Robot Shaping

An Experiment in Behavior Engineering

Marco Dorigo and Marco Colombetti

Intelligent Robotics and Autonomous Agents
Ronald C. Arkin, editor

Behavior-Based Robotics
Ronald C. Arkin, 1998
Robot Shaping: An Experiment in Behavior Engineering
Marco Dorigo and Marco Colombetti, 1998

ROBOT SHAPING

AN EXPERIMENT IN BEHAVIOR ENGINEERING

Marco Dorigo and Marco Colombetti

A Bradford Book
The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Times New Roman on the Monotype "Prism Plus" PostScript Imagesetter by Asco Trade Typesetting Ltd., Hong Kong.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Dorigo, Marco.
Robot shaping : an experiment in behavior engineering / Marco Dorigo and Marco Colombetti.
p. cm.
"A Bradford book."
Includes bibliographical references and index.
ISBN 0-262-04164-2
1. Robots—Control systems. 2. Machine learning. 3. Robots—Motion. I. Colombetti, Marco. II. Title.
TJ211.35.D67 1998
629.8'92—dc21 97-16342
CIP

To Emanuela, Laura, and Luca

Contents

Foreword

As robots are used to perform increasingly difficult tasks in complex and unstructured environments, it becomes more of a challenge to reliably program their behavior. There are many sources of variation for a robot control program to contend with, both in the environment and in the performance characteristics of the robot hardware and sensors. Because these are often too poorly understood or too complex to be adequately managed with static behaviors hand-coded by a programmer, robotic systems sometimes have undesirable limitations in their robustness, efficiency, or competence. Machine learning techniques offer an attractive and innovative way to overcome these limitations. In principle, machine learning techniques can allow a robot to successfully adapt its behavior in response to changing circumstances without the intervention of a human programmer.

The exciting potential of robot learning has only recently been investigated in a handful of groundbreaking experiments. Considerably more work must be done to progress beyond simple research prototypes to techniques that have practical significance. In particular, it is clear that robot learning will not be a viable practical option for engineers until systematic procedures are available for designing, developing, and testing robotic systems that use machine learning techniques. This book is an important first step toward that goal.

Marco Dorigo and Marco Colombetti are two of the pioneers making significant contributions to a growing body of research in robot learning. Both authors have years of hands-on experience developing control programs for a variety of real robots and simulated artificial agents. They have also made significant technical contributions in the subfield of machine learning known are "reinforcement learning". Their considerable knowledge about both robotics and machine learning, together

with their sharp engineering instincts for finding reliable and practical solutions to problems, has produced several successful implementations of learning in robots. In the process, the authors have developed valuable insights about the methodology behind their success. Their insights, ideas, and experiences are summarized clearly and comprehensively in the chapters that follow.

Robot Shaping: An Experiment in Behavior Engineering provides useful guidance for anyone interested in understanding how to specify, develop, and test a robotic system that uses machine learning techniques. The behavior analysis and training methodology advocated by the authors is a well-conceived procedure for robot development based on three key ideas: analysis of behavior into modular components; integration of machine learning considerations into the basic robot design; and specification of a training strategy involving step-by-step reinforcement (or "shaping"). While more work remains to be done in areas like behavior assessment, the elements of the framework described here provide a solid foundation for others to use and build on. Additionally, the authors have tackled some of the open technical issues related to reinforcement learning. The family of techniques they use, referred to as "classifier systems", typically requires considerable expertise and experience to implement properly. The authors have devised some pragmatic ways to make these techniques both easier to use and more reliable.

Overall, this book is a significant addition to the robotics and machine learning literature. It is the first attempt to specify an orderly "behavior engineering" procedure for learning robots. One of the goals the authors set for themselves was to help engineers develop high-quality robotic systems. I think this book will be instrumental in providing the guidance engineers need to build such systems.

Lashon B. Booker
The MITRE Corporation

Preface

This book is about designing and building learning autonomous robots. An autonomous robot is a remarkable example of a device that is difficult to design and program because it must carry out its task in an environment at least partially unknown and unpredictable, with which it interacts through noisy sensors and actuators. For example, a robot might have to move in an environment with unknown topology, trying to avoid still objects, people walking around, and so forth; to sense its surroundings, it might have to rely on sonars and dead reckoning, which are notoriously inaccurate. Moreover, the actual effect of its actions might be very difficult to describe a priori due to the complex interactions of the robot's effectors with its physical environment.

There is a fairly widespread opinion that a good way of solving such problems would be to endow autonomous robots with the ability to acquire knowledge from experience, that is, from direct interaction with their environments. This book explores the construction of a learning robot that acquires knowledge through reinforcement learning, with reinforcements provided by a trainer that observes the robot and evaluates its performance.

One of our goals is to clarify what the two terms in our title, *robot shaping* and *behavior engineering*, mean and how they relate to our main research aim: designing and building learning autonomous robots. *Robot shaping* uses learning to translate suggestions from an external trainer into an effective control strategy that allows a robot to achieve a goal. We borrowed "shaping" from experimental psychology because training an artificial robot closely resembles what experimental psychologists do when they train an experimental subject to produce a predefined response. Our approach differs from most current research on learning autonomous robots in one important respect: the trainer plays a fundamental role in

the robot learning process. Most of this book is aimed at showing how to use a trainer to develop control systems for simulated and real robots.

We propose the term *behavior engineering* to characterize a new technological discipline, whose objective is to provide techniques, methodologies, and tools for developing autonomous robots. In chapter 7 we describe one such behavior engineering methodology—"behavior analysis and training" or BAT. Although discussed only at the end of the volume, the BAT methodology permeates all our research on behavior engineering and, as will be clear from reading this book, is a direct offspring of our robot-shaping approach.

The book is divided into eight chapters. Chapter 1 serves as an introduction to the research presented in chapters 2 through 7 and focuses on the interplay between learning agents, the environments in which the learning agents live, and the trainers with which they interact during learning.

Chapter 2 provides background on ALECSYS, the learning tool we have used in our experimental work. We first introduce LCS_0, a learning classifier system strongly inspired by Holland's seminal work, and then show how we improved this system by adding new functionalities and by parallelizing it on a transputer.

Chapter 3 discusses architectural choices and shaping policies as a function of the structure of behavior in building a successful learning system. Chapter 4 presents the results of experiments carried out in many different simulated environments, while chapter 5 presents the practical results of our robot-shaping approach in developing the control system of a real robot.

Extending our model to nonreactive tasks, in particular to sequential behavior patterns, chapter 6 shows that giving the learning system a very simple internal state allows it to learn a dynamic behavior; chapter 7 describes BAT, a structured methodology for the development of learning autonomous systems that learn through interaction with a trainer.

Finally, chapter 8 discusses related work, draws some conclusions, and gives hints about directions the research presented in this book may take in the near future.

Acknowledgments

Our work was carried out at the Artificial Intelligence and Robotics Project of Politecnico di Milano (PM-AI&R Project). We are grateful to all members of the project for their support, and in particular, to Marco Somalvico, the project's founder and director; to Andrea Bonarini, who in addition to research and teaching was responsible for most of the local organization work; and to Giuseppe Borghi, Vincenzo Caglioti, Giuseppina Gini, and Domenico Sorrenti, for their invaluable work in the robotics lab.

The PM-AI&R Project has been an ideal environment for our research. Besides enjoying many stimulating discussions with our colleagues, we made fullest use of the laboratory facilities for our experiments on real robots. Giuseppe Borghi had an active role in running the HAMSTER and CRAB experiments presented in chapter 7; for the CRAB experiment, we could rely on the cooperation of Mukesh Patel, who spent a year at the PM-AI&R Project on an ERCIM Research Fellowship funded by the EC Human Capital and Mobility Programme. Emanuela Prato Previde, of the Institute of Psychology, Faculty of Medicine, Università di Milano, discussed with us several conceptual issues connected with experimental psychology.

Our work has received the financial support of a number of Italian and European institutions. We gratefully acknowledge a "60%" grant from MURST (Italian Ministry for University and Scientific and Technological Research) to Marco Colombetti for the years 1992–1994; a "University Fund" grant from Politecnico di Milano to Marco Colombetti for the year 1995; and an Individual EC Human Capital and Mobility Programme Fellowship to Marco Dorigo for the years 1994–1996.

Many students have contributed to our research by developing the software, building some of the robots, and running the experiments. We

wish to thank them all. In particular, Enrico Sirtori, Stefano Michi, Roberto Pellagatti, Roberto Piroddi, and Rino Rusconi contributed to the development of ALECSYS and to the experiments presented in chapters 4 and 5. Sergio Barbesta, Jacopo Finocchi, and Maurizio Gorla contributed to the experiments presented in chapter 6; Franco Dorigo and Andrea Maesani to the experiments presented in sections 5.3 and 7.3; Enrico Radice to the experiments presented in section 7.4; Massimo Papetti to the experiments presented in section 7.5. AutonoMouse II was designed and built by Graziano Ravizza of Logic Brainstorm; AutonoMouse IV and AutonoMouse V were designed and built by Franco Dorigo.

Marco Dorigo is grateful to all his colleagues at the IRIDIA lab of Université Libre de Bruxelles for providing a relaxed yet highly stimulating working environment. In particular, he wishes to thank Philippe Smets and Hugues Bersini for inviting him to join their group, and Hugues Bersini, Gianluca Bontempi, Henri Darquenne, Christine Decaestecker, Christine Defrise, Gianni Di Caro, Vittorio Gorrini, Robert Kennes, Bruno Marchal, Philip Miller, Marco Saerens, Alessandro Saffiotti, Tristan Salomé, Philippe Smets, Thierry Van de Merckt, and Hong Xu for their part in the many interesting discussions he and they had on robotics, learning, and other aspects of life.

Some of the ideas presented in this book originated while Marco Dorigo was a graduate student at Politecnico di Milano. Many people helped make that period enjoyable and fruitful. Marco Dorigo thanks Alberto Colorni for his advice and support in writing a good dissertation, Francesco Maffioli and Nello Scarabottolo for their thoughtful comments on combinatorial optimization and on parallel architectures, and Stefano Crespi Reghizzi for his work and enthusiasm as coordinator of the graduate study program.

Finally, a big thank-you to our families, and to our wives, Emanuela and Laura, in particular. This book is dedicated to them, and to Luca, the newborn son of Laura and Marco Dorigo.

Chapter 1
Shaping Robots

1.1 INTRODUCTION

This book is about our research on the development of learning robotic agents at the Artificial Intelligence and Robotics Project of the Politecnico di Milano. Our long-term research goal is to develop autonomous robotic agents capable of complex behavior in a natural environment. The results reported in the book can only be viewed as a first step in this direction—although, we hope, a significant one.

Although the potential impact of advances in autonomous agents on robotics and automation speaks for itself, the development of autonomous robots is a particularly demanding task, involving highly sophisticated mechanics, data collection and processing, control, and the like. Indeed, we believe there is need for a unified discipline of agent development, which we call "behavior engineering"; we hope this book will provide a contribution in this direction.

The term *behavior engineering* is reminiscent of similar, well-established terms like *software engineering* and *knowledge engineering*. By introducing a new term, we want to stress the specificity of the problems involved in the development of autonomous robots, and the close relationship between the development of artificial agents and the natural sciences' study of the behavior of organisms. Although the relevance of the notion of behavior in robotics has been explicitly recognized (see, for example, Brooks 1991; Arkin 1990; Maes 1990, 1994; Maes and Brooks 1990; Steels 1990, 1994; Dorigo and Schnepf 1991, 1993; Saffiotti, Ruspini, and Konolige 1993; and Saffiotti, Konolige, and Ruspini 1995), we believe that the links between robotics and behavioral sciences have not yet been fully appreciated.

It can be said that the mechanisms of learning, and the complex balance between what is learned and what is genetically determined, are the main concern of behavioral sciences. In our opinion, a similar situation arises in autonomous robotics, where a major problem is in deciding what should be explicitly designed and what should be left for the robot to learn from experience. Because it may be very difficult, if not impossible, for a human designer to incorporate enough world knowledge into an agent from the very beginning, we believe that machine learning techniques will play a central role in the development of robotic agents. Agents can reach a high performance level only if they are capable of extracting useful information from their "experience," that is, from the history of their interaction with the environment. However, the development of interesting behavior patterns is not merely a matter of agent self-organization. Robotic agents will also have to perform tasks that are useful to us. Agents' behavior will thus have to be properly "shaped" so that a predefined task is carried out. Borrowing a term from experimental psychology (Skinner 1938), we call such a process "robot shaping". In this book we advocate an approach to robot shaping based on reinforcement learning.

In addition to satisfying intellectual curiosity about the power of machine learning algorithms to generate interesting behavior patterns, autonomous robots are an ideal test bed for several machine learning techniques. Many journals have recently dedicated special issues to the application of machine learning techniques to the control of robots (Gaussier 1995; Kröse 1995; Dorigo 1996; Franklin, Mitchell and Thrun 1996). Moreover, the use of machine learning methods opens up a new range of possibilities for the implementation of robot controllers. Architectures as neural networks and learning classifier systems, for example, cannot be practically programmed by hand—if we want to use them, we have to rely on learning. And there are good reasons why we might want to use them: such architectures have properties like robustness, reliability, and the ability to generalize that can greatly contribute to the overall quality of the agent.

Even when learning is not strictly necessary, it is often suggested that an appropriate use of learning methods might reduce the production cost of robots. In our opinion, this statement has yet to be backed with sufficient evidence; indeed, nontrivial learning systems are not easy to develop and use, and the computational complexity of learning can be very high.

On the other hand, it seems to us that the reduction of production costs should not be sought too obsessively. Recent developments in applied computer science, particulary in software engineering (see, for example, Sommerville 1989; Ghezzi, Jazayeri, and Mandrioli 1991) have shown that a reduction of the global cost of a product, computed on its whole life cycle, cannot be obtained by reducing the costs of the development phase. On the contrary, low global costs demand high product quality, which can only be achieved through a more complex and expensive development process. The use of learning techniques should be viewed as a means to achieve higher behavioral quality for an autonomous agent; the usefulness of the learning approach should therefore be evaluated in a wider context, taking into account all relevant facets of agent quality. We believe this to be a strong argument in favor of behavior engineering, that is, an integrated treatment of all aspects of agent development, including design and the use of learning techniques.

1.2 LEARNING TO BEHAVE

The term *learning* is far too generic; we therefore need to delimit the kind of learning models we have in mind. From the point of view of a robot developer, an important difference between different learning models can be found in the kind and complexity of input information specifically required to guide the learning process. In these terms, the two extreme approaches are given by supervised learning and by self-organizing systems. While the latter do not use any specific feedback information to implement a learning process, the former require the definition of a training set, that is, a set of labeled input-output pairs. Unfortunately, in a realistic application the construction of a training set can be a very demanding task, and much effort can be wasted in labeling sensory configurations that are very unlikely to occur. On the other hand, self-organization has a limited range of application. It can only be used to discover structural patterns already existing in the environment; it cannot be used to learn arbitrary behaviors.

In terms of information requirements, an intermediate case is represented by *reinforcement learning* (RL). RL can be seen as a class of problems in which an agent learns by trial and error to optimize a function (often a discounted sum) of a scalar called "reinforcement" (Kaelbling, Littman, and Moore 1996). Therefore, algorithms for solving RL problems do not require that a complete training set be specified: they only

need a scalar evaluation of behavior (that is, a way to compute reinforcement) in order to guide learning. Such an evaluation can be either produced by the agent itself as the result of its interaction with the environment or provided by a *trainer*, that is, an external observer capable of judging how well the agent approximates the desired behavior (Dorigo and Colombetti 1994a, 1994b; Dorigo 1995).

In principle, of course, it would be preferable to do without an external trainer: an artificial system able to learn an appropriate behavior by itself is the dream of any developer. However, the use of a strictly internal reinforcement procedure poses severe limitations. A robot can evaluate its own behavior only when certain specific events take place, thus signaling that a desirable or undesirable state has been reached. For example, a positive reinforcement, or *reward*, can be produced once the robot's sensors detect that a goal destination has been reached, and a negative reinforcement, or *punishment*, can be generated once an obstacle has been hit. Such *delayed reinforcements* have then to be used to evaluate the suitability of the robot's past course of action. While this process is theoretically possible, it tends to be unacceptably inefficient: feedback information is often too rare and episodic for an effective learning process to take place in realistic robotic applications.

A way to bypass this problem is to use a trainer to continuously monitor the robot's behavior and provide *immediate reinforcements*. To produce an immediate reinforcement, the trainer must be able to judge how well each single robot move fits into the desired behavior pattern. For example, the trainer might have to evaluate whether a specific movement brings the robot closer to its goal destination or keeps the robot far enough from an obstacle.

Using a human trainer would amount to a direct translation of the high-level, often implicit knowledge a human being has about problem solving into a robotic control program, avoiding the problems raised by the need to make all the intermediate steps explicit in the design of the robotic controller. Unforeseen problems could arise, however. Human evaluations could be too noisy, inaccurate, or inconsistent to allow the learning algorithm to converge in reasonable time. Also, the reaction time of humans is orders of magnitude slower than that of *simulated* robots, which could significantly increase the time required to achieve good performance when training can be carried out in simulation. On the other hand, this problem will probably not emerge with real robots, whose reaction time is much slower due to the dynamics of their mechanical parts.

More research is necessary to understand whether the human trainer approach is a viable one. For the time being, we limit our attention to robots that learn by exploiting the reinforcements provided by a *reinforcement program* (RP), that is, by an artificial trainer implemented as a computer program.

The use of an RP might appear as a bad idea from the point of view of our initial goals. After all, we wanted to free the designer of the need to completely specify the robot's task. If the designer has to implement a detailed RP, why should this be preferable to directly programming the robot's controller?

One reason is of an experimental nature. Making our robots learn by using an RP is a first, and simpler, step in the direction of using human-beings: a success with RPs opens up the door to further research in using human-generated reinforcements. But even if we limit our attention to RPs, there are good motivations to the RP approach, as opposed to directly programming the robot controller. Writing an RP can be an easier task than programming a controller because the information needed in the former case is more abstract. To understand why it is so, let us restrict our attention to *reactive agents*, that is, to agents whose actions are a function solely of the current sensory input.[1] The task of a reactive controller is to produce a control signal associated with the signals coming from sensors; the control message is sent to the robot effectors, which in turn produce a physical effect by interacting with the environment. Therefore, to define a controller, the designer must know the exact format and meaning of the messages produced by the sensors and required by the effectors. Moreover, the designer has to rely on the assumption that in all situations the physical sensors and effectors will be consistently correct.

Suppose, for example, that we want a mobile robot with two independent wheels to move straight forward. The designer will then specify that both wheels must turn at the same angular velocity. But suppose that for some reason one wheel happens to be somewhat smaller than the other. In this situation, for the robot to move straight on, the two wheels must turn at different velocities. Defining a reactive controller for a *real* robot (not an *ideal* one) thus appears to be a demanding task: it requires that the designer know all the details of the real robot, which often differ slightly from the ideal robot that served as a model.

Now consider the definition of an RP to train the robot to move straight forward. In this case, the designer might specify a reinforcement related to the direction in which the agent is moving, so that the highest

evaluation is produced when such a direction is straight ahead. There is no need to worry about smaller wheels and similar problems: the system will learn to take the control action, whichever it may be, that produces the desired result by taking into account the actual interaction between the robot and its environment. As often happens, the same problem could be solved in a different way. For example, the designer might design a feedback control system that makes the robot move straight forward even with unequal wheels. However, this implies that the designer is aware of this specific problem in order to implement a specific solution. By contrast, RL solves the same problem as a side effect of the learning process and does not require the designer even to be aware of it.

1.3 SHAPING AN AGENT'S BEHAVIOR

As we have pointed out in the previous section, we view learning mechanisms as a way of providing the agent with the capacity to be trained. Indeed, we have found that effective training is an important issue per se, both conceptually and practically.

First, let us analyze the relationship between the RP and the final controller built by the agent through learning. A way of directly programming a controller, without resorting to learning, would be to describe the robot's behavior in some high-level task-oriented language. In such a language, we would like to be able to say things like "Reach position P" or "Grasp object O." However, similar statements cannot be directly translated into the controller's machine code unless the translator can rely on a *world model*, which represents both general knowledge about the physical world and specific knowledge about the particular environment the robot acts in (figure 1.1). Therefore, world modeling is a central issue for task-based robot programming.

Unfortunately, everybody agrees that world modeling is not easy, and some people even doubt that it is possible. For example, the difficulty of world modeling has motivated Brooks to radically reject the use of symbolic representations, with the argument that "the world is its own best model" (Brooks 1990, 13). Of course, this statement should not be taken too literally (a town may be its own best map, but we do find smaller paper maps useful after all). The point is that much can be achieved *without* a world model, or at least without a human-designed world model.

While there is, in our opinion, no a priori reason why symbolic models should be rejected out of hand, there is a problem with the models that

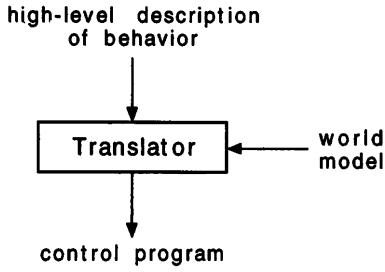


Figure 1.1
Translating high-level description of behavior into control program

human designers can provide. Such models are biased for at least two reasons: (1) they tend to reflect the worldview determined by the human sensory system; and (2) they inherit the structure of linguistic descriptions used to formulate them off-line. An autonomous agent, on the other hand, should build its own worldview, based on its sensorimotor apparatus and "cognitive abilities" (which might not include anything resembling a linguistic ability).

One possible solution is to use a machine learning system. The idea is to let the agent organize its own behavior starting from its own "experience," that is, the history of its interaction with the environment. Again, we start from a high-level description of the desired behavior, from which we build the RP. In other words, the RP is an implicit procedural representation of the desired behavior. The learning process can now be seen as the process of translating such a representation into a low-level control program—a process, however, that does not rely on a world model and that is carried out through continuous interaction between agent and environment (figure 1.2). In a sense, then, behavioral learning with a trainer is a form of *grounded translation* of a high-level specification of behavior into a control program.

If we are interested in training an agent to perform a complex behavior, we should look for the most effective way of doing so. In our experience, a good choice is to impose some degree of modularity on the training process. Instead of trying to teach the complete behavior at once, it is better to split the global behavior into components, to train each component separately, and then to train the agent to correctly coordinate the component behaviors. This procedure is reminiscent of the shaping procedure that psychologists use in their laboratories when they train an animal to produce a predefined response.

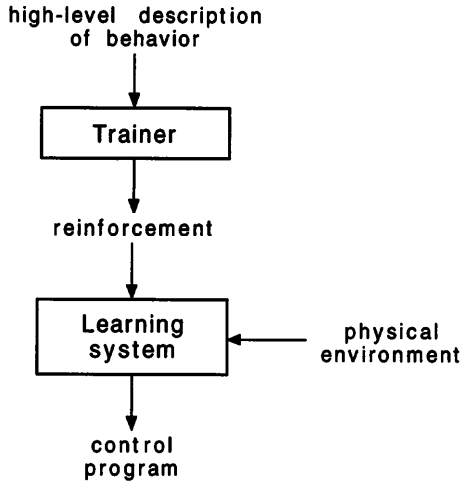


Figure 1.2
Learning as grounded translation

1.4 REINFORCEMENT LEARNING, EVOLUTIONARY COMPUTATION, AND LEARNING CLASSIFIER SYSTEMS

As we said, reinforcement learning can be defined as the problem faced by an agent that learns by trial and error how to act in a given environment, receiving as the only source of learning information a scalar feedback known as "reinforcement".

