines its new position according to its *real* behaviour (defined by the user at the beginning of the simulation) and the system tries to **guess** this new position by calculating it from the set of rules describing its *expected* behaviour.

The two processes are completely separated: although the program obviously "knows" the rules programmed inside the mind of the Agent, one part of the program is never "aware" of these rules, and tries to learn them by observing **only the movements** of the Agent. The two behaviours (the *real* one and the *expected* one) never interact during the experiment.

Therefore the Agent itself is not *learning*; it is just moving according to a set of rules. But we have "plugged" a learning system which follows its movements and tries to learn these rules (cf the analogy described page 41).

In order to use a Learning Classifier System technique, the behaviours are implemented as set of classifiers, using the class ClassifierSet. The meaning of these rules is given by the *Template* they used, which should be a subclass of the abstract class Template.

5.3.3 ClassifierSet and Classifier

A ClassifierSet is simply a set of objects of type Classifier. We used the implementations of the Java abstract class Abstract Set provided by the Java library, such as HashSet for a basic set and TreeSet when we needed to sort the classifiers.

The superclass ClassifierSet is used to represent a behaviour, and the subclasses might implement several functions to deal with the Learning System.

The superclass contains already some useful functions, such as:

- addClassifier,removeClassifier;
- getIterator, which returns a way to access sequentially to all the classifiers (described as the *Iterator Pattern* by [Gamma et al., 1994], it is now directly provided by the Java library when using a Set);
- getMatchSet, which extracts all the classifiers matching a given situation (cf explanations page 38 and procedure 6 in Appendix A);
- getActionSet, which by default returns simply the Match Set (no selection); this function has to be overriden by a specific implementation of a LCS.

Here we can recognize the idea of the *Template Design Pattern* (see page 325 of [Gamma et al., 1994], or page 419 of [Grand, 1998]). Indeed the algorithm in the method moveAndLearn of the class AgentClassifierLearning is just a "skeleton" of the method described in procedure 2 (p. 41) using some basic elements (it calls getMatchSet, getActionSet, etc.) but the details of these elements depend on the implementation chosen (by extending ClassifierSet). We have developed the ZClassifierSet as a subclass of ClassifierSet, to implement the specific features of our learning system, such as the Roulette Wheel Selection for the getActionSet method, the UPDATE SET function, the Genetic Algorithm, etc.

The basic algorithms are very common and some implementations can be found: see for example, Martin V. Butz's implementation[2] of the XCS [Butz, 2000]. How-

[2]freely available from the Illinois Genetic Algorithms Laboratory FTP.

ever, it was difficult to re-use this work since Butz has developed an eXtended Classifier System (XCS), which is a different kind of Learning Classifier System (see [Butz and Wilson, 2001] for a good definition of the XCS). His implementation was also very different than ours, since his XCS is not adapted for multi-agent system. However the basic functions such as how to do a crossover or how to do a mutation are the same for most of the applications using a Genetic Algorithm and at the end we re-used around 100 lines of code from his implementation.

The class Classifier represents a classifier, with a condition part and an action part. This class provides several functions to deal with the bits of the classifier (getBits,setBits), and to check if a classifier matches a given sitation (cf procedure 7 in Appendix A).

Here again, to provide a new implementation for the Learning Classifier System, one just needs to create a subclass of Classifier to implement the specific features of his system. We have developed the ZClassifier class which adds to the classifier the notion of **strength** (the eXtended Classifier System, XCS, would use the *accuracy* and the *fitness* instead).

A ClassifierSet contains classifiers whose meaning is given by a Template. Therefore each ClassifierSet contains a reference to the template used.

5.3.4 Template

This is one of the important feature of our system. For the meaning of the rules, we have created a system of *Templates*, which can also be related to the *Template Design Pattern*. The idea is to create an abstract super class providing several generic functions which need some sub-functions that the subclasses must implement.

A *Template* should give the meaning of each bit of the condition and action of a classifier. Several bits can be grouped together to obtain a meaning (in our dog example, the two bits of the Action part are used to describe four different actions, but each bit doesn't have a particular meaning), so we used another class ClassifierComponent, to describe these groups of bits.

The four important roles of a *Template* are:

1. giving the detail of the number of bits used for the Condition and Action parts.

2. giving the description of a given Entity for each bit of the Condition part through the function testCondition(int i, Entity e). In our dog example, testCondition(1,Entity) would call the property isBig() of the Entity, and return the result; testCondition(2,Entity) would call the property isRed() and return the result.

3. giving the description of the meaning of the bits of the Action part, with getVectorFromAction. In our example, an Action 01 would generate a call to the method destroy() for the other Entity. However, in our problem we only consider the movements of the agents, so the Action does not directly generate a call, it just returns a Vector2D which will be used to calculate the movement vector.

4. giving a description of those bits in a way that can be understood by our GUI, so that the user can modify the bits by clicking on a checkbox or by choosing a value in a list (rather than directly modifying the bits).

With these informations, the superclass Template can determine the sequence of bits that define the Entity according to the template (this is done with the method testCondition(Entity)), and can calculate the resultant vector for all the classifiers in an Action Set (getVectorFromActions) by calling sub-procedures implemented by the subclasses.

Therefore, to test different ways of describing the behaviours of the ducks, we just need to extend the superclass Template and give the details of our description in the subclass.

We give in Appendix C the UML diagram of the abstract class Template and one implementation (which corresponds to a definition of a behaviour discussed in chapter 6).

5.3.5 Utility classes

Several small classes were designed for specific tasks:

- Vector2D which manipulates a Vector in two dimensions. It can for example create a Vector from two points, or a unit vector between two points, etc.;
- Utils which is used for saving the coordinates of the cursor, in order to manipulate manually the agent AgentInteractive; it also contains several functions used to record the results (writing the data files, etc);
- ZCSConfig which is used to fix the values of some constants for our ZCS implementation (size of the population, probabilities, etc.);
- Config which deals with some other global constants such as PRINT_MODE used to tell the program how ”verbose” it should be, or FOLDER_DATA giving the folder used to save the data files, etc.

5.4 The Graphical Interface

The Graphical Interface (package simuLCS.graphics) was designed using mostly the Swing library. It has been designed so that the user can modify the behaviour of the Agents and see their reactions in the simulator; the user can also have an idea of how the Learning system is working.

Figure 5.3 gives an overview of the main components of the GUI.

The panel on the right shows the simulation running and the agents moving within the Arena; the user can move the position of the sheepdog (if there is an AgentInteractive) and observe the reactions of the other agents.

The panel on the left is the customization and control panel. The Control Buttons are linked to the functions described above for the class Simulation (start(), stop(),

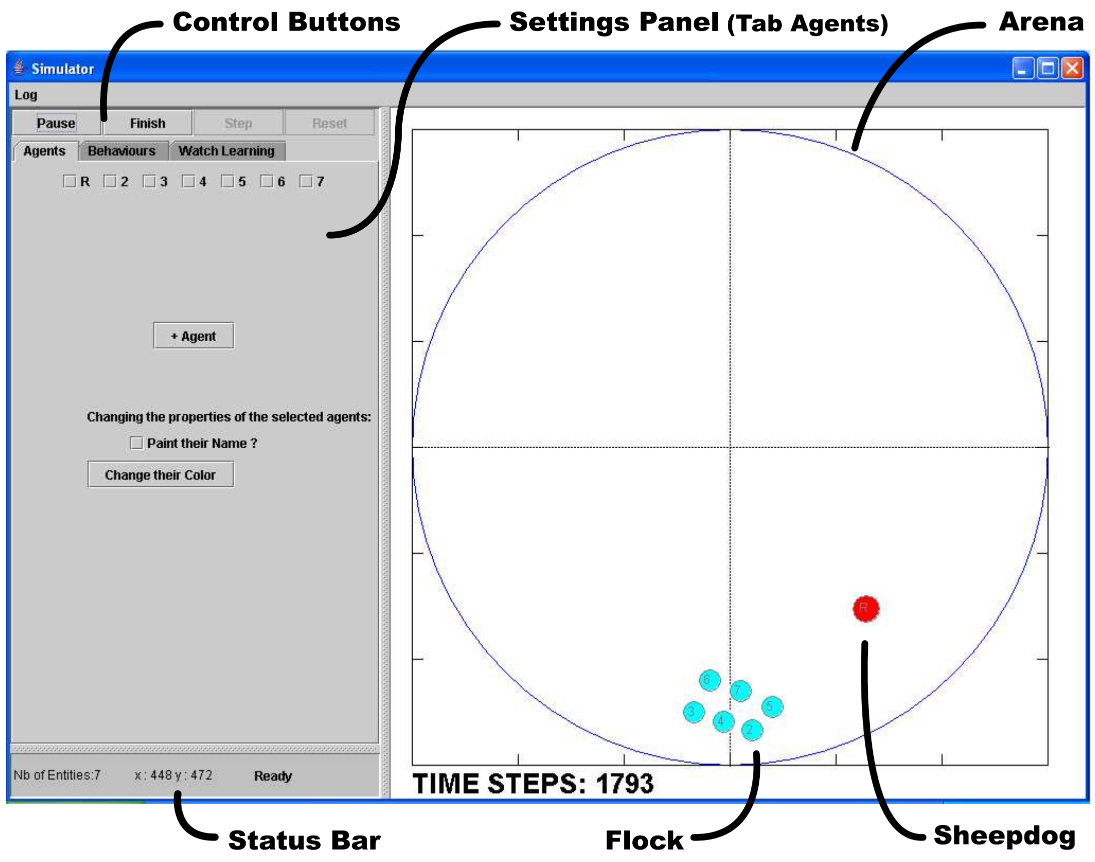


Figure 5.3: Overview of the GUI with the Tab Agents active. For the two other tabs, see partial screenshots on Figures 5.4 and 5.5.

etc.). Below the Control Buttons, there is the customization panel, and at the bottom a status bar gives information to the user.

5.4.1 Implementation of the GUI

We used several techniques and design choices to develop our GUI.

First, in order to be sure that only one window is displayed at a given time, we used the *Singleton Design Pattern* ([Gamma et al., 1994] page 127). This pattern ensures that a class can only have one instance, by protecting the constructor of the class, and providing access to only one instance through a static method `getInstance`.

Second, the GUI was constructed with a tree structure, as it is usually the case with a GUI: a top component is the frame, and inside the frame we can find two panels, the panel for drawing and the panel for customizing; inside these panels, we put other panels, and so on.

The diagram on Figure C.5 in Appendix C shows the top of this tree structure.

However, each of these components must access to the same objects, namely the entities and their classifiers. Furthermore, when a component modifies one entity, the other components should be notified of the changes. In order to do that, we used the following method: all the graphical components (except the main frame which *must* be a subclass of JFrame in order to be displayed) are subclasses of a superclass G_Panel which we created. G_Panel itself is a subclass of the Swing class JPanel, a basic GUI component.

The G_Panel class has the following features:

- static members pointerSimulation, pointerMainWindow, pointerCustom which are initialized at the beginning to provide access to the Simulation, the Window and the Customization Panel;
- static methods repaintAll() which causes all the window to be repainted, and repaintCustom() which causes only the Customization to be repainted;
- static members currentEntitiesSelected and currentEntityToWatch with static methods to access them.

Thus whenever an interactive component (eg. a checkbox) is used by the user, it applies the changes to all the entities in the current selection, by having access to those entities through the static method getSelectedEntities (inherited from G_Panel), and calls the appropriated static method repaint so that all the components are updated.

For example, the following scenario is used to change the color of an Agent (eg. to distinguish it inside a flock):

1. The user clicks on the checkbox corresponding to this Agent below the header of the Tab *Agent*. The program knows which entity is associated to this checkbox because the GUI has access to the instance of the Simulation; the G_Panel containing the checkbox updates the set of selectedEntities by adding this entity, and repaint the Customization Panel (in case there is another component which should display some information about the current selected Entitites).
2. The user click on the button "Change Color" and selects a color: the container of this button will get access to the current selected entities through a call to the corresponding static method inherited from G_Panel. All the selected entities will have their property color changed, and a call to repaintAll will update the drawing so that the new color is displayed.

The call to the repaint method uses a property of the containers directly provided by Swing: whenever a repaint() method is called for a graphic component, the component automatically calls repaint() again for all its "children" ie. all the components it contains. By reimplementing the final paintComponents() (used by repaint()) for a component, we can ensure that it updates its state and displays an updated information.

As we can see, we don't really use the "pure" *Observer Pattern*, because there is only one window, the components can call themselves the repaint() function, since there is no other external display to repaint. The Observer Pattern is more useful when several displays are used to present one information, and each of them doesn't know about the others, which is not the case here.

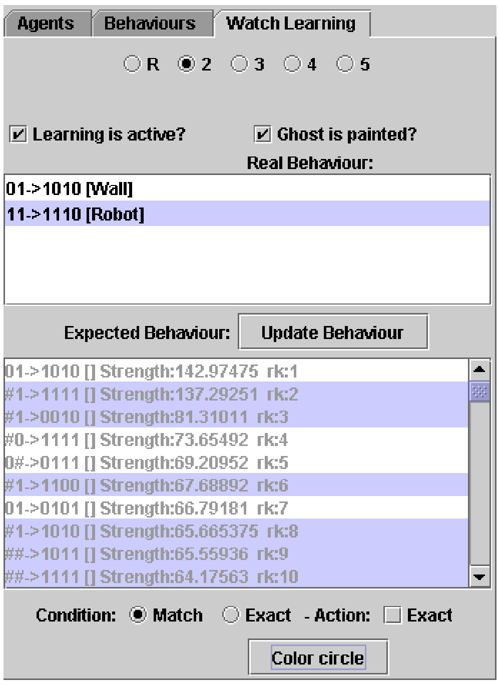
The last feature of our GUI is that we have created re-usable components, so that they can be used in several places of the interface.

