TO APPEAR IN MACHINE LEARNING JOURNAL, VOL.19, N0.3, 1995

ALECSYS and the AutonoMouse:
Learning to Control a Real Robot by Distributed Classifier Systems

Marco Dorigo*

Progetto di Intelligenza Artificiale e Robotica
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci 32
20133 Milano, Italy
dorigo@elet.polimi.it

Correspondence address

Marco Dorigo
IRIDIA
Université Libre de Bruxelles
Avenue Franklin Roosevelt 50
CP 194/6
1050 Bruxelles, Belgium
Voice: +32-2-6502729
Fax: +32-2-6502715
mdorigo@ulb.ac.be

Running head: ALECSYS AND THE AUTONOMOUSE

* This work was partially written while the author was at International Computer Science Institute, 1947 Center Street, Suite 600, Berkeley, 94704-1198 California, USA.

Abstract

In this article we investigate the feasibility of using learning classifier systems as a tool for building adaptive control systems for real robots. Their use on real robots imposes efficiency constraints which are addressed by three main tools: parallelism, a distributed architecture, and training. Parallelism is useful to speed up computation and to increase the flexibility of the learning system design. A distributed architecture helps in making it possible to decompose the overall task into a set of simpler learning tasks. Finally, training provides guidance to the system while learning, shortening the number of cycles required to learn. These tools and the issues they raise are first studied in simulation, and then the experience gained with simulations is used to implement the learning system on the real robot. Results have shown that with this approach it is possible to let the AutonoMouse, a small real robot, learn to approach a light source under a number of different noise and lesion conditions.

Keywords: Learning Classifier Systems, Reinforcement Learning, Genetic Algorithms, Animat Problem.

Running head: ALECSYS AND THE AUTONOMOUSE

1. Introduction

The main goal of the research presented in this article is to investigate the feasibility of using learning classifier systems as a tool for building adaptive control systems for real robots. This is achieved by first experimenting with a variety of design solutions in a simulated world, and then using the best solutions found to implement the learning system for the real robot.

The main focus of the article is on the investigation of design solutions which make the learning system a rapid learner. We identified three tools which could help us in this direction: parallelism, a distributed architecture, and training.

To develop our system we started from Holland's learning classifier system (CS) (Booker, Goldberg and Holland, 1989; Holland, 1975). The first step to make it more efficient was the design of the Improved Classifier System (ICS). ICS is a CS enhanced by automatic activation of the genetic algorithm (GA) when the bucket brigade has reached a steady state, by the introduction of *mutespec*, a new genetic operator whose task is to eliminate over general classifiers, and by a mechanism to dynamically change at run time the set of classifiers which are used by the performance system, so that those classifiers which caused a bad performance of the system are no longer used (see Section 3.1 and, for more details, (Dorigo, 1993)). Although improvements in performance were achieved, still they were not enough for our goals. We turned therefore to a parallel version of ICS, which was made faster by means of *low-level parallelism* involving distribution of the ICS over a set of transputers. Moreover, through what we call *high-level parallelism*, it is possible to define many concurrent ICSs, each one with a specific behavior. The resulting system was called ALECSYS and is described in Section 3.2 (see also (Dorigo and Sirtori, 1991b)). ALECSYS has been used as a tool to build learning controllers composed of many cooperating modules.

The investigation of what kind of structure to give to the learning system was made easier by the use of ALECSYS, which permitted the direct implementation on a distributed architecture of the different behavior decomposition solutions we decided to investigate. In this article we propose as a good solution to behavior decomposition a hierarchical architecture in which a set of learning classifier systems cooperate in the solution of a learning problem. This architecture has some basis on ethology grounds and has been described in detail in (Dorigo and Schnepf, 1993). An

architecture built using ALECSYS is composed of a (hierarchically structured) set of cooperating modules, each one implementing either a simple behavior pattern or a coordination activity. In this respect our work is related to that of Brooks (1990). A major difference is that Brooks' behaviors are designed and not learned. Also closely related is the work of Mahadevan and Connell (1992), which uses a subsumption architecture (Brooks, 1991a) whose components are learning systems instead of being finite state machines, like in Brook's approach. A main difference is that they use Q-learning with statistical clustering instead of classifier systems; moreover, their coordination mechanisms (that is, inhibitions among behaviors) is designed, while in ALECSYS coordination is accomplished by a particular learning module (see also Dorigo and Colombetti, 1994, for more details on architectures and coordination). Lin (1993a) also proposed a hierarchical architecture which resembles the one proposed here, but he uses Q-learning instead of classifier systems. Another approach loosely related to the one presented in this article was proposed by Brooks in (Brooks, 1991b). His work is on automatic translation of genetically evolved high-level programs into robot executable programs, while we are developing programs directly at the robot language level. He suggests that a major difficulty in using artificial life techniques lies in the transferring of programs evolved in simulated environments to actual robots. On the contrary, we believe that one of simulation's main tasks is as a tool to acquire knowledge about how to build learning systems.

The last issue discussed in the article is the *training problem*. The use of a trainer[1] can make the learning process much quicker, particularly so in the first phases of learning when the robot's moves are chosen mostly random. The use of a trainer brings in the following issues, which we investigate experimentally: (i) whether or not to use punishment in training a single classifier system; (ii) which *shaping* procedure should be used, that is, how should reinforcements be distributed among the modules comprising a (hierarchical) learning system, and finally (iii) the role that different kinds of data used by the trainer to evaluate the real robot moves have on the development of a robust system.

The article is organized as follows. In Section 2 we discuss related work. In Section 3 are presented ICS and ALECSYS. Section 4 introduces the experimental setting we used for

[1] In this article a trainer is a reinforcement program which provides step-by-step reinforcements.

simulations. We defined and developed the simulation environment and a simulated AutonoMouse in order to avoid spending a lot of time in doing real world experiments. Experience gained with this system was later used to implement the learning system for the real AutonoMouse. The results obtained in simulations with different kinds of architectures and shaping policies are presented in Section 5. In Section 6 we show how the methodology developed for simulated robots can be smoothly transferred to the real robot. Here we describe the real AutonoMouse's capabilities and report experimental results. In Section 7 we draw some conclusions and present some thoughts about future work.

The reader should note the distinction we make between the terms "ALECSYS" and "AutonoMouse[2]" (real or simulated): We refer to the physical body of the learning system as AutonoMouse and to its brain as ALECSYS. The distinction is most striking when reporting experiments with the real AutonoMouse, because at that point ALECSYS must be able to model not only external environment regularities, but also inaccuracies with which the AutonoMouse senses the world or performs actions (noise and lesions).

2. Related work

The research presented in this article belongs to the reinforcement learning research field. Holland's classifier system (Holland and Reitmann, 1978; Booker, Goldberg and Holland, 1989), Sutton's Dyna architectures (Sutton, 1990), Watkin's Q-learning (1989), and the connectionist approach (e.g., Barto, Sutton and Anderson, 1983; Williams, 1992) are some among the most used techniques applied by researchers to solve reinforcement learning problems. These techniques often overlap, with cross-fertilization between the different approaches. For example neural nets have been used by Lin (1992) in connection with Dyna architectures, while Compiani and others (Compiani, Montanari, Serra and Valastro, 1989) have shown a structural correspondence between

[2] The AutonoMouse project was born at Politecnico of Milano in early 1990 with the goal of building mouse-sized, autonomous robots. The name given to these robots came quite naturally: AutonoMice. The robot described in the last part of this paper is the second of a growing generation. References to the first AutonoMouse can be found in (Dorigo and Sirtori, 1991a). While phase I of the project was devoted to testing the feasibility of the robot design, phase II is concerned with the development of autonomous learning capabilities.

neural nets and classifier systems. A recent result (Dorigo and Bersini, 1994) is a construction which shows Q-learning being the same as a simplified version of a CS.

From the learning classifier system side, ALECSYS has been inspired by Booker (1988), Holland and Reitman (1978), Wilson (1987), and Zhou (1990). All these authors have faced, at different levels of depth, the problem of building a knowledge base appropriate to solve the "Animat" problem, i.e. the problem, posed by Wilson (1987), of designing an artificial animal with learning capabilities sufficient to let it adapt to its environment and meet the goal of survival.

The research presented in this article is related to Wilson's as far as general background ideas are concerned. It addresses some of the problems Wilson considered to be of great importance to the development of working Animats: how to control the growth in complexity faced by a learning classifier system that has to solve real world problems, how to coordinate behavioral modules, and how to use classifier systems organized in a hierarchical structure.

We consider the GOFER system, developed by Booker (Booker, 1988), to be the father of ALECSYS; in GOFER are investigated many of the open problems we are also interested in. A major difference is that in GOFER the distinction between sets of rules implementing different behaviors is not made as explicit as in our system. ALECSYS uses an explicit hierarchical architecture composed of many cooperating CSs and different behavior patterns are implemented as different learning systems, while this is not the case in GOFER. This makes ALECSYS a very flexible system which has allowed the solution of some of the problems Booker reported (e.g., competition between classifiers that realize completely different behaviors, extinction of behavioral sequences not relevant in a particular situation, insufficient distinction between coordination messages and action rules).

Zhou (Zhou, 1990) investigated the Animat problem using CSs. In particular, he was concerned with the problem of how to accumulate experience in a way that can be easily retrieved when needed to solve previously solved problems. Up to now, we have not inserted any memory management mechanism similar to those proposed in Zhou's work.

SAMUEL, the system developed by Grefenstette (Grefenstette, Ramsey and Schultz, 1990; Grefenstette, 1991) addresses the problem of learning decision rules for a sequential task. A major difference is that he uses symbolic condition-action rules, while ALECSYS uses low-level bit

strings. Also, in (Grefenstette, Ramsey and Schultz, 1990) the authors seem to be mostly interested in studying the effects that sensor noise has on the learning process. Their results suggest that a good policy would be to use noisy sensors during the learning phase unless it is certain that the sensors will not be noisy in the target environment. We approach the problem from a different point of view (see experiments presented in Section 6). In fact, instead of trying to develop in simulation rules robust enough to be used also with the real robot, we use simulations only to speed up the general tuning of ALECSYS software. Rules to be used in the real environment are developed during experiments in the real environment. We also stress the importance of some aspects of the reinforcement program, like the kind of data which it uses to judge the learning system performance, in realizing a robust system.

3. ALECSYS: a parallel learning classifier system

ALECSYS is the main subject of this section. We first briefly describe ICS (Improved Classifier System), a version of the sequential classifier system (CS) in which we introduced many innovations. These innovations proved very useful in increasing the CS efficiency, but inadequate for our needs. We therefore implemented a parallel version of ICS, called ALECSYS, whose goal was to both increase ICS speed, and to allow the design and implementation of many concurrent cooperating ICSs. ALECSYS is described in the second part of this section.

3.1 Description of ICS

ICS is an improved version of the typical Holland learning classifier system (Booker, Goldberg and Holland, 1989). The following major innovations have been introduced (for a detailed description and experimental evaluation of these changes to the classical CS, see (Dorigo, 1993)).

Calling the genetic algorithm when a steady state is reached

In classic implementations of CSs, the genetic algorithm (i.e., reproduction, crossover and mutation) is called every N cycles with N a constant whose optimal value is experimentally determined.

A drawback of this approach is that experiments to find N are necessary, and that, even if using the optimal value of N, the genetic algorithm will not be called, in general, at the exact moment when rule strength accurately reflects rule utility. This happens because the optimal value of N changes in time, as it depends on the environment dynamics and on the stochastic processes embedded in the learning algorithms. If the genetic algorithm (GA) is called too soon, it uses inaccurate information; if it is called long after a steady state was reached, there is a loss of computing time (from the GA point of view, the time spent after the steady state has been reached is useless). A better solution is to call the genetic algorithm when the bucket brigade (Holland, 1980) has reached a steady state. In this way the genetic algorithm is called exactly when the strength of every classifier correctly reflects its usefulness to the system. The problem is how to correctly evaluate the attainment of a steady state. We have introduced a variable $E_{CS}(t)$, called *energy* of the CS at time t, defined as the sum of the strength of all classifiers. A CS is said to be at a steady state when $E_{CS}(t) \in [E_{Min}, E_{Max}] \quad \forall t \in [t, t-k]$, where $E_{Min} = \min\{E_{CS}(t), t \in [t-k, t-2k]\}$, $E_{Max} = \max\{E_{CS}(t), t \in [t-k, t-2k]\}$, and k is a parameter. In this way are excluded both cases in which $E_{CS}(t)$ is increasing or decreasing, and situations in which $E_{CS}(t)$ is still oscillating too much. Experiments have shown that the value of k is very robust; in the experiments presented in this article it was set to 50, but no substantial differences were found for values of k in the range between 20 and 100.