Recently there has been much research on algorithms that lend themselves to solving this type of problem efficiently. Typically, the result of the agent learning activity is a *policy*, that is, a mapping from states to actions that maximizes some function of the reward intake (where *rewards* are positive reinforcements, while *punishments* are negative reinforcements).

The study of algorithms that find optimal policies with respect to some objective function is the object of *optimal control*. Techniques to solve RL problems are part of "adaptive optimal control" (Sutton, Barto, and Williams 1991) and can take inspiration from existing literature (see Kaelbling, Littman, and Moore 1996). Indeed, an entire class of algorithms used to solve RL problems modeled as Markov decision processes (Bertsekas 1987; Puterman 1994) took inspiration from dynamic programming (Bellman 1957; Ross 1983). To these techniques belong, among others, the "adaptive heuristic critic" (Barto, Sutton, and Anderson 1983) and

"Q-learning" (Watkins 1989), which try to learn an optimal policy without learning a model of the environment, and "Dyna" (Sutton 1990, 1991) and "real-time dynamic programming" (Barto, Bradtke, and Singh 1995), which try to learn an optimal policy while learning a model of the environment at the same time.

All the previously cited approaches try to learn a value function. Informally, this means that they try to associate a value to each state or to each state-action pair such that it can be used to implement a control policy. The transition from value functions to control policies is typically very easy to implement. For example, if the value function is associated to state-action pairs, a control policy will be defined by choosing in every state the action with the highest value.

Another class of techniques tries to solve RL problems by searching the space of policies instead of the space of value functions. In this case, the object of learning is the entire policy, which is treated as an atomic object. Examples of these methods are evolutionary algorithms to develop neural network controllers (Dress 1987; Fogel, Fogel, and Porto 1990; Beer and Gallagher 1992; Jacoby, Husbands, and Harvey 1995; Floreano and Mondada 1996; Moriarty and Mikkulainen 1996; and Tani 1996), or to develop computer programs (Cramer 1985; Dickmanns, Schmidhuber, and Winklhofer 1987; Hicklin 1986; Fujiki and Dickinson 1987; Grefenstette, Ramsey, and Schultz 1990; Koza 1992; Grefenstette and Schultz 1994; and Nordin and Banzhaf 1996).

In a *learning classifier system* (LCS), the machine learning technique that we have investigated and that we have used to run all the experiments presented in this book, the learned controller consists of a set of rules (called "classifiers") that associate control actions with sensory input in order to implement the desired behavior.

The learning classifier system was originally proposed within a context very different from optimal control theory. It was intended to build rule-based systems with adaptive capabilities in order to overcome the brittleness shown by traditional handmade expert systems (Holland and Reitmann 1978; Holland 1986). Nevertheless, it has recently been shown that by operating a strong simplification on an LCS, one obtains a system equivalent to Q-learning (Dorigo and Bersini 1994). This makes the connection between LCSs and adaptive optimal control much tighter.

The LCS learns the usefulness ("strength") of classifiers by exploiting the "bucket brigade algorithm," a temporal difference technique (Sutton

1988) strongly related to Q-learning. Also, the LCS learns new classifiers by using a genetic algorithm. Therefore, in a LCS there are aspects of both the approaches discussed above: it learns a value function (the strength of the classifiers), and at the same time it searches the space of possible rules by exploiting an evolutionary algorithm. A detailed description of our implementation of an LCS is given in chapter 2.

1.5 THE AGENTS AND THEIR ENVIRONMENTS

Our work has been influenced by Wilson's "Animat problem" (1985, 1987), that is, the problem of realizing an artificial system able to adapt and survive in a natural environment. This means that we are interested in behavioral patterns that are the artificial counterparts of basic natural responses, like feeding and escaping from predators. Our experiments are therefore to be seen as possible solutions to fragments of the Animat problem.

Behavior is a product of the interaction between an agent and its environment. The set of possible behavioral patterns is therefore determined by the structure and the dynamics of both the agent and the environment, and by the interface between the two, that is, the agent's sensorimotor apparatus. In this section, we briefly introduce the agents, the behavior patterns, and the environments we shall use in our experiments.

1.5.1 The Agents' "Bodies"

Our typical artificial agent, the AutonoMouse, is a small moving robot. Two examples of AutonoMice, which we call "AutonoMouse II" and "AutonoMouse IV", are shown in figures 1.3 and 1.4. The simulated AutonoMice used in the experiments presented in chapters 4 and 6 are, unless otherwise stated, the models of their physical counterparts. We also use AutonoMouse V, a slight variation of AutonoMouse IV, which will be described in chapter 7.

Besides using the homemade AutonoMice, we have also experimented with two agents based on commercial robots, which we call "HAMSTER" and "CRAB." HAMSTER (figure 1.5) is a mobile robot based on Robuter, a commercial platform produced by RoboSoft. CRAB (figure 1.6) is a robotic arm based on a two-link industrial manipulator, an IBM 7547 with a SCARA geometry. More details on the robots and their environments will be given in the relevant chapters.

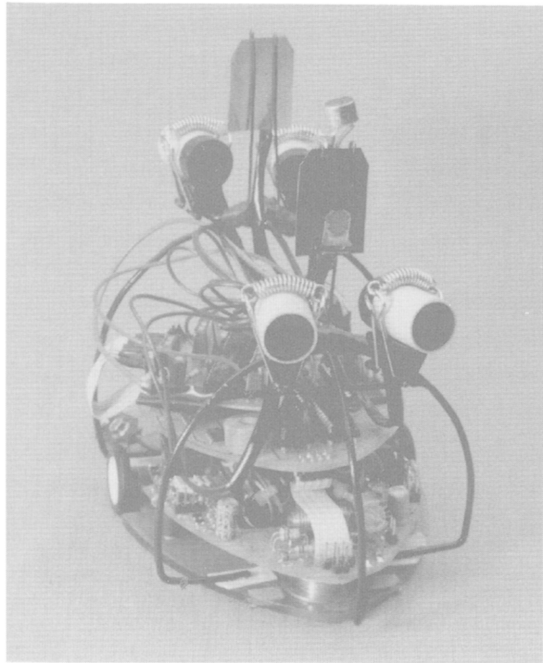


Figure 1.3
AutonoMouse II

1.5.2 The Agents' "Mind"

The AutonoMice are connected to ALECSYS (A LEarning Classifier SYStem), a learning classifier system implemented on a network of transputers (Dorigo and Sirtori 1991; Colombetti and Dorigo 1993; Dorigo 1995), which will be presented in detail in chapter 2. ALECSYS allows one to design the architecture of the agent controller as a distributed system. Each component of the controller, which we call a "behavioral module," can be connected to other components and to the sensorimotor interface.

As a learning classifier system exploiting a genetic algorithm, ALECSYS is based on the metaphor of biological evolution. This raises the question of whether evolution theory provides the right technical language to characterize the learning process.

At the present stage of the research, we are inclined to think it does not. There are various reasons why the language of evolution cannot literally apply to our agents. First, we use an evolutionary mechanism to implement individual learning rather than phylogenetic evolution. Second, the

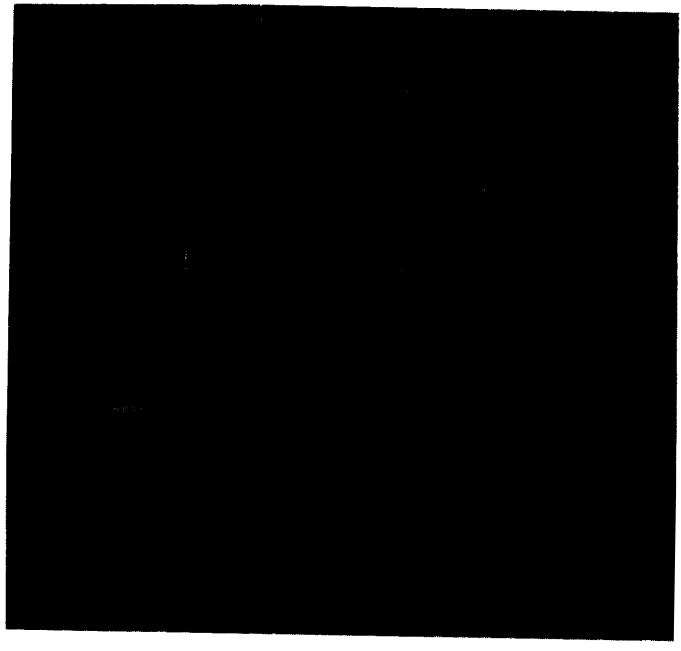


Figure 1.4
AutonoMouse IV

distinction between phenotype and genotype, essential in evolution theory, is in our case rather confused; individual rules within a learning classifier system play both the role of a single chromosome and of the phenotype undergoing natural selection. In our experiments, we found that we tend to consider the learning system as a black box, able to produce stimulus-response (S-R) associations and categorizations of stimuli into relevant equivalence classes. More precisely, we expect the learning system to

- discover useful associations between sensory input and responses; and
- categorize input stimuli so that precisely those categories will emerge which are relevantly associated to responses.

Given these assumptions, the sole preoccupation of the designer is that the interactions between the agent and the environment can produce enough relevant information for the target behavior to emerge. As it will appear from the experiments reported in the following chapters, this concern influences the design of the artificial environment's objects and of the agent's sensory interface.

Figure 1.5
HAMSTER

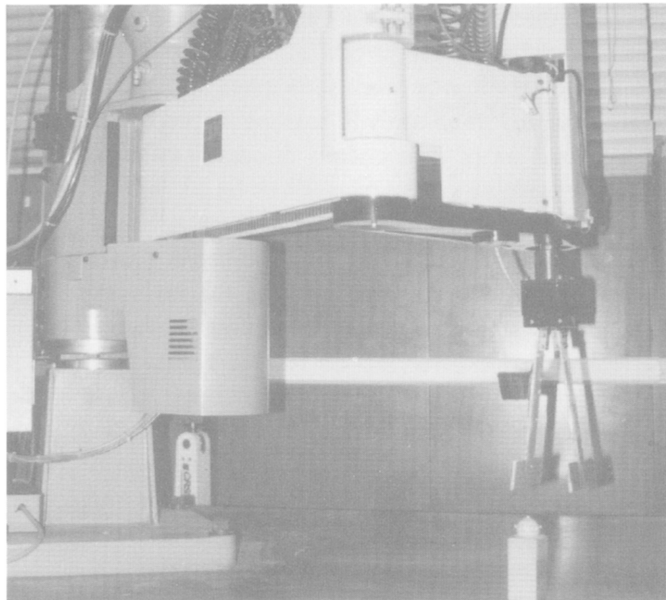


Figure 1.6
CRAB

1.5.3 Types of Behavior

A first, rough classification allows one to distinguish between *stimulus-response (S-R) behavior*, that is, reactive responses connecting sensors to effectors in a direct way, and *dynamic behavior*, requiring some kind of internal state to mediate between input and output.

Most of our experiments (chapters 4 and 5) are dedicated to S-R behavior, although in chapter 6 we investigate some forms of dynamic behavior. To go beyond simple S-R behavior implies that the agent is endowed with some form of internal state. The most obvious candidate for an internal state is a memory of the agent's past experience (Lin and Mitchell 1992; Cliff and Ross 1994; Whitehead and Lin 1995). The designer has to decide *what* has to be remembered, *how* to remember it, and for *how long*. Such decisions cannot be taken without a prior understanding of relevant properties of the environment.

In an experiment reported in section 4.2.3, we added a *sensor memory*, that is, a memory of the past state of the agent's sensors, allowing the learning system to exploit regularities of the environment. We found that a memory of past perceptions first makes the learning process harder, but eventually increases the performance of the learned behavior. By running a number of such experiments, we confirmed an obvious expectation, namely, that the memory of past perceptions is useful only if the relationship between the agent and its environment changes slowly enough to preserve a high correlation between subsequent states. In other words, agents with memory are "fitter" only in reasonably predictable environments.

In a second set of experiments, discussed at length in chapter 6, we improve our learning system capabilities by adding a module to maintain a *state word*, that is, an internal state. We show that, by the use of the state word, we are able to make our AutonoMice learn sequential behavior patterns, that is, behaviors in which the decision of what action to perform at time t is influenced by the actions performed in the past.

We believe that experiments on robotic agents must be carried out in the real world to be truly significant, although such experiments are in general costly and time-consuming. It is therefore advisable to preselect a small number of potentially relevant experiments to be performed in the real world. To carry out the selection, we use a simulated environment, which allows us to have accurate expectations on the behavior of the real agent and to prune the set of possible experiments.

One of the hypotheses we want to explore is that relatively complex behavioral patterns can be built bottom-up from a set of simple responses.

This hypothesis has already been put to test in robotics, for example, by Arkin (1990) and by Saffiotti, Konolige and Ruspini (1995). Arkin's "Autonomous Robot Architecture" integrates different kinds of information (perceptual data, behavioral schemes, and world knowledge) in order to get a robot to act in a complex natural environment. The robot generates complex responses, like walking through a doorway, as a combination of competing simpler responses, like moving ahead and avoiding a static obstacle (the wall, in Arkin's doorway example). The compositional approach to building complex behavior has been given a formal justification in the framework of multivalued logics by Saffiotti, Konolige, and Ruspini. The key point is that complex behavior can demonstrably emerge from the simultaneous production of simpler responses. We have considered four kinds of basic responses:

1. *Approaching behavior*, that is, getting closer to an almost still object with given features; in the natural world, this response is a fundamental component of feeding and sexual behavior.
2. *Chasing behavior*, that is, following and trying to catch a still or moving object with given features; like the approaching behavior, this response is important for feeding and reproduction.
3. *Avoidance behavior*, that is, avoiding physical contact with an object of a given kind; this can be seen as the artificial counterpart of a behavioral pattern that allows an organism to avoid objects that can hurt it.
4. *Escaping behavior*, that is, moving as far as possible from an object with given features; the object can be viewed as a predator.

As we shall see, more complex behavioral patterns can be built from these simple responses in many different ways.

1.5.4 The Environment

Our robots are presently able to act in closed spaces, with smooth floors and constant lighting, where they interact with moving light sources, fixed obstacles, and sounds. Of course, we could fantasize freely in simulations, by introducing virtual sensors able to detect the desired entities, but then results would not be valid for real experimentation. We therefore prefer to adapt our goals to the actual capacities of the agents.

The environment is not interesting per se, but for the kind of interactions that can take place between it and the agent. Consider the four basic responses introduced in the previous subsection. We can call them "objectual" in that they involve the agent's relationship with an external object.

Objectual responses are

- *type-sensitive* in that agent-object interactions are sensitive to the type to which the object belongs (prey, obstacle, predator, etc.); and
- *location-sensitive* in that agent-object interactions are sensitive to the relative location of the object with respect to the agent.

Type sensitivity is interesting because it allows for fairly complex patterns of interaction, which are, however, within the capacity of an S-R agent. It requires only that the agent be able to discriminate some object feature characteristic of the type. Clearly, the types of objects an S-R agent can tell apart depend on the physical interactions between external objects and the agent's sensory apparatus. Note that an S-R agent is not able to *identify* an object, that is, discern two similar but distinct objects of the same type.

The interactions we consider do not depend on the absolute location of the objects and of the agent; they depend only on the relative angular position, and sometimes on the relative distance, of the object with respect to the agent. Again, this requirement is within the capacities of an S-R agent.

In the context of shaping, differences that appear to an external observer can be relevant even if they are not perceived by the agent. The reason is that the trainer will in general base his reinforcing activity on the observation of the agent's interaction with the environment. Clearly, from the point of view of the agent, a single move of the avoidance or the escaping behavior would be exactly the same, although in complex behavior patterns, avoidance and escaping relate differently to other behaviors. In general, avoidance should modulate some other movement response, whereas escaping will be more successful if it suppresses all competing responses. As we shall see in the following chapters, this fact is going to influence both the architectural design and the shaping policy for the agent.

For learning to be successful, the environment must have a number of properties. Given the kind of agent we have in mind, the interaction of a physical object with the agent depends only on the object's type and on its relative position with respect to the agent. Therefore, sufficient information about object types and relative positions must be available to the agent. This problem can be solved in two ways: either the natural objects existing in the environment have sufficient distinctive features that allow them to be identified and located by the agent or the artificial objects must

be designed so that they can be identified and located. For example, if we want the agent to chase light L1 and avoid light L2, the two lights must be of different color, or have a different polarization plane, to be distinguished by appropriate sensors. In any case, identification will be possible only if the rest of the environment cooperates. For example, if light sensing is involved, environmental lighting should not vary too much during the agent's life.

In order for a suitable response to depend on an object's position, objects must be still, or move slowly enough with respect to the agent's speed. This does not mean that a sufficiently smart agent could not evolve a successful interaction pattern with very fast objects, only that such a pattern could not depend on the instantaneous relative position of the object, but would involve some kind of extrapolation of the object's trajectory, which is beyond the present capacities of ALECSYS.

1.6 BEHAVIOR ENGINEERING AS AN EMPIRICAL ENDEAVOR

We regard our own work as part of artificial intelligence (AI). Although certainly departing from work in classical, symbolic AI, it is akin to research in the "new wave" of AI that started in the early 1980s and is focused on the development of simple but complete agents acting in the real world.

In classical AI, computer programs have been regarded sometimes as formal theories and sometimes as experiments. We think that both views are defective. A computer program certainly is a formal object, but it lacks the degree of conciseness, readability, and elegance required from a theory. Moreover, programs usually blend together high-level aspects of theoretical relevance with low-level elements that are there only to make it perform at a reasonable level. But a program is not by itself an experiment; at most, it is the starting point to carry out experiments (see, for example, Cohen 1995). Borrowing the jargon of experimental psychologists, by instantiating a program with specific parameters, we have an "experimental subject," whose behavior we can observe and analyze.

It is surprising to note how little this point of view is familiar in computer science in general, and in AI in particular. The classical know-how of the empirical sciences regarding experimental design, data analysis, and hypothesis testing is still almost ignored in our field. Presumably, the reason is that computer programs, as artificial objects, are erroneously

believed to be easily understandable by their designers. It is important to realize that this is not true.

Although a tiny, well-written computer program can be completely understood by analyzing it statically as a formal object, this has little to do with the behavior of complex programs, especiallly in the area of nonsymbolic processing, which lacks strong theoretical bases. In this case, a great deal of experimental activity is necessary to understand the behavior of programs, which behave almost like natural systems.

A disciplined empirical approach appears to be of central importance in what we have called "behavior engineering," that is, in the principled development of robotic agents. Not only is the agent itself too complex to be completely understood by analytical methods, but the agent also interacts with an environment that often cannot be completely characterized a priori. As we shall argue in chapter 7, the situation calls for an experimental activity organized along clear methodological principles.

The work reported in this book is experimentally oriented: it consists of a reasonably large set of experiments performed on a set of experimental subjects, all of which are instantiations of the same software system, ALECSYS. However, we have to admit that many of our experiments were below the standards of the empirical sciences. For this, we have a candid, but true, explanation. When we started our work, we partly shared the common prejudice about programs, and we only partially appreciated the importance of disciplined experimentation. But we learned along the way, as we hope some of the experiments will show.

1.7 POINTS TO REMEMBER

- Shaping a robot means exploiting the robot's learning capacities to translate suggestions from an external trainer into an effective control strategy that allows the robot to perform predefined tasks.
- As opposed to most work in reinforcement learning, our work stresses the importance of the trainer in making the learning process efficient enough to be used with real robots.
- The learning agent, the sensorimotor capacities of the robot, the environment in which the robot acts, and the trainer are strongly intertwined. Any successful approach to building learning autonomous robots will have to consider the relations among these components.
- Reinforcement learning techniques can be used to implement robot controllers with interesting properties. In particular, such techniques make it

possible to adopt architectures based on "soft computing" methods, like neural networks and learning classifier systems, which cannot be effectively programmed by hand.

- The approach to designing and building autonomous robots advocated in this book is a combination of design and learning. The role of design is to determine the starting architecture and the input-output interfaces of the control system. The rules that will constitute the control system are then learned by a reinforcement learning algorithm.
- Behavior engineering, that is, the discipline of principled robot development, will have to rely heavily on an empirical approach.

Chapter 2

ALECSYS

2.1 THE LEARNING CLASSIFIER SYSTEM PARADIGM

Learning classifier systems are a class of machine learning models which were first proposed by Holland (Holland and Reitmann 1978; Holland 1986; Holland et al. 1986; Booker, Goldberg, and Holland 1989). A learning classifier system is a kind of parallel production rule system in which two kinds of learning take place. First, reinforcement learning is used to change the strength of rules. In practice, an apportionment of credit algorithm distributes positive reinforcements received by the learning system to those rules that contributed to the attainment of goals and negative reinforcements to those rules that caused punished actions. Second, evolutionary learning is used to search in the rule space. A genetic algorithm searches for possibly new useful rules by genetic operations like reproduction, selection, crossover, and mutation. The fitness function used to direct the search is the rule strength.

As proposed by Holland, the learning classifier system or LCS is a general model; a number of low-level implementation decisions must be made to go from the generic model to the running software. In this section we describe LCS_0, our implementation of Holland's learning classifier system, which unfortunately exhibits a number of problems: rule strength oscillation, difficulty in regulating the interplay between the reinforcement system and the background genetic algorithm (GA), rule chains instability, and slow convergence. We identified two approaches to extend LCS_0 with the goal of overcoming these problems: (1) increase the power of a single LCS_0 by adding new operators and techniques, and (2) use many LCS_0s in parallel. The first extension, presented in section 2.2, gave raise to the "improved classifier system" or ICS (Dorigo 1993), a more powerful version of LCS_0 that still retains its spirit. The second extension,

discussed in section 2.3, led to the realization of ALECSYS (Dorigo and Sirtori 1991; Dorigo 1992, 1995), a distributed version of ICS that allows for both *low-level parallelism*, involving distribution of the ICS over a set of transputers, and *high-level parallelism*, that is, the definition of many concurrent cooperating ICSs.

To introduce learning classifier systems, we consider the problem of a simple autonomous robot that must learn to follow a light source (this problem will be considered in detail in chapter 4). The robot interacts with the environment by means of light sensors and motor actions. The LCS controls the robot by using a set of production rules, or *classifiers*, which represent the knowledge the robot has of its environment and its task. The effect of control actions is evaluated by scalar reinforcement. Learning mechanisms are in charge of distributing reinforcements to the classifiers responsible for performed actions, and of discovering new useful classifiers.