5.4.2 GUI Components

While building our GUI, we had to create re-usable GUI components for two main reasons:

1. there are several components which present a different information but with the same display; a typical example is given Figure 5.4: in order to present to the user the real behaviour to compare it with the expected behaviour, the two lists of classifiers need to be displayed. There was scope here to design a re-usable component.

2. when we want to design a new *Template* for the behaviours, it shouldn't require to write the source code for the GUI. By giving the details of the meaning of each bit, the system should generate the adequate GUI so that the user can change the behaviour according to the chosen *Template*. On Figure 5.5 for example, the buttons used to change the Condition and Action of the selected classifier are generated automatically from the description given by the *Template*.



○ R [✓ ● 2] ○ 3 ○ 4 ○ 5

☑ Learning is active? ☑ Ghost is painted?

Real Behaviour:

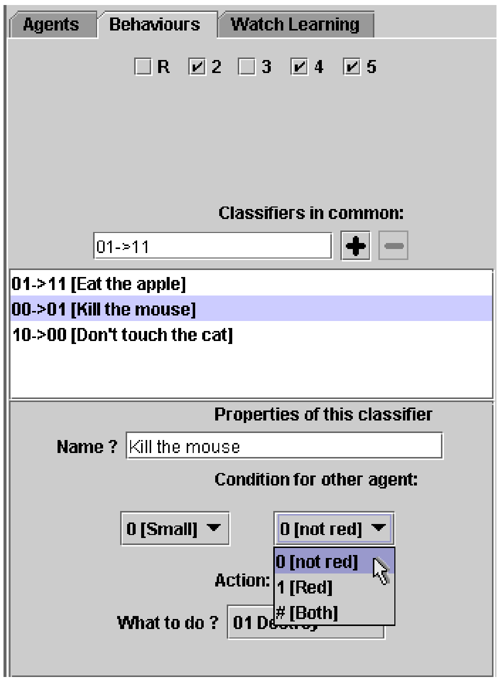
```
01->1010 [Wall]
11->1110 [Robot]
```

Expected Behaviour:

```
01->1010 [] Strength:142.97475  rk:1
#1->1111 [] Strength:137.29251  rk:2
#1->0010 [] Strength:81.31011  rk:3
#0->1111 [] Strength:73.65492  rk:4
0#->0111 [] Strength:69.20952  rk:5
#1->1100 [] Strength:67.68892  rk:6
01->0101 [] Strength:66.79181  rk:7
#1->1010 [] Strength:65.665375  rk:8
##->1011 [] Strength:65.55936  rk:9
##->1111 [] Strength:64.17563  rk:10
```

Condition: [✓ ● Match] ○ Exact - Action: ☐ Exact

Figure 5.4: Two different behaviours shown using the same component.



☐ R ☑ 2 ☐ 3 ☑ 4 ☑ 5

Classifiers in common:

01->11

```
01->11 [Eat the apple]
00->01 [Kill the mouse]
10->00 [Don't touch the cat]
```

Properties of this classifier

Name ? Kill the mouse

Condition for other agent:

0 [Small] 0 [not red]

0 [not red]
1 [Red]
[Both]

Action:

What to do ? 01 De

Figure 5.5: Using components to create the GUI for an example of *Template*.

The component **G_ListClassifiers** displays a behaviour (ClassifierSet) specified as a parameter. It has several options, such as:

- The ability to be enabled (so that the user can select a classifier) and linked with another list: in the figure 5.4, we can see how this works: depending on which *real* classifier is selected by the user, the program highlights the classifiers in the *expected* behaviour (the population of classifiers inside the LCS) whose condition is matched by the condition of the *real* classifier. This is used so that the user can see which classifiers in the population correspond to the same situation (in the example, all the highlighted classifiers in the bottom list could be activated for the robot, since their [Condition] part is more general than **11**, which is the [Condition] part of the selected *real* classifier above).

- The ability to add or not the filters (shown below the bottom panel in Figure 5.4) which change which classifiers are highlighted (for example, if the checkbox *Action: Exact* is activated, only the classifiers having the same action part as the selected *real* one will be highlighted). The change in the highlighting of the classifiers is done by reimplementing the list *renderer* (ListCellRenderer), which tells how the list should look like.

The component **G_ListButtons** displays a list of buttons used at the top of the tabs to select which entities are going to be customized (cf the top of Figures 5.4 and 5.5). As it can be seen on these screenshots, it is also possible to specify if several entities can be selected (in which case the GUI use checkboxes, see Figure 5.5) or just a single one (in which case the GUI use "radiobuttons", see Figure 5.4).

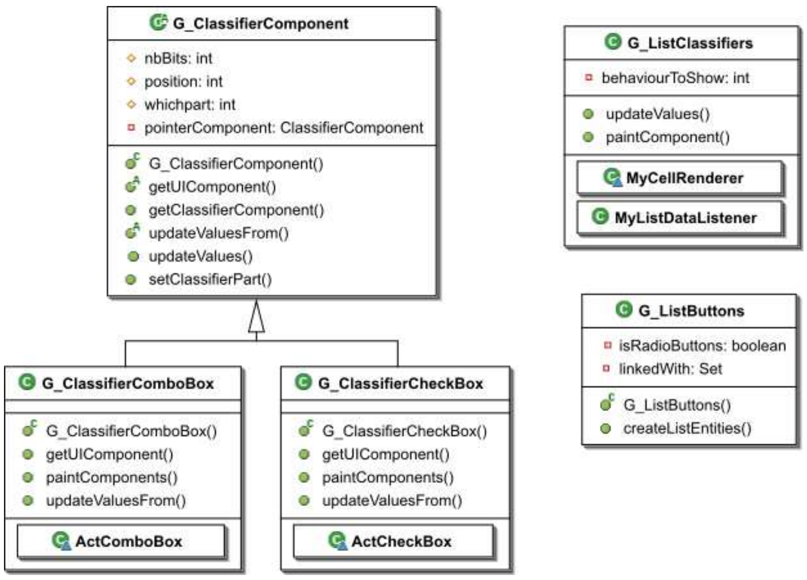


Figure 5.6: Re-usable GUI components developed.

Finally, we created an abstract superclass **G_ClassifierComponent** for the classes implementing components which show one part of the classifier. For example, in Figure 5.5, the *Template* "explains" that the second bit of the Condition is a *"yes/no"* bit with the following meaning: 0 means not red, and 1 means red. The *Template* explains also that another possibility should be both with the usual *don't care* symbol (#). From these informations, the system uses the component G_ClassifierComboBox to generate the list shown on the screenshot. G_ClassifierComboBox is a subclass of

G_ClassifierComponent which just describes how the GUI component should be drawn and how it should update its display for a given value (in the example either `0`, `1` or `#`).

As the diagram on Figure 5.6 shows, we have developed another similar component for this purpose, which provides a checkbox rather than a list of options.

5.5 Interaction with the User

The user can:

1. control the simulation (starting, pausing, resuming, finishing it);
2. customize the agents (color to differentiate an agent among the flock);
3. **customize** their behaviours, by adding or removing interactively rules (Button "+" and "-")and modifying them by clicking on the appropriate components (see Figure 5.5). Thanks to the ability to select several entities, the user can create rules that are shared by several entities so that they have a common behaviour.
4. control an AgentInteractive with the mouse to test the reaction of the other agents.

The system gives the following information to the user:

1. information about the simulation, with the display (position of each agent);
2. information about what the system is doing (through outputs in the console - the level of "verbosity" can be chosen by the user with the menu "Log");
3. visual information about how "well" the system is learning;
4. recorded information about the performance of the system.

For the **third** piece of information, the GUI uses several ways to present this information.

A Thread (UpdateThread, an inner class of the class Simulation, launched when the simulation starts) updates every x seconds (usually 3) the bottom panel in Figure 5.4, to show the current population of classifiers for a given entity. This list is **ranked** so that the user can see which rules are currently the "best" ones in the population (highest strength). Thus the user can check if the first rules of the *expected* behaviour are similar to the *real* rules programmed inside the mind of the Agent. The user can also interactively updates this list at a given time with the button "Update Behaviour" (see Figure 5.4).

Furthermore, at each time step, if the option "Ghost is Painted ?" is selected, the user can see what we called the *ghost* of the entity. The ghost simply represents the *expected* position, guessed by the learning system. When the error between the *expected* position and the *real* position is big, the user can then see two circles, one representing the *real* position of the agent, and one (having a **G** -for **G**host- added to its name) representing the *expected* position. Thus the user can quickly see when the system is having problems to guess the next position of the agent, and when the system is working well (in which case the ghost will hide the agent's drawing). On the example given by the Figure 5.7 the ghost has problems to guess the behaviour of the Agent 2 close to the wall.

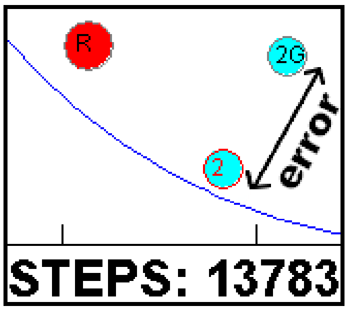


Figure 5.7: Partial screenshot showing the agent 2 and its *ghost* 2G having problems to guess its position. NB: the arrow indicating the error has been added manually to the screenshot.

Finally for the **fourth** piece of information, the system produces data to measure the performance of the learning system for each experiment, with the method detailed in section 3.4.2 page 34. At the beginning of the simulation, the program asks in which file it should put the results, and through the simulation, every x time steps, it writes the average reward obtained by every agent in an output file. When the simulation is finished, the output file is closed and the program generates a command file for the program `gnuplot`, a common plotting program which is then used to create the graphs. It also generates a file with the extension ".rules" giving the set of rules in the population of the LCS for each AgentClassifierLearning at the end of the simulation.

5.6 Summary

We have described in this chapter our program of about 5000 lines of code[3], and explained how we used the *Object-Oriented* paradigm (and *Design Patterns*) to obtain a system able to be easily extended.

We have first built a **simulator** for the following multi-agent system: a set of entities moving together in a 2D environment. The movements of the entities can be:

- pure randomness (class Agent);
- calculated as a function of the positions and features of the other entities (class AgentDuck);
- depending on the position of the user's cursor (class AgentInteractive);
- described by a set of classifiers (classes AgentClassifier and AgentClassifierAutomatic) whose meaning is given by a *Template* (ie. a subclass of the abstract class Template).

Several *Templates* can be tried just by implementing the abstract class Template.
The simulator is working and has been tested with R. Vaughan's model for the ducks (implemented in the class AgentDuck), and we have observed a flocking behaviour as a response to a threat represented by a simulated sheepdog (moved manually using the mouse).

Then a **Learning Classifier System** has been "plugged" in a certain kind of Entity, the AgentClassifierLearning. The specifications of the Learning Classifier System can be implemented by extending the classes ClassifierSet and Classifier. Our adaptation of the Zeroth-Level Classifier System (described in Chapter 4) has been developed with the classes ZClassifierSet and ZClassifier.
Finally, a GUI allows the user to easily customize the simulation between the different experiments and watch the learning system working. Another way to assess the performance of the system is to generate a graph from the data recorded by the program.

[3]statistic generated using David A. Wheeler's 'SLOCCount'.

Chapter 6

Results and Discussion

6.1 Introduction

As we stated previously, our project was experimentally oriented: it consisted of several experiments performed with different *Templates* and different techniques for the Learning System. Following the methodology given page 33, we tried to define a description of an agent's behaviour as a set of rules which can be learnt by our Learning Classifier System. Through a large number of experiments we changed the *Templates*, adapted the algorithms, tuned the parameters of the Learning System, etc. We present in this chapter the biggest steps which improved the performance significantly during this process, and explain how we designed these improvements.

In order to draw useful conclusions from the experiments, we had to specify an experimental setup which should be followed for all the tests. We first present this experimental setup and then give an account of the most important experiments, together with a discussion about the results we obtained.

6.2 Experimental Setup

6.2.1 Motivation

In order to be able to compare the different techniques, we had to design an experiment that could be easily repeated. Because we had in mind an extension to the Robot Sheepdog Project as a major application of our project, we tried to do some experiments with several agents representing the ducks and one agent representing the Robot Sheepdog inside the circular Arena. The Robot was manually moved with the mouse to observe the reaction of the flock (using our class AgentInteractive, cf. page 50) and the Learning System working.

However, it was soon decided that this could not be used as a good benchmark for the performance of our system, since an important part of the experiment was done manually (the Robot's movements). The way the Robot was moved could affect the way the agents were reacting, and as a result the Learning System could work differently. Therefore it was decided to design an **Automatic Robot Sheepdog**, ie. an Agent that would always have the same behaviour towards the others Agents. Since we wanted the system to learn how the Agents are reacting to each other, and especially to the Robot Sheepdog, it was decided that the Robot Sheepdog would have two different behaviours: chasing the agents and going away from them. Thus two different situations can be tested: (1) Agents having the Robot Sheepdog very close to them and (2) Agents having the Robot Sheepdog very far. But using an automatic movement for the Robot Sheepdog we had to face an important problem: what if the flock splits and the Robot chases only some of the Agents?

Furthermore, using several agents for a flock increased a lot the complexity of the behaviour of one agent, since every entity around one Agent can affect its behaviour (cf Procedure 3 page 42). This made the learning very difficult and it was finally decided to make the first experiments with only one agent and the Robot, thus avoiding the splitting effect and reducing the difficulty.