The mutespec operator

Mutespec, a new operator, is used to reduce variance in the reinforcement received by default rules[3]. *Mutespec* takes a classifier, randomly chooses a "don't care" (#) symbol in one of the conditions, and generates two classifiers in which the selected "don't care" symbol is replaced by a 0 and a 1, respectively (the parent classifier remains in the population). The *mutespec* operator is applied to overly general rules that fire in conflicting situations, causing actions that sometimes are useful and sometimes are not. To identify these rules, a measure of each rule's strength variance is

[3] Default rules are rules which cover broad categories of system responses, and are opposed to specific rules which cover situations in which default rules are incorrect. In CSs default rules are implemented by using don't care (#) symbols (Riolo, 1989).

taken. The *mutespec* operator is applied to the rule with the highest variance, provided that its variance is at least K times the average variance of classifiers in the population (a good value is K=1.25, determined experimentally).

Dynamically changing the number of used classifiers

In ICS the number of used rules dynamically changes at runtime (i.e., the classifier set cardinality shrinks as the bucket brigade finds out that some rules are useless or dangerous). The rationale for reducing the number of used rules is that when a rule strength drops to very low values its probability of winning the competition is very low and, in the case that it wins the competition, it is very likely to propose a useless action. As the time spent matching rules against messages in the message list is proportional to the classifier set cardinality, cutting down the number of matching rules results in the ICS performing a greater number of cycles (a cycle goes from one sensing action to the next one) than a standard CS in the same time period. This causes a quicker convergence to a steady state. The genetic algorithm can be called with higher frequency and therefore a greater number of rules can be tested. Experiments presented in (Dorigo, 1993) for a light following task have shown that the average time for a cycle using the dynamical change of the number of activatable classifiers is about 50% of the average time (expressed in seconds) for a standard CS cycle. Moreover, the proposed method causes both the number of cycles required to reach a pre-fixed performance level to diminish, and the maximum performance achieved in a fixed number of cycles to increase.

3.2 ALECSYS architecture

ALECSYS is a tool for experimentation with parallel ICSs, designed to obtain a powerful enhancement of the possibilities of CSs. One of the main problems faced by CSs trying to solve real problems is the presence of heavy limitations to the number of rules which can be employed, due to the linear increase of the basic cycle complexity with the classifier set cardinality. With a personal computer, for instance, in order to have acceptable elaboration times, only small populations, e.g. sets of 100-500 classifiers, can be used. One possible solution to increase the amount of processed information without slowing down the basic elaboration cycle comes from the

use of new parallel architectures, such as the Connection Machine or the transputer. A parallel implementation of the CS on the Connection Machine has been proposed by Robertson (1987). That work has demonstrated the power of such a solution, but still retained, in our opinion, a basic limit. As the Connection Machine is a SIMD (Flynn, 1972) architecture, the most natural design for the parallel version of the CS is based on the "data parallel" modality, i.e., a single flow of control applied to many data. Therefore, the resulting implementation is a more powerful, but still a classic, learning classifier system.

As our main goal was to give our system features such as modularity, flexibility and scalability, we implemented ALECSYS on a transputer system[4]. In fact, because of its MIMD architecture, the use of a transputer system permits both the presence of many simultaneously active flows of control operating on different data sets, and a gradual growth of the learning system without major problems. We organized ALECSYS in such a way as to have both SIMD-like and MIMD-like forms of parallelism concurrently working in the system. The first was called *low-level parallelism* and the second *high-level parallelism*. Low-level parallelism operates within the structure of a single ICS and its role is to increase the speed of the ICS. With high-level parallelism allowing various ICSs to work together, the complete learning task can be decomposed into simpler learning tasks running in parallel.

3.2.1 Low-level parallelism: A solution to speed problems

In the following is described the *micro-structure* of ALECSYS, that is the way parallelism was used to enhance the performance of the ICS model. The ICS, like Holland's CS, can be depicted as a set of three interacting systems (see Fig.1): the performance system, the rule-discovery system (genetic algorithm), and the credit apportionment system (bucket brigade algorithm). In order to simplify the presentation of the low-level parallelization algorithms, the various systems are discussed separately. First we show how the CS performance and credit apportionment systems were parallelized, and then we will make similar considerations regarding the genetic algorithm.

---

[4] ALECSYS is implemented in parallel C and Express, and runs on Quintek boards inserted in PCs (ATs or better). The simulation environment is written in C and runs on the host computer (a PC). Both ALECSYS, the simulation environment, and the communication software for experiments with the real robot are available. Requests should be directed to the author of this paper.

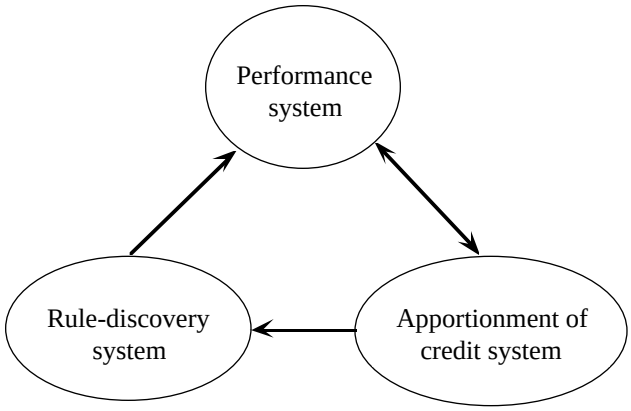


Figure 1. *Functional organization of ICS.*

The performance and credit apportionment systems

A basic execution cycle of the sequential CS can be looked at as the result of the interaction between two data structures: the list of messages, ML, and the set of classifiers, CF. Therefore, we decompose a basic execution cycle into two concurrent processes, MLprocess and CFprocess, that interface to input process DTprocess (DeTector) and output process EFprocess (EFfector), as shown in Fig.2.

The processes communicate by explicit synchronization, with the following sequence of steps.

1 - MLprocess receives messages from DTprocess and places them in the message list ML.

2 - MLprocess sends ML to CFprocess.

3 - CFprocess "matches" ML and CF, calculating bids for each triggered rule.

4 - CFprocess sends MLprocess the list of triggered rules.

5 - MLprocess erases the old message list and makes an auction among triggered rules; the winners, selected with respect to their bid, are allowed to post their own messages, thus composing a new message list ML.

6 - MLprocess sends ML to EFprocess.

7 - EFprocess chooses the action to apply and, if necessary, discards conflicting messages from ML; EFprocess receives reinforcements and is then able to calculate the reinforcements owed to each message in ML; this list of reinforcements is sent back to MLprocess, together with the remaining ML.

8 - MLprocess sends the set of messages and reinforcements to CFprocess.

9 - CFprocess modifies the strengths of CF elements, paying bids, assigning reinforcements and collecting taxes.

10 - While not stopped, go back to step 1.

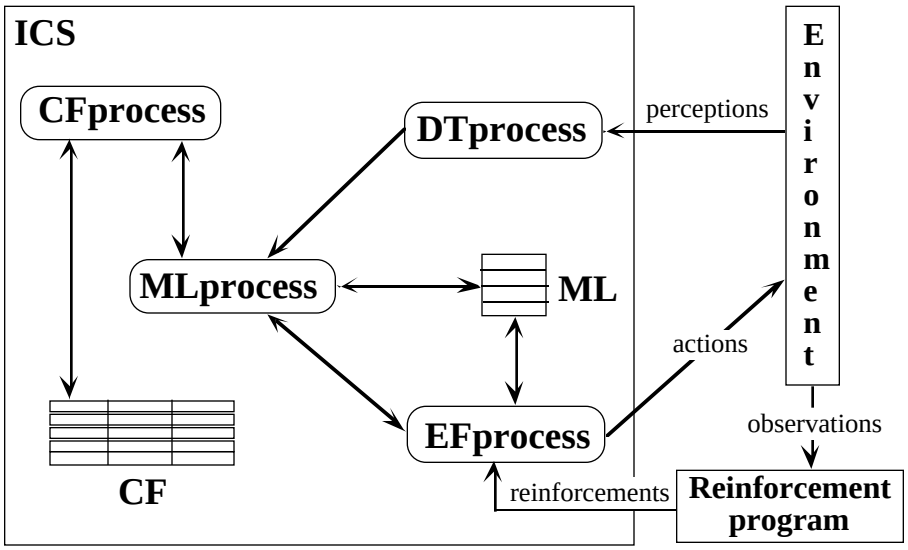


Figure 2. *Concurrent processes in ICS.*

Steps 3 and 4 (matching and message-production) can be executed on each classifier independently. So we split CFprocess into an array of concurrent subprocesses {CFprocess.*1*, ..., CFprocess.*i*, ... , CFprocess.*n*}, each one taking care of *1/n* of CF. The higher *n* goes, the more intensive the concurrency is. In our transputer-based implementation, we allocated about 100-500 rules to each processor (this range was experimentally determined as the most efficient, see (Dorigo, 1992)). CFprocesses can be organized in hierarchical structures, such as a tree (see Fig.3) or a toroidal grid. (It should be noted that the structure actually chosen deeply influences the distribution of computational loads and therefore, the computational efficiency of the overall system, see (Camilli and others, 1990).)

Many other steps can be parallelized. We propagate the message list ML from MLprocess to *i*-th CFprocess and back, obtaining concurrent processing of credit assignment on each CFprocess.*i*. (The auction among triggered rules is subject to a hierarchical distribution mechanism, and the same hierarchical approach is applied to reinforcement distribution.) Further details on the implementation can be found in (Dorigo, 1992).

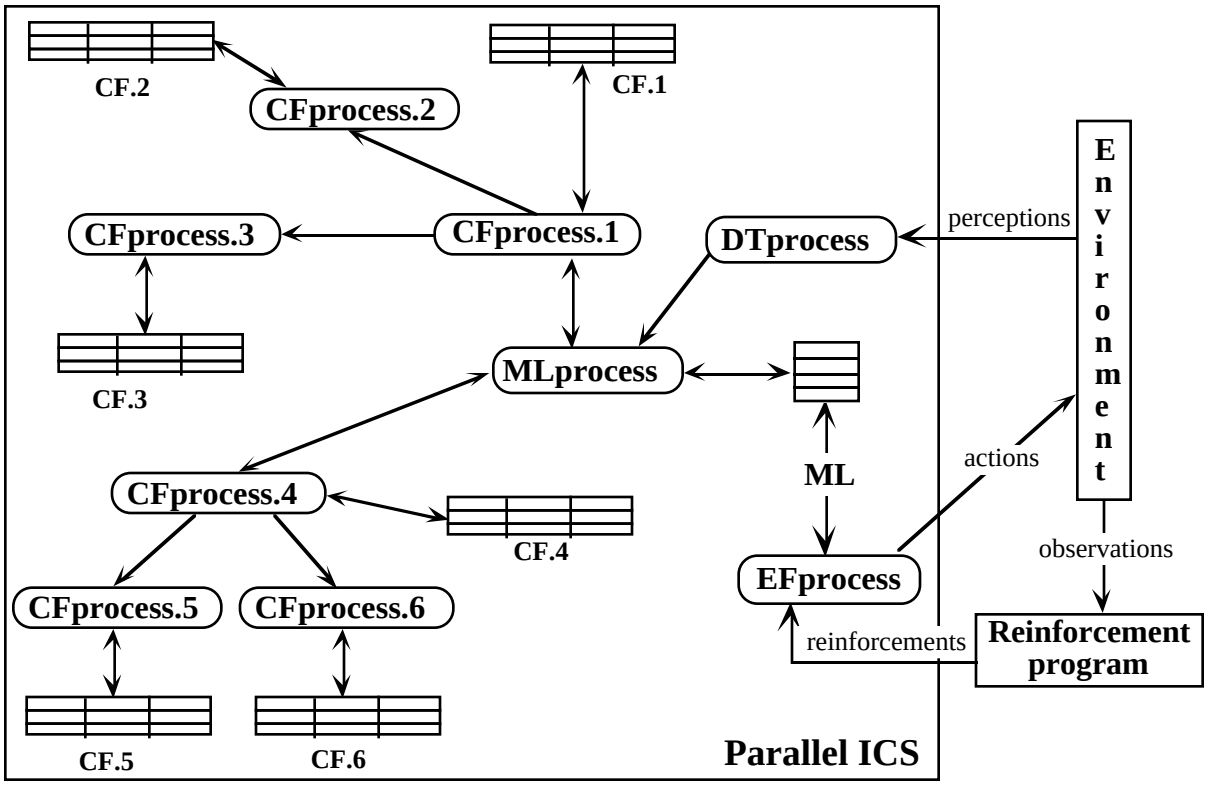


Figure 3. *A parallel version of the ICS. In this example CFprocess of Fig.2 is split into six concurrent processes CFprocess.1, ... , CFprocess.6.*

The rule discovery system

We now illustrate briefly our parallel implementation of the GA. A first process, named GAprocess, can be assigned the duty to select from among CF elements those individuals that are to be either replicated or discarded. It will be up to the (split) CFprocess, after receiving GAprocess decisions, to apply genetic operators, each single CFprocess.i focusing upon its own fraction of CF population (see Fig.4).