An LCS comprises three functional modules that implement the controller and the two learning mechanisms informally described above (see figure 2.1):

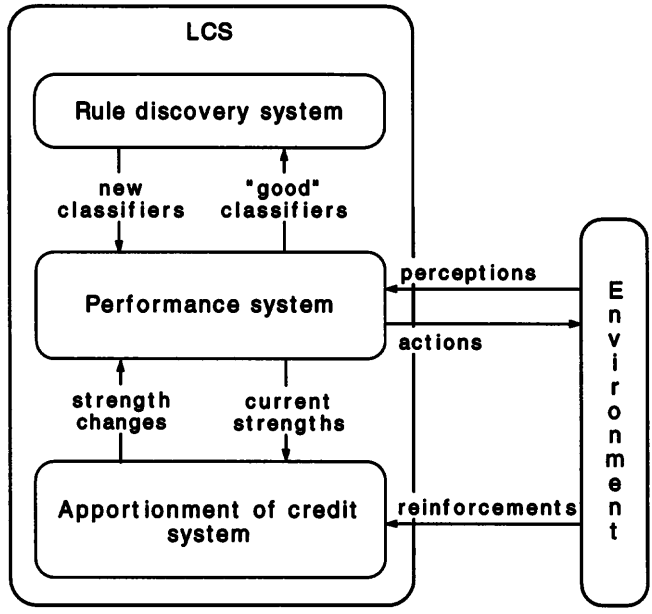


Figure 2.1
Learning classifier system

1. The *performance system* governs the interaction of the robot with the external environment. It is a kind of parallel production system, implementing a behavioral pattern, like light following, as a set of condition-action rules, or classifiers.
2. The *apportionment of credit system* is in charge of distributing reinforcements to the rules composing the knowledge base. The algorithm used is the "bucket brigade" (Holland 1980; in previous LCS research, and in most of reinforcement learning research, reinforcements are provided to the learning system whenever it enters certain states; in our approach they are provided by a *trainer*, or reinforcement program, as discussed in section 1.2).
3. The *rule discovery system* creates new classifiers by means of a genetic algorithm.

In LCSs, learning takes place at two distinct levels. First, the apportionment of credit system learns from experience the adaptive value of a number of given classifiers with respect to a predefined target behavior. Each classifier, maintains a value, called "strength," and this value is modified by the bucket brigade in an attempt to redistribute rewards to useful classifiers and punishments to useless (or harmful) ones. Strength is therefore used to assess the degree of usefulness of classifiers. Classifiers that have all conditions satisfied are fired with a probability that is a function of their strength. Second, the rule discovery mechanism, that is, the genetic algorithm, allows the agent to explore the value of new classifiers. The genetic algorithm explores the classifiers' space, recombining classifiers with higher strength to produce possibly better offspring. Offspring are then evaluated by the bucket brigade.

It must be noted that the LCS is a general model with many degrees of freedom. When implementing an LCS, it is therefore necessary to make a number of decisions that transform the generic LCS into an implemented LCS. In the rest of this section we present the three modules of LCS_0, our implementation of an LCS.[1] ICS and ALECSYS, presented in sections 2.2 and 2.3, were obtained as improvements of LCS_0.

2.1.1 LCS_0: The Performance System

The LCS_0 performance system (see figure 2.2 and terminology box 2.1) consists of

- the *classifier set* CS;
- an *input interface* and an *output interface* with the environment (sensors and effectors) to receive/send messages from/to the environment;

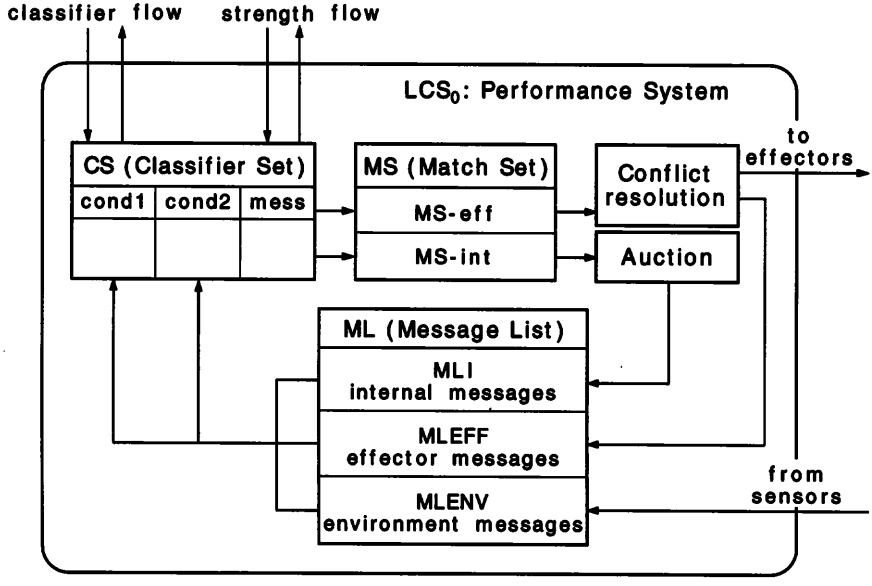


Figure 2.2
LCS_0 performance system

Example Box 2.1

Consider the classifier #1#; 011 → 010. Here the second condition is matched only by the message 011, while the first condition is matched by any message with a 1 in second position. The # symbol stays for "don't care," meaning that both symbols, 0 or 1, match the position. If both conditions are matched by some message, then the rule is activated, and the message/action part, that is, the string 010 in our example, is appended to the message list at the following step (if it is to be interpreted as an internal message) or is sent to the effectors (if it is to be interpreted as an effector message, that is, an action). If some effector messages propose contradictory actions, then conflict resolution is applied and contradictions are removed by discarding some of the messages.

Terminology Box 2.1

- In LCS_0 a *classifier* (or *rule*) is a triple (K_1, K_2, M), where K_1 and K_2 are strings of length n on $\{0, 1, \#\}^*$ and M is a string of length n on $\{0, 1\}^*$. Informally, (K_1, K_2) is the conjunction of two conditions, and M is the action part of the classifier (that is, a rule with two conditions in conjunctive form and a single action, also called *message*). Every K_i $(i = 1, 2)$ denotes a *schema*, that is, the set of all the strings that can be obtained from K_i by means of the substitution $(\# \rightarrow 0, 1)$ in all the possible ways.
- An *effector classifier* is a classifier that sends messages to the effectors, and an *internal classifier* is a classifier that sends messages to other classifiers.
- The *message list* (ML) is the union of an *environment message list* (MLENV), which contains messages coming from the sensors, an *internal message list* (MLI), which contains messages sent from internal classifiers at the preceding time step, and an *effector message list* (MLEFF), which contains messages were used to produce actions by the effectors at the preceding time step.
- The *classifier set* (CS) is the multiset containing all the classifiers.
- The *match set* (MS) is the multiset containing the classifiers that at a given time step of the algorithm have both conditions matched by at least one message. Also, MS = MS-eff ∪ MS-int, where MS-eff is the multiset of classifiers in MS that want to send their messages to effectors (effector classifiers), and MS-int is the multiset of classifiers in MS that want to append their messages to the internal message list MLI (internal classifiers).

- the *message list* ML, composed of the three sublists: MLI, which collects messages sent from classifiers at the preceding time step, MLENV, which collects messages coming from the environment through the sensors, and MLEFF, which contains messages used to choose the actions at the preceding time step;
- the *match set* MS, that is, the multiset containing classifiers that have both conditions matched by at least one message (see example box 2.1). MS is composed of MS-eff and MS-int, that is two multisets containing classifiers in MS that want to send their messages to effectors and to the internal message list MLI, respectively;
- a *conflict resolution* module, to arbitrate conflicts among rules in MS-eff that propose contradictory actions; and
- an *auction module*, to select which rules in MS-int are allowed to append a message to MLI.

In simplified terms, the LCS_0 performance system works like this. At time zero, a set of classifiers is randomly created, and both the message

list and the match set are empty. Then the following loop is executed. Environmental messages, that is, perceptions coming from sensors, are appended to the message list MLENV and matched against the condition part of classifiers. Matching classifiers are added to the match set, and the message list is then emptied. If there is any message to effectors, then corresponding actions are executed (if necessary, a conflict resolution module is called to choose among conflicting actions); messages used to select performed actions are appended to MLEFF. If the cardinality of MS-int does not exceed that of MLI, all messages are appended. Otherwise, the auction module is used to draw a k-sample from MS-int (where k is the cardinality of MS-int); the messages of the classifiers in the k-sample are then appended to MLI. The match set is emptied and the loop is repeated.

The need to run an auction is one reason an apportionment of credit algorithm is introduced. As it redistributes reinforcements to the rules that caused the performed actions, the algorithm changes their strengths. This allows the system to choose which rules to select in accordance to some measure of their usefulness. The rules' strength is also used for conflict resolution among rules that send messages to effectors proposing inconsistent actions (e.g., "Go right" and "Go left"). Algorithm box 2.1 gives a detailed formulation of the performance algorithm.

2.1.2 LCS_0: The Apportionment of Credit System

We have said that the main task of the apportionment of credit algorithm is to classify rules in accordance with their usefulness. In LCS_0, the algorithm works as follows: to every classifier C a real value called "strength," Str(C), is assigned. At the beginning, each classifier has the same strength. When an effector classifier causes an action on the environment, a reinforcement may be generated, which is then transmitted backward to the internal classifiers that caused the effector classifier to fire. The backward transmission mechanism, examined in detail later, causes the strength of classifiers to change in time and to reflect the relevance of each classifier to the system performance (with respect to the system goal).

Clearly, it is not possible to keep track of all the paths of activation actually followed by the rule chains (a *rule chain* is a set of rules activated in sequence, starting with a rule activated by environmental messages and ending with a rule performing an action on the environment) because the number of these paths grows exponentially with the length of the path. It is then necessary to have an appropriate algorithm that solves the problem using only local information, in time and in space.

Algorithm Box 2.1
LCS_0: The performance system

0. Associate an initial strength to every classifier C_i, and clear the message list.

loop

1. Read environmental messages and append them to MLENV.
2. Execute the MATCH operation between messages in ML and the condition part of classifiers in CS.
3. If no classifier is matched by at least a message in MLENV, then call the *cover detector* operator.
4. MS := classifiers $C_1, \ldots, C_r$ which have each condition matched by at least one message.
5. Remove all messages from ML.
6. Apply conflict resolution to classifiers in MS-eff. The resulting messages are both appended to MLEFF and sent to effectors, which perform the corresponding actions.
7. If Length(MS-int) $\leq$ Length(MLI),
   then let $\mathcal{A}$:= MS-int and append to MLI the messages of classifiers in $\mathcal{A}$
   else draw an independent k-sample $\mathcal{A}=[C_1,\ldots,C_k]$ from MS-int with probability $\text{Prob}(C_i) = \text{Strength}(C_i)/\Sigma_h \text{Strength}(C_h)$ {this is the auction} and append to MLI the messages $[M_1, \ldots, M_k]$ of classifiers in $\mathcal{A}$. {Classifiers in $\mathcal{A}$ are *active classifiers*; $\text{Strength}(C_i)$ is the strength of classifier C_i}.
   endif
8. Remove all classifiers from MS.
9. If exists at least one effector action EA such that $\forall\ C \in CS$ $(M \cap EA = \varnothing)$, where M is the message part of classifier C, then apply with probability p_{CE} the *cover effector* operator.

endloop

The symbols used in the algorithm are so defined:

- $CS = \{C_1, \ldots, C_i, \ldots, C_p\}$, where C_i is a classifier;
- $ML = MLI \cup MLENV \cup MLEFF$ is the message list. $MLI = (M_1, \ldots, M_k)$ is the list of internal messages; $MLENV = (M_{k+1}, \ldots, M_q)$ is the list of environmental messages; $MLEFF = (M_{q+1}, \ldots, M_f)$ is the list of messages that caused an action at the previous time step; and M_s is a message.
- MS = MS-int $\cup$ MS-eff is the match set, that is, the multiset of classifiers that have each condition matched by at least one message. MS-int is the set of matched internal classifiers, while MS-eff is the set of matched effector classifiers. The main operation is $\text{MATCH}(K_i, M) \equiv M \in K_i$, where K_i is a classifier's condition and M is a message in the message list.

"Local in time" means that the information used at every computational step is coming only from a fixed recent temporal interval. "Local in space" means that changes in a classifier's strength are caused only by classifiers directly linked to it (we say that classifiers C_1 and C_2 are "linked" if the message posted by C_1 matches a condition of C_2).

The classical algorithm used for this purpose in LCS_0, as well as in most LCSs, is the bucket brigade algorithm (Holland 1980). This algorithm models the classifier system as an economic society, in which every classifier pays an amount of its strength to get the privilege of appending a message to the message list and receives a payment by the classifiers activated because of the presence in the message list of the message it appended during the preceding time step.[2] Also, a classifier can increase (decrease) its strength whenever an action it proposes is performed and results in receiving a reward (punishment). In this way, reinforcement flows backward from the environment (trainer) again to the environment (trainer) through a chain of classifiers. The net result is that classifiers participating in chains that cause highly rewarded actions tend to increase their strength. Also, our classifiers lose a small percentage of their strength at each cycle (what is called the "life tax") so that classifiers that are never used will sooner or later be eliminated by the genetic algorithm.

In algorithm box 2.2, we report the apportionment of credit system. Because the bucket brigade algorithm is closely intertwined with the performance system, a detailed description of how it works is more easily given together with the description of the performance system. Bucket brigade instructions are those steps of the credit apportionment system not already present in the performance system (they can be easily recognized because we maintained the same name for instructions in the two systems, and the bucket brigade instructions were just added in between the performance system instructions).

One of the effects of the apportionment of credit algorithm should be to contribute to the formation of default hierarchies, that is, sets of rules that categorize the set of environmental states into equivalence classes. A rule in a default hierarchy is characterized by its specificity. A maximally specific rule will be activated by a single type of message (see, for example, the set of four rules in the left part of figure 2.3). A maximally general (default) rule is activated by any message (see, for example, the last rule in the right part of figure 2.3).

Consider a default hierarchy composed of two rules, as reported in the right part of figure 2.3. In a default hierarchy, whenever a set of rules have

Algorithm Box 2.2
LCS_0: The apportionment of credit system

0. Associate an initial strength to every classifier C_i, clear the message list, and $\mathscr{A}^- := \varnothing$. {$\mathscr{A}^-$ is the set of active classifiers at the previous time step.}

loop

1. Read environmental messages and append them to MLENV.
2. Execute the MATCH operation between messages in ML and the condition part of classifiers in CS.
3. If no classifier is matched by at least one message in MLENV, then call the *cover detector* operator.
4. MS := classifiers $C_1, \ldots, C_r$, which have each condition matched by at least one message.
5. Remove all messages from ML.
6. Apply conflict resolution to classifiers in MS-eff. The resulting messages are both appended to MLEFF and sent to effectors, which perform the corresponding actions.

6.1 If there are any reinforcements, sum them to the strengths of those rules in MS-eff whose message has been chosen to be sent to effectors.

7. If Length(MS-int) $\leq$ Length(MLI),
   then let $\mathscr{A}$:= MS-int and append to MLI the
   messages of classifiers in $\mathscr{A}$
   else draw an independent k-sample $\mathscr{A} = [C_1, \ldots, C_k]$
   from MS-int with probability
   $\text{Prob}(C_i) = \text{Strength}(C_i)/\Sigma_h \text{Strength}(C_h)$
   {this is the auction}
   and append to MLI the messages $[M_1, \ldots, M_k]$
   of classifiers in $\mathscr{A}$. {Classifiers in $\mathscr{A}$ are
   active classifiers; Strength(C_i) is the
   strength of classifier C_i}
   endif

7.1 $\mathscr{C}_s := \varnothing$ {$\mathscr{C}_s$ is the multiset composed of those classifiers whose messages caused the activation of C_s}.

7.2 For each $C_s \in \mathscr{A}$, add to $\mathscr{C}_s$ those classifiers in $\mathscr{A}^-$ whose messages contributed to the activation of C_s.

7.3 For each $C_s \in \mathscr{A}$,
   for each $C_j \in \mathscr{C}_s$,
   $\text{Strength}(C_j) := \text{Strength}(C_j) + f(\text{Strength}(C_s))/\text{Cardinality of } (\mathscr{C}_s)$ {f is a function of the classifier strength; usually f is a constant in the interval [0, 1]}.

7.4 For each $C_s \in \mathscr{A}$, $\text{Strength}(C_s) := \text{Strength}(C_s) - f(\text{Strength}(C_s))$

7.5 For each $C_s \in CS$, $\text{Strength}(C_s) := \rho \cdot \text{Strength}(C_s)$ {$0 \leq \rho \leq 1$ is the *life tax*}.

Algorithm Box 2.2 (continued)

7.6 $\mathcal{A}^- := \mathcal{A}$.
8. Remove all classifiers from MS.
9. If exists at least one effector action EA such that $\forall$ C $\in$ CS ($M \cap EA = \varnothing$), where M is the message part of classifier C, then apply with probability p_{CE} the *cover effector* operator.

endloop

Nonhierarchical set (homomorphic set)	Hierarchical set (quasi-homomorphic set)
00 → 11	00 → 11
01 → 00	## → 00
10 → 00	
11 → 00	

Figure 2.3
Hierarchical set implements same state categorization with fewer rules.

all conditions matched by at least a message, it is the most specific that is allowed to fire. Default hierarchies have some nice properties. They can be used to build quasi-homomorphic models. Let $\mathcal{E}$ be the set of environmental states that the learning system has to learn to categorize. Then, the system could either try to find a set of rules that partition the whole set $\mathcal{E}$, never making mistakes (left part of figure 2.3), or build a default hierarchy (right part of figure 2.3). With the first approach, a homomorphic model of the environment is built; with the second, a quasi-homomorphic model, which generally requires far fewer rules than an equivalent homomorphic one (Holland 1975; Riolo 1987, 1989). Moreover, after an initial quasi-homomorphic model is built, its performance can be improved by adding more specific rules. In this way, the whole system can learn gracefully, that is, the performance of the system is not too strongly influenced by the insertion of new rules.

2.1.3 LCS_0: The Rule Discovery System

The rule discovery system algorithm used in LCS_0, as well as in most LCSs, is the genetic algorithm (GA). GAs are a computational device

Terminology Box 2.2

- A *population* is a multiset of individuals.
- An *individual* (chromosome) is a string of k positions, or *genes*. Typically, when GAs are applied to optimization problems, an individual is a feasible solution. On the other hand, when in the LCS framework, an individual is a classifier.
- A *gene* can assume an *allelic value* belonging to an alphabet that is usually $A = \{0, 1\}$. The reasons underlying this choice are to be found in the rule discovery algorithm used, that is, the genetic algorithm. It has been shown (Holland 1975) that the lower the cardinality of the alphabet, the higher the efficiency of the algorithm in processing useful information contained in the structure of the chromosomes. When individuals are classifiers, then the alphabet is often augmented with the symbol # ("don't care"), which allows for general (default) rules to be represented. In LCS_0, allelic values of condition genes belong to $\{0, 1, \#\}$, while allelic values of action (message) genes belong to $\{0, 1\}$.

inspired by population genetics. We give here only some general aspects of their working; the interested reader can refer to one of the numerous books about the subject (for example, Goldberg 1989; Michalewicz 1992; Mitchell 1996).

A GA is a stochastic algorithm that works by modifying a population of solutions[3] to a given problem (see terminology box 2.2). Solutions are coded, often using a binary alphabet, although other choices are possible, and a function, called a "fitness function," is defined to relate solutions to performance (that is, to a measure of the quality of the solution). At every cycle, a new population is created from the old one, giving higher probability to reproduce (that is, to be present again in the new population of solutions) to solutions with a fitness higher than average, where *fitness* is defined to be the numeric result of applying the fitness function to a solution. This new population is then modified by means of some genetic operators; in LCS_0 these are

- *crossover*, which recombines individuals (it takes two individuals, operates a cut in a randomly chosen point, and recombines the two in such a way that some of the genetic material of the first individual goes to the second one and vice versa; see figure 2.4); and
- *mutation*, which randomly changes some of the allelic values of the genes constituting an individual.

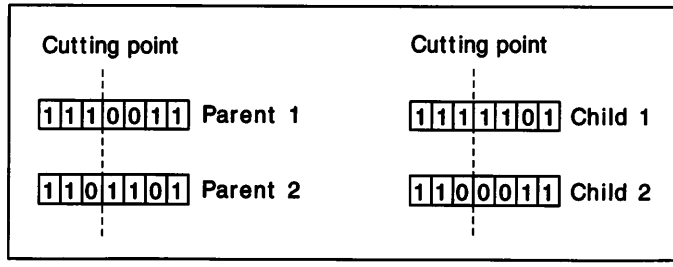


Figure 2.4
Example of crossover operator: (a) two individuals (parents) are mated and crossover is applied, to generate (b) two new individuals (children).

Algorithm Box 2.3
LCS_0: The rule discovery system (genetic algorithm)

0. Generate an initial random population P.

repeat

1. Apply the Reproduction operator to P:
   $P' := Reproduce(P)$.
2. Apply the Crossover operator (with probability p_c) to P':
   $P'' := Crossover(P')$.
3. Apply the Mutation operator (with probability p_m) to P'':
   $P''' := Mutation(P'')$.
4. $P := P'''$.

until EndTest = true.

After these modifications, the reproduction operator is applied again, and the cycle repeats until a termination condition is verified. Usually the termination condition is given by a maximum number of cycles, or by the lack of improvements during a fixed number of cycles. In algorithm box 2.3, we report the rule discovery system (genetic algorithm).

The overall effect of the GAs' work is to direct the search toward areas of the solution space with higher values of the fitness function. The computational speedup we obtain using GAs with respect to random search is due to the fact that the search is directed by the fitness function. This direction is based not on whole chromosomes but on the parts that are strongly related to high values of the fitness function; these parts are called "building blocks" (Holland 1975; Goldberg 1989). It has been proven (Booker, Goldberg, and Holland 1989; Bertoni and Dorigo 1993)

that GAs process at each cycle a number of building blocks at least proportional to a polynomial function of the number of individuals in the population, where the degree of the polynomial depends on the population dimension.

In LCSs, the hypothesis is that useful rules can be derived from recombination of other useful rules.[4] GAs are then used as a rule discovery system. Most often, as in the case of our system, in order to preserve the system performance, they are applied only to a subset of rules: the m best rules (the rules with higher strength) are selected to form the initial population of the GA, while the worst m rules are replaced by those arising from application of the GA to the initial population. After applying the GA, only some of the rules are replaced. The new rules will be tested by the combined action of the performance and credit apportionment algorithms. Because testing a rule requires many time steps, GAs are applied with a much lower frequency than the performance and apportionment of credit systems.

In order to improve efficiency, we added to LCS_0 a *cover detector* operator and a *cover effector* operator (these cover operators are standard in most LCSs).

The cover detector fires whenever no classifier in the classifier set is matched by any message coming from the environment (that is, by messages in MLENV). A new classifier is created with conditions that match at least one of the environmental messages (a random number of #s are inserted in the conditions), and the action part is randomly generated.

The cover effector, which is applied with a probability p_{CE}, assures the same thing on the action side. Whenever some action cannot be activated by any classifier, then a classifier that generates the action and that has random conditions is created. In our experiments, we set $p_{CE} = 0.1$.

Covering is not an absolutely necessary operator because the system is in principle capable of generating the necessary classifiers by using the basic GA operators (crossover and mutation). On the other hand, because this could require a great deal of time, covering is a useful way to speed up search in particular situations easily detected by the learning system.

2.2 ICS: IMPROVED CLASSIFIER SYSTEM

In this section, we describe ICS, an improved version of LCS_0. The following major innovations have been introduced (for the experimental evaluation of these changes to LCS_0, see section 4.2).

2.2.1 ICS: Calling the Genetic Algorithm When a Steady State Is Reached

In LCS_0, as in most classic implementations of LCSs, the genetic algorithm (i.e., reproduction, crossover, and mutation) is called every q cycles, q being a constant whose optimal value is experimentally determined.