However, this reduced experiment cannot be compared to a simple *Prey/Predator* environment, because the Arena is considered by the Agents as a normal entity . Therefore **3 entities** are in fact "interacting" in the environment which we decided to focus on: the arena (which does not move), the agent representing the duck, and the agent representing the Robot Sheepdog.

6.2.2 Description

We used the following experimental setup for the experiments described in sections 6.3 to 6.5.

The entities used were:

- a circular Arena of 600 pixels of diameter (representing 6 meters);
- an AgentClassifierLearning (called *Agent 2*) of 20 pixels of diameter with a Learning Classifier System "plugged" in (using the specifications detailed in Chapter 4 with the constants given in Appendix D);
- an AgentClassifierAutomatic[1] (called R_a) of 25 pixels of diameter which used the two following behaviours: **(1)** chasing the Agent 2 while avoiding the wall and **(2)** going away from Agent 2 while avoiding the wall. These behaviours are presented in equation 6.1 for behaviour (1) and equation 6.2 for behaviour (2); in those equations, D represents the position of the "duck" (Agent 2) and W is the nearest point on the wall (Arena). We found experimentally that $K_D =$ 30,000 and $K_W =$ 2,000 provided a quite agressive behaviour for Agent R_a, which was useful for really testing the reactions of Agent 2, as we will explain later.

[1]For convenience, this automatic agent was implemented using two behaviours described with classifiers whose meaning was given by the *Template* TemplateRSP that we describe in the first experiment.

$$\vec{M}^1_{R_a} \overset{def}{=} +\left(\frac{K_D}{|\overrightarrow{R_aD}|^2}\right)\widehat{R_aD} - \left(\frac{K_W}{|\overrightarrow{R_aW}|^2}\right)\widehat{R_aW} \tag{6.1}$$

$$\vec{M}^2_{R_a} \overset{def}{=} -\left(\frac{K_D}{|\overrightarrow{R_aD}|^2}\right)\widehat{R_aD} - \left(\frac{K_W}{|\overrightarrow{R_aW}|^2}\right)\widehat{R_aW} \tag{6.2}$$

The experiments were run as follows:

- At the beginning, Agent R_a and Agent 2 are placed randomly inside the Arena.
- During 30,000 time steps, the Agents (R_a and *Agent 2*) move according to their behaviour, and a Learning System tries to guess the behaviour of Agent 2, by evolving a set of 40 rules whose condition and action were initialized randomly and whose strength was initialized with the value 5,000 .
- Every 3,000 time steps, the behaviour of R_a was switched so that only one behaviour is active at a given time. Thus at the beginning, behaviour **(1)** was active during the first 3,000 time steps, then behaviour **(2)** during the next 3,000 time steps, then again behaviour **(1)**, and so on.
- Every 100 time steps, the average reward (over the 100 time steps) the Learning System received is recorded to obtain a graph. The maximum reward that can be obtained is 2,000, as described in Procedure 4 page 43.
- At the end of the 30,000 time steps, the set of rules inside the *expected* behaviour (ie. the Learning System) was saved into a file to compare it with the *real* behaviour. The rules are ranked according to their strength, to see which rules were considered to be the "best ones" by the system.

A first problem had to be solved while doing these experiments: when the Agent R_a started to chase the Agent **2**, it went straight to it and could sometimes push it outside the Arena. This happened because our system is, by definition, using discrete time steps: when the Agent 2 was between the wall of the Arena and the Agent R_a, the movement vector $\vec{M}^t_2$ could put it outside the Arena at the next time step if the Agent R_a was very close to Agent 2. In a real-world situation, the Agent 2 (eg. a duck) would come infinitely close to the wall and the repulsion from the wall would become so strong that the Agent would move either on the left or the right in order to avoid being stuck between the wall and the Agent R_a chasing it.

We avoided this problem in our simulation by combining the following two techniques:

1. We limited the maximum speed of the Agent R_a so that it cannot come too close to Agent 2 too quickly. At each time step, the Agent R_a can make a move of m pixels maximum (we took $m = 5$).

2. R_a's behaviour was modified a little bit so that when it comes quite close to the wall it does not go straight but a little bit on the right (cf Figure 6.1). Thus the Agent 2 will react by moving to the opposite direction and it will not stay stuck between the wall and the Agent R_a any more (cf Figure 6.2).

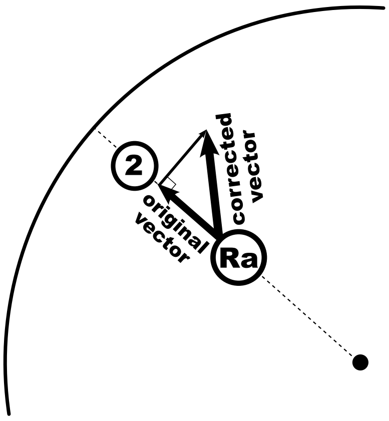


Figure 6.1: Agent R_a does not go straight to Agent 2.

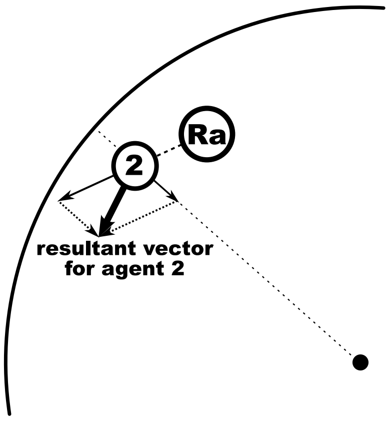


Figure 6.2: After R_a's move, Agent 2 can escape.

We describe now the main steps we took to improve our system, together with the experiments that lead to those changes.

6.3 Experiment 1

6.3.1 Description

This experiment was our first attempt to describe the behaviour of the ducks (equation 6.3) in a form that can be applied to a Learning System.

$$\overrightarrow{d} = \sum_{n=1}^{N} \left[\underbrace{\left(\frac{K_1}{(|\overrightarrow{DD_n}| + L)^2} \right) \widehat{DD_n}}_{(1)} - \underbrace{\left(\frac{K_2}{|\overrightarrow{DD_n}|^2} \right) \widehat{DD_n}}_{(2)} \right] - \underbrace{\left(\frac{K_3}{|\overrightarrow{DW}|^2} \right) \widehat{DW}}_{(3)} - \underbrace{\left(\frac{K_4}{|\overrightarrow{DR}|^2} \right) \widehat{DR}}_{(4)} \tag{6.3}$$

Behaviour of a simulated duck. Refer to Figure 3.1 page 32 for more details.

The idea was to find common characteristics to each of the terms (1) to (4) so that each term can be represented by a rule following a unique *Template*.

We decided to use the following characteristics:

1. each term "generates" a vector $\vec{v}_{e_j \Rightarrow a_i}$ which is part of the resultant vector.

2. the direction of the vector $\vec{v}_{e_j \Rightarrow a_i}$ is given by the sign + or - and the unit vector of the vector between the Agent a_i and the entity e_j. The entity e_j can be the Arena (the nearest point on the wall is used), the Robot, another duck, etc.

3. the scale of the vector is given by a constant (K_1 to K_4) divided by the square distance between the Agent a_i and the entity e_j. A constant L can be added to the denominator to prevent collisions (ie. to ensure that repulsion [term (2)] is greater than attraction [term (1)] at small distances).

Therefore we designed the following *Template*, called TemplateRSP:

- **Condition**: there are three different type of entities which can affect a duck's behaviour (arena, duck, robot). Therefore we used two bits to describe the type of the entity. In order to make the bits more meaningful, the first bit answers the question "Is this entity moving?" and the second bit answers the question "Is this entity dangerous?". Therefore the wall is, by convention, defined by **01**, the robot by **11** and another duck by **10**. Using two bits we can define a new type of entity, namely **00** (not moving and not dangerous, eg. some Food).

- **Action**: all the bits are used to describe the vector corresponding to the reaction of the agent to the other entity. The first two bits define the direction of the vector (**00** for *towards the other entity*, **01** for *on the right of the other entity*, etc. cf Fig. 6.3). We found that good values for the constants $K_{1 \to 4}$ were ranged between 500 and 8,000 to have a quite realistic behaviour, so we used the next four bits to encode 16 different values between 500 and 8,000 (every 500). Finally the seventh bit was set to **1** if the constant L should be added to the denominator.

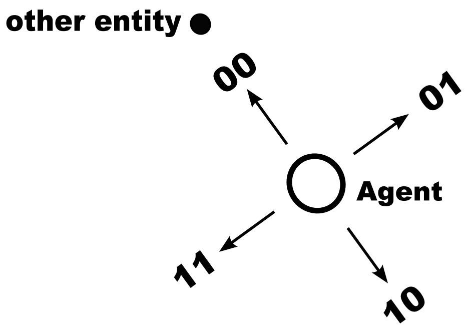


Figure 6.3: Meaning of the two first bits of the Action part: the direction of the vector. Two bits allows us to expand the range of directions, to create more complex behaviours.

With this template, we can then transform the four terms in equation 6.3 into four rules given in Table 6.1 (with the following values for the constants: $K_1 = 3000$; $K_2 = 1000$; $K_3 = 500$ and $K_4 = 8000$).

As an example, if the agent a_i has to react to the robot R whose definition is **11** (*moving* and *dangerous*), it will activate rule (4) and the generated vector will be

	Moving?	Dangerous?		Angle		Scale				Collision	
Duck	1	0	→	0	0	0	1	0	1	1	Term (1)
Duck	1	0	→	1	0	0	0	0	1	0	Term (2)
Wall	0	1	→	1	0	0	0	0	0	0	Term (3)
Robot	1	1	→	1	0	1	1	1	1	0	Term (4)
CONDITION						ACTION					

Table 6.1: Full behaviour of a simulated duck described with the TemplateRSP.

as stated in equation 6.4. Indeed its [Action] part describes a vector in the opposite direction (angle `10`) with a scale of `1111` which corresponds to 15, that we multiply by the step between each value (500) and add to the minimum value (500). The "collision bit" is set to `0` so the constant L is not added to the denominator.

$$\vec{v}_{R\Rightarrow a_i} = -\left(\frac{500 + 15 \times 500}{|\overrightarrow{a_iR}|^2}\right)\widehat{a_iR} \qquad (6.4)$$

With these four rules, **the model given in Equation 6.3 is fully and exactly described**. For each entity, the simulated duck activates the rules whose condition part is the same as the definition of the entity and adds the generated vector to its movement vector. The sum of the generated vectors given by the rules 1 to 4 is exactly the vector $\overrightarrow{d}$ described by Richard Vaughan's model. Indeed, we have tested this behaviour with the class AgentClassifierDuck and we observed exactly the same behaviour as the AgentDuck's one which was directly implemented with the mathematical model (no classifier).

Thus we had a way to describe the behaviour of the ducks with **classifiers**. We then applied the Learning System to see if it could *learn* these four rules that we put inside the "mind" of a simulated duck.

6.3.2 Results

The first results were very disappointing. Although we obtained a good flocking behaviour, the Learning System was not able to learn the four different rules with *several* agents interacting. We tried numerous ways to improve the system, but we did not manage to obtain interesting results.

It was then decided to follow our methodology and to simplify a lot the problem. The first simplification has been given in the Experimental Setup: we restricted the problem to only three entities (one "duck", one robot, and the arena). This was a major decision, since with only one "duck" no flocking behaviour can be observed nor learnt. Therefore we decided to simplify as well the behaviour of the remaining "duck" (we prefer to call it "Agent" in the following since its behaviour is different), and to keep only one "generic" rule given in Table 6.2.

	Moving?	Danger.?		Angle		Scale				Collis.	
Wall & Robot	#	1	→	1	0	1	0	1	1	0	Avoiding Danger
	CONDITION			ACTION							

Table 6.2: Simplified behaviour with the TemplateRSP (only one generic rule).

This rule handles the reaction of the Agent to two different type of entities (thanks to the "don't care" symbol **#**): wall (**01**) and robot (**11**). Therefore this simplified behaviour tells the Agent to go away from *any* dangerous entity.

With this simplified problem, we hoped to get better results, because the Learning System had only one rule to learn.

We ran several experiments following our experimental setup but again the results were not good, as the Figure 6.4 shows.

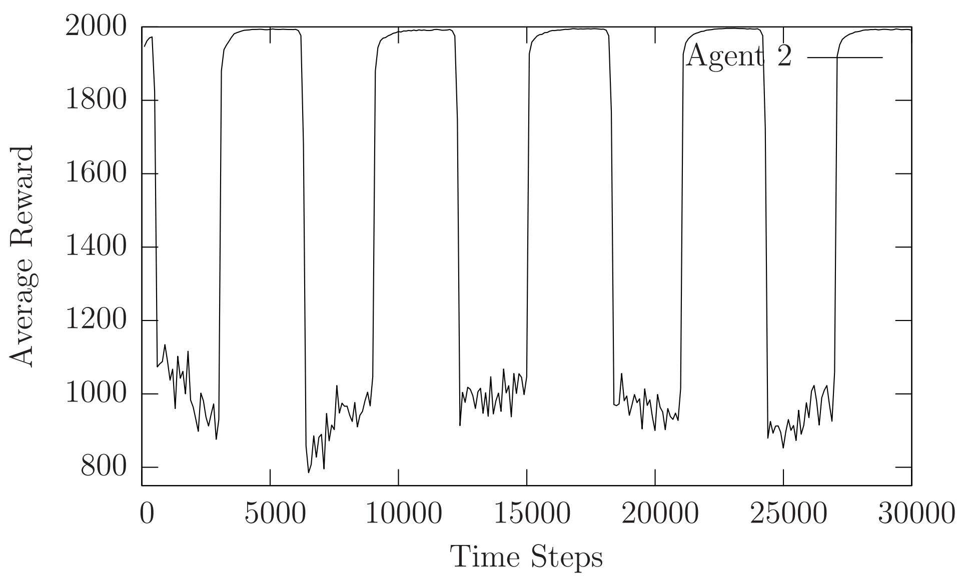


Figure 6.4: Average reward using the TemplateRSP with the *real* behaviour {"#1→1010110"} (only one generic rule).