MLprocess stays idle during GA operations, as it could affect CF strengths, upon which genetic selection is based; likewise GAprocess is "dormant" when MLprocess works.

A typical genetic algorithm works as follows.

1 - Within CF two sets of rules are selected, one composed of rules to be replicated (*parents* classifiers), the other of those to be replaced (*offspring* positions).

2 - *Parents* are mated two by two;

3 - A genetic crossover operator is applied to each couple created at step 2, thus generating a new pair of rules (*offspring*).

4 - *Offspring* undergo a stochastic mutation operator.

5 - Offspring generated at steps 3 and 4 replace rules chosen at step 1 (offspring positions).

Considering our parallel implementation, step 1 may be seen as an auction, which can be distributed over the processor network by a "hierarchical gathering and broadcasting" mechanism, similar to the one we used to propagate the message list. Step 2 (mating of rules) is not easy to parallelize, as it requires a central management unit. Luckily, in CS applications of the GA the number of pairs is usually low and concurrency seems to be unnecessary (at least for moderately sized populations). Step 3 is the hardest to parallelize because of the many communications it requires, both between MLprocess and the array of split CFprocesses and among CFprocesses themselves. Step 4 is a typical example of local data processing, extremely suited to concurrent distribution.

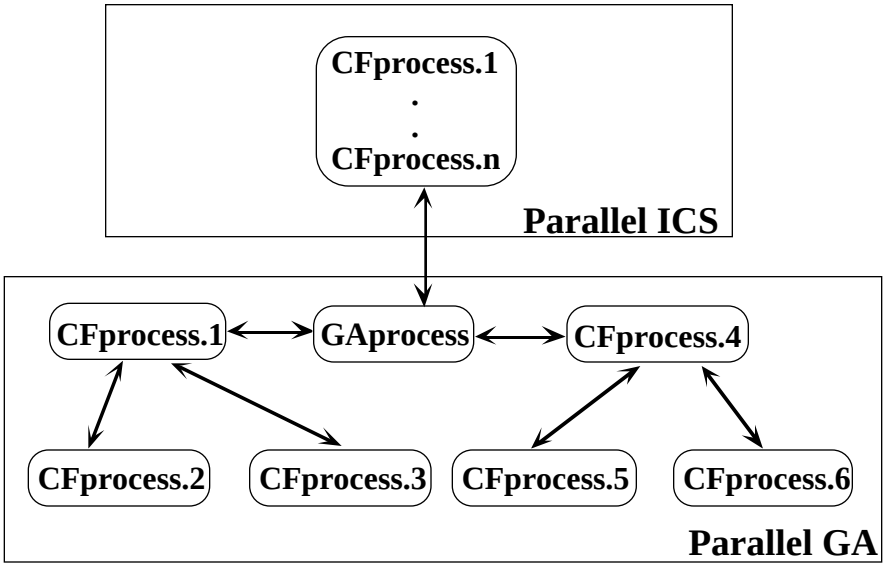


Figure 4. *The same type of architecture used for the credit apportionment and performance systems parallelization is employed to parallelize the genetic algorithm.*

The parallel GA is:

1 - Each CFprocess.*i* selects, within its own subset of CF, *m* rules to replicate and *m* to replace (note: *m* is a system parameter).

2 - Each CFprocess.*i* sends GAprocess some data about each selected classifier, enabling GAprocess to set up a hierarchical auction based on strength values; this process results in selecting 2*m* individuals within the overall CF population.

3 - GAprocess sends the following data to each CFprocess.*i* containing a *parent* classifier:

identifier of the *parent* itself

identifiers of the two *offspring*

CrossOver Point (COP)

at the same time GAprocess sends to the CFprocess.*i* containing a position for any *offspring* the following data:

identifier of the *offspring*

identifiers of the two *parents*

COP.

4 - All the CFprocesses that have *parents* in their own fraction of CF send a copy of the *parent* rule to the CFprocess.*i* that has the corresponding *offspring* position; this process will apply crossover and mutation operators and will overwrite rules to be replaced with newly generated rules.

3.2.2 High-level parallelism: A solution to behavioral complexity problems

In the preceding section we presented a method for parallelizing a single CS, with the goal of obtaining improvements in computing speed. This approach shows its weakness when a CS is applied to problems with multiple goals, which seems to be true of most real problems. We propose to assign the solution of the different goals to different CSs. This approach requires the introduction of a stronger form of parallelism, which we call high-level parallelism.

Moreover, scalability problems arise in a low-level parallelized CS; adding a node to the transputer network implementing the CS makes the communication load grow faster than the computational power, obtaining a less than linear speedup. Therefore, adding processors to an existing network is decreasingly effective. As already said, a better way to deal with complex problems could be to code them as a set of easier subproblems, each one committed to a CS. In ALECSYS the processor network is partitioned into subsets, each having its own size and topology. To each subset is allocated a single CS, which can be parallelized at a low level (see Fig.5).

Each of these CSs learns to solve a specific subgoal, according to the inputs it receives. Since ALECSYS is not provided with any automated way to come up with an optimal or even a good decomposition of a task into subtasks, this work is left to the learning system designer, who should try to identify independent basic tasks and coordination tasks and then assign them to different CSs. The design approach to task decomposition is common to most of the current research on autonomous system (see for example Lin (1993a; 1993b), and Mahadevan and Connell (1992)), but goes beyond the scope of this article (for a discussion of this issue see Dorigo and Colombetti, 1994, and Colombetti and Dorigo, 1994).

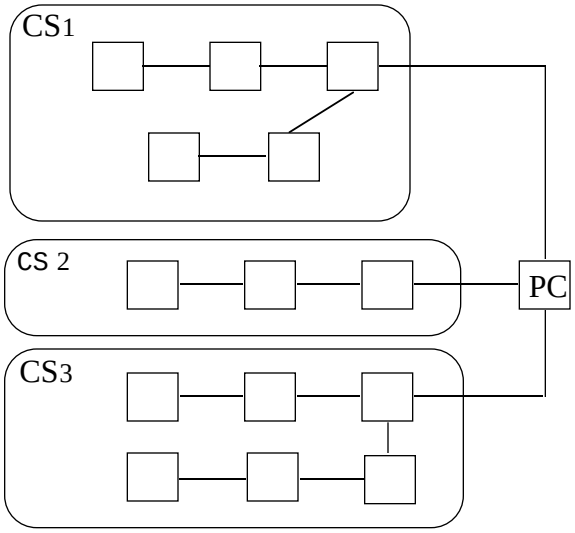


Figure 5. *Example of concurrent use of high-level and low-level parallelism: using ALECSYS the learning system designer divides a problem into three CSs (high-level parallelization) and maps each CS onto a subnet of transputer nodes (low-level parallelization).*

4. The experimental setting

In this work the role of simulation is quite different from other systems such as Grefenstette's SAMUEL system. Rather than using simulation to learn robot control programs, we use them to test the design of robot control systems capable of learning. Once we come up with a good system tested in simulated environments, we transfer the knowledge we, as researchers, have acquired to the design of learning systems which control real robots. The experiments described in this article concern a particular instance of the Animat problem (Wilson, 1987). As our goal is to develop an efficient learning system for real world experiments, as opposed to develop robot controllers in

simulation and then to use them with real robots, we run simulations in environments which are a very coarse model of real target environments.

The following sections describe the objects in the simulation environment, the target behaviors, the simulated AutonoMouse, the learning architectures built using ALECSYS, the details of the representation used, and finally the reinforcement program.

4.1 The simulation environment

Simulations were run placing the simulated AutonoMouse in a two dimensional environment where there were some objects that it could perceive. The objects are as follows.

- A moving light source. The light moves at a speed which is set to be equal to the maximum speed of the simulated AutonoMouse. The light moves along a straight line and bounces against walls (the initial position of the light is random).
- A predator which appears periodically and can only be heard.
- The AutonoMouse's lair. The lair occupies the upper right angle of the environment (see Fig.6).

4.2 Target behavior

The goal of the learning system is to have the simulated AutonoMouse learn the following behavior patterns.

- **Playing behavior**. The simulated AutonoMouse likes to follow the moving light source. (Imagine the light source to be something the simulated AutonoMouse likes to play with.)
- **Hiding behavior**. The simulated AutonoMouse occasionally hears the sound of a predator. Its main goal then becomes to reach the lair as soon as possible and to stay there until the predator goes away.
- **Global behavior**. A major problem for the learning system is not only to learn single behaviors, but also to learn to coordinate them. As we will see, this can be accomplished in different ways.

If the learning process is successful the resulting global behavior should be as follows.

> *The simulated AutonoMouse plays with (follows) the light source. When it happens to hear a predator[5], it suddenly gives up playing, runs to the lair and stays there until the predator goes away.*

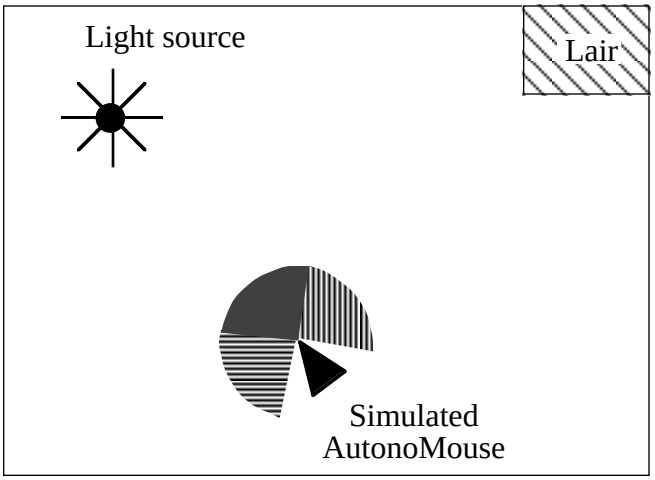


Figure 6. *Experimental setting (simulation): the simulated AutonoMouse sees and likes to play (that is, to follow) with a light source. It can also hear the noise of an approaching predator, in which case it runs to its lair.*

4.3 The simulated AutonoMouse

The simulated AutonoMouse has limited capabilities that should allow it to learn some simple behavior patterns.

Sensory capabilities of the AutonoMouse

The simulated AutonoMouse has two eyes. Each eye covers a visual angle of 180˚ and the two eyes have a 90˚ overlap in the AutonoMouse forward move direction. The eyes partition therefore the environment in four non overlapping regions (see Fig.6). AutonoMouse eyes can sense the position of the light source and of the lair (we say the AutonoMouse can *see* the light and the lair); they can also detect the difference between a close light and a distant light (the same is true for the

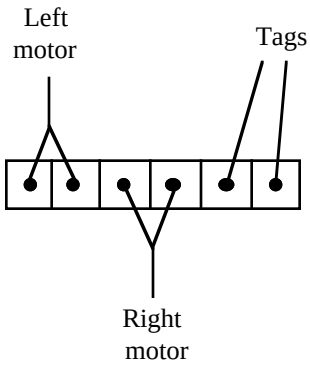
[5] In these simulations the AutonoMouse can hear, but cannot see, the predator. Therefore, to avoid conflict situations like if the predator is between the agent and the lair, we make the implicit assumption that the Animat can hear the predator when the predator is still very far away, so that it always has the time to reach the lair before the predator enters the computer monitor.

lair). The distinction between close and far was necessary because the AutonoMouse should learn two different responses within the hiding behavior: when the lair is far, approach it; when the lair is close (that is the AutonoMouse is into the lair) do not move. The distinction between *close* and *far* was not necessary for the playing behavior, but was maintained for uniformity. The AutonoMouse can also sense the presence of a predator (we say it can hear the predator), but cannot see it. As the sensory capabilities of the AutonoMouse make the perceived environment partitioned into four regions, with each region divided between a close region and a far region, three bits are enough to identify the light or the lair relative position. The AutonoMouse receives therefore seven bits of information from sensors: two bits to identify light position, two bits to identify lair position, one bit for light distance (close/far), one bit for lair distance (close/far), and one bit to signal the presence of the predator.