A drawback of this approach is that experiments to find q are necessary, and that, even if using the optimal value of q, 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 q changes in time, depending on the dynamics of the environment and on the stochastic processes embedded in the learning algorithms. If the genetic algorithm is called too soon, it uses inaccurate information; if it is called long after a steady state has been reached, there is a waste 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 has reached a steady state; in this way, we can reasonably expect that when the genetic algorithm is called, the strength of every classifier reflects its actual usefulness to the system. The problem is how to correctly evaluate the attainment of a steady state. We have introduced a function $E_{LCS}(t)$, called "energy"[5] of the LCS at time t, defined as the sum of the strengths of all classifiers. An LCS is said to be "at a steady state at time t" when

$$E_{LCS}(t') \in [E_{min}, E_{max}], \qquad \text{for all } t' \in [t-k, t],$$

where

$$E_{min} = \min\{E_{LCS}(t''), t'' \in [t-2k, t-k]\},$$

and

$$E_{max} = \max\{E_{LCS}(t''), t'' \in [t-2k, t-k]\},$$

k being a parameter. This excludes cases in which $E_{LCS}(t)$ is increasing or decreasing and those in which $E_{LCS}(t)$ is still oscillating too much. Experiments have shown that the value of k is very robust; in our experiments (see section 4.2; and Dorigo 1993), k was set to 50, but no substantial differences were found for values of k in the range between 20 and 100.

2.2.2 ICS: The Mutespec Operator

Mutespec is a new operator we introduced to reduce the variance in the reinforcement received by default rules. A problem caused by the presence

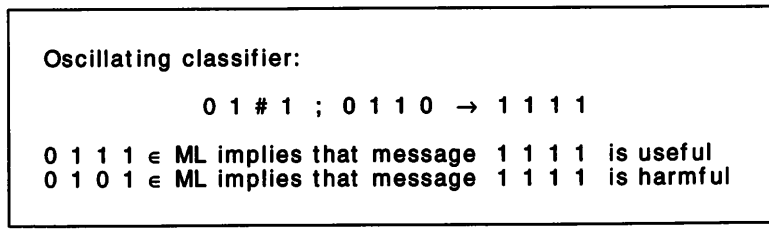


Figure 2.5
Example of oscillating classifier

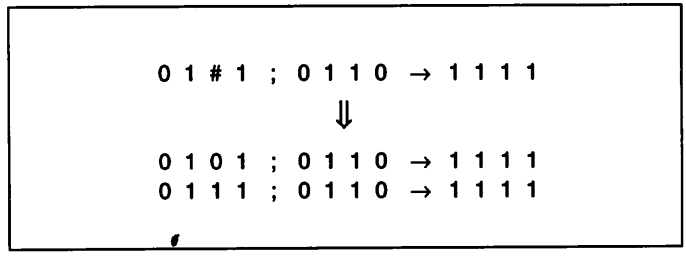


Figure 2.6
Example of application of mutespec operator

of "don't care" (#) symbols in classifier conditions is that the same classifier can receive high rewards when matched by some messages, and low rewards (or even punishments if we use negative rewards) when matched by others. We call these classifiers "oscillating classifiers." Consider the example in figure 2.5; the oscillating classifier has a "don't care" symbol in the third position of the first condition. Suppose that whenever the classifier is matched by a message with a 1 in the position corresponding to the # in the classifier condition, the message 1 1 1 1 is useful, and that whenever the matching value is a 0, the message 1 1 1 1 is harmful. As a result, the strength of that classifier cannot converge to a steady state but will oscillate between the values that would be reached by the two more specific classifiers in the bottom part of figure 2.6. The major problem with oscillating classifiers is that, on average, they will be activated too often when they should not be used and too seldom when they could be useful. This causes the performance of the system to be lower than it could be.

The mutespec operator tries to solve this problem using the oscillating classifier as the parent of two offspring classifiers, one of which will have a

0 in place of the #, and the other, a 1 (see figure 2.6). This is the optimal solution when there is a single #. When the number of # symbols is greater than 1, the # symbol chosen could be the wrong one (i.e., not the one responsible for oscillations). In that case, because the mutespec operator did not solve the problem, it is highly probable the operator will be applied again. The mutespec operator can be likened to the mutation operator. The main differences are that mutespec is applied only to oscillating classifiers and that it always mutates # symbols to 0s and 1s. And while the mutation operator mutates a classifier, mutespec introduces two new, more specific, classifiers (the parent classifier remains in the population).

To decide which classifier should be mutated by the mutespec operator, we monitor the variance of the rewards each classifier gets. This variance is computed according to the following formula:

$$\text{var}(R_C(t)) = \frac{1}{t}\sum_{j=1}^{t} R_C^2(j) - \left(\frac{\sum_{j=1}^{t} R_C(j)}{t}\right)^2,$$

where $R_C(t)$ is the reward received by classifier C at time t.

A classifier C is an oscillating classifier if $\text{var}(R_C) \geq r \cdot \overline{\text{var}}$, where $\overline{\text{var}}$ is the average variance of the classifiers in the population and r is a user defined parameter (we experimentally found that a good value is $r = 1.25$). At every cycle, mutespec is applied to the oscillating classifier with the highest variance. (If no classifier is an oscillating classifier, then mutespec is not applied.)

2.2.3 ICS: Dynamically Changing the Number of Classifiers Used

In ICS, the number of used rules dynamically changes at run time (i.e., the classifier set's 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 also very low; should it win 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's 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) than LCS_0 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. In our experiments, the use of a classifier is inhibited when its strength becomes lower than $h \cdot \overline{S}(t)$, where $\overline{S}(t)$ is the average strength of classifiers in the population at time t, and h is a user defined parameter ($0.25 \leq h \leq 0.35$ was experimentally found to be a good range for h).

2.3 THE ALECSYS SYSTEM

ALECSYS is a tool for experimenting with parallel ICSs. One of the main problems faced by ICSs trying to solve real problems is the presence of heavy limitations on the number of rules that can be employed due to the linear increase in the basic cycle complexity with the classifier set's cardinality. One possible way to increase the amount of processed information without slowing down the basic elaboration cycle would be the use of parallel architectures, such as the "Connection Machine" (Hillis 1985) or the "transputer" (INMOS 1989). A parallel implementation of an LCS on the Connection Machine, proposed by Robertson (1987), has demonstrated the power of such a solution, while still retaining, in our opinion, a basic limit. Because the Connection Machine is a single-instruction-multiple-data architecture (SIMD; Flynn 1972), the most natural design for the parallel version of an LCS is based on the "data parallel" modality, that is, a single flow of control applied to many data. Therefore, the resulting implementation is a more powerful, though still classic, learning classifier system.

Because our main goal was to give our system features such as modularity, flexibility, and scalability, we implemented ALECSYS on a transputer system.[6] The transputer's multiple-instruction-multiple-data architecture (MIMD) permits many simultaneously active flows of control to operate on different data sets and the learning system to grow gradually 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. High-level parallelism allows various ICSs to work together; therefore, the complete learning task can be decomposed into simpler learning tasks running in parallel.

2.3.1 Low-Level Parallelism: A Solution to Speed Problems

We turn now to the *microstructure* of ALECSYS, that is, the way parallelism was used to enhance the performance of the ICS model. ICS, like LCS_0 and most LCSs, can be depicted as a set of three interacting systems (see figure 2.1): (1) the performance system; (2) the credit apportionment system (bucket brigade algorithm); and (3) the rule discovery system (genetic algorithm). In order to simplify the presentation of the low-level parallelization algorithms, the various ICS systems are discussed separately. We show how first the performance and credit apportionment systems and then the rule discovery system (genetic algorithm) were parallelized.

2.3.1.1 The Performance and Apportionment of Credit Systems

As we have seen in section 2.1, the basic execution cycle of a sequential learning classifier system like LCS_0 can be looked at as the result of the interaction between two data structures: the list of messages, ML, and the set of classifiers, CS. Therefore, we decompose a basic execution cycle into two concurrent processes, MLprocess and CSprocess. MLprocess communicates with the input process, SEprocess (SEnsor), and the output process, EFprocess (EFfector), as shown in figure 2.7.

The processes communicate by explicit synchronization, following an algorithm whose high-level description is given in algorithm box 2.4.

Because steps 3 and 4 (matching and message production) can be executed on each classifier independently, we split CSprocess into an array of concurrent subprocesses $\{\text{CSprocess}_1, \ldots, \text{CSprocess}_i, \ldots, \text{CSprocess}_n\}$, each taking care of $1/n$ of CS. 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 to be the most efficient; see Dorigo 1992). CSprocesses can be organized in hierarchical structures, such as a tree (see figure 2.8) or a toroidal grid (the structure actually chosen deeply influences the distribution of computational loads and, therefore, the computational efficiency of the overall system; see Camilli et al. 1990).

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

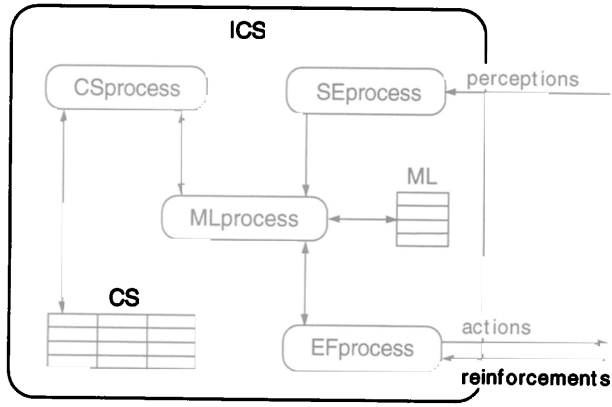


Figure 2.7
Concurrent processes in ICS

Algorithm Box 2.4
Process-oriented view of learning classifier systems

while not stopped **do**
1. MLprocess receives messages from SEprocess and places them in the message list ML.
2. MLprocess sends ML to CSprocess.
3. CSprocess "matches" ML and CS, calculating bids for each triggered rule.
4. CSprocess 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 CSprocess.
9. CSprocess modifies the strengths of CS elements, paying bids, assigning reinforcements, and collecting taxes.

endwhile

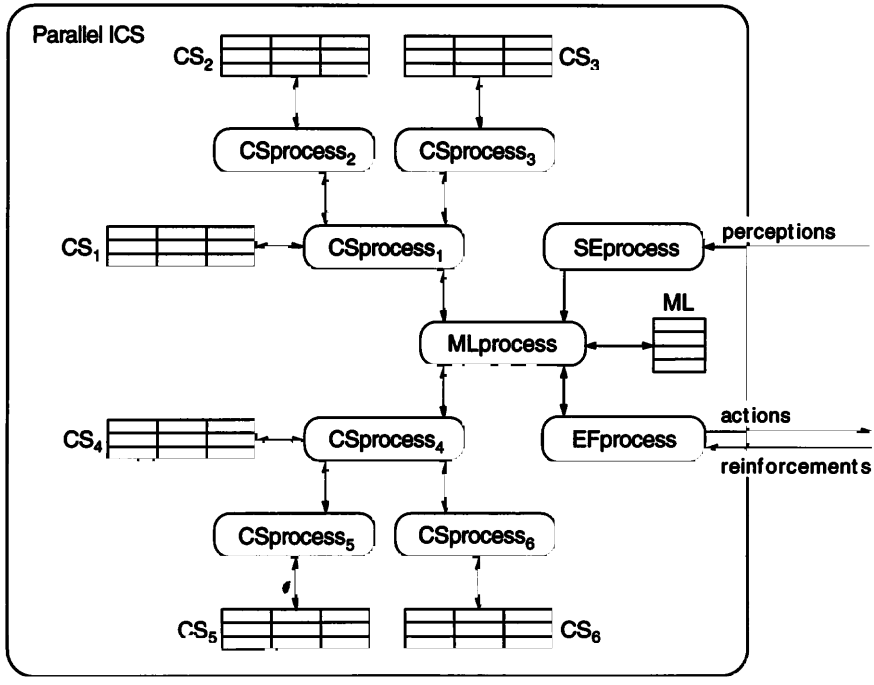


Figure 2.8
Parallel version of the ICS. In this example, CSprocess of figure 2.7 is split into six concurrent processes: $\text{CSprocess}_1, \ldots, \text{CSprocess}_6$.

2.3.1.2 The Rule Discovery System

We now illustrate briefly our parallel implementation of the rule discovery system—the genetic algorithm or GA. A first process, GAprocess, can be assigned the duty to select from among CS elements those individuals that are to be either replicated or discarded. It will be up to the (split) CSprocess, after receiving GAprocess decisions, to apply genetic operators, each single CSprocess_i focusing upon its own fraction of CS population.

Because it could affect CS strengths, upon which genetic selection is based, MLprocess stays idle during GA operations. Likewise, GAprocess is "dormant" when MLprocess works. Our parallel version of the genetic algorithm is reported in the algorithm box 2.5.

Considering our parallel implementation, step 1 may be seen as an auction, which can be distributed over the processor network by a

Algorithm Box 2.5
Parallel genetic algorithm in ALECSYS

1. Each CSprocess$_i$ selects, within its own subset of classifiers, m rules to replicate and m to replace (note: m is a system parameter).
2. Each CSprocess$_i$ 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 $2m$ individuals within the overall CS population.
3. GAprocess sends the following data to each CSprocess$_i$ containing a *parent* classifier:
   identifier of the parent itself,
   identifiers of the two offspring,
   crossover point.
   At the same time GAprocess sends to the CSprocess$_i$ containing a position for any *offspring* the following data:
   identifier of the offspring,
   identifiers of the two parents,
   crossover point.
4. All the CSprocesses that have parents in their own fraction of CS send a copy of the parent rule to the CSprocess$_i$ 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.

"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 because it requires a central management unit. Luckily, in LCS applications of the GA, the number of pairs is usually low and concurrency seems to be unnecessary (at least for moderate-sized populations). Step 3 is the hardest to parallelize because of the amount of communication it requires, both between MLprocess and the array of split CSprocesses and among CSprocesses themselves. Step 4 is a typical example of local data processing, extremely well suited to concurrent distribution.

2.3.2 High-Level Parallelism: A Solution to Behavioral Complexity Problems

In section 2.3.1 we presented a method for parallelizing a single ICS, with the goal of obtaining improvements in computing speed. Unfortunately, this approach shows its weakness when an ICS is applied to problems involving multiple tasks, which seems to be the case for most real problems. We propose to assign the execution of the various tasks to different

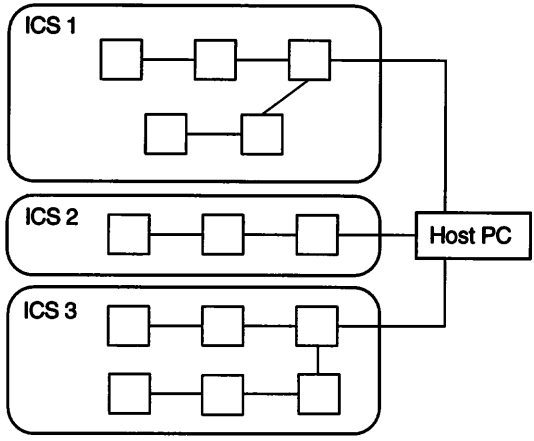


Figure 2.9
Example of concurrent high-level and low-level parallelism. Using ALECSYS, learning system designer divides problem into three ICSs (high-level parallelization) and maps each ICS onto subnet of nodes (low-level parallelization).

ICSs. 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 ICS. Adding a node to the transputer network implementing the ICS makes the communication load grow faster than computational power, which results in a less than linear speedup. Adding processors to an existing network is thus decreasingly effective. As already said, a better way to deal with complex problems would be to code them as a set of easier subproblems, each one allocated to an ICS. In ALECSYS the processor network is partitioned into subsets, each having its own size and topology. To each subset is allocated a single ICS, which can be parallelized at a low level (see figure 2.9). Each of these ICSs learns to solve a specific subgoal, according to the inputs it receives. Because 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 ICSs. The design approach to task decomposition is common to most of the current research on autonomous systems (see, for example, Mahadevan and Connell 1992; and Lin 1993a, 1993b), and will be discussed in chapter 3.

2.4 POINTS TO REMEMBER

- The learning classifier system (LCS) is composed of three modules: (1) the performance system; (2) the credit apportionment system; and (3) the rule discovery system.
- The LCS can be seen as a way of automatically synthesizing a rule-based controller for an autonomous agent. The synthesized controller consists of the performance system alone.
- The learning components in an LCS are the credit apportionment system and the rule discovery system. The first evaluates the usefulness of rules, while the second discovers potentially useful rules.
- ICS (improved classifier system) is an improved version of LCS_0, a particular instantiation of an LCS. ICS's major improvements are (1) the GA is called only when the bucket brigade has reached a steady state; (2) a new genetic operator called "mutespec" is introduced to overcome the negative effects of oscillating classifiers; and (3) the number of rules used by the performance system is dynamically reduced at runtime.
- ALECSYS is a tool that allows for the distribution of ICSs on coarse-grained parallel computers (transputers in the current implementation). By means of ALECSYS, a designer can take advantage of both parallelization of a single ICS and cooperation among many ICSs.

Chapter 3
Architectures and Shaping Policies

3.1 THE STRUCTURE OF BEHAVIOR

As we have noted in chapter 1, behavior is the interaction of the agent with its environment. Given that our agents are internally structured as an LCS and that LCSs operate in cycles, we might view behavior as a chain of elementary interactions between the agent and the environment, each of which is caused by an elementary action performed by the agent.

This is, however, an oversimplified view of behavior. Behavior is not just a flat chain of actions, but has a *structure*, which manifests itself in basically two ways:

1. The chain of elementary actions can be segmented in a sequence of different segments, each of which is a chain of actions aimed at the execution of a specific task. For example, an agent might perform a number of actions in order to reach an object, then a number of actions to grasp the object, and finally a chain of actions to carry the object to a container.
2. A single action can be made to perform two or more tasks simultaneously. For example, an agent might make an action in a direction that allows it both to approach a goal destination and to keep away from an obstacle.

Following well-established usage, we shall refer to the actions aimed at accomplishing a specific task as a "behavior"; we shall therefore speak of activities such as approaching an object, avoiding obstacles, and the like as "specific behaviors."

Some behaviors will be treated as *basic* in the sense that they are not structured into simpler ones. Other behaviors, seen to be made up of simpler ones, will be regarded as *complex behaviors*, whose component behaviors can be either basic or complex.

Complex behaviors can be obtained from simpler behaviors through a number of different composition mechanisms. In our work, we consider the following mechanisms:

- *Independent sum*: Two or more behaviors involving independent effectors are performed at the same time; for example, an agent may assume a mimetic color while chasing a prey. The independent sum of behaviors α and β will be written as

$$\alpha|\beta.$$

- *Combination*: Two or more behaviors involving the same effectors are combined into a resulting one; for example, the movement of an agent following a prey and trying to avoid an obstacle at the same time. The combination of behaviors α and β will be written as

$$\alpha+\beta.$$

- *Suppression*: A behavior suppresses a competing one; for example, the agent may give up chasing a prey in order to escape from a predator. When behavior α suppresses behavior β, we write

$$\frac{\alpha}{\beta}.$$

- *Sequence*: A behavior is built as a sequence of simpler ones; for example, fetching an object involves reaching the object, grasping it, and coming back. The sequence of behavior α and behavior β will be written as

$$\alpha\cdot\beta;$$

moreover, we shall write "σ^*" when a sequence σ is repeated for an arbitrary number of times.

In general, several mechanisms can be at work at the same time. For example, an agent could try to avoid fixed hurting objects while chasing a moving prey and being ready to escape if a predator is perceived.

With respect to behavior composition, the approach we have adopted so far is the following. First, the agent designer specifies complex behaviors in terms of their component simpler behaviors. Second, the designer defines an architecture such that each behavior is mapped onto a single *behavioral module*, that is, a single LCS of ALECSYS. Third, the agent is trained according to a suitable *shaping policy*, that is, a strategy for making the agent learn the complex behavior in terms of its components. In the next sections, we review the types of architectures we have used in

our experiments and analyze the ways in which they can match different classes of behaviors.

3.2 TYPES OF ARCHITECTURES

By using ALECSYS, the control system of an agent can be implemented by a network of different LCSs. It is therefore natural to consider the issue of *architecture*, by which we mean the problem of designing the network that best fits some predefined class of behaviors. As we shall see, there are interesting connections between the structure of behavior, the architecture of controllers, and shaping policies (i.e., the ways in which agents are trained). In fact, the structure of behavior determines a "natural" architecture, which in turn constrains the shaping policy.

Within ALECSYS, realizable architectures can be broadly organized in two classes: *monolithic architectures*, which are built by one LCS directly connected to the agent's sensors and effectors; and *distributed architectures*, which include several interacting LCSs. Distributed architectures, in turn, can be more or less "deep"; that is, they can be *flat architectures*, in which all LCSs are directly connected to the agent's sensors and effectors, or *hierarchical architectures*, built by a hierarchy of levels.

In principle, any behavior that can be implemented by ALECSYS can be implemented through a monolithic architecture, although distributed architectures allow for the allocation of specific tasks to different behavioral modules, thus lowering the complexity of the learning task. At the present stage, architectures cannot be learned by ALECSYS, but have to be designed; in other words, an architecture serves to include a certain amount of a priori knowledge into our agents so that their learning effort is reduced. To reach this goal, however, the architecture must correctly match the structure of the global behavior to be learned.

3.2.1 Monolithic Architectures

The simplest choice is the monolithic architecture, with one LCS in charge of controlling the whole behavior (figure 3.1). As we have already noted, this kind of architecture in principle can implement any behavior of which ALECSYS is capable, although for nontrivial behaviors the complexity of learning a monolithic controller can be too high.

Using a monolithic architecture allows the designer to put into the system as little initial domain knowledge as possible. Such initial knowledge

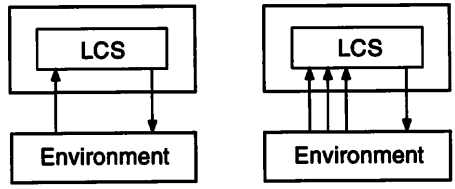


Figure 3.1
Monolithic architectures

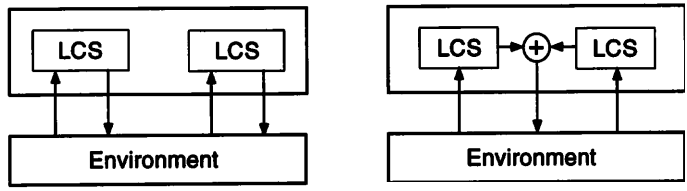


Figure 3.2
Flat architectures

is embedded in the design of the sensorimotor interface, that is, in the structure of messages coming from sensors and going to effectors.

If the target behavior is made up of several basic responses, and if such basic responses react only to part of the input information, the complexity of learning can be reduced even by using a monolithic architecture. Instead of wrapping up the state of all sensors in a single message (figure 3.1a), one can distribute sensor input into a set of independent messages (figure 3.1b). We call the latter case "monolithic architecture with distributed input." The idea is that inputs relevant to different responses can go into distinct messages; in such a way, input messages are shorter, the search space of different classifiers is smaller, and the overall learning effort can be reduced (see the experiment on monolithic architecture with distributed input in section 4.2). The only overhead is that each input message has to be tagged so that the sensor it comes from can be identified.