Figure 6.4 is a typical graph obtained for the experiment described in section 6.2.2. It shows that when the Agent R_a is chasing the Agent 2 (every 3,000 time steps, starting at time step 0), the reward obtained by the Learning System is very low (around 1,000) and **not increasing**. This means that the system is using the wrong rules to guess the next position of the agent; furthermore, it does not manage to increase the number of good rules which could lead to a better reward. The high rewards obtained when the Agent R_a is far from the Agent 2 are easily explained: when Agent 2 is not "stressed" (ie. no entity close to it), its movements are *very* small (it almost doesn't move). Thus any rule will generate a very small vector, since the scale of the vector is divided by the square distance between the Agent and the other entity (denominator) which is big. Therefore even with using ”wrong” rules, the system can still get good rewards since its error is very small (both *real* position and *expected* position are very close to the current position, no matter which rules are used).

6.3.3 Discussion

Even with only one rule and three entities, the Learning System did not manage to learn the behaviour of the Agent 2. At the end of the 30,000 time steps, the *expected* rules (inside the Learning System) are quite different from the *real* one, even though some of the top ranked rules appear to have some similarity with it (see listing of the rules in Table E.1 in Appendix E).

Since the Learning Classifier System paradigm has already been applied with some success reported by researchers in the literature, we tried to understand those bad results by comparing their methods to ours. It appears that most of the time, their Learning Classifier System had to learn rules which have only a limited number of possible actions. Wilson’s ***Boole*** system was learning rules whose Action part can take only two different values [Wilson, 1987]. [Dorigo and Colombetti, 1998] and [Sigaud and Gérard, 2001a] allowed the Action part to have only 8 or 16 values. Indeed, for a given input, if the number of possible actions is high, it is more difficult for the LCS to learn which action is the best one, since there are too many different actions to try.

In our TemplateRSP, the number of possible actions is $2^7 = 128$ (7 bits) and since there are 9 possible conditions (because each bit can take 3 values for the Condition part: 0, 1 or #), the total number of possible rules is $3^2 \times 2^7 = 9 \times 128 = 1152$. These high numbers seem to play an important role in the difficulties encountered by our Learning System, which doesn’t manage to find the good rule out of more than one thousand.

6.4 Experiment 2

6.4.1 Motivation

Following our methodology, we decided to simplify the problem again, since it seems too complicated for the Learning System. In order to do that, we kept only the essential common features of the term of the Equation 6.3, to obtain a new *Template* called TemplateRSPVerySimple with a much lower number of possible actions.

6.4.2 Description

TemplateRSPVerySimple is based upon TemplateRSP (indeed, this class inherits from TemplateRSP to re-use part of the code) while simplifying a lot the Action part:

- **Condition**: unchanged, two bits used as in TemplateRSP.

- **Action**: the direction is given by only **one** bit (thus allowing only two possible values, `0` for *towards the entity* and `1` for *on the opposite direction*). The scale is described by two bits with the minimum value set to 0 and the step to 2,000. There is no "collision" bit: no constant can be added to the denominator.

This simpler *Template*, with only 3 bits for the Action part, was designed in such a way that the behaviour of the previous experiment can still be reproduced. Indeed, the rule "`#1→1010110`" with the TemplateRSP generates the same vector as the rule "`#1→111`" with the TemplateRSPVerySimple (cf. equations 6.5 and 6.6).

$$\begin{aligned}\vec{v}_{E\Rightarrow a_i}(\mathtt{1010110}_{TempRSP}) &= \overbrace{-}^{bits\ \mathtt{10}}\left(\frac{500+\overbrace{11}^{bits\ \mathtt{1011}}\times 500}{|\overrightarrow{a_iE}|^2+\underbrace{0}_{bit\ \mathtt{0}}}\right)\widehat{a_iE} \\ &= -\left(\frac{6000}{|\overrightarrow{a_iE}|^2}\right)\widehat{a_iE}\end{aligned} \tag{6.5}$$

$$\begin{aligned}\vec{v}_{E\Rightarrow a_i}(\mathtt{111}_{TempRSPVerySimple}) &= \overbrace{-}^{bit\ \mathtt{1}}\left(\frac{0+\overbrace{3}^{bits\ \mathtt{11}}\times 2000}{|\overrightarrow{a_iE}|^2}\right)\widehat{a_iE} \\ &= -\left(\frac{6000}{|\overrightarrow{a_iE}|^2}\right)\widehat{a_iE}\end{aligned} \tag{6.6}$$

Thus we put the same behaviour (but described in a simpler way, cf. Table 6.3) into an AgentClassifierLearning to measure how the Learning System is performing.

	Moving?	Danger..?		Angle	Scale		
Wall & Robot	#	1	→	1	1	1	Avoiding Danger
	CONDITION			ACTION			

Table 6.3: Simplified behaviour with the TemplateRSPVerySimple.

6.4.3 Results

The Figure 6.5 shows a typical graph that can be obtained with this new *Template*. The average reward when Agent 2 is chased by Agent R_a is getting closer to 1200, which is a bit better than the previous experiments.

It is important to notice that with this simple *Template*, the Learning System managed to improve its performance when the chasing occurs (see the curve between 0 and 3,000 time steps, between 6,000 and 9,000 time steps and between 12,000 and 15,000 time steps). Indeed it was more easy for the Learning System to find the

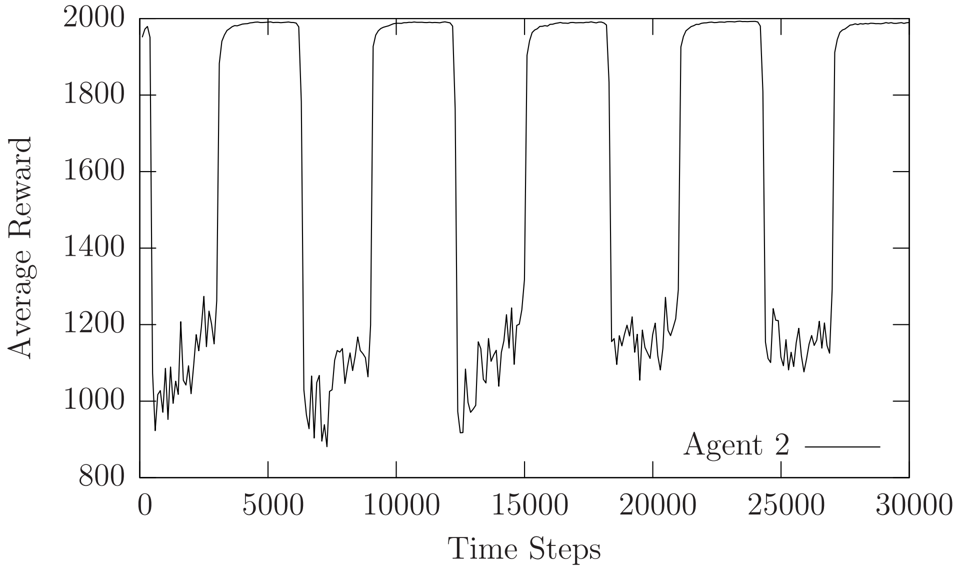


Figure 6.5: Average reward using the TemplateRSPVerySimple with the *real* behaviour {"#1→111"}.

"good rule" and once it has found it, to multiply the number of times it occurs in the population (through the reproduction step of the Genetic Algorithm) and to increase the strengths of these copies. However, it seemed that the information the system had learnt during these 3,000 time steps was somehow *lost* during the next 3,000 time steps (no chasing), since the average reward after that was again quite low.

Table E.2 in Appendix E gives the set of ranked rules after 30,000 time steps.

6.4.4 Discussion

With a simpler *Template*, learning can occur when Agent 2 is stressed. Indeed, when Agent R_a is chasing Agent 2 and stays very close to it, Agent 2 has to make relatively big moves to avoid R_a. Thus if the Learning System uses "bad rules", the *guessed* position will be relatively far from the *real* position: the "bad rules" will be severely punished. This was already the case for the previous experiment, but in this experiment the "good rule" is easier to find (since there are only $2^3 = 8$ possible actions) in the search space; and once it is found it can be rewarded and multiplied in the population.

However, when the Agent 2 is not chased any more, the system "looses" its experience. One possible reason for this problem was explained before: when the Agent 2 is not stressed, it does only very small movements, because all the other entities are far from it, thus almost not affecting it. Hence every activated rule would be rewarded, even if it is not a "good rule", because its generated vector would be very small: the *guessed* position and the *real* position are still relatively close, thus the error is small, and "bad rules" can overcome the "good rules" previously learnt during the chase. When the chase starts again, the "bad rules" are strong and generate low rewards (because now the movements are bigger), and the LCS has to learn *again* that these rules should be weakened.

6.5 Experiment 3

6.5.1 Motivation

In order to overcome the problem previously described, we wanted to find a way that would avoid the loss of experience when the other entities are far from the Agent.

But a more important, recurring problem was to be addressed as well.

This problem that we called **rule interference** is quite specific to the research question we chose, and we need to explain it in more details.

In most of the applications of the Learning Classifier System paradigm, the idea is to learn which actions are the best to execute for **a given situation**. In the example from [Dorigo and Colombetti, 1994] detailed on Figure 1.2 page 16, for instance, the Animat can encounter 16 different situations (corresponding to the 16 different positions of the light) and for each of these situations, only one of the possible Action is *correct*.

In our problem, we decided to learn the rules inside the mind of the Agent *only* from its movements, **without knowing the details** which generated this movement. Let's take our example with one Robot, one Agent, and one circular Arena. To make the explanations more clear, we consider that the rules inside the mind of the Agent are of the form *"**if** entity E is around, **then** move X steps away/towards it"*. We put the rules *"**if** Robot is around, **then** move 2 steps away from it"* and *"**if** Arena is around, **then** move 1 step away from it"*.

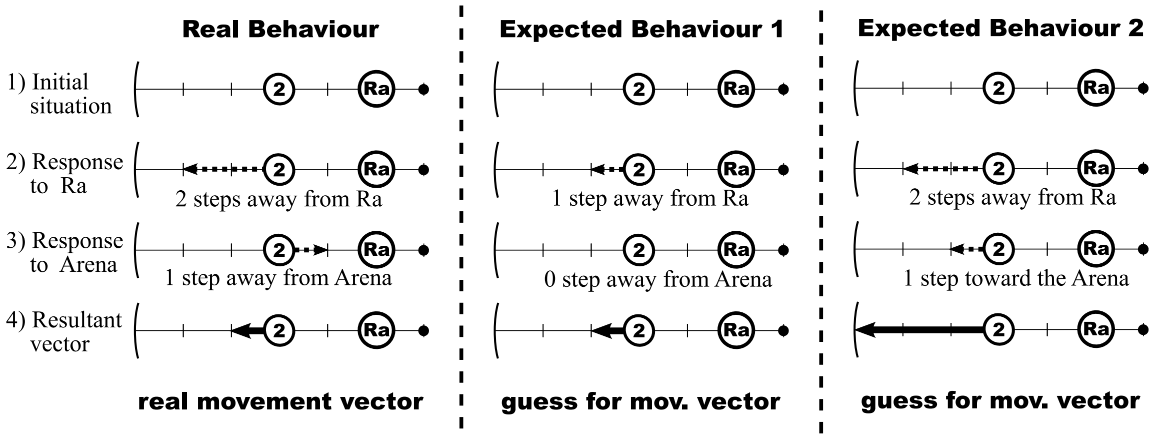


Figure 6.6: Comparison of the resultant vector computed with the *real* behaviour and two different *expected* behaviours (EB) from the same situation. EB 1 generates a correct guess with wrong rules, and EB 2 generates a wrong guess but one of the rules was correct.

If we consider the situation given at the top of Figure 6.6, where the Arena is three steps on the left of Agent 2 and the Agent Ra is two steps on the right of Agent 2, the real movement vector is given at the bottom left, calculated by applying successively the two rules.

Let's imagine that the Learning System uses the Expected Behaviour 1 shown on Figure 6.6: *"**if** Robot is around, **then** move 1 step away from it"* and *"**if** Arena is around, **then** move 0 step away from it"* (ie. no reaction to the Arena). The final guess of the Learning System is *correct*, but the decomposition of the movement is *wrong*. Hence two "bad rules" are going to be rewarded although they do not correspond to the *real* behaviour.

Even worst, when one rule is correct and the other is wrong (Expected Behaviour 2 in Figure 6.6), the resultant is not *correct*, and **the two rules are going to be punished**, even though one was good. Indeed, the Learning System does not have any information in that situation that can help it distinguishing the "good rules" from the "bad rules".

Therefore our problem is more difficult to learn for a LCS than a simple function (f : input $\mapsto$ output), because several rules are combined to obtain the output.

The input is always the same (R_a and Arena around, so two rules must be activated), but the output might be different (depending on the position of the Agents) so the LCS has difficulties to build a record of the form "in *this* situation, *this* action is correct", which is what should normally happened for a classical LCS.