Motor capabilities of the AutonoMouse

The AutonoMouse has a right and a left motor and it can give the following movement commands to each of them: *stay still*, *move one step backward*, *move one step forward*, and *move two steps forward*. The maximum speed of the AutonoMouse, that is two steps per cycle, was set to be the same as the speed of the moving light (it was found that setting a higher speed, e.g. four steps per cycle, made the task much easier, while a lower speed made it impossible).

At every cycle, the AutonoMouse decides whether to move or not. As there are four possible actions for each motor, a move action can be described by four bits. The messages ALECSYS sends to the AutonoMouse motors are six bits long, the two additional bits being used to identify the messages as motor messages. The structure of a message sent to motors is shown in Fig.7.



The two bits going to each motor have the following meaning:

00 - one step backward,

01 - stay still,

10 - one step forward,

11 - two steps forward.

Figure 7. *Structure of a message going to motors.*

4.4 The learning architectures: Monolithic and Hierarchical

In the experiments presented in the next section a distinction is made between a *monolithic* architecture (Fig.8) and a *switch* (hierarchical) architecture (Fig.9). In the monolithic architecture the learning system is implemented as a single low-level parallel classifier system (on three transputers), called CS-global. In the switch architecture the learning system is implemented as a set of three classifier systems organized in a hierarchy (see Fig.9); two classifier systems learn the basic behaviors (playing and hiding), while one learns to switch between the two basic behaviors. To implement the switch architecture we took advantage of the high-level parallelism facility of ALECSYS; we used one transputer for the playing behavior, one transputer for the hiding behavior, and one transputer to learn to switch.

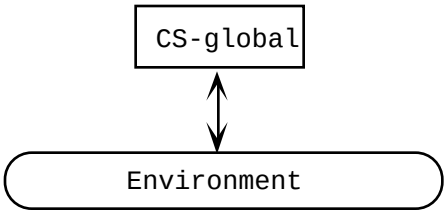


Figure 8. *The monolithic architecture. The problem is not decomposed; one single CS learns the global behavior.*

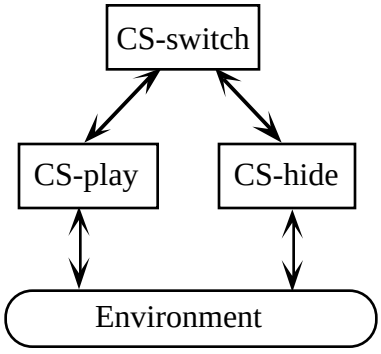


Figure 9. *The switch architecture. The problem is decomposed into three subproblems; a different CS learns to solve each subproblem.*

4.5 Representation

In ALECSYS classifiers are rules with two condition parts and one action part. Conditions are connected by an AND operator; a classifier enters the activation state if and only if it has both conditions matched by at least a message. Conditions are strings on $\{0, 1, \#\}^k$, and actions are strings on $\{0, 1\}^k$. The value of k is set to be the same as the length of longest between sensor and motor messages.

In our experiments the length of motor messages is always six bits, while the length of sensor messages depends on the type of architecture chosen.

When using the monolithic architecture, sensor messages are composed of nine bits, seven bits being sensory information and two bits for the tag, which identifies the message as being a sensor message[6]. (Therefore a classifier will be 3 x 9 = 27 bits long.)

In the case of the switch architecture sensory information is split to create two messages, a five bit message for the playing behavioral module (two bits for light position, one bit for light distance, and two bits as a tag), and a six bit message for the hiding behavioral module (two bits for light position, one bit for lair distance, one bit for the predator presence message, and two bits as a tag).

When using a switch architecture it is also necessary to define the format of the interface messages among basic CSs and CS-switch. At every cycle each basic CS sends, besides the message directed to motors if it is the case, a one-bit message to CS-switch. This one-bit message is intended to signal to CS-switch the intention to propose an action. A message composed by all the bits coming from basic CSs (a two bit message in our experiment: one bit from CS-play and one bit from CS-hide) goes as input to CS-switch. Hence CS-switch selects one of the basic CSs, which is then allowed to send its action to the AutonoMouse motors. The value and the meaning of the bit sent from the basic CSs to CS-switch is not predefined, but is learned by the system.

In Fig.10 is shown the format of a message from sensory input in the monolithic architecture; in Fig.11 the format of a message from sensory input to basic CSs in the switch architecture.

---

[6] This implies that three bits in the action part are useless, as actuator messages are 6-bit long.

Consider for example Fig.11b, which shows the format of the sensor message going to CS-hide. The message has six bits. Bits one and two are tags. They indicate whether the message comes from the environment or was generated by a classifier at the preceding time step (in which case they distinguish between a message that caused an action and one which did not). Bits three and four indicate lair position using the following encoding

00 - no lair,

01 - lair on the right,

10 - lair on the left,

11 - lair ahead.

Bit five indicates lair distance; it is set to one if the lair is far away and to zero if the lair is very close. Bit six is set to 1 if the AutonoMouse hears the sound produced by an approaching predator, to zero otherwise.

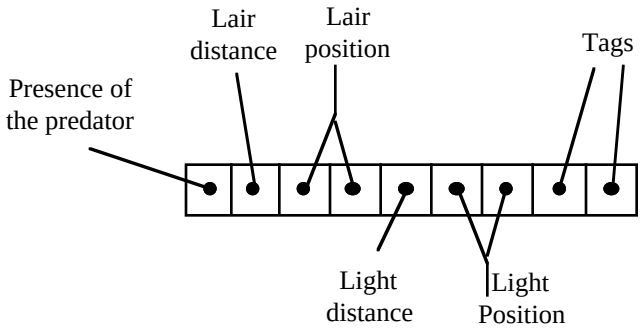


Figure 10. *Structure of a message from sensory input in the case of monolithic architecture.*

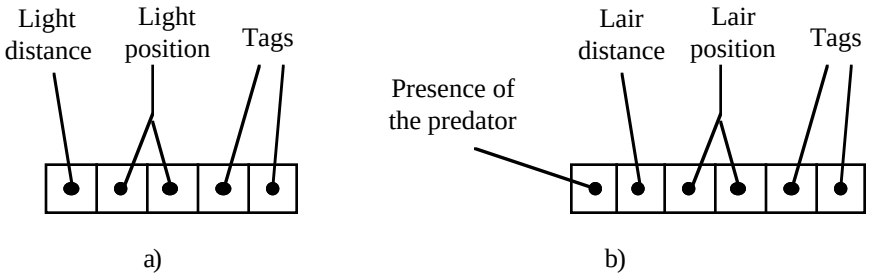


Figure 11. *Structure of a message from sensory input in the case of switch architecture.*
a) Message going to CS-play.
b) Message going to CS-hide.

4.6 The reinforcement program

In our experiments a trainer observes the learning agent actions and provides reinforcements after each action. In the experiments presented in Section 5 the trainer is implemented as a reinforcement program (RP). When implementing a RP a few interesting issues arise. The first issue we studied is whether the RP should provide only rewards (*only rewards policy*) or also punishments (*rewards and punishments policy*). An experiment in which only the playing behavior was considered, reported in Section 5.2, has shown that the use of punishments increases the learning performance. In the experiment the reinforcement program is called RP-play. RP-play gives the AutonoMouse a reward if it moves so that its distance from the light source does not increase. On the contrary, if the distance from the light source increases it punishes ALECSYS in case it uses the rewards and punishments policy, or it gives a null reinforcement if it uses the only rewards policy.

A second issue we studied is how the RP should be structured to make the best use of the architecture. We ran experiments to compare the monolithic and the switch architecture. In the monolithic architecture the RP is composed of the RP-play and the RP-hide procedures. RP-play is used to reinforce the AutonoMouse when the predator is not present, and is the same as in the previous experiment. RP-hide, which is used when the predator is present, rewards the AutonoMouse when it makes a move which diminishes the distance from the lair, if the lair is far; or when it stays still, if the lair is close. Although the RP for the switch architecture is the same as that for the monolithic architecture, there is a major difference in that a decision must be taken about how to shape the learning system. Experiments were run with the following two shaping policies.

- *Holistic shaping.* In holistic shaping the whole learning system is considered as a black box. The actual behavior of a single classifier systems is not used to evaluate how to distribute reinforcements; when the system receives a reinforcement, it is given to all of the three CSs comprising the switch architecture. This can make the learning task difficult because there can be ambiguous situations in which the trainer cannot give the correct reinforcement to the component CSs. Examples of ambiguous situations are: a correct action is the result of two wrong messages (e.g., CS-switch chooses to give control to the wrong basic CS, which in

turn proposes a normally wrong move that in that context happens to be correct) or a CS gets a punishment because of a mistake another CS made. Nevertheless, this method of distributing reinforcements is an interesting one because it does not require access to all the internal modules of the system and is much more plausible from an "ethological" point of view. (After all, you do not teach children to walk by rewarding single muscle contractions!).

- *Modular shaping.* In modular shaping first the basic CSs are trained, then they are frozen (that is, basic CSs are no longer learning, but only performing) and CS-switch is trained. Training of basic CSs is done in a separate session (it can be done in parallel): RP-play is used to train the CS-play, and RP-hide to train CS-hide. After the two basic CSs have reached a good performance level, they are frozen and CS-switch is turned on. CS-switch is reinforced by the complete RP, as in the monolithic architecture.

5. Learning complex interacting behaviors

The central issue investigated in this section is the role that the architectural choice and the reinforcement program play in making a learning system an efficient learner. All experiments were run with the simulated AutonoMouse and using ALECSYS.

In the first experiment, the issue was whether the use of punishments is a good thing or not; we also tested the optimal length for the message list, a parameter which can greatly influence the performance of the learning system. This experiment was run on the playing task, and only the low-level parallelism capacity of ALECSYS was used.

In the second set of experiments we explored some issues which arise when the system is required to learn more complex behaviors. In particular, we compared different architectural solutions and shaping policies. These experiments were run using as test-bed the global task: playing and hiding. The logical structure of the learning system consisted, in this case, of the two basic behaviors plus coordination of the basic behaviors. (Coordination can be made explicit, as in the switch architecture, or implicit, as in the monolithic architecture.) This logical structure, which can be mapped onto ALECSYS in a few different ways, was tested using both the monolithic

architecture and the switch architecture. In the case of the switch architecture we also compared holistic and modular shaping.

5.1 Experimental methodology

In all the experiments in the simulated worlds, we used the following performance measure: $P = \frac{\text{Number of rewarded moves}}{\text{Total number of moves}} \leq 1$. Each experiment was composed of a learning phase and of a test session. The test session was run after the end of the learning phase; during the test session the learning algorithm was switched off. The index P, measured considering only the moves done in the test session, was used to compute the performance achieved by the system after learning. (For basic behaviors P is computed only for the moves in which they were active; instead, we compute the global performance as the ratio of globally correct moves to total moves during the whole test session, where at every cycle a globally correct move is a move correct with respect to the current goal.)

We report tables with averages and standard deviations for each experiment (each experiment was repeated twenty times). When the performances of different architectural or shaping policies were compared we executed the Kruskal-Wallis (non parametric ANOVA) test. When this test indicated a significant difference, the performances were then pairwise compared through Mann-Whitney tests.

5.2 The role of punishments and of internal messages

In this experiment only the two light sensors were active; the structure of messages coming from the sensors was the one reported in Fig.11a. Messages going to motors had the format of Fig.7.

We tried to give an answer to these questions:

- Is it better to use only positive reward or is punishment also useful?
- Are internal messages useful?

Regarding the first point, results have shown, see Table 1, that the use of rewards and punishments is advantageous over using only rewards for any number of internal messages used.

Mann-Whitney tests showed that the differences observed between the first and the second row of Table 1 are highly significant for every message list (ML) length tested.

Results have also shown that, for the given task, a one-message ML is the best. Learning curves for the rewards and punishments policy are shown in Fig.12, and results obtained in test session are reported in Table 1. As shown in Table 2, the differences in average performance when changing the ML length are significant (except when moving from ML=2 to ML=4), and significativity tends to increase with the difference in the number of internal messages used. (The Kruskal-Wallis test was also performed and resulted to be highly significant, $p < 10^{-4}$). This result is what we expected, given that the considered behavior is a stimulus-response one, which does not require the use of internal messages. On the contrary, it is easy to understand that internal messages can slow down the convergence of the classifiers strength to good values.