3.2.2 Flat Architectures

A distributed architecture is made up of more than one LCS. If all LCSs are directly connected to the agent's sensors, then we use the term *flat architecture* (figure 3.2). The idea is that distinct LCSs implement the dif-

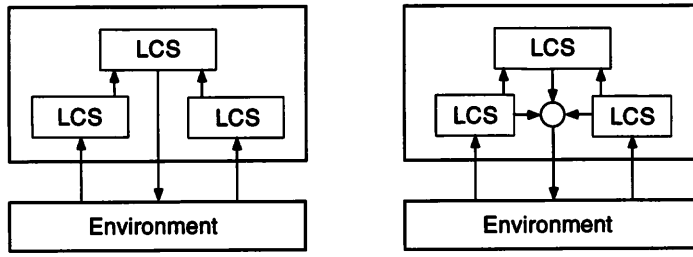


Figure 3.3
Two-level hierarchical architectures

ferent basic responses that make up a complex behavior pattern. There is a further issue, here, regarding the way in which the agent's response is built up from the actions proposed by the distinct LCSs. If such actions are independent, they can be realized in parallel by different effectors (figure 3.2a); for example, by rotating a videocamera while moving straight on. Actions that are not independent, on the other hand, have to be integrated into a single response before they are realized (figure 3.2b). For example, the action suggested by an LCS in charge of following an object and the action suggested by an LCS in charge of avoiding obstacles have to be combined into a single physical movement. (One of the many different ways of doing this might be to regard the two actions as vectors and add them.) In the present version of ALECSYS, the combination of different actions in a flat architecture cannot be learned but has to be programmed in advance.

3.2.3 Hierarchical Architectures

In a flat architecture, all input to LCSs comes from the sensors. In a *hierarchical architecture*, however, the set of all LCSs can be partitioned into a number of *levels*, with lower levels allowed to feed input into higher levels.

By definition, an LCS belongs to level N if it receives input from systems of level $N-1$ at most, where level 0 is defined as the level of sensors. An N-level hierarchical architecture is a hierarchy of LCSs having level N as the highest one; figure 3.3 shows two different two-level hierarchical architectures. In general, first-level LCSs implement basic behaviors, while higher-level LCSs implement coordination behaviors.

With an LCS in a hierarchical architecture we have two problems: first, how to receive input from a lower-level LCS; second, what to do with the

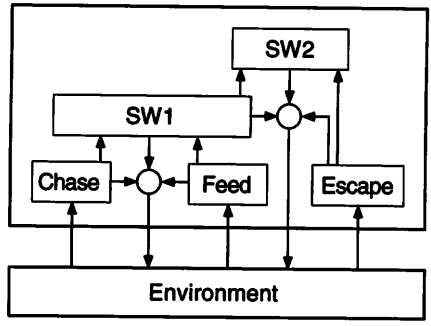


Figure 3.4
Example of three-level switch architecture for `Chase/Feed/Escape` behavior. Besides three basic behaviors, there are two switches, SW1 and SW2.

output. Receiving input from a lower-level LCS is easy: both input and output messages are bit strings of some fixed length, so that an output message produced by an LCS can be treated as an input message by a different LCS.

The problem of deciding what to do with the output of LCSs is more complex. In general, the output messages from the lower levels go to higher-level LCSs, while the output messages from the higher levels can go directly to the effectors to produce the response (see figure 3.3a) or can be used to control the composition of responses proposed by lower LCSs (see figure 3.3b). Most of the experiments presented in this book were carried out using suppression as composition rule; we call the resulting hierarchical systems "switch architectures." In figure 3.4, we show an example of three-level switch architecture implementing an agent that has to learn a `Chase/Feed/Escape` behavior, that is, a complex behavior taking place in an environment containing a "sexual mate," pieces of food, and possibly a predator. This can be defined as

```
if there is a predator
  then Escape
  else if hungry
         then Feed {i.e., search for food}
         else Chase {i.e., try to reach the sexual
             mate}
       endif
endif.
```

In this example, the coordinator of level two (SW1) should learn to suppress the `Chase` behavior whenever the `Feed` behavior proposes an action, while the coordinator of level three (SW2) should learn to suppress SW1 whenever the `Escape` behavior proposes an action.

3.3 REALIZING AN ARCHITECTURE

3.3.1 How to Design an Architecture: Qualitative Criteria

The most general criterion for choosing an architecture is to make the architecture naturally match the structure of the target behavior. This means that each basic response should be assigned an LCS, and that all such LCSs should be connected in the most natural way to obtain the global behavior.

Suppose the agent is normally supposed to follow a light, while being ready to reach its nest if a specific noise is sensed (revealing the presence of a predator). This behavior pattern is made up of two basic responses, namely, following a light and reaching the nest, and the relationship between the two is one of suppression. In such a case, the switch architecture is a natural choice.

In general, the four mechanisms for building complex behaviors defined in section 3.1 map onto different types of architecture as follows:

- *Independent sum* can be achieved through a flat architecture with independent outputs (figure 3.2a). This choice is the most natural because if two behaviors exploit different effectors, they can run asynchronously in parallel.
- *Combination* can be achieved either through a flat architecture with integrated outputs (figure 3.2b) or through a hierarchical architecture (figure 3.3). A flat architecture is advisable when the global behavior can be reduced to a number of basic behaviors that interact in a uniform way; for example, by producing a linear combination of their outputs. As we have already pointed out, the combination rule has to be explicitly programmed. On the other hand, if the agent has to learn the combination rule, the only possible choice with ALECSYS is to use a hierarchical architecture: two LCSs can send their outputs to a higher-level LCS in charge of combining the lower-level outputs into the final response. However, complex combination rules may be very difficult to learn.
- *Suppression* can be achieved through a special kind of hierarchical architecture we have called "switch architecture." Suppression is a simple limit

case of combination: two lower-level outputs are "combined" by letting one of the two become the final response. In this case, the combination rule can be effectively learned by ALECSYS, resulting in a switch architecture.

• *Sequence* can be achieved either through a flat or switch architecture or through a hierarchical architecture with added internal states. As we shall see in chapter 6, there are actually two very different types of sequential behaviors. The first type, which we have called "pseudo-sequential behavior," occurs when the perception of changes in the external environment is sufficient to correctly chain the relevant behaviors. For example, exploring a territory, locating food, reaching for it, and bringing it to a nest is a kind of pseudo-sequential behavior because each behavior in the sequence can be chosen solely on the basis of current sensory input. This type of behavior is not substantially different from nonsequential behavior and can be implemented by a flat or switch architecture. Proper sequential behavior occurs when there is not enough information in the environment to chain the basic behaviors in the right way. For example, an agent might have to perform a well-defined sequence of actions that leave no perceivable trace on the environment, like moving back and forth between two locations. This type of behavior requires some kind of "internal state" to keep track of the different phases the agent goes through. We have implemented proper sequential behavior by adding internal states to agents, and by allowing an LCS to update such a state within a hierarchical architecture (see chapter 6).

3.3.2 How to Design an Architecture: Quantitative Criteria

The main reason for introducing an architecture into LCSs is to speed up the learning of complex behavior patterns. Speedup is the result of factoring a large search space into smaller ones, so that a distributed architecture will be useful only if the component LCSs have smaller search spaces than a single LCS able to perform the same task.

We can turn this consideration into a quantitative criterion by observing that the size of a search space grows exponentially with the length of messages. This implies that a hierarchical architecture can be useful only if the lower-level LCSs realize some kind of informational abstraction, thus transforming the input messages into shorter ones (see, for example, the experiment on the two-level switch architecture in section 4.4.3). Consider an architecture in which a basic behavioral module receives from its sensors four-bit messages saying where a light is located. If this basic behavioral module sends the upper level four-bit messages indicat-

ing the proposed direction of motion, then the upper level can use the sensorial information directly, by-passing the basic module. Indeed, even if this basic behavioral module learns the correct input-output mapping, it does not operate any information abstraction, and because it sends the upper level the same number of bits it receives from its sensors, it makes the hierarchy computationally useless. We call this the "information compression" requirement.

An LCS whose only role is to compress sensory messages can be viewed as a "virtual sensor," able to extract from physical sensory data a set of relevant features. As regards the actual sensors, we prefer to keep them as simple as possible, even in the simulated experiments: all informational abstraction, if any, takes place within ALECSYS and has to be learned. In this way, we make sure that the "intelligence" of our agents is located mainly within ALECSYS and that simulated behaviors can be also be achieved by real robots endowed with low-cost physical sensors.

3.3.3 Implementing an Architecture with ALECSYS

With ALECSYS it is possible to define two classes of learning modules: *basic behaviors* and *coordination behaviors*. Both are implemented as learning classifier systems.

Basic behaviors are directly interfaced with the environment. Each basic behavior receives bit strings as input from sensors and sends bit strings to effectors to propose actions. Basic behaviors can be inserted in a hierarchical architecture; in this case, they also send bit strings to connected higher-level coordination modules.

Consider AutonoMouse II and the complex behavior `Chase/Feed/Escape`, which has already been mentioned in section 2.3 and will be extensively investigated in chapter 4. Figure 3.5 shows one possible innate architecture of an agent with such a complex behavior. A basic behavior has been designed for each of the three behavioral patterns used to describe the learning task. In order to coordinate basic behaviors in situations where two or more of them propose actions simultaneously, a coordination module is used. It receives a bit string from each connected basic behavior (in this case a one-bit string, the bit indicating whether the sending ICS wants to do something or not) and proposes a *coordination action*. This coordination action goes into the composition rule module, which implements the composition mechanism. In this example, the composition rule used is suppression, and therefore only one of the basic actions proposed is applied.

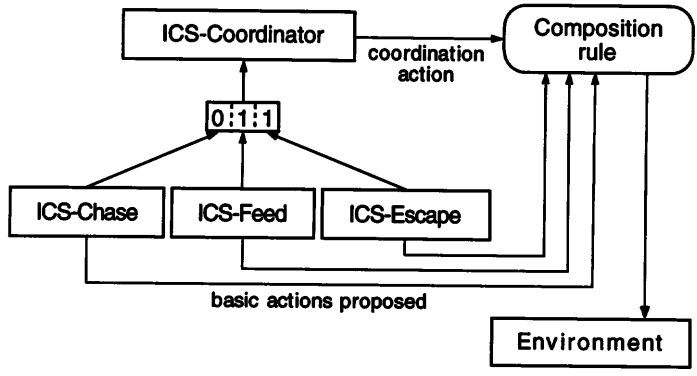


Figure 3.5
Example of innate architecture for three-behavior learning task

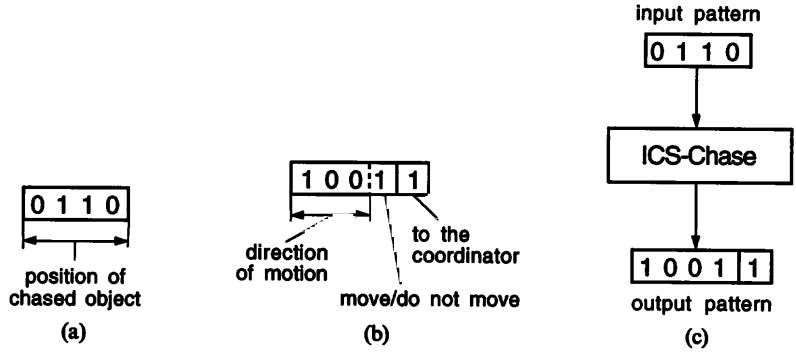


Figure 3.6
(a) Example of input message; (b) example of output message; (c) example of input-output interface for ICS-Chase behavior

All behaviors (both basic and coordination), are implemented as ICSs. For example, the basic behavioral pattern Chase is implemented as an ICS, which for ease of reference we call "ICS-Chase" (figure 3.5). Figure 3.6 shows the input-output interface of ICS-Chase. In this case, the input pattern only says which sensors see the light. (AutonoMouse II has four binary sensors, each of which is set to 1 if it detects a light intensity higher than a given threshold, and to 0 otherwise.) The output pattern is composed of a proposed action, a direction of motion plus a move/do not move command, and a bit string (in this case of length 1) going to ICS-Coordinator (figure 3.5). The bit string is there to let the coordinator

know when ICS-Chase is proposing an action. Note that the value of this bit string is not designed but must be learned by ICS-Chase.

In general, coordination behaviors receive input from lower-level behavioral modules and produce an output action that, with different modalities depending on the composition rule used, influences the execution of actions proposed by basic behaviors.

3.4 SHAPING POLICIES

The use of a distributed system, either flat or hierarchical, brings in the new problem of selecting a *shaping policy*, that is, the order in which the various tasks are to be learned. In our work, we identified two extreme choices:

- *Holistic shaping*, in which the whole learning system is considered as a black box. The actual behavior of a single learning classifier system is not used to evaluate how to distribute reinforcements; when the system receives a reinforcement, it is given to all of the LCSs within the system architecture. This procedure has the advantage of being independent of the internal architecture, but can make the learning task difficult because there can be ambiguous situations in which the trainer cannot give the correct reinforcement to the component LCSs. A correct action might be the result of two wrong messages. For example, in our Chase/Feed/Escape task (see figure 3.4), the LCS called SW1 might in a feeding situation choose to give control to the wrong basic LCS, that is, LCS-Chase, which in turn would propose a normally wrong action, "Move away from the light," that, by chance, happened to be correct. Nevertheless, this method of distributing reinforcements is interesting because it does not require access to the internal modules of the system and is much more plausible from an ethological point of view.
- *Modular shaping*, in which each LCS is trained with a different reinforcement program, taking into account the characteristics of the task that the considered LCS has to learn. We have found that a good way to implement modular shaping is first to train the basic LCSs, and then, after they have reached a good performance level, to "freeze" them (that is, basic LCSs are no longer learning, but only performing) and to start training upper-level LCSs. Training basic LCSs is usually done in a separate session (in which training of different basic LCSs can be run in parallel).

In principle, training different LCSs separately, that is, modular shaping, makes learning easier, although the shaping policy must be designed in a sensible way. Hierarchical architectures are particularly sensitive to the shaping policy; indeed, it seems reasonable that the coordination modules be shaped after the lower modules have learned to produce the basic behaviors. The experiments in chapter 4 on two-level and three-level switch architectures show that good results are obtained, as we said above, by shaping the lower LCSs, then freezing them and shaping the coordinators, and finally letting all components be free to go on learning together.

3.5 POINTS TO REMEMBER

- Complex behaviors can be obtained from simpler behaviors through a number of composition mechanisms.
- When using ALECSYS to build the learning control system of an autonomous agent, a choice must be made regarding the system architecture. We have presented three types of architectures: monolithic, distributed flat, and distributed hierarchical.
- The introduction of an LCS module into an architecture must satisfy an "information compression" criterion: the input string to the LCS must be longer than the output string sent to the upper-level LCS.
- When designing an architecture, a few qualitative criteria regarding the matching between the structure of the behavioral task to be learned and the type of architecture must be met. In particular, the component LCSs of the architecture should correspond to well-specified behavioral tasks, and the chosen architecture should naturally match the structure of the global target behavior.
- Distributed architectures call for a shaping policy. We presented two extreme choices: holistic and modular shaping.

Chapter 4
Experiments in Simulated Worlds

4.1 INTRODUCTION

In this chapter we present some results obtained with simulated robots. Because it takes far less time to run experiments in simulated worlds than in the real world, a much greater number of design possibilities can be tested.

The following questions guided the choice of the experiments:

- Does ICS perform better than a more standard LCS like LCS_0?
- Can a simple form of memory help the AutonoMouse to solve problems that require it to remember what happened in the recent past?
- Does decomposition in subtasks help the learning process?
- How must shaping be structured? Can basic behaviors and coordination of behaviors be learned at the same time, or is it better to split the learning process into several distinct phases?
- Is there any relation between the agent's architecture and the shaping policy to be used?
- Can an inappropriate architecture impede learning?
- Can the different architectural approaches we used in the first experiments be composed to build more complex hierarchical structures?
- Can a bad choice of sensor granularity with respect to the agent's task impede learning?

In the rest of this section, we explain our experimental methodology, illustrate the simulated environments we used to carry out our experiments, and describe the simulated AutonoMouse we used in our experiments. In section 4.2, we present experiments run in a simple environment in which our simulated AutonoMouse had to learn to chase a moving light source. The aim of these experiments was to test ICS and ALECSYS,

and to study a simple memory mechanism we call "sensor memory." In section 4.3, we give a rather complete presentation of a multibehavior task and discuss most of the issues that arise when dealing with distributed architectures. In particular, we focus on the interplay between the shaping policy and the architecture chosen to implement the learning system. In section 4.4, we slightly complicate the overall learning problem by adding a new task, feeding, and by augmenting the dimension of the sensory input, making sensor messages longer and adding multiple copies of the same class of objects in the environment. We report extensively on experiments comparing different architectural choices and shaping policies. Section 4.5 briefly reports on an experiment run in a slightly more complex environment which will be repeated, using this time a real robot, in chapter 5.

4.1.1 Experimental Methodology

In all the experiments in simulated worlds, we use

$$P = \frac{\text{Number of correct responses}}{\text{Total number of responses}} \leq 1$$

as performance measure. That is, performance is measured as the ratio of correct moves to total moves performed from the beginning of the simulation. Note that the notion of "correct" response is implicit in the RP: a response is correct if and only if it receives a positive reinforcement. Therefore, we call the above defined ratio the "cumulative performance measure induced by the RP."

The experiments reported in sections 4.2 and 4.3 were run as follows. Each experiment consisted of a learning phase and a test session. The test session was run after the learning phase; during the test session the learning algorithm was switched off (but actions were still chosen probabilistically, using strength). 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 was computed only for the moves in which the behaviors were active; we computed the global performance as the ratio of globally correct moves to total moves during the whole test session, where at every cycle (the interval between two sensor readings) a globally correct move was a move correct with respect to the current goal. Thus, for example, if after ten cycles the `Chase` behavior had been active for 7 cycles, proposing a correct move 4 times, and the `Escape` behavior had been active for 6 cycles, proposing a correct move 3 times, then the `Chase` behavior performance was 4/7, the

`Escape` behavior performance 3/6, and the global performance $(4+3)/(7+6)=7/13$. To compare the performances of different architectural or shaping policies, we used the Mann-Whitney nonparametric test (see, for example, Siegel and Castellan 1988), which can be regarded as the nonparametric counterpart of Student's t-test for independent samples. The choice of nonparametric statistics was motivated by the failure of our data, even if measured on a numeric scale, to meet the requirements for parametric statistics (for example, normal distribution).

The experiments presented in section 4.4 were run somewhat differently. First, instead of presetting a fixed number of learning cycles, we ran an experiment until there was at least some evidence that the performance was unlikely to improve further; this evidence was collected automatically by a steady state monitor routine, checking whether in the last k cycles the performance had significantly changed. In experiments involving multiphase shaping strategies, a new phase was started when the steady state monitor routine signaled that learning had reached a steady state. Second, experiments were repeated several times (typically five), but typical results were graphed instead of average results. In fact, the use of the steady state monitor routine made it difficult to show averaged graphs because new phases started at different cycles in different experiments. Nevertheless, all the graphs obtained were very similar, which makes us confident that the typical results we present are a good approximation of the average behavior of our learning system. The reason we used this different experimental approach is that we wanted to have a system that learned until it could profit from learning, and then was able to switch automatically to the test session.

Finally, section 4.5 presents simple exploratory experiments run to provide a basis of comparison for those to be run with a real robot (see section 5.3). In these experiments we did not distinguish between a learning and a test session, but ran a single learning session of appropriate length (50,000 cycles in the first experiments and 5,000 in the second).

4.1.2 Simulation Environments

From chapter 3 it is clear that to test all the proposed architectures and shaping policies, we need many different simulated worlds. As we need a basic task for each basic behavior, in designing the experimental environments, we were guided by the necessity of building environments in which basic tasks, and their coordination, could be learned by the tested agent architecture. We used the following environments:

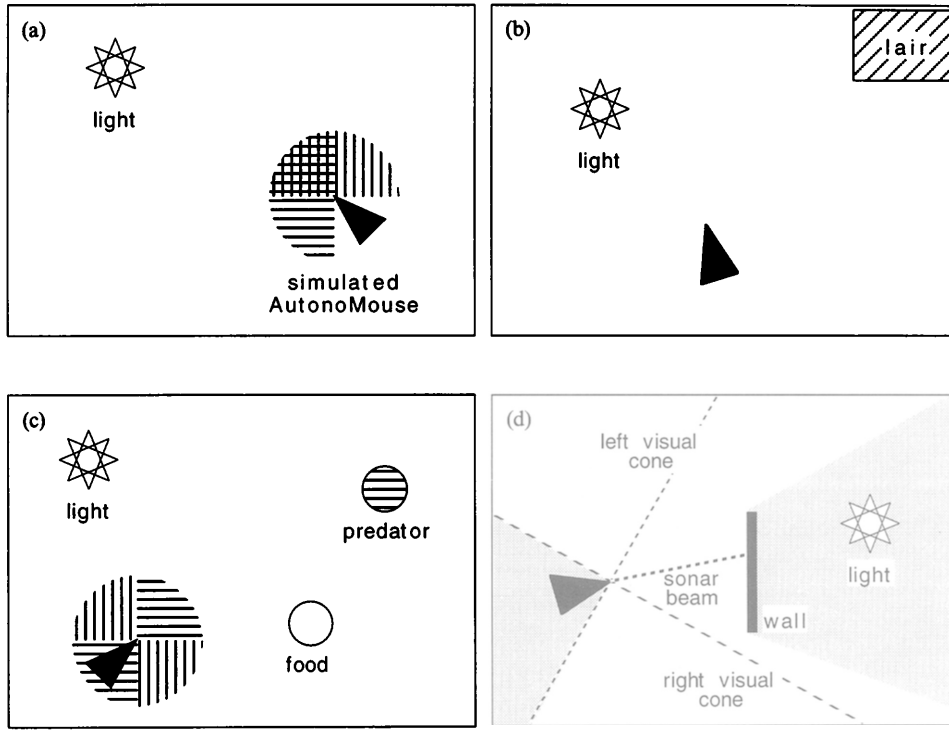


Figure 4.1
Simulated environment setup: (a) `Chase` environment: the AutonoMouse and its visual sensors; (b) `Chase/Escape` environment; (c) `Chase/Feed/Escape` environment; and (d) `FindHidden` environment: AutonoMouse does not see light when it is in the shaded area.

- `Chase` environment (single-behavior environment).
- `Chase/Escape` environment (two-behavior environment).
- `Chase/Feed/Escape` environment (three-behavior environment).
- `FindHidden` environment (two-behavior environment).

In the `Chase` environment (see figure 4.1a, in which the AutonoMouse is shown with its two binary eyes, which each cover a half-space), the task is to learn to chase a moving object. This environment was studied primarily to test the learning classifier system capabilities and as a test bed for proposed improvements in the LCS model. In particular, the issue was whether the use of punishments is a good thing; we also tested the optimal length for the message list, a parameter that can greatly influence the performance of the learning system. The `Chase` environment was used to

fine-tune the performance of ICS. The environment dimensions, for this and all the following environments, were 640 × 480 pixels, an AutonoMouse's step was set to 3 pixels, and the maximum speed of the AutonoMouse was the same as the speed of the light source.

In the `Chase/Escape` environment there are two objects: a light and a lair. There is also a noisy predator, which appears now and then, and which can be heard, but not seen, by the AutonoMouse. When the predator appears, the AutonoMouse, otherwise predisposed to chase the moving light source, must run into its lair and wait there until the predator has gone. The light source is always present. The predator appears at random time intervals, remains in the environment for a random number of cycles, and then disappears. The predator never enters the environment: it is supposed to be far away and its only effect is to scare the AutonoMouse, which runs to hide in its lair. Figure 4.1b shows a snapshot of the environment.