6.5.2 Description

It was decided to change a bit the specification given by the mathematical model of a duck. In this model, a duck should be affected by any other entity inside the arena, because each entity plays a role in the mathematical equation. However, as we explained, this caused the Learning to be difficult, since several rules had to be activated. In order to change the situation over the time (not always the same input from the environment), we decided that the Agent would react **only to the closest entities**. If an entity is too far from the Agent, this Agent does not activate the corresponding rule and this entity does not affect its movement. Thus the Agent can encounter several situations, including situations where only one entity is close to it: in those situations, only one rule can be activated, and if a big reward is obtained by the LCS, it necessarily means that this rule was good and corresponded to an actual rule in the *real* behaviour, since there is no interference with another rule.

This idea solves the two problems we mentioned:

1. When the other entities are far from the Agent, no rules are activated so the "experience" is not damaged by the activation of "bad rules" which could get rewarded because of the small movement vectors;

2. The situation is not always the same for the Agent, and the the problem described in Figure 6.6 would be detected by the LCS when the Agent only reacts to one entity. Indeed, even if the two rules of Expected Behaviour 1 can get a reward in the situation shown on the figure, the LCS would punish separately those two rules in a situation where only one should be activated. For example, when the Agent R_a is the only entity close to Agent 2, the *real* behaviour

generates a movement "*2 steps away from* R_a" and the Expected behaviour 1 generates a guess "*1 step away from* R_a": the rule will be punished.

To implement this idea, we designed what we called a **Visibility Circle**, ie. a Circle whose center is the Agent and whose radius can be understood as the "detection limit" for the Agent: any entity outside the Circle is not detected by the Agent (too far), thus no rule is activated. The *Visibility Circle* is shown on the screenshot E.1 in Appendix E. We took the value 150 for the radius of the *Visibility Circle* (1/4 of the Arena diameter).

6.5.3 Results

Using this technique, we finally obtained good results. We ran several experiments and the Learning System usually managed to find which rule could get the maximum reward and multiplied the number of copies of this rule inside the population. Indeed, only by watching the simulation through the display, we could see very clearly the number of such rules increasing throughout the experiment. These rules can easily be seen in the *expected* behaviour because they are highlighted and the mention "*EXACT*" appears near to their Strength (cf screenshot in Figure E.2 in Appendix E). A typical graph obtained with the TemplateRSPVerySimple and the technique of the *Visibility Circle* is given on Figure 6.7.

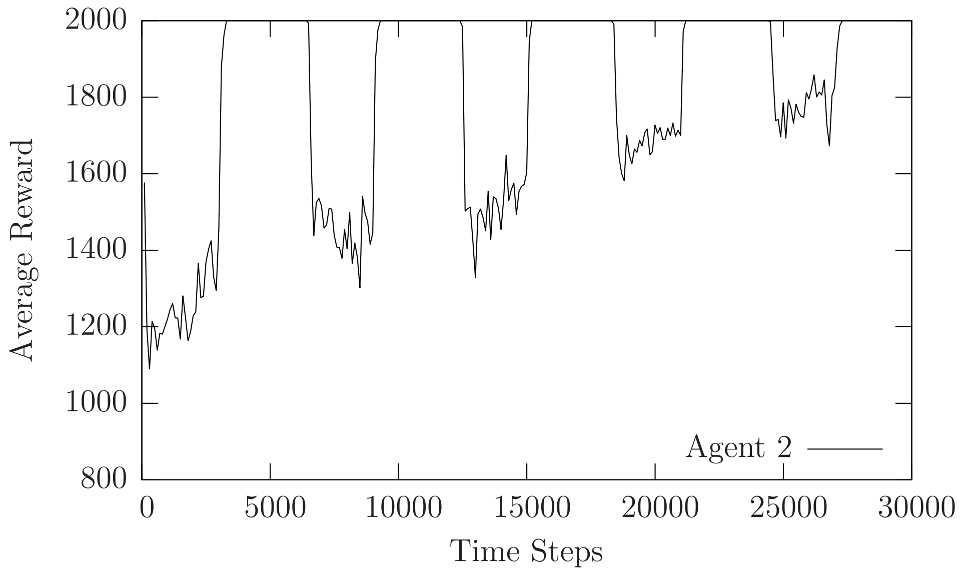


Figure 6.7: Average reward using the TemplateRSPVerySimple and the *Visibility Circle* technique.

Sometimes the learning took more time than the 30,000 time steps, so we let some experiments running until 100,000 time steps. Figure 6.8 shows an example of an experiment where the learning was quite "slow".

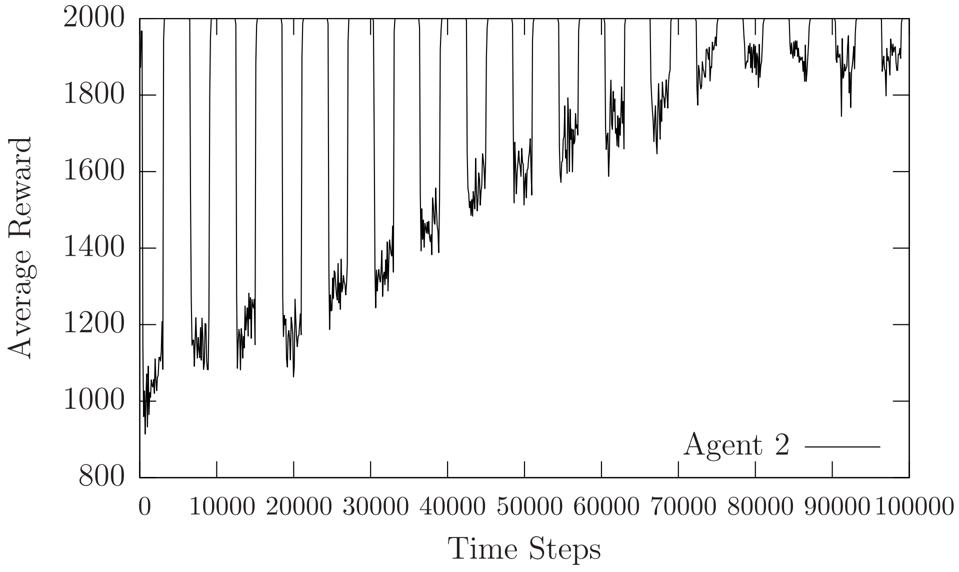


Figure 6.8: Another experiment run until 100,000 time steps.

6.5.4 Discussion

The *Visibility Circle* improved a lot our system, by providing a way to change the number of rules activated depending on the situation.

It can be noticed as well that the experience was not lost when the chase is finished. Indeed, when Agent R_a stopped chasing Agent 2, Agent quickly went away from the Arena, until no wall was inside the *Visibility Circle* any more. Once Agent 2 was far from the Arena (distance greater than the "detection limit") and far from Agent R_a, no entity was stressing it: it did not move *at all*. No rule was activated, and of course the Learning System got the highest reward (cf Figure 6.7, when there is no chase), but this reward is not distributed among the rules, since the Action Set is empty. Thus the learning "paused" during these periods. However it was important that Agent R_a stopped and started again the chase, in order to have some situations where Agent R_a is *coming* close to Agent 2, or when the Arena is the only entity affecting Agent 2.

At the end of these experiments, the number of "good" rules in the population was quite high; Table 6.4 gives the rules obtained for the experiment shown on Figure 6.8. Although most of the rules are of the form "`##`→`111`", they are in fact equivalent to "`#1`→`111`" because the environment does not provide any non-dangerous entity. Therefore the Learning Classifier System tries to obtain the most generic rules, and since it does not experience situations where there is a non-dangerous entity, it does not specialize the Condition `##` into `#1`.

Finally, it should be noticed that the best classifiers are reaching the maximum strength they could get (50,000 - cf. explanations in section A.3 page 82).

It is concluded that a Learning Classifier System *needs* the situation to change in order to learn which actions are best suited for a given situation. If the situation

Real **behaviour**					
#1 → 111 [Avoiding Danger]					
Expected **behaviour** (rules inside the LCS)					
1:	1# → 111	Str.:49481.22	21:	## → 111	Str.:40735.07
2:	## → 111	Str.:48691.996	22:	## → 110	Str.:39465.223
3:	## → 111	Str.:47790.508	23:	## → 111	Str.:39283.445
4:	## → 111	Str.:47618.824	24:	## → 111	Str.:36862.055
5:	## → 111	Str.:47593.445	25:	#1 → 110	Str.:34848.16
6:	## → 111	Str.:47590.78	26:	## → 111	Str.:32832.7
7:	## → 111	Str.:47283.22	27:	## → 111	Str.:28290.654
8:	## → 111	Str.:47024.113	28:	#0 → 011	Str.:24445.97
9:	#1 → 111	Str.:46635.45	29:	## → 111	Str.:24189.408
10:	## → 111	Str.:46474.03	30:	#0 → 111	Str.:23697.78
11:	## → 111	Str.:46442.758	31:	11 → 110	Str.:23411.918
12:	## → 111	Str.:46184.945	32:	11 → 110	Str.:23280.273
13:	## → 111	Str.:45771.82	33:	0# → 111	Str.:23091.12
14:	## → 111	Str.:44562.45	34:	## → 101	Str.:23006.137
15:	## → 111	Str.:44464.508	35:	#1 → 110	Str.:19650.48
16:	## → 111	Str.:42035.598	36:	11 → 110	Str.:18931.193
17:	## → 111	Str.:41972.38	37:	## → 011	Str.:18226.93
18:	## → 111	Str.:41388.0	38:	#1 → 011	Str.:17812.541
19:	## → 111	Str.:41144.824	39:	01 → 110	Str.:17784.512
20:	## → 111	Str.:40867.332	40:	11 → 110	Str.:16079.815

Table 6.4: *Expected* behaviour learnt for Agent 2 after 100,000 time steps, compared with the *real* behaviour. Same experiment as in Figure 6.8. ["Str." means Strength]

is always the same, and especially if several rules are always activated, a LCS has difficulties to learn, because of rule interference (cf Figure 6.6).

It is also concluded that rewards should be given to the Learning System only when there is a high presumption that the activated rules are quite good. In this experiment, when the entities were too far from the Agent, their influence on the Agent were considered too low to be able to distinguish which rules are the good ones, since every rule would generate a small vector, because of the high value of the common denominator (square of the distance between the Agent and the other entity).

6.6 Summary

We have found a way to describe the behaviour of a simulated duck as a set of rules, by creating the TemplateRSP. However, it quickly appeared that a simulated duck had a too complex behaviour to be learnt by our system. Indeed, our Learning Classifier System encountered difficulties to learn a behaviour programmed as a set of several rules. It is believed that most of the difficulties came from the the research question we chose: since we assumed that the system could only observe the resultant movement of the agent, without knowing the details (ie. each vector), a complex behaviour (several rules) generates interferences during the learning process.

Our problem was not to learn a function $f : x \mapsto y$, where x is a situation and y is the correct action. Since our agents were reacting to *several* situations at the same time (each entity around the agent generates a situation), several rules were activated at the same time and the LCS had difficulties to reward only the good ones.

We decided to simplify the problem, considering only three entities and a simpler behaviour (one generic rule), in order to avoid those problems. We designed several improvements to our system, and managed to get good results at the end, for the simpler problem.

Chapter 7

Conclusion

In this chapter a summary of the project is given, and we provide some suggestions for further research.

7.1 Overview of the Project

After having studied the RSP project and the LCS literature, we decided to try to implement some learning in a Multi-Agent system. The idea was to learn the behaviours of autonomous agents, whose "mind" is unknown to the system.

We reached our first objective by building a simulator using the *Object-Oriented Paradigm*, and we added to the simulator a Learning System. We found a way to describe a simulated duck's behaviour as a set of rules (second objective) and a lot of experiments were run (third objective). However, we encountered intrinsic difficulties for the learning process, due to the complexity of the problem we chose, and the way we described the behaviours. Following our methodology, we simplified the problem to obtain better results.

At the end, the system managed to learn a simple behaviour in a world consisting of three entities.

7.2 A Last Learning Agent

There is a last Learning Agent that we didn't describe throughout the thesis: the author. Building the simulator and the LCS was a very good training to *Object-Oriented* paradigm, and we enhanced our knowledge of *Design Patterns*.

The Artificial Intelligence concepts we used throughout this thesis went beyond what was learnt from the courses *Intelligent Systems I & II* and this was very challenging. Indeed, we presented in this dissertation the main steps which improved the Learning System, but we had to do much more experiments to obtain those results. We tried to apply Wilson's XCS [Butz and Wilson, 2001] but this eXtended Classifier System did not work at the beginning. Since its mechanisms are much more complex than a simple ZCS, it was very difficult to understand why it was not working, so we decided instead to adapt the ZCS. However, the ZCS is not the best

Learning Classifier System around now, and it is believed that some further research (see next section) using a more advanced LCS might get better results. Nevertheless we needed to start from the basics of the Learning Classifier System theory to understand the mechanisms, and this project was very interesting in that respect.

7.3 Further Research

Questions abound.
J.H. Holland in [Holland, 1986], p. 622.

It appeared that even the behaviour of a simulated duck was too difficult to be learnt by our system, but we believe that the Learning Classifier System paradigm can be a good approach to learn the behaviour of autonomous agents. We provide below some suggestions to continue our work.

If an agent has a *complex* behaviour (ie. more than one rule), the learning becomes difficult as soon as there are several entities involved, because of the problem of rule interference. It might be a good idea to try to avoid this problem by defining the behaviour in such a way that only one rule is active at a given time. Although the search space might become bigger, because more information should be inside one rule, the learning process should work better since the rewards would be distributed more accurately.