Table 1. *Comparisons between the use of the only rewards training policy and the use of the rewards and punishments training policy. Results obtained in the test session (1000 cycles run after 1000 cycles of learning). Averages on twenty runs.*

		ML=1	*ML=2*	*ML=4*	*ML=8*
Rewards and punishments	Average	0.904	0.883	0.883	0.851
	Std.Dev.	0.021	0.023	0.026	0.026
Only rewards	Average	0.744	0.729	0.726	0.682
	Std.Dev.	0.022	0.028	0.027	0.025

Table 2. *Significance levels obtained using the Mann-Whitney test on the data of Table 1 regarding rewards and punishments.*

	ML=1	*ML=2*	*ML=4*
ML=8	$< 10^{-5}$	$< 10^{-3}$	$< 10^{-3}$
ML=4	0.017	0.465	
ML=2	0.003		

It is also interesting to note that, using rewards and punishments and ML=1, the system achieves a good performance level (about 0.8) after only about 200 cycles (see Fig.12). (100

cycles done in about 60 seconds with 300 rules on 3 transputers.) This makes it feasible to use ALECSYS to control the real robot in real-time.

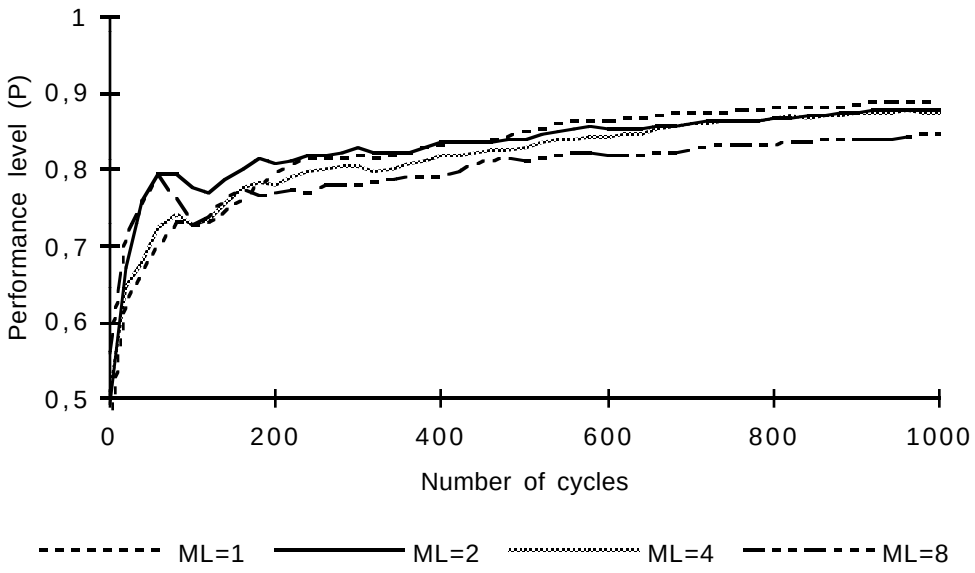


Figure 12. *Learning to follow a light source (the playing behavior). A comparison of different ML lengths, using the rewards and punishments training policy. Averages on twenty runs.*

5.3 Choice of an architecture and of a shaping policy

The number of computing cycles a learning classifier system requires to learn to solve a given task is a function of the length of its rules, which in turn depends on the complexity of the task. A straightforward way to reduce this complexity, which was used in the experiments reported in the following, is to split the task into many simpler learning tasks. Whenever this can be done, the learning system performance should improve. This is what is tested by the experiments presented in this section, where we compare the monolithic and the switch architectures, which were described in Section 4.4. Also, as discussed in Section 4.6, the use of a distributed behavioral architecture calls for the choice of a shaping policy; we ran experiments with both holistic and modular shaping. Given the results of the previous experiment, we use rewards and punishments and a short message list (ML=1). Experiments were designed as follows.

- Experiment A. We compared the monolithic architecture and the switch architecture with holistic shaping (holistic-switch hereafter). Differences in performance between these two architectures, if any, are due only to the different architectural organization. Each experiment was run for 15000 learning cycles, and was followed by a 1000 cycles test session.
- Experiment B. We ran two experiments using the switch architecture with modular shaping (modular-switch). In experiment B1 we gave the modular-switch architecture roughly the same amount of computing resources as in experiment A, so to allow a fair comparison. The experiment was organized as follows: 5000 cycles to train the playing behavioral module, 5000 cycles to train the hiding behavioral module, and 5000 cycles to train the behavior coordination module (for ease of reference we call the architecture of this experiment modular-switch-long). It is important to note that, although the number of learning cycles is the same as in experiment A, the actual time (in seconds) required is shorter as the two basic behaviors can be trained in parallel. Each learning phase was followed by a 1000 cycles test session.

  In the second experiment (B2) we tried to find out what was the minimal amount of resources to give to the modular-switch architecture to let it reach the same performance level as the best performing of the two architectures used in experiment A. In this experiment we ran 2000 cycles to train the playing behavioral module, 2000 cycles to train the hiding behavioral module, and 2000 cycles to train the behavior coordination module; we will refer to this architecture as to the modular-switch-short. Again, each learning phase was followed by a 1000 cycles test session.

Results regarding the test session are reported in Table 3. The best performing architecture was the modular-switch. It was both the best performing given approximately the same amount of resources during learning, and was able to achieve the same performance of the holistic-switch architecture using much less computing time. These results can be explained by the fact that in the switch architecture each single CS, having shorter classifiers, has a smaller search space[7];

[7] Rules in CS-play are 15 bits long, in CS-hide 18 bits long, and in CS-switch they are 6 bits long.

therefore, the overall learning task is easier. The worst performing resulted to be the monolithic architecture. These results are consistent with previously obtained results in similar, although different, Animat environments (see Dorigo and Colombetti, 1994). The significance of the differences in mean performances across architectures was tested using the Kruskal-Wallis test, which resulted to be highly significant ($p < 10^{-4}$). Table 4, 5, and 6 report the significance levels of the Mann-Whitney tests, which also resulted to be highly significant for all the pairs of architectures compared, except for the comparison modular-switch-short versus the holistic-switch, as the first was designed to have approximately the same final performance as the second.

Table 3. *Comparison across architectures during the test session. Averages computed on twenty runs.*

	Monolithic Architecture		Holistic-switch Architecture		Modular-switch Architect. (short)		Modular-switch Architecture (long)	
	Avg.	Std.Dev.	Avg.	Std.Dev.	Avg.	Std.Dev.	Avg.	Std.Dev.
Playing	0.830	0.169	0.941	0.053	0.942	0.034	0.980	0.015
Hiding	0.744	0.228	0.919	0.085	0.920	0.077	0.978	0.022
Global	0.784	0.121	0.933	0.073	0.931	0.085	0.984	0.022

Table 4. *Mann-Whitney tests on the playing behavioral module. Comparison across architectures.*

	Monol. Arch.	Hol.-sw. Arch.	Mod.-sw. Arch.(short)
Mod.-switch Arch. (long)	$<10^{-6}$	$<10^{-5}$	$<10^{-5}$
Mod.-switch Arch. (short)	$<10^{-3}$	0.850	
Hol.-switch Architecture	$<10^{-3}$		

Table 5. *Mann-Whitney tests on the hiding behavioral module. Comparison across architectures.*

	Mon. Arch.	Hol.-sw. Arch.	Mod.-sw. Arch.(short)
Mod.-switch Arch. (long)	$<10^{-6}$	$<10^{-5}$	$<10^{-3}$
Mod.-switch Arch. (short)	$<10^{-4}$	0.787	
Hol.-switch Architecture	$<10^{-4}$		

Table 6. *Mann-Whitney tests on the global behavior. Comparison across architectures.*

	Monol. Arch.	Hol.-sw. Arch.	Mod.-sw. Arch.(short)
Mod.-switch Arch. (long)	$<10^{-7}$	$<10^{-5}$	$<10^{-4}$
Mod.-switch Arch. (short)	$<10^{-5}$	0.245	
Hol.-switch Architecture	$<10^{-4}$		

As it can be observed in Table 3, the performance of the hiding behavioral module tended to be slightly lower than that of the playing behavioral module in all the architectures. Although this could be easily explained with the fact that the search space for the hiding behavioral module is slightly bigger than in the case of the playing behavioral module (messages are one bit longer, due to the presence of the extra predator bit), the Mann-Whitney test has shown that for all the architectures the difference is not significant.

6. Experiments with the real AutonoMouse

In this section we deal with some issues which arise when moving from simulated to real robots. Often the role of simulation is that of building a coarse control system for a simulated robot, to be refined by learning on the real robot. This is not the approach taken in this article, where simulation is considered to be a useful tool to speed up the designer's understanding of the learning processes (given that simulation experiments are much less time consuming than real robot experiments). Once a working learning system is obtained in simulation, the experience gained by the designer can be transferred to the design of a learning system for the real robot. Still, when moving from simulated to real robots new issues arise. Sensor noise can appear, motors can work imprecisely, and in general there can be differences of any sort between the designed and the real function of the robot. These problems can be studied from many different points of view. In this article we focus on the role of the trainer (reinforcement program); that is, we study how different trainers can influence the robustness to noise and to lesions in the real AutonoMouse.

Results obtained with simulations presented in the preceding section suggested using rewards and punishments (we experimentally found that a good value for reinforcements was: rewards=+10, punishments=-15) to train the AutonoMouse, setting the message list length to ML=1, and using 300 classifiers on a three-transputer configuration. In all the experiments presented in this section we used the monolithic architecture.

6.1 AutonoMouse hardware

Fig.13 shows a functional schematic of the AutonoMouse, while Fig.14 presents a photograph of the real robot. Its sensory capabilities are provided by four directional eyes, and one microphone, as shown in Fig.13. The AutonoMouse moves by activating two motors, one for the left wheel and one for the right. The available commands for engines are *move one step backward*, *stay still*, *move one step forward*, *move two steps forward*. From combinations of these basic commands arise forms of movements like *advance*, *retreat*, *rotate*, *rotate while advancing* and so on (there are 16 different composite movements). The directional eyes sense light source when it is in either (or both) eyes' field of view. In order to change this field of view the AutonoMouse must rotate as the eyes cannot do so independently; they are fixed with respect to the robot. The sensor value of each eye is either on or off, depending on the relation of the light intensity to a threshold.

The AutonoMouse is also provided with two eyes that can sense light within a halfspace, which are used solely by the trainer using the reward-the-result policy. These central eyes evaluate the absolute increase or decrease in light intensity (they discriminate between 256 levels).

The format of input messages is given in Fig.15. In the experiments reported in this article (Section 6.4) the AutonoMouse can see the light only with the two frontal eyes (the two rear eyes and microphone were used in other experiments not reported here).