In the `Chase/Feed/Escape` environment there are three objects: a light, a food source, and a predator. This environment is very similar to the previous one, but for the presence of food and for the different robot-predator interaction. As above, the AutonoMouse is predisposed to chase the moving light source. When its distance from the food source is less than a threshold, which we set to 45 pixels, then the AutonoMouse feels hungry and thus focuses on feeding. When the chasing predator appears, then the Autonomouse's main goal is to run away from it (there is no lair in this environment). The maximum speed of the AutonoMouse is the same as the speed of the light source and of the chasing predator. The light source and the food are always present (but the food can be seen only when closer than the 45 pixels threshold). As above, the predator appears at random time intervals, remains in the environment for a random number of cycles, and then disappears. Figure 4.1c shows a snapshot of the environment.

In the `FindHidden` environment the AutonoMouse's task is composed of two subtasks: `Chase` and `Search`. As in the `Chase` environment, the AutonoMouse has to chase a moving light source. The task is complicated by the presence of a wall. Whenever it is interposed between the light and the AutonoMouse (see figure 4.1d), the AutonoMouse cannot see the light any longer, and must `Search` for the light.

4.1.3 The Simulated AutonoMice

The simulated AutonoMice are the simulated-world counterparts of the real AutonoMice in the experiments presented in the next chapter. Their

limited capabilities are designed to let them learn the simple behavior patterns discussed in the section 4.1.2.

4.1.3.1 Sensory Capabilities

In all experimental environments except the `Chase/Feed/Escape` environment, the simulated AutonoMouse has two eyes. Because each eye covers a visual angle of 180 degrees and the two eyes have a 90-degree overlap in the AutonoMouse's "forward move" direction, the eyes partition the environment into four nonoverlapping regions (see figure 4.1a). AutonoMouse eyes can sense the position of the light source, of food (if it is closer than a given threshold), of the chasing predator, and of the lair (we say the AutonoMouse can "see" the light and the lair). We call these the "basic AutonoMouse sensor capabilities." In our work we chose a particular disposition of sensors we considered reasonable. Obviously, a robot endowed with different sensors or even a different disposition or granularity of existing sensors would behave differently (see, for example, Cliff and Bullock 1993).

In one of the experiments run in the `Chase` environment (the experiment on sensor memory), the two eyes of the AutonoMouse covered a visual cone of 60 degrees, overlapping for 30 degrees; as a result, the visual space in front of the AutonoMouse was 90 degrees, partitioned into three sectors of 30 degrees each. The reason to restrict the visual field of the AutonoMouse was to make more evident the role of memory in learning to solve the task.

In experiments run in the `Chase/Escape` environment, in addition to the basic sensor capabilities, the AutonoMouse can also detect the difference between a close light and a distant light (the same is true for the lair), and can sense the presence of the noisy predator (we say it can "hear" the predator), but cannot see it. The distinction between "close" and "far" was necessary because the AutonoMouse had to learn two different responses within the hiding behavior: "When the lair is far, approach it"; "When the lair is close [the AutonoMouse is into the lair], do not move." The distinction between "close" and "far" was not necessary for the chasing behavior, but was maintained for uniformity.

In experiments run in the `Chase/Feed/Escape` environment, the simulated AutonoMouse has four eyes (like AutonoMouse II; see figure 1.3). Because each eye covers a visual angle of 90 degrees without overlap, the four eyes partition the environment into four nonoverlapping regions. As in the case of basic sensor capabilities, AutonoMouse eyes can sense

the position of the light source, of the food (if it is closer than a given threshold), and of the chasing predator. The reason to have sensors with different characteristics from those in the previous environments is that we wanted to see whether there was any significant change in the AutonoMouse's performance when it was receiving longer than necessary sensory messages (it actually took twice the number of bits to code the same amount of sensory information using four eyes than it did using two eyes, as in the `Chase` case).

In the `FindHidden` environment, the AutonoMouse was enriched with a sonar and two side bumpers. Both the sonar and the bumpers were used as on/off sensors.

AutonoMice with two eyes require two bits to identify an object (light, lair, or predator) relative position, while AutonoMice with four eyes require four bits. In the experiments where the AutonoMouse differentiates between a close and a far region, one extra bit of information is added. Moreover, one bit of information is used for the sonar and one for each of the bumpers.

In the `Chase` environment, the AutonoMouse receives two bits of information from sensors to identify light position. In the `Chase/Escape` environment, the AutonoMouse receives 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. In the `Chase/Feed/Escape` environment the AutonoMouse receives twelve bits of information from sensors: four bits to identify light position, four bits to identify lair position, and four bits to identify predator position. In the `FindHidden` environment the AutonoMouse receives five bits of information from sensors: two bits to identify light position, one bit from the sonar, and one bit from each of the two side bumpers.

Table 4.1 summarizes the sensory capabilities of the AutonoMouse in the different environments. Please note that two bits of information were added to messages to distinguish between sensor messages, internal messages, and motor messages.

4.1.3.2 Motor Capabilities

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

Table 4.1
Sensory input in different experimental environments

	Light sensor	Lair sensor	Food sensor	Predator sensor	Sonar	Bumpers	Total
Chase environment	2 bits						2 bits
Chase/Escape environment	3 bits	3 bits		1 bit			7 bits
Chase/Feed/Escape environment	4 bits		4 bits	4 bits			12 bits
FindHidden environment	2 bits				1 bit	2 bit	5 bits

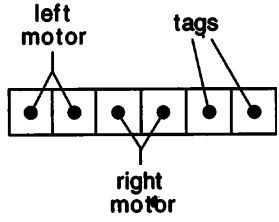


Figure 4.2
Structure of message going to motors

per cycle, was set to be the same as the speed of the moving light or of the chasing predator (it was found that setting a higher speed, say, 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. 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 therefore 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 figure 4.2.

In the case of the `FindHidden` experiment, we used a different coding for motor messages: the output interface included two bits, coding the following four possible moves: "still," "straight ahead," "ahead with a left turn," and "ahead with a right turn."

4.2 EXPERIMENTS IN THE `Chase` ENVIRONMENT

We ran experiments in the `Chase` environment to answer the following questions:[1]

Table 4.2
Comparison between rewards and punishments training policy and only rewards training policy

		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

Note: Results obtained in test session; 1,000 cycles run after 1,000 cycles of learning. Values reported are means of performance index P over 20 runs, and standard deviations.

- Is it better to use only positive rewards, or are punishments also useful?
- Are internal messages useful?
- Does ICS perform better than LCS_0?

In particular, we

- compared the results obtained using only positive rewards with those obtained using both positive rewards and punishments;
- compared the results obtained using message lists of different lengths;
- evaluated the usefulness of applying the GA when the bucket brigade has reached a steady state;
- evaluated the usefulness of the mutespec operator; and
- evaluated the usefulness of dynamically changing the number of classifiers used.

We also ran an experiment designed to provide the AutonoMouse with some simple dynamic behavior capabilities, using a general mechanism based on the idea of sensor memory.

4.2.1 The Role of Punishments and of Internal Messages

Our experimental results have shown (see table 4.2) that using rewards and punishments works better than using only rewards for any number of internal messages. Mann-Whitney tests showed that the differences observed between the first and the second row of table 4.2 are highly significant for every message list (ML) length tested ($p < 0.01$).

Results have also indicated that, for any given task, a one-message ML is the best. This is shown both by learning curves for the rewards and punishments policy (see figure 4.3) and by results obtained in the test session (see table 4.2, mean performance for different values of ML). We

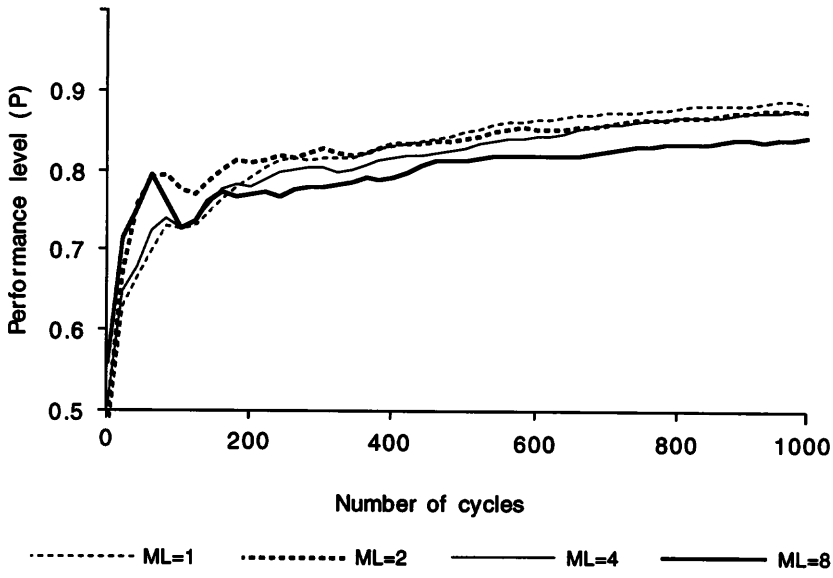


Figure 4.3
Learning to chase light source (chasing behavior). Comparison of different ML lengths, using rewards and punishments training policy; averages over 20 runs.

studied the statistical significance of the differences between adjacent means by applying the Mann-Whitney test. Samples were ordered by increasing ML length, and the choice of comparing adjacent samples was done a priori (which makes the application of the Mann-Whitney test a correct statistical procedure). The differences considered were everywhere found to be highly significant ($p < 0.01$) by the Mann-Whitney test except between ML = 2 and ML = 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. Although Holland's original LCS used internal messages, Wilson (1994) has recently proposed a minimal LCS, called "ZCS," which uses none, although to learn dynamic behavior, some researchers (Cliff and Ross 1994) have recently extended Wilson's ZCS by adding explicit memory (see also our experiments in chapter 6). Indeed, it is easy to understand that internal messages can slow down the convergence of the classifiers' strength to good values.

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 210 cycles (see figure 4.3; 100 cycles done in about 60 seconds

Table 4.3
Comparison of LCS_0 and LCS_0 + energy

Type of LCS used	NC	NGA	*P*
LCS_0	2	500	0.80
LCS_0	4	500	0.82
LCS_0	6	500	0.84
LCS_0	12	500	0.87
LCS_0	24	500	0.88
LCS_0	48	500	0.79
LCS_0	72	500	0.86
LCS_0 + energy	2	222	0.90
LCS_0 + energy	4	261	0.92
LCS_0 + energy	6	287	0.92
LCS_0 + energy	12	290	0.95
LCS_0 + energy	24	332	1
LCS_0 + energy	48	497	0.96
LCS_0 + energy	72	568	0.92

Note: NC = Number of classifiers introduced by application of GA. NGA = Number of cycles between two applications of GA. For LCS_0 + energy, we report average number of cycles. For LCS_0, we call GA every 500 iterations (500 was experimentally found to be good value). *P* = Performance index averaged over 10 runs.

with 300 rules on 3 transputers). This makes it feasible to use ALECSYS to control the real AutonoMouse in real time.

Given the results presented in this section, all the following experiments were run using punishments and rewards and a one-message ML.

4.2.2 Experimental Evaluation of ICS

4.2.2.1 Energy and Steady State

In table 4.3, we report some results obtained comparing LCS_0 with its derived ICS, which differs only in that the GA is called automatically when the bucket brigade algorithm has reached a steady state. The attainment of a steady state was measured by the energy variable, introduced in section 2.2.1. Similar results were obtained in a similar environment in which AutonoMouse had slightly different sensors (Dorigo 1993). The experiment was repeated for different values of the number of new

Table 4.4
Comparison between LCS_0 without and with dynamical change of number of classifiers

	Maximum performance P achieved	Iterations required to achieve $P = 0.67$	Time required to run 10,000 iterations (sec)	Average time of iteration (sec)
LCS_0	0.85	3,112	7,500	0.75
LCS_0 with dynamical change of number of classifiers	0.95	1,797	3,900	0.39

Note: 10,000 cycles per run; results averaged over 10 runs.

classifiers introduced in the population as a result of applying the GA because this was found to be a relevant parameter. Results show that the use of energy to identify a steady state, and the subsequent call of the GA when the steady state has been reached, improves LCS_0 performance, as measured by the P index. It is also interesting to note that, the number of cycles between two applications of the GA is almost always smaller for LCS_0 with GA called at steady state, which means that a higher number of classifiers will be tested in the same number of cycles.

4.2.2.2 The Mutespec Operator

Experiments were run to evaluate the usefulness of both the mutespec operator and the dynamical change of the number of classifiers used. Adding mutespec to LCS_0 resulted in an improvement of both the performance level achieved, and the speed with which that performance was obtained. On average, after 100 calls of the GA, the performance P with and without the mutespec operator was, respectively, 0.94 and 0.85 (the difference was found to be statistically significant). Also using mutespec, a performance $P = 0.99$ was obtained after 500 applications of the GA, while after the same number of GA calls, LCS_0 without mutespec reached a $P = 0.92$.

4.2.2.3 Dynamically Changing the Number of Classifiers Used

Experiments have shown (see table 4.4) that adding the dynamic change of the number of activatable classifiers to LCS_0 decreases the average time for a cycle by about 50 percent. 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.

4.2.3 A First Step toward Dynamic Behavior

The experiments described thus far concern S-R behavior, that is, direct associations of stimuli and responses. Clearly, the production of more complex behavior patterns involves the ability to deal with dynamic behavior, that is, with input-output associations that exploit some kind of internal state.

In a dynamic system, a major function of the internal state is memory. Indeed, the limit of S-R behavior is that it can relate a response only to the current state of the environment. It must be noted that ALECSYS is not completely without memory; both the strengths of classifiers and the internal messages appended to the message list embody information about past events. However, it is easy to think of target behaviors that require a much more specific kind of memory.

For example, if physical objects are still or move slowly with respect to the agent, their current position is strongly correlated with their previous position. Therefore, how an object was sensed in the past is relevant to the actions to be performed in the present, even if the object is not currently perceived. Suppose that at cycle t the agent senses a light in the leftmost area of its visual field, and that at cycle $t+1$ the light is no longer sensed. This piece of information is useful to chase the light because at cycle $t+1$ the light is likely to be out of the agent's visual field on its left.

To study whether chasing a light can be made easier by a memory of past perceptions, we have endowed our learning system with a *sensor memory*, that is, a kind of short-term memory of the state of the AutonoMouse's sensors. In order to avoid an ad hoc solution to our problem, we have adopted a sensor memory that functions uniformly for all sensors, regardless of the task. The idea was to provide the AutonoMouse with a representation of the previous state of each sensor, for a fixed period of time. That is, at any given time t the AutonoMouse can establish, for each sensor S, whether

(i) the state of S has *not* changed during the last k cycles (where the *memory span* k is a parameter to be set by the experimenter);
(ii) the state of S *has* changed during the last k cycles; in this case, enough information is given so that the previous state of S can be reconstructed.

This design allows us to define a sensor memory that depends on the input interface, but is independent of the target behavior (with the exception of k, whose optimal value can be a function of the task). More precisely, the sensor memory is made up of

- a memory word, isomorphic to the input interface;
- an algorithm that updates the memory word at each cycle, in accordance to specifications (i) and (ii), on the basis of the current input, the previous memory word, and the number of cycles elapsed from the last change;
- a mechanism that appends the memory word to the message list, with a specific tag identifying it as a memory message.

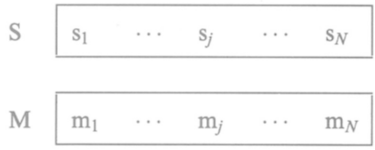
The memory process is described in more detail in algorithm box 4.1. Note that, coherently with our approach, the actual "meaning" of memory messages must be learned by the classifier systems. In other words, memory messages are just one more kind of messages, whose correlation with the overall task has to be discovered by the learning system.

The results obtained in a typical simulation are reported in figures 4.4–4.6, which compare the performances of the AutonoMouse with and without memory. The memory span was $k = 10$.

Figure 4.4 shows that the performance of the AutonoMouse with memory tends to become asymptotically better than the performance of the AutonoMouse without memory, although the learning process is slower. This is easy to explain: the "intellectual task" of the AutonoMouse with memory is harder because the role of the memory messages has to be learned; on the other hand, the AutonoMouse can learn about the role of memory only when it becomes relevant, that is, when the light disappears from sight—and this is a relatively rare event.[2] To show that the role of memory is actually relevant, we have decomposed AutonoMouse's performance into the performance produced when the light is not visible (and therefore memory is relevant; see figure 4.5); and the performance when the light is visible (and thus memory is superfluous; see figure 4.6).[3] It can be seen that in the former case the performance of the AutonoMouse with memory is better.

We conclude that even a very simple memory system can improve the performance of ALECSYS in cases where the target behavior is not intrinsically S-R.

Algorithm Box 4.1
The sensor memory algorithm



S = *environment message*, coding the state of the sensors; each sensor is represented by a bit sequence s_j (N being the number of distinct sensors);
M = *memory message*; for each s_j, there is an m_j of equal length.

The memory message at time t, M(t), is computed from S(t), S($t-1$), and M($t-1$). Given a *memory span* k, the following algorithm is applied at each noninitial cycle t (at cycle $t = 0$, $m_j(0) := s_j(0)$):

```
for j from 1 to N do                    {for each sensor j}
   delta_s_j := s_j(t) xor s_j(t − 1)   {compute the change of s_j from
                                         t − 1 to t}
   if delta_s_j ≠ [0...0...0]           {if there is a change}
      then
         m_j(t) := delta_s_j            {set m_j to such a change}
         clock(j) := 0                  {and set the clock of sensor j to 0}
      else
         m_j(t) := m_j(t − 1)           {else set m_j to its previous value}
         clock(j) := clock(j) + 1       {and increment the clock of
                                         sensor j}
   endif
   if clock(j) > k                      {if the memory span of sensor j
                                         has elapsed}
      then
         m_j(t) := 0                    {set m_j to zero}
         clock(j) := 0                  {and set the clock of sensor j to
                                         zero}
   endif
endfor
```

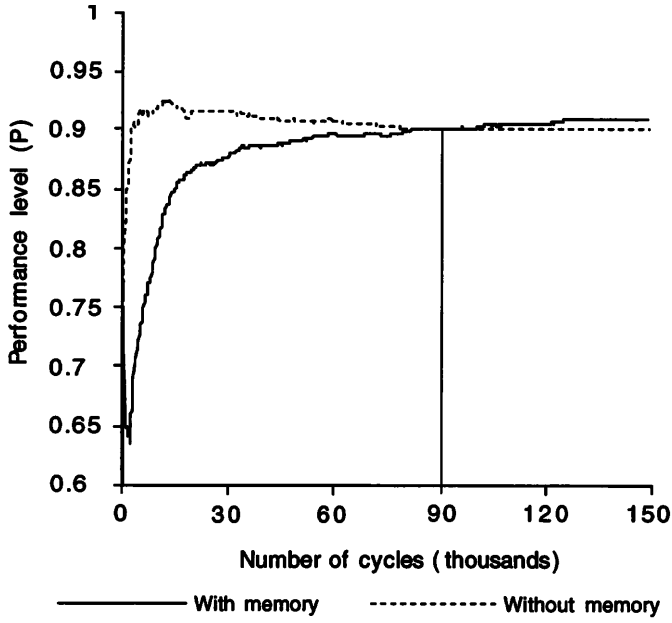


Figure 4.4
Chasing moving light with and without sensor memory

4.3 EXPERIMENTS IN THE Chase/Escape ENVIRONMENT

The central issue investigated in this section is the role that the system architecture and the reinforcement program play in making a learning system an efficient learner.

In particular, we compare different architectural solutions and shaping policies. The logical structure of the learning system consists, 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, is tested using both the monolithic architecture and the switch architecture. In the case of the switch architecture, we also compare holistic and modular shaping.

Another goal of this section is to help the reader understand how our experiments were run by giving a detailed specification of the objects in the simulation environment, the target behaviors, the learning architectures built using ALECSYS, the representation used, and, finally, the reinforcement program.

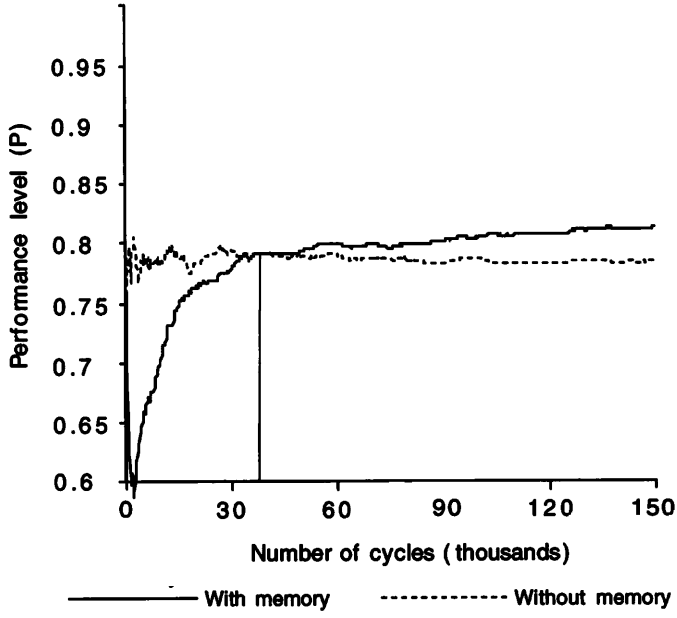


Figure 4.5
Light-chasing performance when light is not visible

4.3.1 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, which moves at a speed set 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, which occupies the upper right angle of the environment (see figure 4.1b).

4.3.2 Target Behavior

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

1. *Chasing behavior.* The simulated AutonoMouse likes to chase the moving light source.

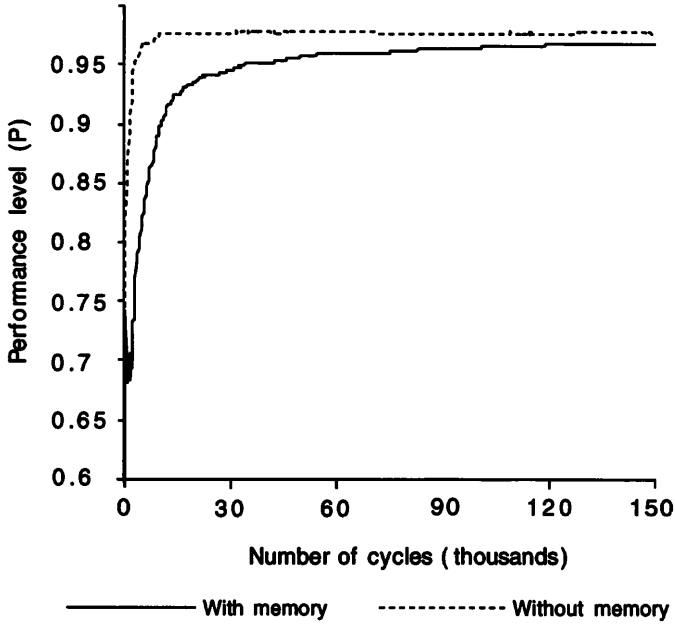


Figure 4.6
Light-chasing performance when light is visible

2. *Escaping 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.
3. *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 chases the light source; when it happens to hear a predator, it suddenly gives up chasing, runs to the lair, and stays there until the predator goes away.

In these simulations the AutonoMouse can hear, but cannot see, the predator. Therefore, to avoid conflict situations like having the predator between the agent and the lair, we make the implicit assumption that the AutonoMouse 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.

4.3.3 The Learning Architectures: Monolithic and Hierarchical

Experiments were run with both a monolithic architecture (see figure 3.1) and a switch architecture (see figures 3.3 and 3.4). In the monolithic architecture the learning system is implemented as a single low-level parallel learning classifier system (on three transputers), called "LCS-global." In the switch architecture, the learning system is implemented as a set of three classifier systems organized in a hierarchy: two classifier systems learn the basic behaviors (`Chase` and `Escape`), 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 chasing behavior, one transputer for the escaping behavior, and one transputer to learn to switch.