Another way to improve the results would be to improve the Learning Classifier System. We did not use the advanced features of the recent studies, and it might be interesting to try them. Several improvements to the standard LCS are provided in Dorigo's ICS and ALECSYS [Dorigo and Colombetti, 1994]. A new kind of LCS, the Anticipatory Learning Classifier System (ALCS) [Butz et al., 2003], might also be useful in our context. In an ALCS, classifiers have a third component *[Effect]* so that the system can record the effect of a classifier; this allows *planning* to be introduced in the system.

Finally further research could also be done with practical applications in mind. By studying the abilities of some basic real robots (such as an automatic vacuum-cleaner), it could be possible to try real-world applications, where a more "advanced" robot is learning the behaviour of the basic robots in order to avoid collisions, for example.

To conclude, the task of learning the behaviour of autonomous agent might seem quite complicated, since by assumption we have no idea about the possibilities and the level of intelligence of the external agents. However we believe that it is important to explore this problem, if we are to build *really* autonomous robots, dealing with unknown environment including unknown agents in this environment. We hope that the system we built and the conclusions we drew from our experiments will be useful for further research on this problem.

Appendix A

More procedures for our LCS

We give below some common algorithms for a Learning Classifier System; since most of them can be found in the literature (see for example [Butz and Wilson, 2001]), we put them only in Appendix to give a comprehensive description of our LCS. We made significant modifications to the classic APPORTIONMENT OF CREDIT SYSTEM (the way strengths are updated), they are described in section A.3.

A.1 Generate match set

Procedure 6 details how the Match Set is generated. This procedure has three sub-procedures: DOES MATCH (procedure 7) which tells whether or not a classifier's condition matches the situation given by d_j, GENERATE COVERING CLASSIFIER (procedure 8) which creates a new classifier when Match Set is too small and ADD CLASSIFIER cl_c WHILE MAINTAINING SIZE (described below).

Procedure 6 GENERATE MATCH SET FROM $[P]$ SATISFYING d_j **returns** $[M]$

```
1: $[M] \leftarrow [\,]$                                   {empty match set}
2: for each classifier $cl$ in $[P]$ do
3:    if DOES MATCH classifier $cl$ FOR DEFINITION $d_j$ then
4:       $[M] \leftarrow [M] \cup \{cl\}$
5:    end if
6: end for
7: while number of different actions in $[M]$ < minNbActionsForMatchSet do
8:    $cl_c \leftarrow$ GENERATE COVERING CLASSIFIER CONSIDERING $[M]$ FOR $d_j$
9:    ADD CLASSIFIER $cl_c$ WHILE MAINTAINING SIZE IN $[P]$
10:   $[M] \leftarrow [M] \cup \{cl_c\}$
11: end while
12: return $[M]$
```

The procedure DOES MATCH is straightforward. Every bit of the [Condition] part of the classifier is checked: if this bit is # (*"either 0 or 1"*), then it matches, otherwise we need to check that the bit is the same than the corresponding one in the situation d_j (information given by the detectors).

Procedure 7 DOES MATCH classifier *cl* FOR DEFINITION d_j **returns** *boolean*

```
1: for each attribute $x$ in $cl.C$ do
2:   if ($x \neq$ # and $x \neq$ the corresponding attribute in $d_j$) then
3:     return false
4:   end if
5: end for
6: return true
```

The procedure GENERATE COVERING CLASSIFIER creates a new classifier with a condition part which matches the situation d_j (with some `#` with a probability `proba#`) and a random action chosen among the ones that don't exist already in $[M]$ (in order to explore the search space, the system needs to test different action to find which one can get the maximum payoff).

Procedure 8 GENERATE COVERING CLASSIFIER CONSIDERING $[M]$ (AND $[P]$) FOR DEFINITION d_j **returns** *cl*

```
1: create new classifier $cl$
2: for each attribute $x$ in $cl.C$ do
3:   if RandomNumber$[0, 1)$ < proba# then
4:     $x \leftarrow$ #
5:   else
6:     $x \leftarrow$ the corresponding attribute in $d_j$
7:   end if
8: end for
9: $cl.A \leftarrow$ random action not already present in $[M]$
10: $cl.s \leftarrow$ average strength in the population $[P]$
11: return $cl$
```

We don't give the detail of the procedure ADD CLASSIFIER cl_c WHILE MAINTAINING SIZE , which simply ensures that the size of $[P]$ doesn't exceed the maximum size chosen for the population. If it does, then a classifier is selected for deletion, using an **inverse** Roulette Wheel (see procedure 10 for a normal Roulette Wheel), ie. giving more chances to be deleted to the weakest classifiers.

A.2 Generate action set

We generate the action set (procedure 9) by using a Roulette Wheel. The Roulette Wheel is given in the sub-procedure 10, SELECT CLASSIFIER USING RW.

The Roulette Wheel works as follows: a random number between 0 and 1 is chosen, and the classifier returned is the one which has the corresponding "slot" in the Roulette. Each classifier has a slot sized according to its strength divided by the

Procedure 9 GENERATE ACTION SET OF SIZE sizeActionSet FROM $[M]$ **returns** $[A_i]$

```
1: if size of $[M] \leq$ sizeActionSet then
2:    return $[M]$
3: else
4:    $[A_i] \leftarrow [\,]$                                  {initially empty}
5:    for i = 0 to sizeActionSet do
6:       $cl \leftarrow$ SELECT CLASSIFIER USING RW FROM $[M]$
7:       while $cl \in [A_i]$ do
8:          $cl \leftarrow$ SELECT CLASSIFIER USING RW FROM $[M]$
9:       end while
10:      $[A_i] \leftarrow [A_i] \cup \{cl\}$
11:   end for
12: end if
13: return $[A_i]$
```

sum of all the strengths (to obtain a slot which is a part of the interval $[0, 1]$). Thus the classifiers with the highest strengths will have higher probabilities to be selected.

Procedure 10 SELECT CLASSIFIER USING RW FROM SET $[S]$ **returns** cl

```
1: currentPosition $\leftarrow 0$
2: sumStrengths $\leftarrow \sum_{cl\in[S]} cl.s$          {sum of all the strengths in the set}
3: choicePoint $\leftarrow$ RandomNumber[0,1)
4: for each classifier $cl$ in $[S]$ do
5:    currentPosition $\leftarrow$ currentPosition + $(cl.s/$sumStrengths$)$
6:    if currentPosition > choicePoint then
7:       return $cl$
8:    end if
9: end for
```

A.3 Updating the strengths

The procedure 11 updates the strengths of the selected classifiers considering the reward the system got from the environment.

It also considers the maximum reward that can be obtained from the environment, because the strength of a rule should not be decreased if it gets the maximum reward. However, in the method described page 43, each rule looses a portion β of its strength (giving it to a common bucket) before getting the reward. This ensure that a rule is weakened when it does not get rewarded. But if the strength of a rule is too high, it might loose more than it can get, even with the biggest reward. Thus a "good rule" (which got the highest reward) might still be punished. In order to avoid this effect, it was decided that a rule cannot give more to the bucket than it can possibly get with the highest reward. The old method was:

1. each classifier cl in the Action Set $[A]$ with a strength $cl.s$ gives a portion of its strength ($\beta \times cl.s$ with $\beta < 1$) to the common *bucket*.
2. Then each classifier receives its share of the reward (ie. $\frac{r}{|[A]|}$, where $|[A]|$ is the number of classifiers in $[A]$) plus its share of a portion γ of the bucket (ie. $\frac{\gamma \times bucket}{|[A]|}$).

Thus if we call r_{max} the maximum reward the environment can give to the system, we should ensure that a classifier cl should not give more than it can get, ie.:

$$\beta \times cl.s \quad \leq \quad \frac{r_{max} + \gamma \times bucket}{|[A]|} \tag{A.1}$$

Now if we consider that each classifier in $[A]$ had the same strength, it gave the same amount to the bucket and we obtain:

$$\begin{aligned}
\beta \times cl.s &\leq \frac{r_{max} + \gamma \times |[A]| \times \beta \times cl.s}{|[A]|} \\
\beta \times cl.s &\leq \frac{r_{max}}{|[A]|} + \gamma \times \beta \times cl.s \\
(1-\gamma) \times \beta \times cl.s &\leq \frac{r_{max}}{|[A]|} \\
\beta \times cl.s &\leq \frac{r_{max}}{(1-\gamma) \times |[A]|}
\end{aligned} \tag{A.2}$$

The procedure UPDATE (procedure 11) takes this into account to compute the share a classifier should give to the bucket.

Procedure 11 UPDATE $[A]$ CONSIDERING THE REWARD r AND r_{max}

```
1: bucket ← 0
2: for each classifier $cl$ in $[A]$ do
3:     giveToBucket ← $\beta \times cl.s$
4:     if giveToBucket > $r_{max}$ / $((1-\gamma) \times |[A]|)$ then
5:         giveToBucket ← $r_{max}$ / $[(1-\gamma) \times |[A]|]$
6:     end if
7:     bucket ← bucket + giveToBucket
8:     $cl.s$ ← $cl.s$ − giveToBucket
9: end for
10: finalRewardForEach ← $(r + \gamma \times$bucket$)$ / $|[A]|$        {$|[A]|$ is the size of $[A]$}
11: for each classifier $cl$ in $[A]$ do
12:     $cl.s$ ← $cl.s$ + finalRewardForEach
13: end for
```

It is interesting to notice that we can calculate the **maximum strength** a classifier can have, if we know r_{max}. In our problem, we used $r_{max} = 2,000$ (cf procedure 4 page 43). In order to get the maximum reward, the classifier should be the only one to be activated (so that it does not share the reward). When a classifier gives exactly to the bucket the maximum reward it can get, the strength of the classifier has reached the limit (it cannot increase any more). Thus we have:

$$\beta \times cl.s_{max} = \frac{r_{max}}{(1-\gamma) \times |[A]|} \tag{A.3}$$

Hence, if the classifier can be alone in the Action Set $[A]$ (ie. $|[A]| = 1$):

$$cl.s_{max} = \frac{r_{max}}{\beta \times (1-\gamma)} \tag{A.4}$$

With the parameters we chose ($\beta = 0.1$, $\gamma = 0.6$, cf Appendix D), the limit for the strength of a classifier is:

$$cl.s_{max} = \frac{2,000}{0.1 \times (1-0.6)} = \frac{2,000}{0.04} = 50,000. \tag{A.5}$$

Appendix B

Instructions to test the program

The program is available from the following web pages:

- `http://helicos.com.free.fr/mscproject/`
- (mirror) `http://benoit.helicos.com/mscproject/`

On these pages a `jar` file (Java archive) of the program is provided, and there is an access to the documentation.

The `jar` file should be downloaded and tried from a local directory, because it creates data files in order to plot the results.

A Java Run-Time Environment (JRE) should be installed in order to run the program. The program has been successfully tested with a version **1.4.2** of the JRE, but it should work as well with more recent versions of the JRE.

Once downloaded, the program can be run usually by double-clicking on the jar file or by the command: `java -jar simuLCS.jar`.

In order to generate the graphs, `gnuplot`[1] is required. Once the data file (extension `.dat`) has been created, a command file for gnuplot (extension `.gp`) is created as well with the same base name. Run `gnuplot myfile.gp` (where `myfile` is the base name chosen) to obtain a *Postscript* graph (`myfile.ps`).

[1]Freely available at `http://www.gnuplot.info`.

Appendix C

More class diagrams

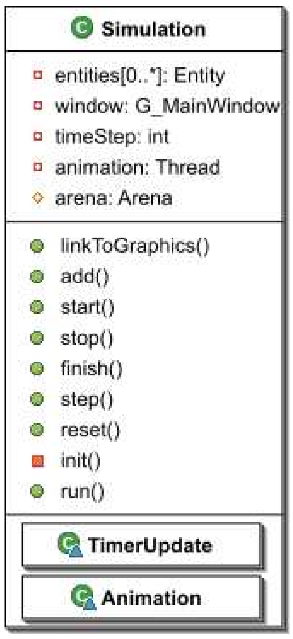


Figure C.1: The main features of the Simulation class.

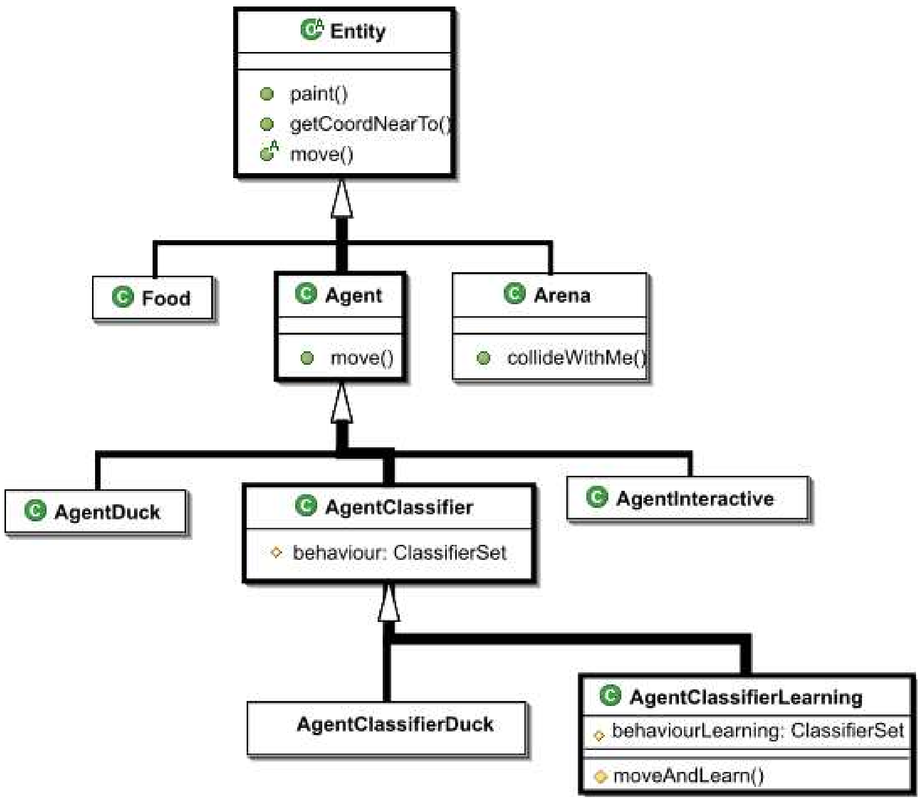


Figure C.2: The Entity hierarchy.