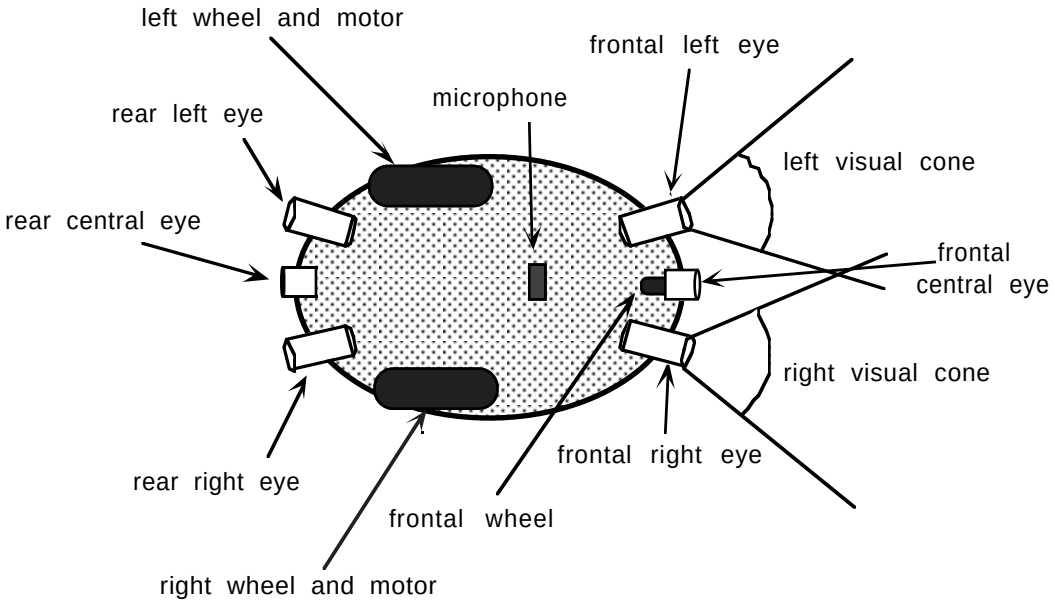


Figure 13. *A functional schematic of the AutonoMouse.*

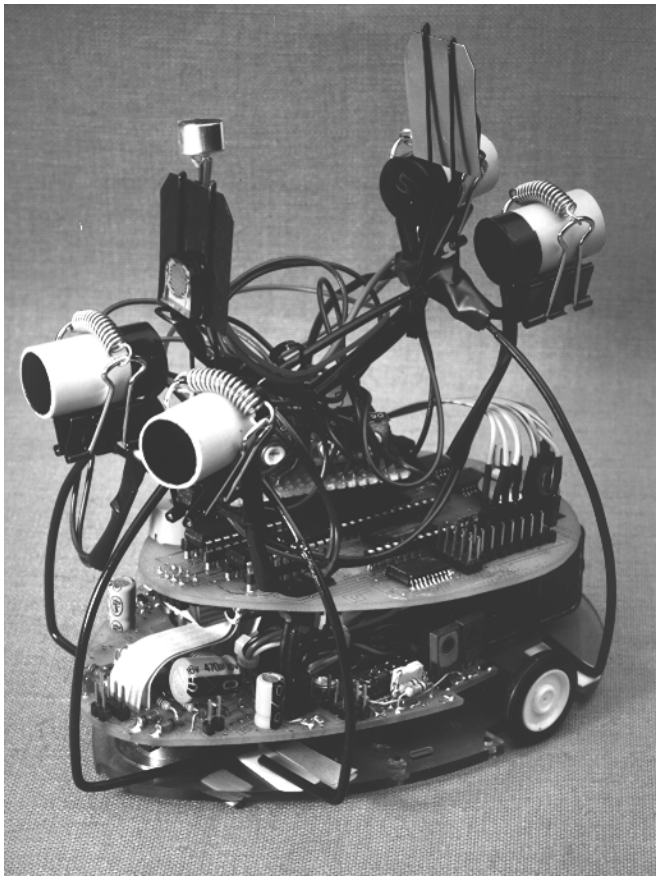


Figure 14. *The AutonoMouse.*

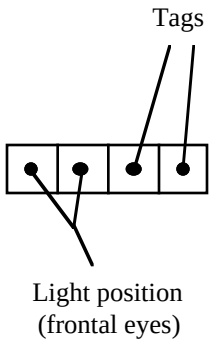


Figure 15. *Format of sensor messages in the real AutonoMouse. The AutonoMouse uses only the two frontal eyes.*

6.2 Experimental methodology

In the real world, experiments were run until either the goal was achieved or the experimenter was convinced that the robot was not going to achieve the goal (at least in a reasonable time).

Experiments with the real robots were repeated only occasionally, as they are highly time consuming. Experiments which were repeated showed that the differences between different runs were marginal.

The performance index used is the light intensity perceived by the frontal central eye (which increases with proximity, up to a maximum level of 256). To study the relation between the reinforcement program and the actual performance, we plot the average reward over intervals of 20 cycles.

6.3 The training policies

As we have seen, given that the environment used in the simulations was designed to be as close as possible to the real environment, messages going from AutonoMouse's sensors to ALECSYS and from ALECSYS to AutonoMouse's motors have a very similar structure to that of the simulated AutonoMouse. On the other hand, the real robot differs from simulation in that sensor input and motor output are liable to be inaccurate[8]. This is a major point in machine learning research, since simulated environments can be but an approximation of real ones.

---

[8] We note that, although not investigated in this paper, some of our experiments exhibited a certain amount of reward noise, due to the use of sensors to evaluate performance, in addition to the two kinds of noise mentioned before.

To study the effect that using real sensors and motors has on the learning process, and its relation to the training procedure, we introduced two different training policies, called *reward-the-intention* policy, and *reward-the-result* policy (see Fig.16).

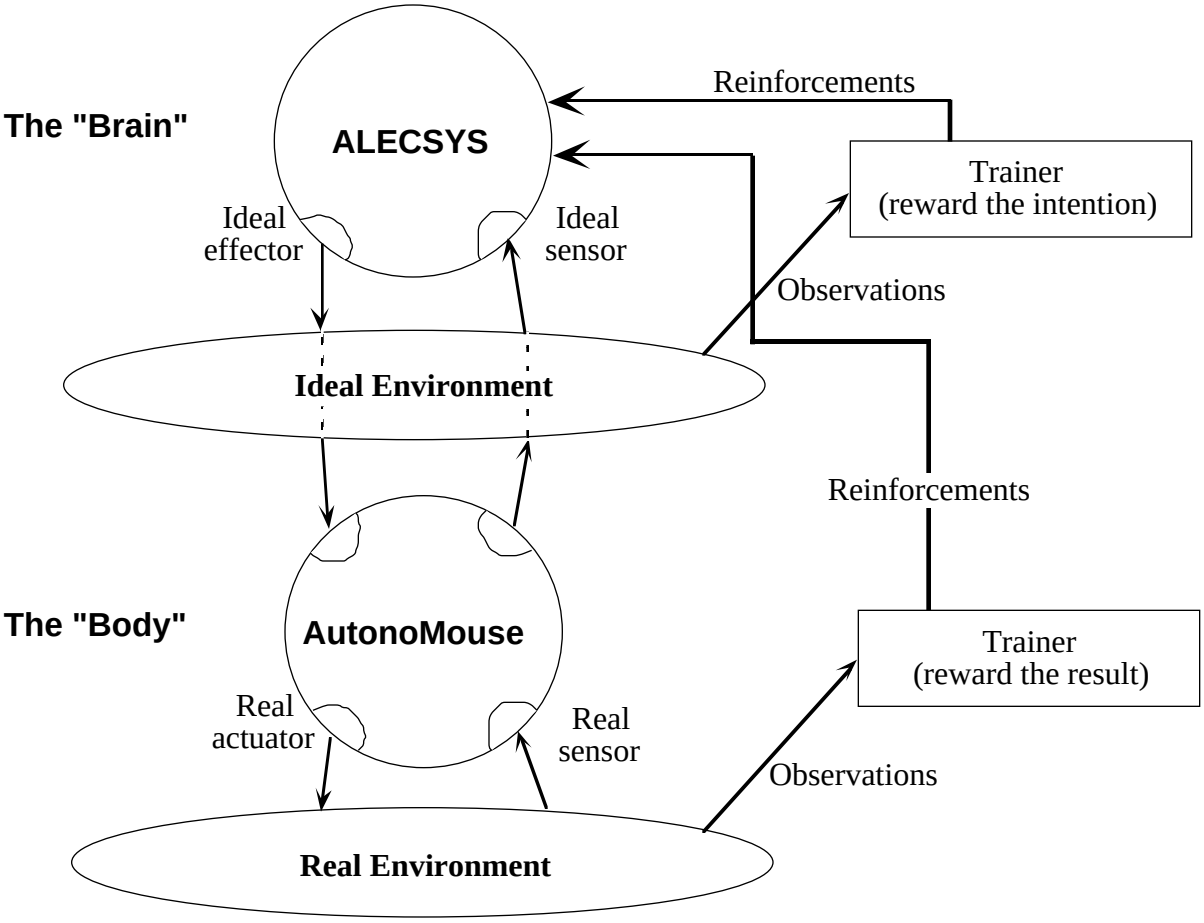


Figure 16. *The reward-the-intention and reward-the-result trainers.*

The reward-the-intention policy

We say that a trainer uses the reward-the-intention policy if, in order to decide about the reinforcement to give, it uses observations from an *ideal environment* positioned between ALECSYS and the AutonoMouse. This environment is said to be ideal because there is no interference with the real world (it is the same type of environment ALECSYS senses in simulations). The reward-the-intention trainer rewards ALECSYS if it proposes a move that is correct with regard to the input-output mapping he wants to teach (i.e., the reinforcement is

computed observing the responses in the ideal, simulated world). A reward-the-intention trainer knows the desired input-output mapping and rewards the learning system if it learns that mapping, regardless of the resulting actions in the real world.

The reward-the-result policy

We say that a trainer uses the reward-the-result policy if, in order to decide about the reinforcement to give, it uses observations from the real world. The reward-the-result trainer rewards ALECSYS if it proposes an action that diminishes the distance from the goal. (In the example the rewarded action is a move that causes the AutonoMouse to approach the light source.) A reward-the-result trainer has knowledge about the goal, and rewards ALECSYS if the actual move is consistent with the achievement of that goal.

In the case of reward-the-intention, the observed behavior can be the desired one only if there is a correct mapping between ALECSYS's interpretation of sensor (and/or motor) messages and the real world significance of AutonoMouse's sensory input (and/or motor commands)[9]. Consider the extreme case in which the two AutonoMouse's eyes are inverted (i.e., ALECSYS believes the right eye is the left one and *vice versa*). A reward-the-intention trainer will reward ALECSYS for actions that an external observer would judge as wrong; in the example of inverted eyes, while the reinforcement program intends to reward the "turn towards the light" behavior, the learned behavior will be a "turn away from the light" one. In case of a reward-the-result trainer, the mapping problem disappears; ALECSYS learns any mapping between sensor messages and motor messages that maximizes the reward received by the trainer.

In the following of this section we present the results of some experiments that had the objective of testing ALECSYS's capability to learn under the two different reinforcement policies. We also introduced the following handicaps to test ALECSYS's adaptive capabilities: inverted eyes, blindness of one eye, and incorrect calibration of motors. All graphs presented are relative to a single experiment.

---

[9] If this condition holds, we say that ALECSYS and the AutonoMouse are well calibrated.

6.4 An experimental study of training policies

In this section we experimentally compare the reward-the-intention and the reward-the-result training policies. In particular we investigate the influence that the choice of a training policy has on noisy sensors and motors, and on a set of lesions[10] we on purpose inflicted to the AutonoMouse to study the robustness of learning. The experiment is set in a room in which a lamp has been arbitrarily positioned. The AutonoMouse is allowed to wander freely. The AutonoMouse senses the environment using only its two frontal directional eyes. ALECSYS receives a high reward whenever, in the case of reward-the-result policy, the light intensity perceived by the frontal central sensor increases, or, in the case of reward-the-intention policy, the proposed move is the correct one with respect to the received sensory input. Sometimes, especially in the early phases of the learning process, the AutonoMouse loses contact with the light source (i.e., it ceases to see the lamp). In these cases, ALECSYS receives a reward if the AutonoMouse turns twice in the same direction. This favors sequences of turns in the same direction, assuring that the AutonoMouse will, sooner or later, see the light again. In the following, we report and comment on the results of experiments with the standard AutonoMouse and with AutonoMice using noisy or faulty sensors or motors. For all of the experiments, ALECSYS was initialized with a set of randomly generated classifiers.

Noisy sensors and motors

In Fig.17, we compare the performance obtained with the two different reinforcement policies using well calibrated sensory or motor devices. Note that the performance is better for the reward-the-result policy. Falls in performance at cycle 230 for the reward-the-intention and at cycle 250 for the reward-the-result policies are due to sudden changes in the light position caused by the experimenter moving the lamp far away from the AutonoMouse. Fig.18 shows the rewards received by ALECSYS for the same two runs. Strangely enough, the average reward is higher (and optimal after 110 cycles) for the reward-the-intention policy. This indicates that, given perfect

[10] In this paper we call lesions those malfunctionings which make the behavior of a sensor, or of a motor, sistematically different from the design specifications. This is different from noise, which in our experiments could be modeled as a Gaussian distribution around a mean behavior which meets the designer specifications.

information, ALECSYS learns to do the right thing. In the case of reward-the-result however, ALECSYS does not always get the expected reward. This is due to the fact that, despite our efforts to build good, noise free, sensors and motors, and to calibrate them appropriately, we did not completely succeed.

In Fig.19 light intensity and average reward are directly compared for the reward-the-result policy. (In the graph average reward is multiplied by 25 to ease visual comparison.) Observe that the two graphs are strongly correlated; when the AutonoMouse's performance starts to increase, so does the reward received by ALECSYS. Drops in rewards reflect, with a delay due to the time required to move away from the light, drops in performance.

These results confirm our expectation, as discussed in Section 6.3, that a trainer using the reward-the-intention policy, in order to be a good trainer, needs a very accurate, low-level, knowledge of the input-output mapping it is teaching and of how to compute this mapping, in order to be able to give the correct reinforcements. Often this is a non-reasonable requirement, as this accurate knowledge could be directly used to design the behavioral modules, making learning useless. On the other hand, a reward-the-result trainer only needs to be able to evaluate the moves of the learning system with respect to the behavioral pattern that it is teaching.