4.3.4 Representation

As we have seen in chapter 2, ALECSYS classifiers have 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 the longest among 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 used for sensory information and two bits for the tag, which identifies the message as being a sensor message.[4] (Therefore, a classifier will be $3 \cdot 9 = 27$ bits long.)

In the case of switch architecture, sensory information is split to create two messages, a five-bit message for the chasing 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 escaping 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 LCSs and switches. At every cycle each basic LCS sends, besides the message directed to motors if that is the case, a one-bit message to the switch. This one-bit message is intended to signal the switch the intention to propose an action. A message composed by all the bits coming from basic LCSs (a two-bit

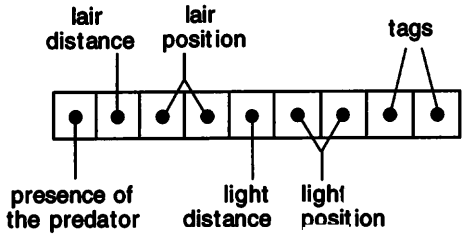


Figure 4.7
Structure of message from sensory input using monolithic architecture

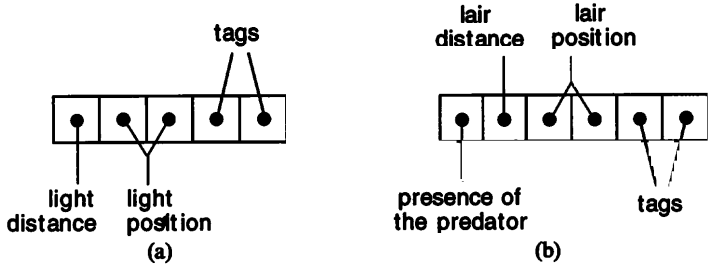


Figure 4.8
Structure of message from sensory input using switch architecture: (a) message going to LCS-`Chase`; (b) message going to LCS-`Escape`

message in our experiment: one bit from the `Chase` LCS and one bit from the `Escape` LCS) goes as input to the switch. Hence the switch selects one of the basic LCSs, which is then allowed to send its action to the AutonoMouse motors. The value and the meaning of the bit sent from the basic LCSs to the switch is not predefined, but is learned by the system.

In figure 4.7, we show the format of a message from sensory input in the monolithic architecture; in figure 4.8, the format of a message from sensory input to basic LCSs in the switch architecture.

Consider, for example, figure 4.8b, which shows the format of the sensor message going to the `Escape` LCS. The message has six bits. Bits one and two are tags that 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 that 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 1 if the lair is far away and to 0 if the lair is very close. Bit six is set to 1 if the AutonoMouse hears the sound produced by an approaching predator; to 0 otherwise.

4.3.5 The Reinforcement Program

As we have seen in section 1.2, the role of the trainer (i.e., the RP), is to observe the learning agent's actions and to provide reinforcements after each action. An interesting issue regarding the use of the trainer is how the RP should be structured to make the best use of the architecture. To answer this question, we ran experiments to compare the monolithic and the switch architecture. Our RP comprises two procedures called "RP-Chase" and "RP-Escape." RP-Chase gives the AutonoMouse a reward if it moves so that its distance from the light source does not increase. On the other hand, if the distance from the light source increases, it punishes ALECSYS. RP-Escape, which is used when the predator is present, rewards the AutonoMouse when it makes a move that diminishes the distance from the lair, if the lair is far; or when it stays still, if the lair is close.

Although the same RP is used for both the monolithic architecture and the switch architecture, when using the switch architecture a decision must be made about how to shape the learning system.

4.3.6 Choice of an Architecture and 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 below, 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 following experiments, where we compare the monolithic and the switch architectures, which were described in chapter 3. Also, the use of a distributed behavioral architecture calls for the choice of a shaping policy; we ran experiments with both holistic and modular shaping.

Table 4.5
Comparison across architectures during test session

	Monolithic architecture		Holistic-switch architecture		Modular-switch-long architecture		Modular-switch-short architecture	
	Avg.	Std.dev.	Avg.	Std.dev.	Avg.	Std.dev.	Avg.	Std.dev.
Chasing	0.830	0.169	0.941	0.053	0.980	0.015	0.942	0.034
Escaping	0.744	0.228	0.919	0.085	0.978	0.022	0.920	0.077
Global	0.784	0.121	0.933	0.073	0.984	0.022	0.931	0.085

Note: Values reported are for performance index P averaged over 20 runs.

In experiment A, we compared the monolithic architecture and the switch architecture with holistic shaping ("holistic-switch"). Differences in performance between these two architectures, if any, are due only to the different architectural organization. Each architecture was run for 15,000 learning cycles, and was followed by a 1,000-cycle test session.

In experiment B, we ran two subexperiments (B1 and B2) using the switch architecture with modular shaping ("modular-switch"). In subexperiment B1, we gave the modular-switch architecture roughly the same amount of computing resources as in experiment A, so as to allow a fair comparison. We ran a total of 15,000 cycles: 5,000 cycles to train the chasing behavioral module, 5,000 cycles to train the escaping behavioral module, and 5,000 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 was the same as in experiment A, the actual time (in seconds) required was shorter because the two basic behaviors could be trained in parallel. Each learning phase was followed by a 1,000-cycle test session.

In subexperiment 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. We ran a total of 6,000 cycles: 2,000 cycles to train the chasing behavioral module, 2,000 cycles to train the escaping behavioral module, and 2,000 cycles to train the behavior coordination module; we refer to this architecture as "modular-switch-short." Again, each learning phase was followed by a 1,000-cycle test session.

Results regarding the test session are reported in table 4.5. The best-performing architecture was the modular-switch. It performed best given

approximately the same amount of resources during learning, and it was able to achieve the same performance as the holistic-switch architecture using much less computing time. These results can be explained by the fact that in the switch architecture each single LCS, having shorter classifiers, has a smaller search space;[5] therefore, the overall learning task is easier. The worst-performing architecture was the monolithic. These results are consistent with those obtained in a similar environment, presented in section 4.4. We studied the significance of the difference in mean performance between the monolithic and the holistic-switch architectures, and between the holistic-switch and the modular-switch-long architectures (the choice of which architectures to compare was made a priori). Both differences were found to be highly significant ($p < 0.01$) by the Mann-Whitney test for all the behavioral modules (`Chase`, `Escape`, and the global behavior).

4.4 EXPERIMENTS IN THE `Chase/Feed/Escape` ENVIRONMENT

In this section, we systematically compare different architecture and shaping policy combinations using the `Chase/Feed/Escape` environment. We try to find a confirmation to some of the answers we already had from the experiments presented in the previous section. Additionally, we make a first step toward studying the architecture scalability and the importance of choosing a good mapping between the structured behavior and the hierarchical architecture that implements it.

4.4.1 Monolithic Architecture

Because the monolithic architecture is the most straightforward way to apply LCSs to a learning problem, results obtained with the monolithic architecture were used as a reference to evaluate whether we would obtain improved performance by decomposing the overall task into simpler subtasks, by using a hierarchical architecture, or both. In an attempt to be impartial in comparing the different approaches, we adopted the same number of transputers in every experiment. And because, for a given number of processors, the system performance is dependent on the way the physical processors are interconnected, that is, on the hardware architecture, we chose our hardware architecture only after a careful experimental investigation (discussed in Piroddi and Rusconi 1992; and Camilli et al. 1990).

Figure 4.9 shows the typical result. An important observation is that the performance of the `Escape` behavior is higher than the performance

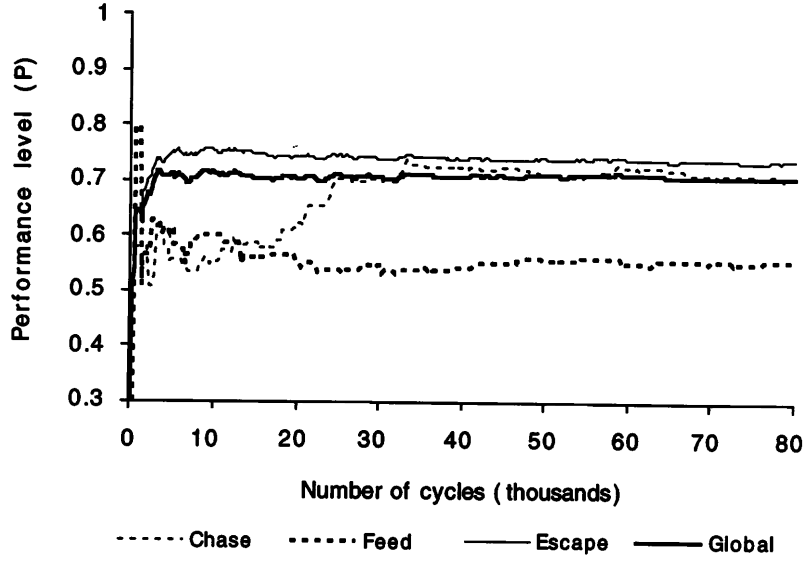


Figure 4.9
Cumulative performance of typical experiment using monolithic architecture

of the `Chase` behavior, which in turn is higher than that of the `Feed` behavior. This result holds for all the experiments with all the architectures. The reasons are twofold. The `Escape` behavior is easier to learn because our learning agent can choose the escaping movement among 5 out of 8 possible directions, while the correct directions to `Chase` an object are, for our AutonoMouse, 3 out of 8 (see figure 4.10).

The lower performance of the `Feed` behavior is explained by the fact that, in our experiments, the AutonoMouse could see the object to be chased and the predator from any distance, while the food could be seen only when closer than a given threshold. This caused a much lower frequency of activation of `Feed`, which resulted in a slower learning rate for that behavior.

Another observation is that, after an initial, very quick improvement of performance, both basic and global performances settled to an approximately constant value, far from optimality. After 80,000 cycles, the global performance reached the value 0.72 and did not change (we ran the experiment up to 300,000 cycles without observing any improvement). Indeed, given the length of the classifiers in the monolithic architecture, the search space (cardinality of the set of possible different classifiers) is

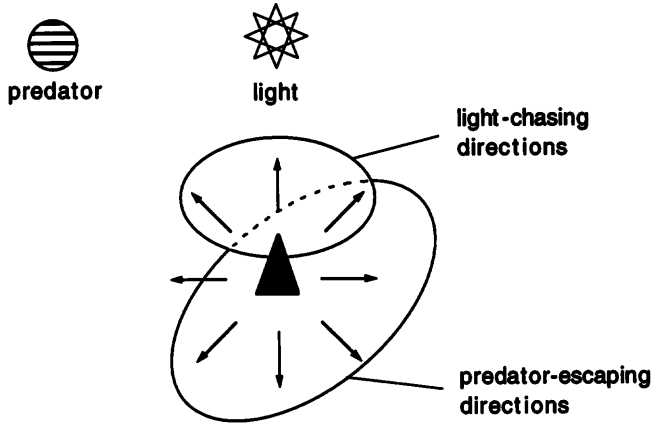


Figure 4.10
Difference between chasing and escaping behaviors

huge: the genetic algorithm, together with the apportionment of credit system, appears unable to search this space in a reasonable amount of time.

4.4.2 Monolithic Architecture with Distributed Input

With distributed input, environmental messages are shorter than in the previous case, and we therefore expect a better performance. More than one message can be appended to the message list at each cycle (maximum three messages, one for each basic behavior).

The results, shown in figure 4.11, confirmed our expectations: global performance settled to 0.86 after 80,000 cycles, and both the `Chase` and `Escape` behaviors reached higher performance levels than with the previous monolithic architecture. Only the `Feed` behavior did not improve its performance. This was partially due to the shortened run of the experiment. In fact, in longer experiments, in which it could be tested adequately, the `Feed` behavior reached a higher level of performance, comparable to that of the `Chase` behavior. It is also interesting to note that the graph qualitatively differs from that of figure 4.9; after the initial steep increase, performance continues slowly to improve, a sign that the learning algorithms are effectively searching the classifiers' space.

4.4.3 Two-level Switch Architecture

In this experiment we used a two-level hierarchical architecture, in which the coordination behavior implemented suppression. The results, reported

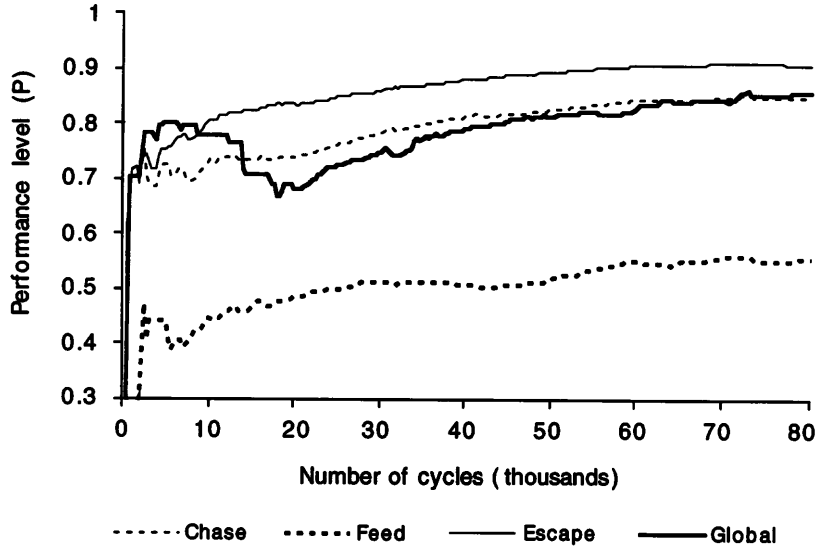


Figure 4.11
Cumulative performance of typical experiment using monolithic architecture with distributed input

in figures 4.12 and 4.13, give the following interesting information. First, as shown in figure 4.12, where we report the performance of the three basic behaviors and of the coordinator (switch) in the first 50,000 cycles, and the global performance from cycle 50,000 to the end of the experiment, the use of the holistic shaping policy results in a final performance that is comparable to that obtained with the monolithic architecture. This is because, with holistic shaping, reinforcements obtained by each individual LCS are very noisy. With this shaping policy, we give each LCS the same reinforcement, computed from observing the global behavior, which means there are occasions when a double mistake results in a correct, and therefore rewarded, final action. Consider the situation where `Escape` is active and proposes a (wrong) move toward the predator, but the coordinator fails to choose the `Escape` module and chooses instead the `Chase` module, which in turn proposes a move away from the chased object (wrong move), say in the direction opposite to that of the predator. The result is a correct move (away from the predator) obtained by the composition of a wrong selection of the coordinator with a wrong proposed move of two basic behaviors. It is easy to understand that it is difficult to learn good strategies with such a reinforcement function.

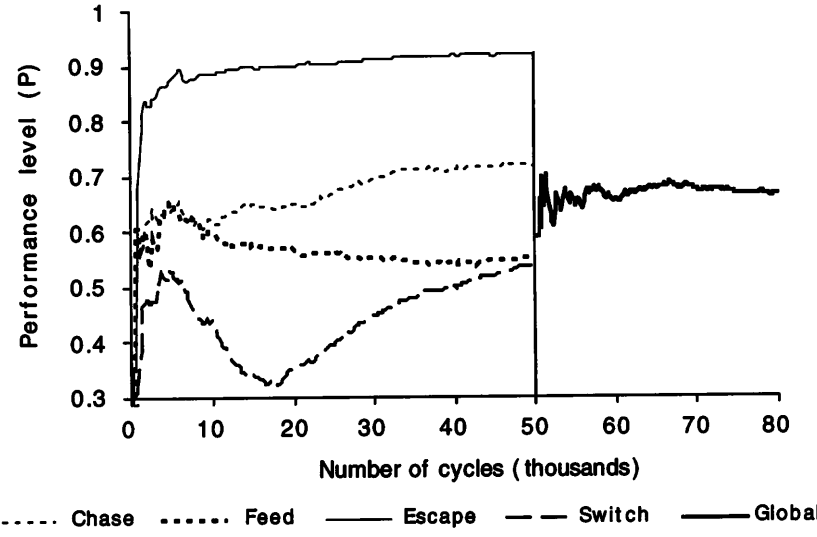


Figure 4.12
Cumulative performance of typical experiment using two-level switch architecture. Holistic shaping.

Second, using the modular shaping policy, performance improves, as expected. The graph of figure 4.13 shows three different phases. During the first 33,000 cycles, the three basic behaviors were independently learned. Between cycles 33,000 and 48,000, they were frozen, that is, learning algorithms were deactivated, and only the coordinator was learning. After cycle 48,000, all the components were free to learn, and we observed the global performance. The maximum global performance value obtained with this architecture was 0.84.

Table 4.6 summarizes the results from using the monolithic and two-level architectures already presented in figures 4.9, 4.11, 4.12, and 4.13. The table shows that, from the point of view of global behavior, the best results were obtained by the monolithic architecture with distributed input and by the two-level switch architecture with modular shaping. The performance of the basic behaviors in the second was always better. In particular, the `Feed` behavior achieved a much higher performance level; using the two-level switch architecture with modular shaping, each basic behavior is fully tested independently of the others, and therefore the `Feed` behavior has enough time to learn its task. It is also interesting to note that the monolithic architecture and the two-level switch architecture with holistic shaping have roughly the same performance.

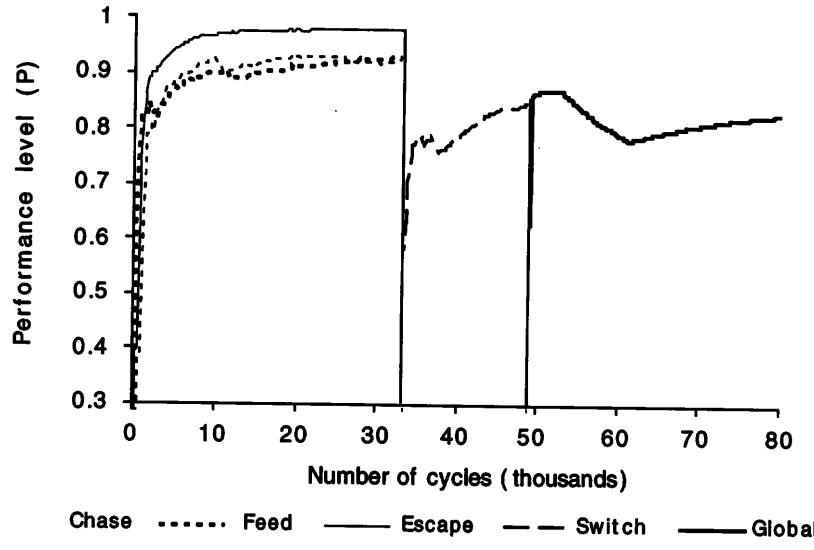


Figure 4.13
Cumulative performance of typical experiment using two-level switch architecture. Modular shaping.

Table 4.6
Comparison of monolithic and two-level hierarchical architectures

Architecture	Chase	Feed	Escape	Switch 1	Global
Monolithic	0.71 (80,000)	0.56 (80,000)	0.75 (80,000)		0.72 (80,000)
Monolithic with distributed input	0.85 (80,000)	0.56 (80,000)	0.92 (80,000)		0.85 (80,000)
Two-level switch holistic shaping	0.73 (50,000)	0.56 (50,000)	0.93 (50,000)	0.54 (50,000)	0.66 (85,000)
Two-level switch modular shaping	0.93 (33,000)	0.92 (33,000)	0.98 (33,000)	0.84 (15,000)	0.82 (85,000)

Note: Values reported are for performance index P, with number of iterations in underlying parentheses.

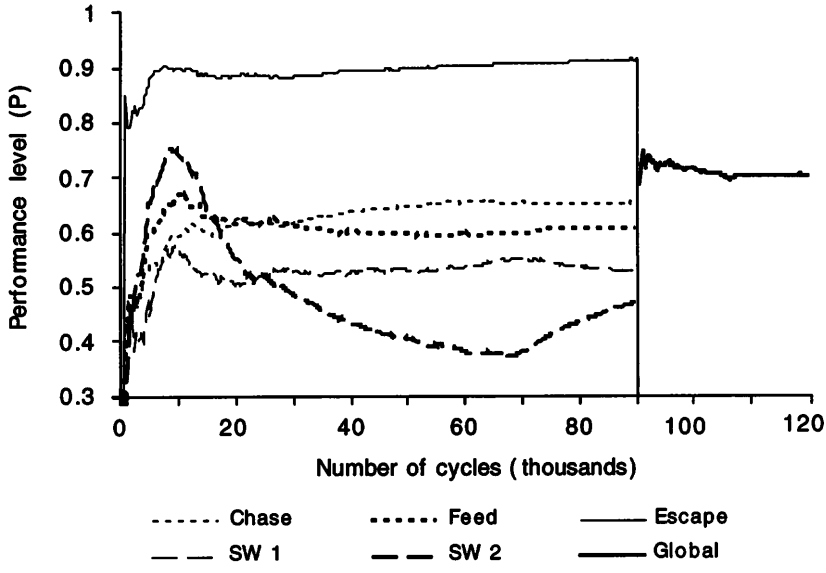


Figure 4.14
Cumulative performance of typical experiment using three-level switch architecture of figure 3.4. Holistic shaping.

4.4.4 Three-level Switch Architecture

The three-level switch architecture stretches to the limit the hierarchical approach (a three-behavior architecture with more than three levels looks absurd). Within this architecture, the coordinator used in the previous architecture was split into two simpler binary coordinators (see figure 3.4). Using holistic shaping, results have shown that two- or three-level architectures are comparable (see figures 4.12 and 4.14, and table 4.7). More interesting are the results obtained with modular shaping. Because we have three levels, we can organize modular shaping in two or three phases. With two-phase modular shaping, we follow basically the same procedure used with the two-level hierarchical architecture: in the second phase, basic behavioral modules are frozen and the two coordinators learn at the same time. In three-phase modular shaping, the second phase is devoted to shape the second-level coordinator (all the other modules are frozen), while in the third phase, the third-level coordinator alone learns. Somewhat surprisingly, the results show that, given the same amount of resources (computation time in seconds), two-phase modular shaping gave slightly better results. With two-phase modular shaping, both coordination

Table 4.7
Comparison of two- and three-level hierarchical architectures

Architecture	Chase	Feed	Escape	Switch 1	Switch 2	Global
Two-level switch, holistic shaping	0.73 (50,000)	0.56 (50,000)	0.93 (50,000)	0.54 (50,000)		0.66 (85,000)
Two-level switch, modular shaping	0.93 (33,000)	0.92 (33,000)	0.98 (33,000)	0.84 (15,000)		0.82 (85,000)
Three-level switch, holistic shaping	0.66 (90,000)	0.61 (90,000)	0.92 (90,000)	0.53 (90,000)	0.47 (90,000)	0.70 (120,000)
Three-level switch, two-phase modular shaping	0.94 (33,000)	0.92 (33,000)	0.98 (33,000)	0.97 (56,000)	0.86 (56,000)	0.99 (120,000)
Three-level switch, three-phase modular shaping	0.93 (33,000)	0.93 (33,000)	0.98 (33,000)	0.80 (10,000)	0.80 (10,000)	0.95 (120,000)

Note: Values reported are for performance index P, with number of iterations in underlying parentheses.

behaviors are learning for the whole learning interval, whereas with three-phase modular shaping, the learning interval is split into two parts during which only one of the two coordinators is learning, and therefore the two switches cannot adapt to each other. The graph in figure 4.15 shows the very high performance level obtained with two-phase modular shaping.