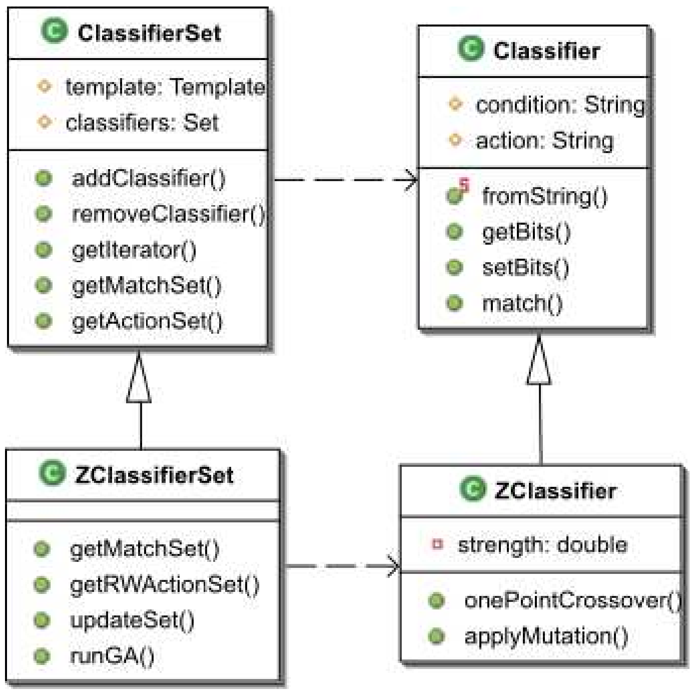


Figure C.3: ClassifierSet and Classifier and our ZCS implementation.

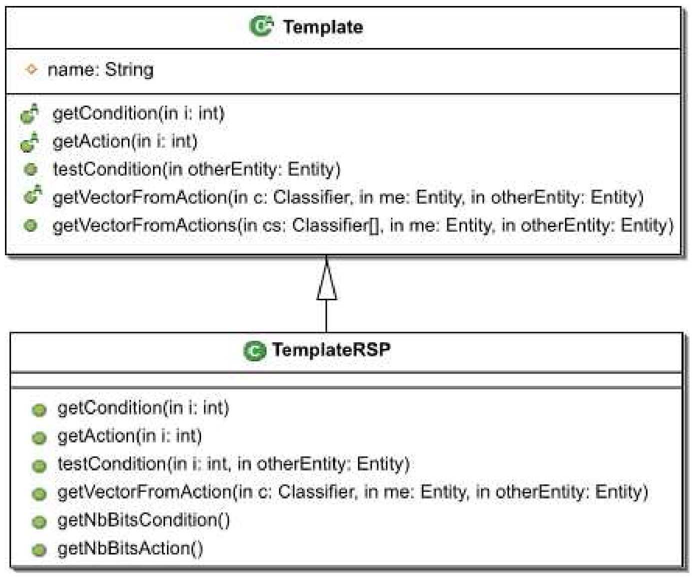

Template
name: String
getCondition(in i: int)
getAction(in i: int)
testCondition(in otherEntity: Entity)
getVectorFromAction(in c: Classifier, in me: Entity, in otherEntity: Entity)
getVectorFromActions(in cs: Classifier[], in me: Entity, in otherEntity: Entity)
TemplateRSP
getCondition(in i: int)
getAction(in i: int)
testCondition(in i: int, in otherEntity: Entity)
getVectorFromAction(in c: Classifier, in me: Entity, in otherEntity: Entity)
getNbBitsCondition()
getNbBitsAction()

Figure C.4: The Template class and an implementation for the RSP.

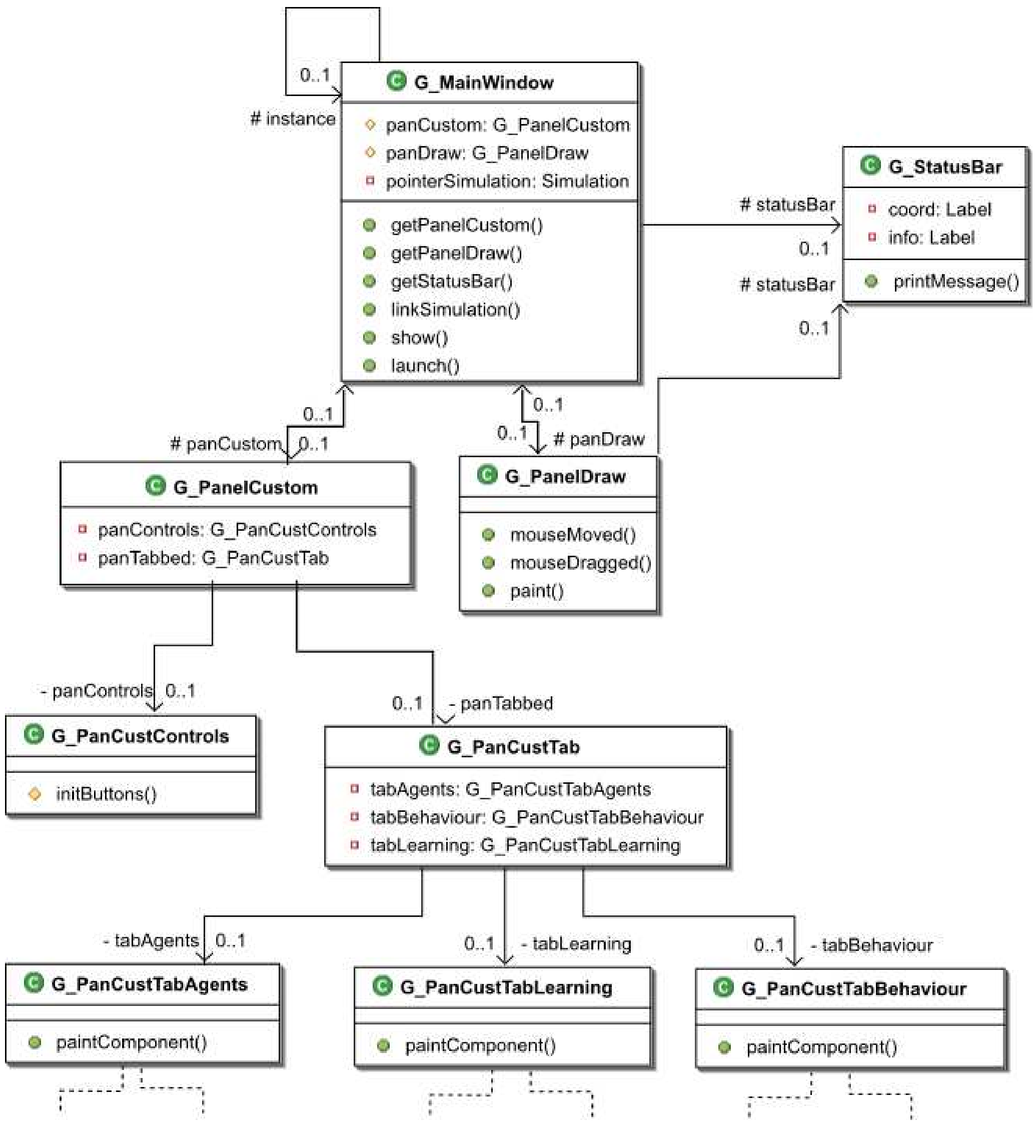


Figure C.5: The top of the tree structure for the GUI.

Appendix D

Parameters

We give below the parameters we used for the experiments described in Chapter 6. It is a well-known problem in Evolutionary Computation that a Genetic-Based Machine Learning system might perform much differently depending on those parameters. We ran several experiments in order to find the best parameters, but we had to face other problems, so we didn't focus so much on tuning the parameters.

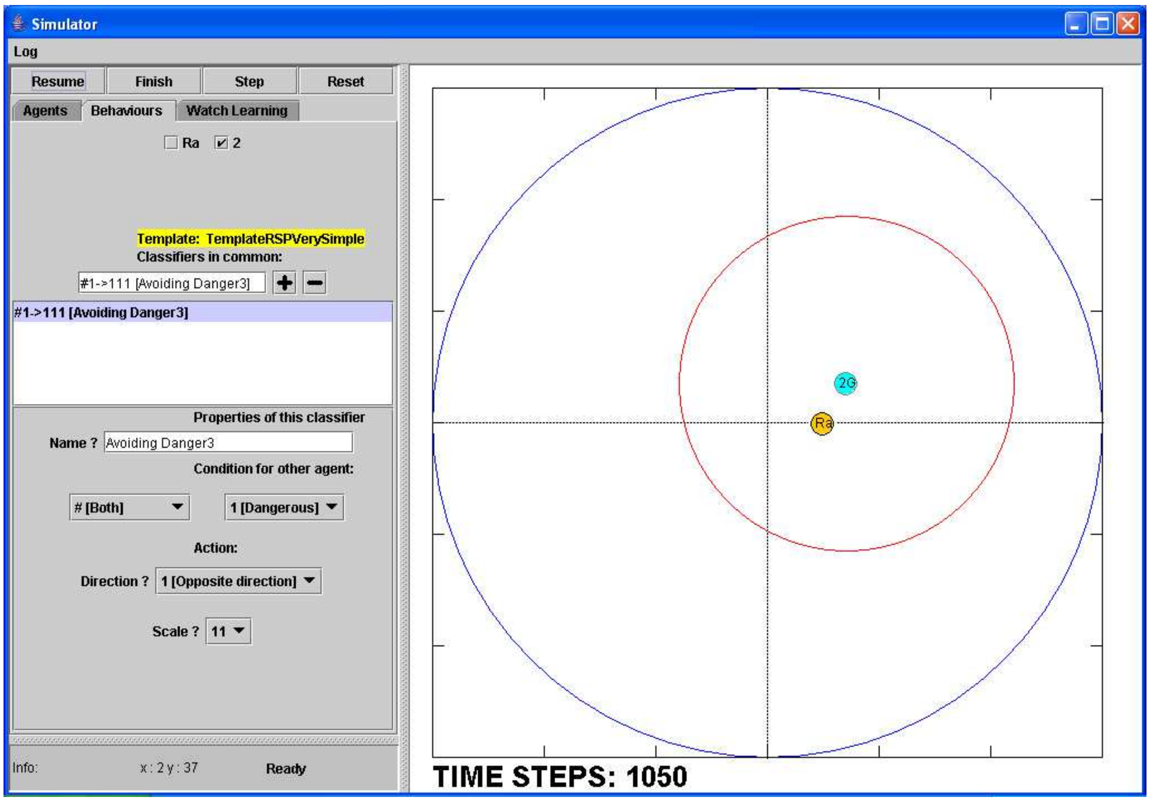
Name	**Value**	**Meaning**	**See**
maxPopSize	40	Max. number of classifiers in the population	p 80
β	0.1	Portion of strength a classifier should give to the bucket	proc. 11 p 82
γ	0.6	Portion of the bucket redistributed	proc. 11 p 82
θ_{GA}	100	`GAperiodicity`	proc. 2 p 41
pX	0.8	Probability of applying the crossover operator to a pair of classifiers	section 4.3.3 p 43
pM	0.1	Probability of applying the mutation operator to one bit of a classifier	section 4.3.3 p 43
p#	0.3	Probability of using a "don't care" symbol for a bit of a classifier when using covering	proc. 8 p 80 (`proba#`)
strengthIni	5,000	The initial strength value for a new classifier	p 63
maxError	10	Maximum error to get a reward	proc. 4 p 43
sizeActionSet	2†, 1‡	The number of classifiers to activate in order to calculate the response for *one* entity	p 42
minNbActions ForMatchSet	4†, 0‡	Miminum number of different actions to have in a Match Set; below this number, covering occurs.	proc. 6 p 79

Table D.1: Parameters used for the Genetic Algorithm and the LCS. Key: †: for TemplateRSP, ‡: for TemplateRSPVerySimple.

Appendix E

Outputs and screenshots

This appendix contains some screenshots of the program during the experiments and some results obtained. Figure E.1 shows the *Visibility Circle* described in Experiment 3. Figure E.2 shows how the user can see that "good" rules are appearing in the population of the LCS (*expected* behaviour).