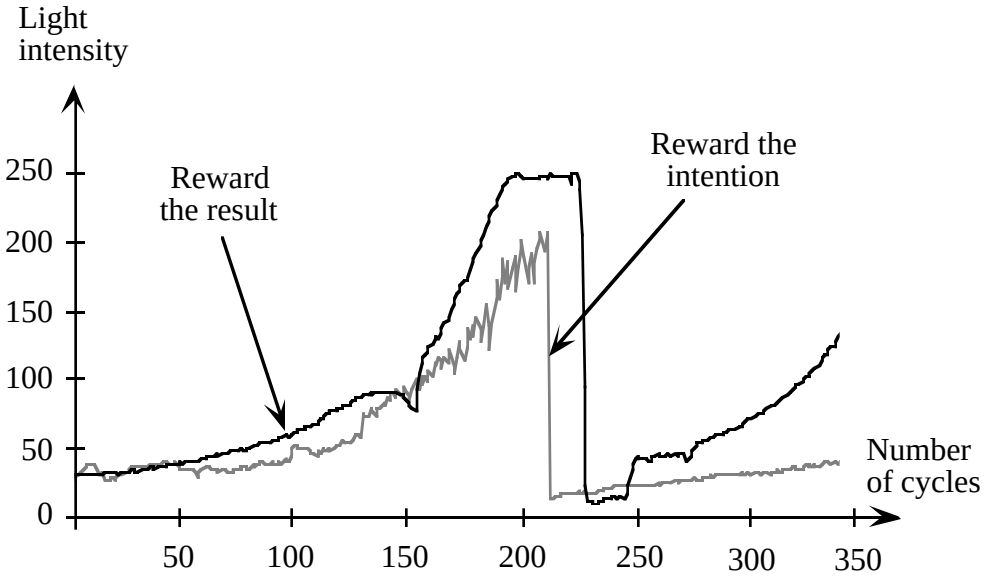


Figure 17. *Difference in performance between reward-the-result and reward-the-intention training policies.*

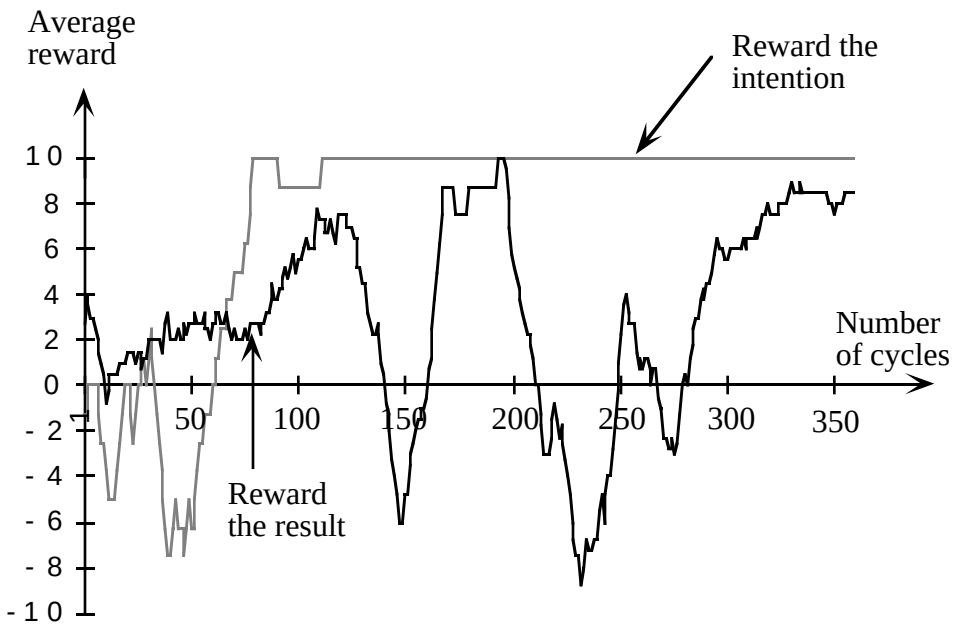


Figure 18. *Difference in average reward between reward-the-result and reward-the-intention training policies.*

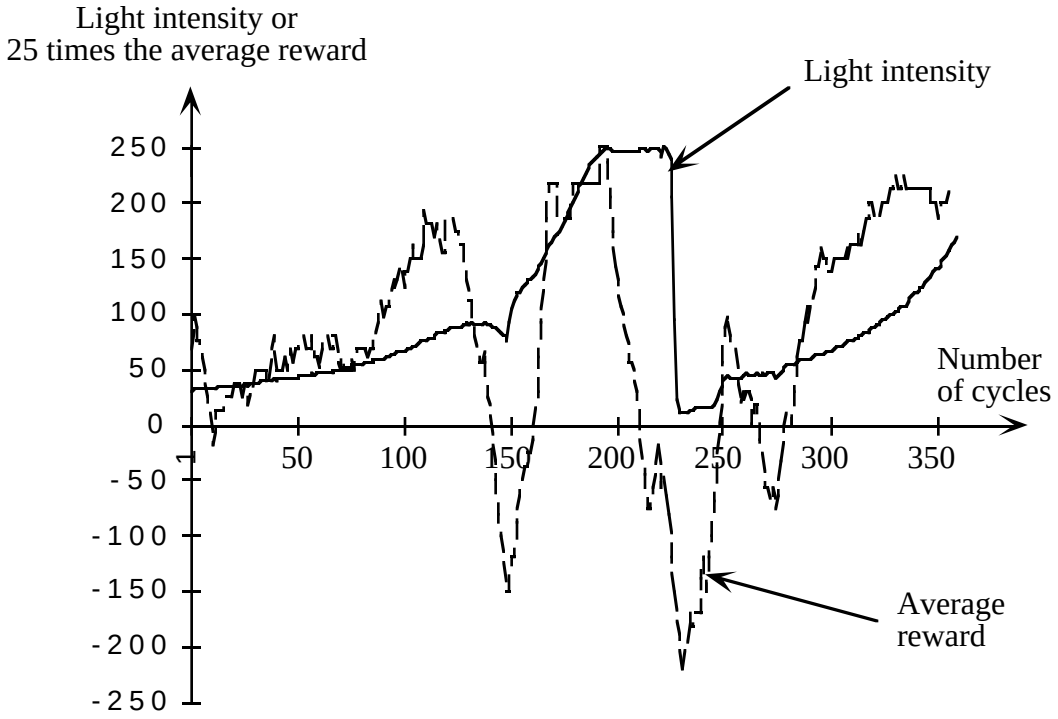


Figure 19. *System performance (light intensity) and average reward (multiplied by 25 to ease visual comparison) for the reward-the-result training policy.*

Lesions study

Lesions differ from noise in that they cause a sensor (or a motor) to systematically deviate from its design specifications. In this section we study the robustness of our system for three kinds of lesions: inverted eyes, one blind eye, and incorrectly regulated motors. Experiments have shown that for each type of lesion the reward-the-result training policy was able to teach the target behavior, while the reward-the-intention training policy was not.

Results of the experiment in which we inverted the two frontal eyes of the AutonoMouse are shown in Figs.20 and 21 (these graphs are analogous to those regarding the standard eye configuration of Figs.17 and 18). While graphs obtained for the reward-the-result training policy are qualitatively comparable, graphs obtained for the reward-the-intention training policy differ greatly (see Figs.17 and 20). As discussed before (see Section 6.3), with inverted eyes the reward-the-intention policy fails (Note that in this experiment the light source was moved after 150 cycles.)

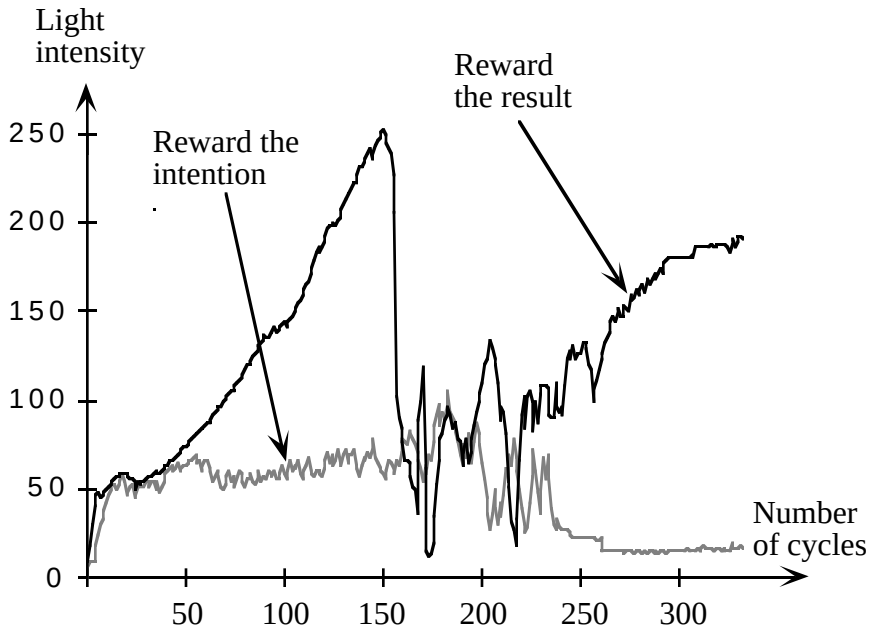


Figure 20. *Difference in performance between reward-the-result and reward-the-intention training policies in the case of an AutonoMouse with inverted eyes.*

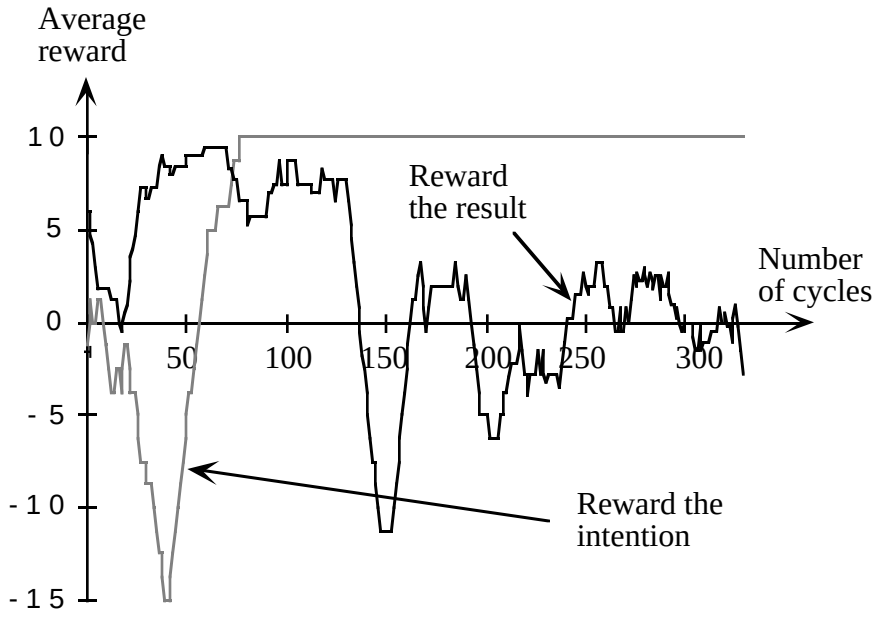


Figure 21. *Difference in average reward between reward-the-result and reward-the-intention training policies in the case of an AutonoMouse with inverted eyes.*

Similar qualitative results were obtained using only one of the frontal directional eyes (one blind eye experiment). Fig.22 shows that the reward-the-intention policy is not capable of letting the AutonoMouse learn to approach the light source. In this experiment the light was never moved because the task was already hard enough. The reward-the-result policy, however, allows the AutonoMouse to approach the light, although it requires more cycles than with two working eyes. (At cycle 135 there is a drop in performance where the AutonoMouse lost sight of the light and turned right until it saw the light source again.)