☐ Ra ☑ 2

Template: TemplateRSPVerySimple
Classifiers in common:
#1->111 [Avoiding Danger3]

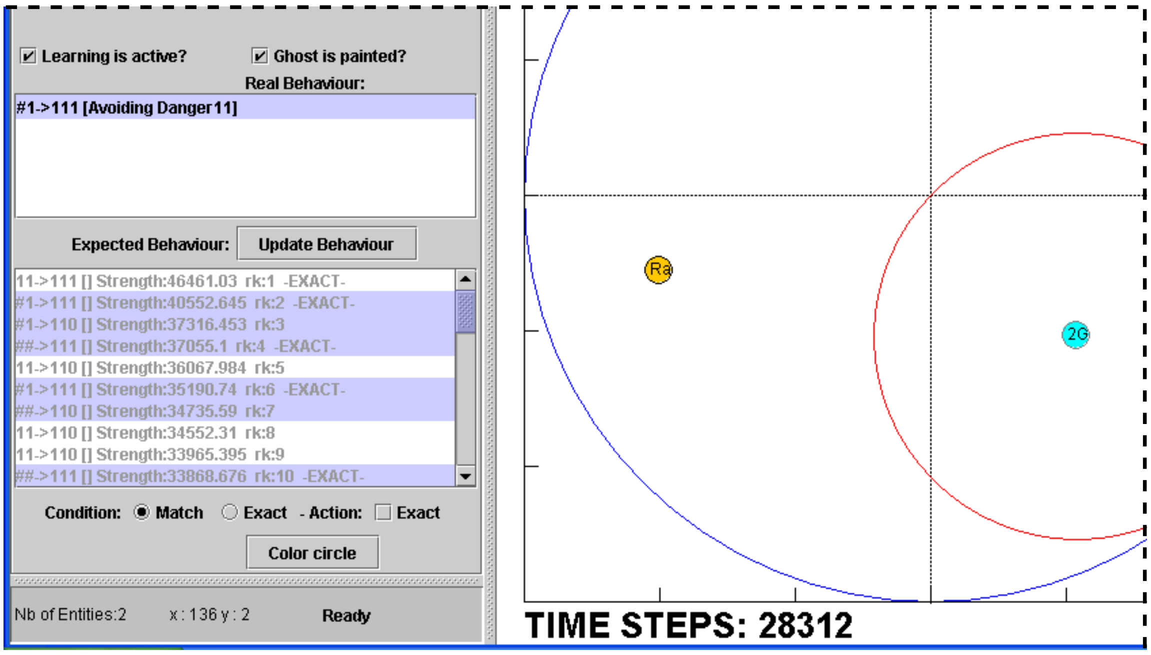
#1->111 [Avoiding Danger3]

Properties of this classifier
Name ? Avoiding Danger3
Condition for other agent:
[Both] 1 [Dangerous]
Action:
Direction ? 1 [Opposite direction]
Scale ? 11

Info: x : 2 y : 37 Ready

TIME STEPS: 1050

Figure E.1: Screenshot with the Visibility Circle activated for Agent 2. Only Agent R_a can affect the movement of Agent 2 at this time step.



```
#1->111 [Avoiding Danger11]

11->111 [] Strength:46461.03  rk:1  -EXACT-
#1->111 [] Strength:40552.645  rk:2  -EXACT-
#1->110 [] Strength:37316.453  rk:3
##->111 [] Strength:37055.1  rk:4  -EXACT-
11->110 [] Strength:36067.984  rk:5
#1->111 [] Strength:35190.74  rk:6  -EXACT-
##->110 [] Strength:34735.59  rk:7
11->110 [] Strength:34552.31  rk:8
11->110 [] Strength:33965.395  rk:9
##->111 [] Strength:33868.676  rk:10  -EXACT-
```

Learning is active? ✓ Ghost is painted? ✓

Condition: [✓ Match] Exact - Action: ☐ Exact

Nb of Entities:2   x : 136 y : 2   Ready

Figure E.2: Partial Screenshot. The number of "good" rules in the *expected* behaviour can be seen during the experiment (highlighted + mention "*-EXACT-*").

Table E.1 presents the rules in the population of the LCS after 30,000 time steps in Experiment 1 (ie. with TemplateRSP). Table E.2 presents the rules after 30,000 in Experiment 2 (ie. with TemplateRSPVerySimple, without the *Visibility Circle*). The rules for Experiment 3 are given in Table 6.4 page 75.

***Real* behaviour**					
#1 → 1010110 [Avoiding Danger]					
***Expected* behaviour** (rules inside the LCS)					
1:	## → 1001111	Str.:24708.082	21:	#1 → 0010111	Str.:23795.25
2:	## → 0011111	Str.:24696.562	22:	#1 → 1100111	Str.:23783.709
3:	## → 0011111	Str.:24670.314	23:	#1 → 0111111	Str.:23780.504
4:	#1 → 0100111	Str.:24489.498	24:	## → 0100001	Str.:23756.434
5:	#1 → 1100101	Str.:24335.17	25:	## → 0100111	Str.:23744.793
6:	## → 0101010	Str.:24328.95	26:	#1 → 0101001	Str.:23701.146
7:	## → 0000000	Str.:24323.71	27:	## → 1011111	Str.:23647.383
8:	#1 → 1101001	Str.:24270.807	28:	#1 → 0110111	Str.:23636.826
9:	## → 0100001	Str.:24250.121	29:	#1 → 1100001	Str.:21047.639
10:	#1 → 0100111	Str.:24225.47	30:	0# → 0011111	Str.:19830.535
11:	#1 → 0110111	Str.:24167.277	31:	## → 0110000	Str.:19126.08
12:	## → 0100101	Str.:24149.822	32:	01 → 0100111	Str.:18694.133
13:	#1 → 1101101	Str.:24146.357	33:	0# → 0100000	Str.:17208.88
14:	#1 → 1011111	Str.:24141.46	34:	1# → 0111111	Str.:16210.034
15:	## → 0100000	Str.:24102.63	35:	## → 0111111	Str.:15862.073
16:	#1 → 0101111	Str.:24055.652	36:	## → 0000101	Str.:15840.572
17:	#1 → 0100001	Str.:24049.596	37:	## → 0100000	Str.:13696.718
18:	## → 1001111	Str.:24032.27	38:	01 → 1010111	Str.:11897.625
19:	#1 → 0100101	Str.:24022.95	39:	#0 → 0000010	Str.:11633.008
20:	#1 → 0100101	Str.:23823.195	40:	#1 → 1111001	Str.:10523.819

Table E.1: Experiment 1 (TemplateRSP); *Expected* behaviour learnt for Agent 2 after 30,000 time steps, compared with the *real* behaviour. ["Str." means Strength]

***Real* behaviour**
#1 → 111 [Avoiding Danger]

***Expected* behaviour** (rules inside the LCS)					
1:	## → 100	Str.:26176.555	21:	#1 → 001	Str.:23705.43
2:	## → 100	Str.:25725.127	22:	#1 → 110	Str.:23652.434
3:	## → 111	Str.:25459.479	23:	#1 → 001	Str.:23615.555
4:	## → 101	Str.:25193.545	24:	## → 111	Str.:23580.506
5:	#1 → 010	Str.:24858.32	25:	#1 → 101	Str.:22974.986
6:	#1 → 001	Str.:24764.258	26:	0# → 100	Str.:22839.021
7:	#1 → 101	Str.:24585.984	27:	01 → 101	Str.:22737.582
8:	## → 100	Str.:24493.748	28:	01 → 011	Str.:22663.19
9:	## → 100	Str.:24339.602	29:	1# → 010	Str.:21984.979
10:	#1 → 101	Str.:24326.63	30:	## → 001	Str.:21824.307
11:	1# → 000	Str.:24191.344	31:	## → 100	Str.:21726.928
12:	## → 010	Str.:24166.566	32:	## → 111	Str.:20732.775
13:	## → 010	Str.:24150.863	33:	#1 → 111	Str.:20627.117
14:	## → 010	Str.:24025.504	34:	#1 → 101	Str.:20387.291
15:	#1 → 101	Str.:23997.0	35:	#1 → 101	Str.:20037.656
16:	#1 → 001	Str.:23955.957	36:	#1 → 001	Str.:17892.3
17:	11 → 001	Str.:23948.387	37:	#1 → 110	Str.:14858.605
18:	## → 111	Str.:23914.191	38:	01 → 110	Str.:12705.314
19:	11 → 101	Str.:23882.092	39:	1# → 010	Str.:11537.886
20:	1# → 100	Str.:23845.623	40:	## → 010	Str.:11537.886

Table E.2: Experiment 2 (TemplateRSPVerySimple without *Visibility Circle*); *Expected* behaviour learnt for Agent 2 after 30,000 time steps, compared with the *real* behaviour.

List of Procedures

Bibliography

Arkin, R. C. (1998). *Behavior-based Robotics*. MIT Press, Cambridge, MA, USA. 19

Balch, T. and Arkin, R. C. (1998). Behavior-Based Formation Control for Multirobot Teams. *IEEE Transactions on Robotics and Automation*, 14(6):926–939. 19

Balch, T. R. (1998). *Behavioral diversity in learning robot teams*. PhD thesis, Georgiate Institute of Technology. Director-Ronald C. Arkin. 19, 45

Batterink, E. (2004). The robot sheepdog. Master's thesis, Oxford University Computing Laboratory. 14

Brusey, J. P. (2002). *Learning Behaviours for Robot Soccer*. PhD thesis, RMIT University, Melbourne, Victoria, Australia. 18

Butz, M. V. (2000). XCSJava1.0: An Implementation of the XCS classifier system in Java. Technical report, Illinois Genetic Algorithms Laboratory. 51

Butz, M. V., Sigaud, O., and Gérard, P., editors (2003). *LNCS 2684: Anticipatory Behavior in Adaptive Learning Systems*. Springer-Verlag. 78

Butz, M. V. and Wilson, S. W. (2001). An Algorithmic Description of XCS. In Lanzi, P. L., Stolzmann, W., and Wilson, S. W., editors, *Advances in Learning Classifier Systems*, number 1996 in LNAI, pages 253–272, London, UK. Springer-Verlag. 15, 40, 52, 77, 79

Dorigo, M. and Colombetti, M. (1994). Robot Shaping: Developing Autonomous Agents through Learning. *Artificial Intelligence*, 71:321–370. 15, 16, 71, 78

Dorigo, M. and Colombetti, M. (1998). *Robot Shaping: An Experiment in Behavior Engineering*. MIT Press/Bradford Books, Cambridge, Ma. 20, 68

Gamma, E., Helm, R., Johnson, R., Vlissides, J., and Booch, G. (1994). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. 29, 51, 54

Geyer-Schulz, A. (1995). Holland classifier systems. In *APL '95: Proceedings of the international conference on Applied programming languages*, pages 43–55, New York, NY, USA. ACM Press. 36

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization and machine learning*. Reading: Addison-Wesley, 1989. 15, 33, 35, 36, 43

Grand, M. (1998). *Patterns in Java*, volume 1. Wiley. 29, 49, 51

Holland, J. H. (1975). *Adaptation in natural and artificial systems.* University of Michigan Press, Ann Arbor, MI, USA. 14, 33

Holland, J. H. (1986). *Escaping Brittleness: The Possibilities of General Purpose Machine Learning Algorithms applied to Parallel Rule-based systems.* Morgan Kaufman, Los Alamos, CA. 15, 78

Kitano, H., editor (1998). *LNCS 1395: RoboCup 1997: Robot Soccer. World Cup I.* Springer-Verlag. 100

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., and Osawa, E. (1997). RoboCup: The Robot World Cup Initiative. In *AGENTS '97: Proceedings of the first international conference on Autonomous agents*, pages 340–347, New York, NY, USA. ACM Press. 17

Kumar, A. (2001). Extending the Robot Sheepdog. Master's thesis, Oxford University Computing Laboratory. 14

Luke, S., Hohn, C., Farris, J., Jackson, G., and Hendler, J. A. (1998). Coevolving Soccer Softbot Team Coordination with Genetic Programming. In [Kitano, 1998], pages 398–411. 18

Lurcock, P. (2001). Computation Project: Robotic Animal Herding. 14, 20

McMillan, M. (2004). Developing Intelligent Control Systems for Simulated Soccer Agents. Master's thesis, St. John's College, University of Oxford. 23

Michalewicz, Z. (1996). *Genetic algorithms + data structures = evolution programs.* Springer-Verlag, Berlin Heidelberg, 3rd edition. 15

Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach.* Prenctice Hall. 11, 25

Salustowicz, R., Wiering, M., and Schmidhuber, J. (1998). Learning Team Strategies: Soccer Case Studies. *Machine Learning*, 33(2-3):263–282. 18

Sigaud, O. and Gérard, P. (2000). The use of roles in a multiagent adaptive simulation. In *Proceedings of the 14th European Conference in Artificial Intelligence, Workshop on Balancing reactivity and Social Deliberation in Multiagent Systems*, Berlin, Germany. 20, 23

Sigaud, O. and Gérard, P. (2001a). Being Reactive by Exchanging Roles: an Empirical Study. In Hannebauer, M., Wendler, J., and Pagello, E., editors, *LNAI 2103 : Balancing reactivity and Social Deliberation in Multiagent Systems*, pages 138–157. Springer-Verlag. 20, 21, 68

Sigaud, O. and Gérard, P. (2001b). Using Classifier Systems as Adaptive Expert Systems for Control. In Lanzi, P.-L., Stolzmann, W., and Wilson, S. W., editors, *LNAI 1996 : Advances in Classifier Systems*, pages 138–157. Springer-Verlag. 20

Stone, P. and Veloso, M. (1998). Layered Approach to Learning Client Behaviors in the Robocup Soccer Server. *Applied Artificial Intelligence*, 12(2):165–188. 18

Vaughan, R. (1999). *Experiments in Animal-Interactive Robotics.* PhD thesis, Oriel College, University of Oxford. 13, 22, 32

Wilson, S. W. (1985). Knowledge growth in an artificial animal. In *Proceedings of an International Conference on Genetic Algorithms and Their Application.*, pages 16–23, Pittsburgh, PA. 19

Wilson, S. W. (1987). Classifier Systems and the Animat Problem. *Machine Learning*, 2(3):199–228. 19, 68

Wilson, S. W. (1994). ZCS: A Zeroth Level Classifier System. *Evolutionary Computation*, 2(1):1–18. 35, 37, 38