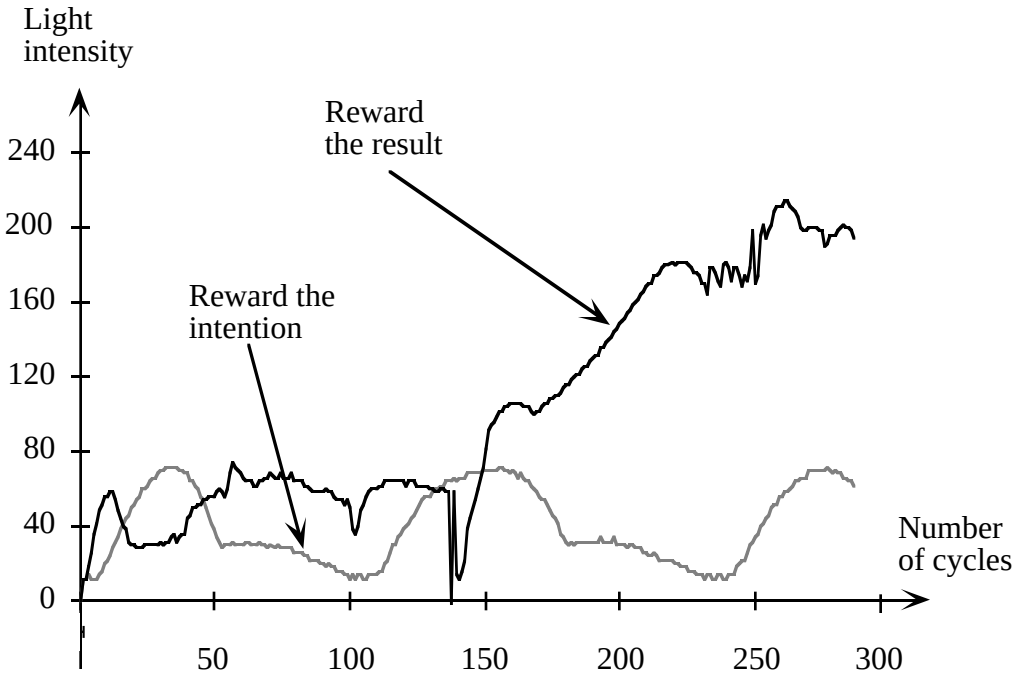


Figure 22. *Difference in performance between reward-the-result and reward-the-intention training policies in the case of one blind eye.*

As a final experiment the AutonoMouse was given badly regulated motors. In this experiment bits going to each motor had the following new meaning:

00 - stay still,
01 - one step forward,
10 - two steps forward,
11 - four steps forward.

The net effect was an AutonoMouse that could not move backward any more, and that on the average moved much more quickly than before. The result was that it was much easier to lose contact with the light source, as happened with the reward-the-intention policy after the light was moved (in this experiment the light source was moved after 90 cycles); but, as the average speed of the AutonoMouse was higher, it took fewer cycles to reach the light. Fig.23, the result of a typical experiment, shows that also in this case the reward-the-result training policy was better than the reward-the-intention training policy.

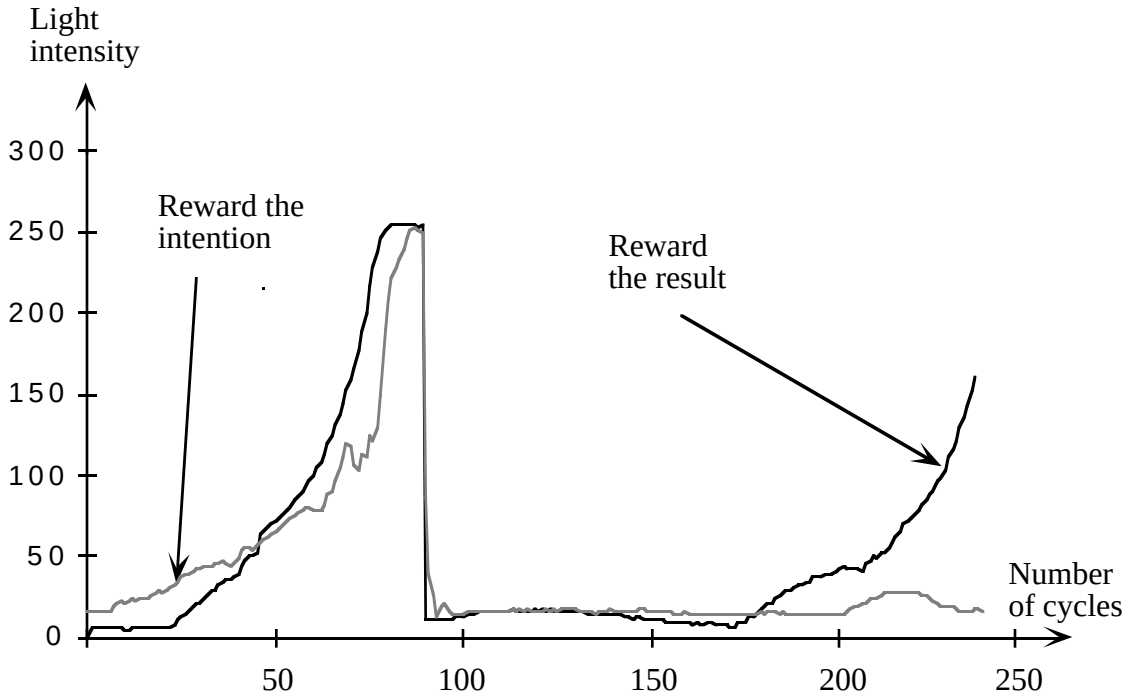


Figure 23. *Difference in performance between reward-the-result and reward-the-intention training policies in the case of incorrect regulation of motors.*

7. Conclusions

Attempts to use learning classifier systems for real robot control applications raise a number of problems, the most prominent of which is that they are in general too slow to allow the on-line learning of even simple behaviors. We attack this problem along three main directions: (i) increase of the speed with which a CS learns to solve a given task, (ii) reduction of the learning complexity, and (iii) the use of a trainer.

Point (i) is addressed by a variety of improvements of Holland's CS and by means of parallelization. These are discussed at length in Section 3 of this article. The resulting learning system, called ALECSYS, beside being quicker than the original Holland's CS, also provides a flexible way to define many cooperating modules, each module being a parallel CS.

To reduce learning complexity we design a distributed architecture in which each module is a basic behavior or a coordination behavior. Modules are CSs and the use of ALECSYS allows to run them in parallel. We experimentally show that the use of a distributed, hierarchical architecture can help to control the complexity of learning.

The use of a trainer brings in same new issues. First we experimentally show that, in our Animat application, the use of negative reinforcements to punish wrong actions speeds up learning. Then we study how to distribute reinforcements to the different modules which comprise a distributed architecture. We propose and experimentally compare two policies called holistic and modular shaping. Experimental results show that modular shaping, when feasible, performs better than holistic shaping. To use modular shaping the internal components (CSs in our approach) of the learning system must be accessible.

Finally, we study the influence that different kinds of data used by the trainer to judge the learning system behavior have on the learning process in case of real robot experiments. A result of our research is that, in order for the robot to be robust to sensor or motor noise, or to major changes (lesions) in the characteristics of its sensors or motors, the trainer must observe the behavior of the real robot, as opposed to the intended behavior of the learning system. In fact, the use of the reward-the-result policy resulted in the development of a robust robot, while the reward-the-intention policy was much less effective. This result underlines again the importance of using real robots to develop really adaptive and robust systems.

It is interesting to note that to analyze the data obtained with the real robot we had to film the experiments. This allowed us to study and compare the AutonoMouse behavior in different situations and to compare the observed behavior with the corresponding data plots. This activity is not part of the traditional computer science methodology used to evaluate software systems and further research will be necessary to understand all of his implications (first steps in this direction can be found in Colombetti, 1994).

In conclusion, our study shows that learning classifier systems are a feasible tool to build robust robot control systems. To achieve this goal we found very helpful to decompose the desired overall robotic behavior into a set of simpler interacting behaviors organized in a hierarchy. These interacting behaviors were implemented as a set of CSs using a distributed architecture in which each CS runs on a different set of processors. This choice, together with the use of a trainer providing immediate reinforcements, was sufficient to achieve adequate levels of performance for a variety of behaviors.

Acknowledgments

We thank Lisa Desjarlais for the detailed comments on a earlier version of this article. Thanks are also due to Andrea Bonarini, Marco Colombetti and Vittorio Maniezzo for their help in revising this article, and to Graziano Ravizza and the Logic Brainstorm for help in the design and realization of the AutonoMouse used in some of the experiments presented in this article. Finally, a special thank to the three anonymous reviewers and to Ken De Jong for providing so many and helpful comments.

References

Barto, A.G., Sutton, R.S., and Anderson, C.W. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics, 13,* 834–846.

Booker, L.B. (1988). Classifier systems that learn internal world models. *Machine Learning, 3,* 3, 161–192.

Booker, L.B., Goldberg, D.E., and Holland, J.H. (1989). Classifier systems and genetic algorithms. *Artificial Intelligence, 40,* 2, 235–282.

Brooks, R.A. (1990). Elephants don't play chess. *Robotics and Autonomous Systems, 6,* 1–2, 3–16.

Brooks, R.A. (1991a). Intelligence without representation. *Artificial Intelligence, 47,* 1-3, 139–159.

Brooks, R.A. (1991b). Artificial life and real robots. *Proceedings of the First European Conference on Artificial Life,* Paris, MIT Press/Bradford Books, 3–10.

Camilli, A., Di Meglio, R., Baiardi, F., Vanneschi, M., Montanari, D., and Serra, R., (1990). Classifier system parallelization on MIMD architectures. Tech. Rep. 3/17 CNR.

Colombetti, M. (1994). Adaptive agents: Steps to an ethology of the artificial. In F. Masulli, P. Morasso and A. Schenone (Eds.), *Neural Networks in Biomedicince,* World Scientific, in press.

Colombetti, M. and Dorigo, M. (1994). Training agents to perform sequential behavior. *Adaptive Behavior, 2,* 3, 247–275, MIT Press.

Compiani, M., Montanari, D., Serra, R. and Valastro, G. (1989). Classifier systems and neural networks. In E. R. Caianiello (Ed.), *Parallel architectures and neural networks,* World Scientific.

Dorigo, M. (1992). Using transputer to increase speed and flexibility of genetics-based machine learning systems. North Holland, *Microprocessing and Microprogramming, Euromicro Journal,* 34, 147–152.

Dorigo, M. (1993). Genetic and non genetic operators in ALECSYS. *Evolutionary Computation, 1,* 2, 151–164, MIT Press.

Dorigo, M. and Bersini, H. (1994). A Comparison of Q-Learning and Classifier Systems. *Proceedings of From Animals to Animats, Third International Conference on Simulation of Adaptive Behavior (SAB94),* Brighton, UK, MIT Press, 248–255.

Dorigo, M. and Colombetti, M. (1994). Robot shaping: Developing autonomous agents through learning. *Artificial Intelligence, 71,* 2.

Dorigo, M. and Schnepf, U. (1993). Genetics-based machine learning and behavior based robotics: A new synthesis. *IEEE Transactions on Systems, Man, and Cybernetics, 23,* 1, 141–154.

Dorigo, M. and Sirtori, E. (1991a). A learning environment for robots. *Proceedings of GAA91 - Second Italian Workshop on Machine Learning,* Bari - Italy.

Dorigo, M. and Sirtori, E. (1991b). ALECSYS: A parallel laboratory for learning classifier systems. *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Diego - CA, Morgan Kaufmann, UCSD, 296–302.

Flynn, M.J. (1972). Some computer organizations and their effectiveness. *IEEE Transaction on Computers C-21, 9*, 948–960.

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

Grefenstette, J.J. (1991). Lamarckian learning in multi-agent environments. *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Diego - CA, Morgan Kaufmann, UCSD, 303–310.

Holland, J.H. (1975). *Adaptation in natural and artificial systems*, Ann Arbor: The University of Michigan Press.

Holland, J.H. (1980). Adaptive algorithms for discovering and using general patterns in growing knowledge bases. *International Journal of Policy Analysis and Information Systems, 4*, 2, 217-240.

Holland, J.H. and Reitman, J.S. (1978). *Cognitive systems bases on adaptive algorithms*, Academic Press.

Lin, L-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning, 8*, 3-4, 293–322.

Lin, L-J. (1993a). Hierarchical learning of robot skills by reinforcement. *Proceedings of the 1993 IEEE International Conference on Neural Networks*, IEEE, 181–186.

Lin, L-J. (1993b). Scaling up reinforcement learning for robot control. *Proceedings of the Tenth International Conference on Machine Learning*, Morgan Kaufmann, 182–189.

Mahadevan, S. and Connell, J. (1992). Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence, 55*, 2, 311–365.

Riolo, R.L. (1989). The emergence of default hierarchies in learning classifier systems. *Proceedings of the Third International Conference on Genetic Algorithms*, J.D. Schaffer (Ed.), Morgan Kaufmann, 322–327.

Robertson, G.G. (1987). Parallel implementation of genetic algorithms in a classifier system. *Proceedings of the Second International Conference on Genetic Algorithms*, Lawrence Erlbaum, MIT - Cambridge - MA, 140–147.

Sutton, R.S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. *Proceedings of the Seventh International Conference on Machine Learning*, Morgan Kaufmann, Palo Alto, CA, 216–224.

Watkins, C.J.C.H., (1989). *Learning with delayed rewards*. Ph.D. dissertation, Psychology Department, University of Cambridge, England.

Williams, R.J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning, 8*, 3-4, 229–256.

Wilson, S. (1987). Classifier systems and the Animat problem. *Machine Learning, 2*, 3, 199–228.

Zhou, H.H. (1990). CSM: A computational model of cumulative learning. *Machine Learning, 5,* 4, 383–406.