Table 4.7 compares the results obtained with the two- and three-level switch architectures: we report the performance of basic behaviors, of switches, and of the global behavior, as measured after k iterations, where k is the number in parentheses below each performance value. Remember that, as we already said, the number of iterations varied across experiments due to the steady state monitor routine, which automatically decided when to shift to a new phase and when to stop the experiment.

We have run another experiment using a three-level switch architecture to show that the choice of an architecture that does not correspond naturally to the structure of the target behavior leads to poor performance. The architecture we have adopted is reported in figure 4.16; it is similar to that of figure 3.4, except that the two basic behaviors `Feed` and `Escape` have been swapped. It can be expected that the new distribution of tasks between SW1 and SW2 will impede learning: because SW2 does not know whether SW1 is proposing a `Chase` or an `Escape` action, it cannot

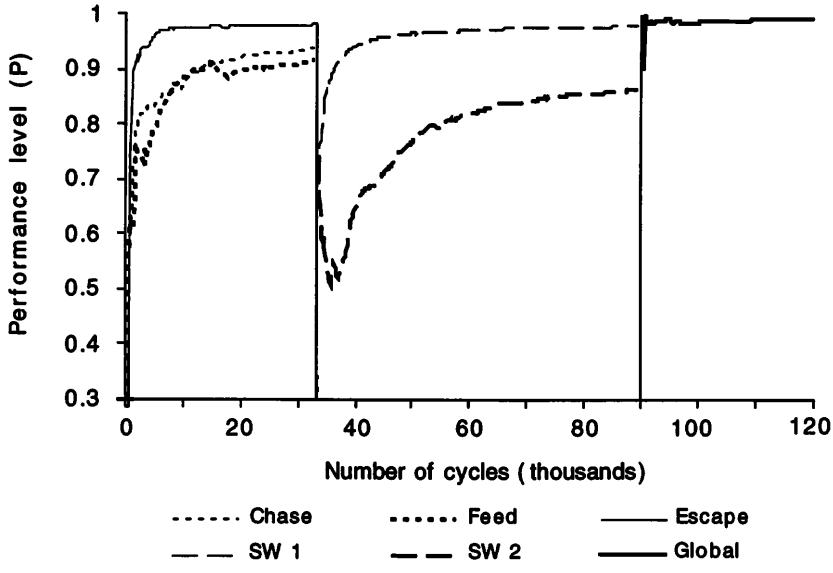


Figure 4.15
Cumulative performance of typical experiment uisng three-level switch architecture of figure 3.4. Two-phase modular shaping.

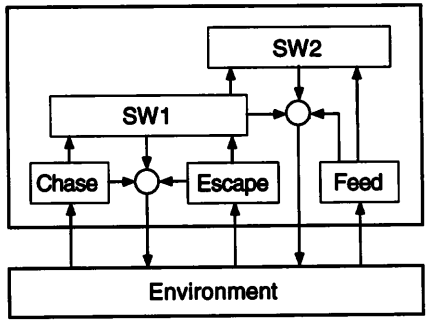


Figure 4.16
Three-level switch architecture with "unnatural" disposition of coordination modules

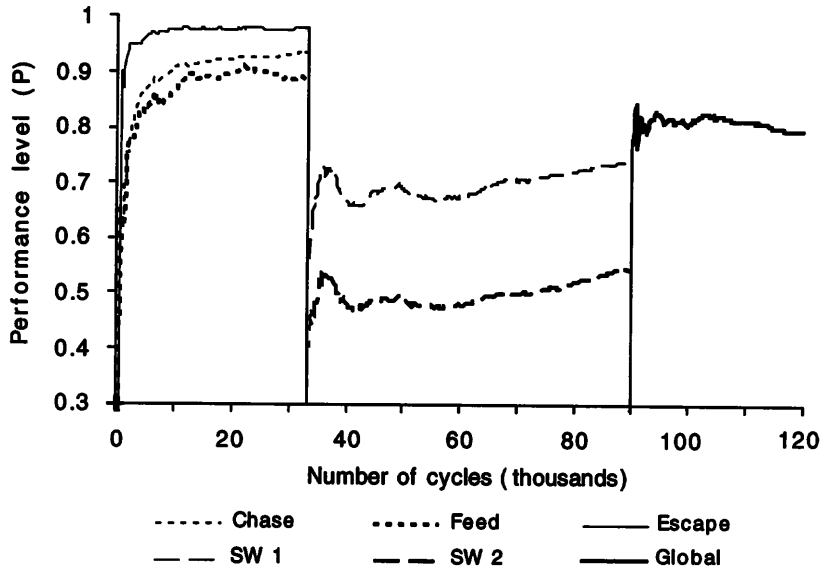


Figure 4.17
Cumulative performance of typical experiment using "unnatural" three-level switch architecture. Two-phase modular shaping.

decide (and therefore learn) whether to suppress SW1 or the `Feed` behavioral module.

Results are shown in figures 4.17 and 4.18. As in the preceding experiment, two-phase shaping gave better results than three-phase. It is clear from figure 4.18 that the low level of global performance achieved was due to the impossibility for SW2 to learn to coordinate the SW1 and the `Feed` modules.

4.4.5 The Issue of Scalability

The experiment presented in this subsection combines a monolithic architecture with multiple inputs and a two-level hierarchical architecture. We use a `Chase/Feed/Escape` environment in which four instances of each class of objects (lights, food pieces, and predators) are scattered around. Only one instance in each class is relevant for the learning agent (i.e., the AutonoMouse likes only one of the four light colors and one of the four kinds of food, and fears only one of the four potential predators). Therefore the basic behaviors, in addition to the basic behavioral pattern, have to learn to discriminate between different objects of the same class.

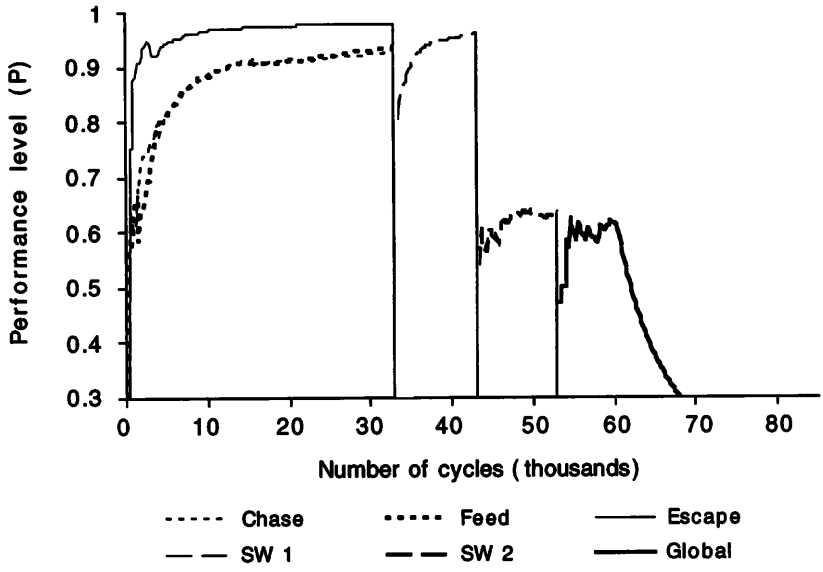


Figure 4.18
Cumulative performance of typical experiment using "unnatural" three-level switch architecture. Three-phase modular shaping.

For example, the `Escape` behavior, instead of receiving a single message indicating the position of the predator (when present), now receives many messages regarding many different "animals," only one of which is a real predator. Different "animals" are distinguished by a tag, and `Escape` must learn to run away only from the real predator (it should be unresponsive to other animals). Our experiments show (see figure 4.19) that the AutonoMouse learns the new, more complex, task, although a greater number of cycles is necessary to reach a performance level comparable to that obtained in the previous experiment (figure 4.13).

4.5 EXPERIMENTS IN THE `FindHidden` ENVIRONMENT

The aims of the `FindHidden` experiment are twofold. First, we are interested to see whether our system is capable of learning a reasonably complex task, involving obstacle detection by sonar and bumpers and searching for a hidden light. The target behavior is to chase a light (`Chase` behavior), searching for it behind a wall when necessary (`Search`

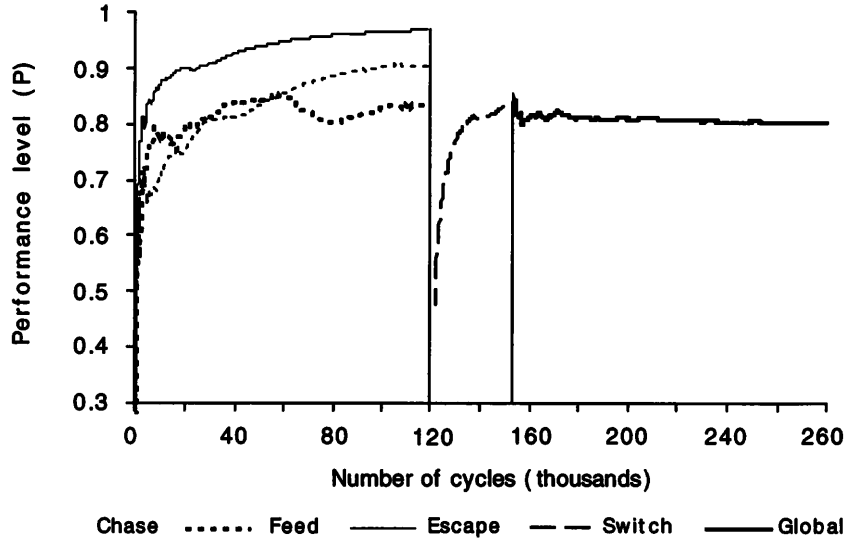


Figure 4.19
Cumulative performance of typical experiment with multi-input, two-level switch architecture. Modular shaping.

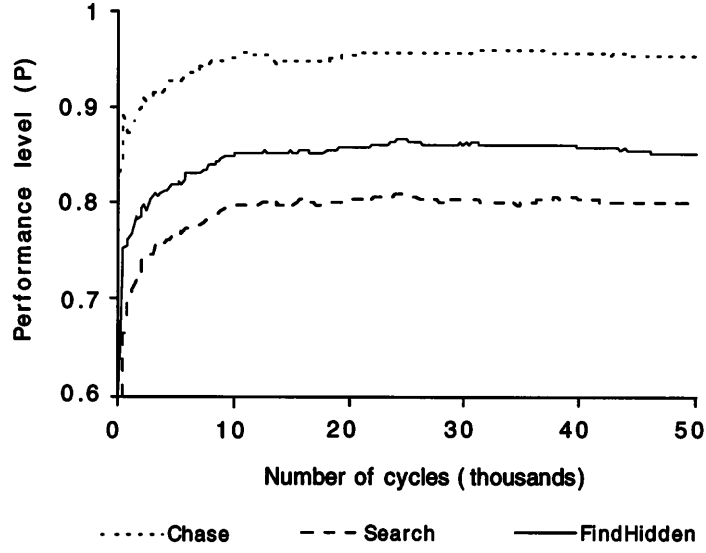


Figure 4.20
Cumulative performance for `FindHidden` task

behavior). The wall has a fixed position, while the light automatically hides itself behind the wall each time it is reached by the AutonoMouse. Second, we are interested in the relation between the sensor granularity, that is, the number of different states a sensor can discriminate, and the learning performance. In these experiments, we pay no attention to the issue of architecture, adopting a simple monolithic LCS throughout.

4.5.1 The `FindHidden` Task

To shape the AutonoMouse in the `FindHidden` environment, the reinforcement program was written with the following strategy in mind:

```
if a light is seen
  then {Chase behavior} Approach it
  else {Search behavior}
       if a distal obstacle is sensed by sonars
         then Approach it
         else if a proximal obstacle is sensed by
                  bumpers
                then Move along it
                else Turn consistently {go on
                    turning in the same direction,
                    whichever it is}
              endif
       endif
endif.
```

The distances of the AutonoMouse from the light and from the wall are computed from the geometric coordinates of the simulated objects. The simulation was run for 50,000 cycles. In figure 4.20, we separately show the `Chase` behavior performance (when the light is visible), the `Search` behavior performance (when the light is not visible), and the `FindHidden` performance. Chasing the light appears to be easier to learn than searching for it. This is easy to explain given that searching for the light is a rather complex task, involving moving toward the wall, moving along it, and turning around when no obstacle is sensed. It will be interesting to compare these results with those obtained with the real robot (see section 5.3).

4.5.2 Sensor Granularity and Learning Performance

This experiment explored the effect that different levels of sensor granularity have on the learning speed. To make things easier, we concen-

trated on the sonar; very similar results were obtained using the light sensors. As a test problem, we chose to let the simulated AutonoMouse learn a modified version of the `Search` task. The reinforcement program used was as follows.

```
case distance of
   very_close: if backward then reward else punish;
   close:      if turn then reward else punish;
   far:        if slowly forward then reward else
               punish;
   very_far:   if quickly forward then reward else
               punish
endcase.
```

In our experiments, robot sensors were used by both the trainer and the learner. In fact, the trainer always used the sensor information with the maximum available granularity (in the sonar case, 256 levels) because in this way it could more accurately judge the learner performance. On the other hand, the learner had to use the minimum amount of information necessary to make learning feasible because a higher amount of information would make learning slower, without improving the long-term performance. The correspondence between the distance variable and the sonar reading was set by the trainer as follows:

```
distance := very_close, if sensor_reading ∈ [0 . . . 63],
distance := close, if sensor_reading ∈ [64 . . . 127],
distance := far, if sensor_reading ∈ [128 . . . 191],
distance := very_far, if sensor_reading ∈ [192 . . . 256].
```

We investigated the learning performance with the simulated robot for the following granularities of the sonar messages: 2, 4, 8, 16, 32, 64, 128, 256. This means that the learner was provided with messages from the sonar that ranged from a very coarse granularity (one-bit messages, that is, binary division of the space) to very high granularity (eight-bit messages, that is, 256 levels, the same as the trainer). Figure 4.21 shows the results obtained with the simulated robot. We also report the performance of a random controller for comparison.

Results show that, as expected, when the sonar was used to discriminate among four levels, the performance was the best. In fact, this granularity level matched the four levels used by the trainer to give reinforcements. It is interesting to note that, even with a too-low granularity

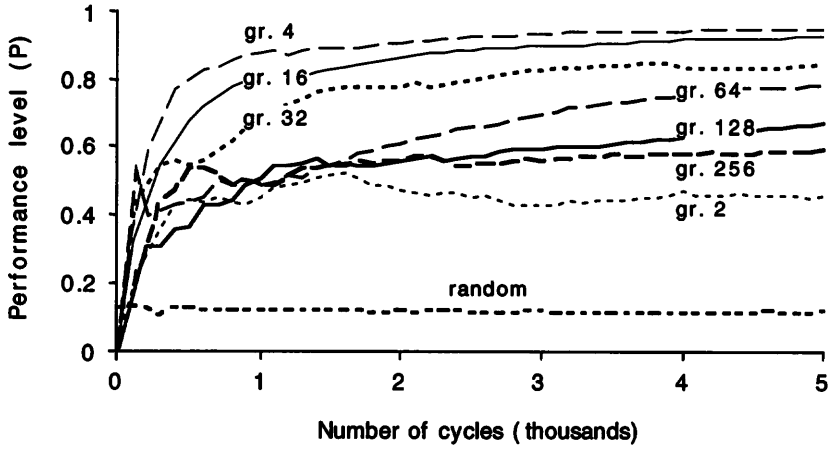


Figure 4.21
Granularity of sonar sensor and learning performance *P* with simulated AutonoMouse

(i.e., granularity = 2), the robot was able to perform better than in the random case. Increasing the granularity decreased monotonically the performance achieved by the learner. It is also interesting to note that the learner, in those cases where the granularity was higher than necessary (> 4), was able to create rules that (by the use of "don't care" symbols matching the lower positions of input messages) clustered together all inputs belonging to the same output class as defined by the trainer policy.

4.6 POINTS TO REMEMBER

- ALECSYS allows the AutonoMouse to learn a set of classifiers that are tantamount to solving the learning problems posed by the simple tasks we have worked with in this chapter.
- Our results suggest that it is better to use rewards and punishments than only rewards.
- Using the mutespec operator, dynamically changing the number of classifiers, and calling the genetic algorithm when the bucket brigade has reached a steady state all improve the LCS's performance.
- AutonoMouse can benefit from a simple type of memory called "sensor memory."
- The factorization of the learning task into several simpler learning tasks helps. This was shown by the results of our experiments.

- The system designer must take two architectural decisions: how to decompose the learning problem; and how to make the resulting modules interact. The first issue is a matter of efficiency: a basic behavior should not be too difficult to learn. In our system, this means that classifiers should be no longer than about 30 bits (and therefore messages cannot be longer than 10 bits). The second issue is a matter of efficiency, comprehensibility, and learnability.
- The best shaping policy, at least in the kind of environments investigated in this chapter, is modular shaping.
- The granularity of sensor messages is best when it matches the granularity chosen by the trainer to discriminate between different reinforcement actions.

Chapter 5

Experiments in the Real World

5.1 INTRODUCTION

Moving from simulated to real environments is challenging. New issues arise: sensor noise can appear, motors can be inaccurate, and in general there can be differences of any sort between the designed and the real function of the robot. Moreover, not only do the robot's sensors and effectors become noisy, but also the RP must rely on real, and hence noisy, sensors to evaluate the learning robot moves. It is therefore important to determine to what extent the real robot can use the ideas and the software used in simulations.

Although the above problems can be studied from many different points of view, running experiments with real robots has constrained us to limit the extent of our investigation. In section 5.2, we focus on the role that the trainer plays in making ALECSYS learn a good control policy: we study how different trainers can influence the robustness to noise and to lesions in the real AutonoMouse. We present results from experiments in the real world using AutonoMouse II. In section 5.3, we repeat, with the real robot (AutonoMouse IV), the `FindHidden` experiment and the sensor granularity experiment already run in simulation.

Results obtained with simulations presented in the preceding chapter suggest setting the message list length to 1, using 300 classifiers on a three-transputer configuration, and using both rewards and punishments. In particular, we experimentally found that learning tended to be faster when punishments were somewhat higher (in absolute value) than rewards; a typical choice would be using a reinforcement of $+10$ for rewards, and of -15 for punishments. In all the experiments presented in this chapter, we used the monolithic architecture.

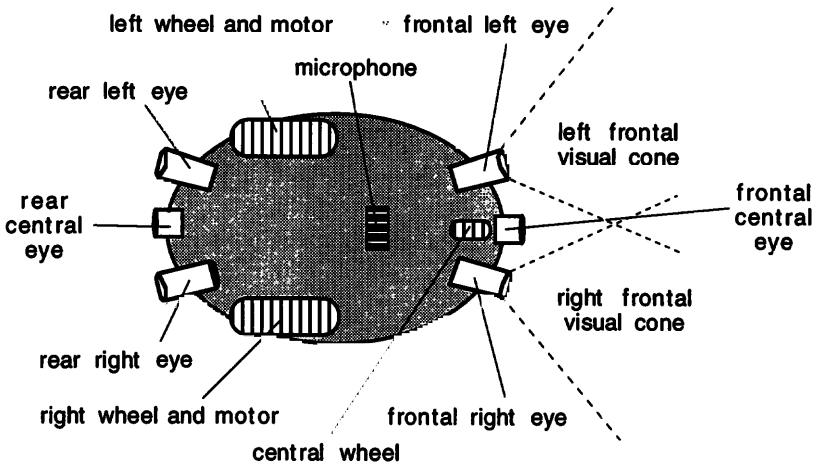


Figure 5.1
Functional schematic of AutonoMouse II

5.2 EXPERIMENTS WITH AUTONOMOUSE II

5.2.1 AutonoMouse II Hardware

Our smallest robot is AutonoMouse II (see picture in figure 1.3 and sketch in figure 5.1). It has four directional eyes, a microphone, and two motors. Each directional eye can sense a light source within a cone of about 60 degrees. Each motor can stay still or move the connected wheel one or two steps forward, or one step backward. AutonoMouse II is connected to a transputer board on a PC via a 9,600-baud RS-232 link. Only a small amount of processing is done onboard (the collection of data from sensors and to effectors and the management of communications with the PC). All the learning algorithms are run on the transputer board.

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, AutonoMouse II must rotate because the eyes cannot do so independently; they are fixed with respect to the robot body. The sensor value of each eye is either on or off, depending on the relation of the light intensity to a threshold. The robot is also provided with two central eyes that can sense light within a half-space; these are used solely by the trainer while implementing the reward-the-result policy. These central eyes evaluate the absolute increase or decrease in light intensity (they distinguish 256 levels). The format of input messages is given in figure 5.2. We used three message formats

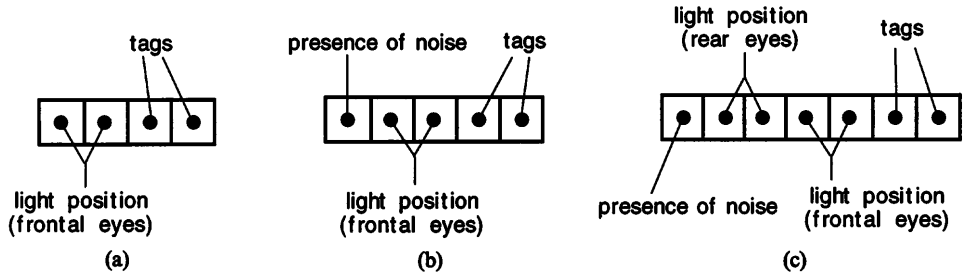


Figure 5.2
Meaning of messages from sensors in AutonoMouse II: (a) format of sensor messages in first and second experiments; AutonoMouse II uses only two frontal eyes; (b) format of sensor messages in third experiment; AutonoMouse II uses two frontal eyes and microphone; (c) format of sensor messages in fourth experiment; AutonoMouse II uses all four eyes and microphone.

because in different experiments AutonoMouse II needed different sensory capabilities. In the experiments on noisy sensors and motors, in the lesion studies, and in the experiment on learning to chase a moving light source (sections 2.4.1, 2.4.2, and 2.4.3, respectively), the robot can see the light only with the two frontal eyes; this explains the short input message shown in figure 5.2a. In the first of the two experiments presented in section 2.4.4, AutonoMouse II was also able to hear a particular environmental noise, which accounts for the extra bit in figure 5.2b. Finally, in the second experiment presented in section 2.4.4, using all the sensory capabilities resulted in the longest of the input messages (figure 5.2c).

AutonoMouse II moves by activating the 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 format of messages to effectors is the same as for the simulated AutonoMouse (see figure 4.2).

5.2.2 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 period).

Experiments with the real robots were repeated only occasionally because they took up a great deal of time. When experiments were

repeated, however, differences between different runs were shown to be marginal.

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

5.2.3 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 the robot's sensors to ALECSYS and from ALECSYS to the robot's motors have a structure very similar to that of the simulated AutonoMouse. On the other hand, the real robot differs from the simulated one in that sensor input and motor output are liable to be inaccurate. This is a major point in machine learning research: simulated environments can only approximate real ones.

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 figure 5.3).

5.2.3.1 The Reward-the-Intention Policy

We say that a trainer "rewards the intention" if, to decide what reinforcement to give, it uses observations from an *ideal environment* positioned between ALECSYS and the real 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 the trainer 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.

5.2.3.2 The Reward-the-Result Policy

On the other hand, we say that a trainer "rewards the result" if, to decide what reinforcement to give, it uses observations from the real world. The reward-the-result trainer rewards ALECSYS if it proposes an action that

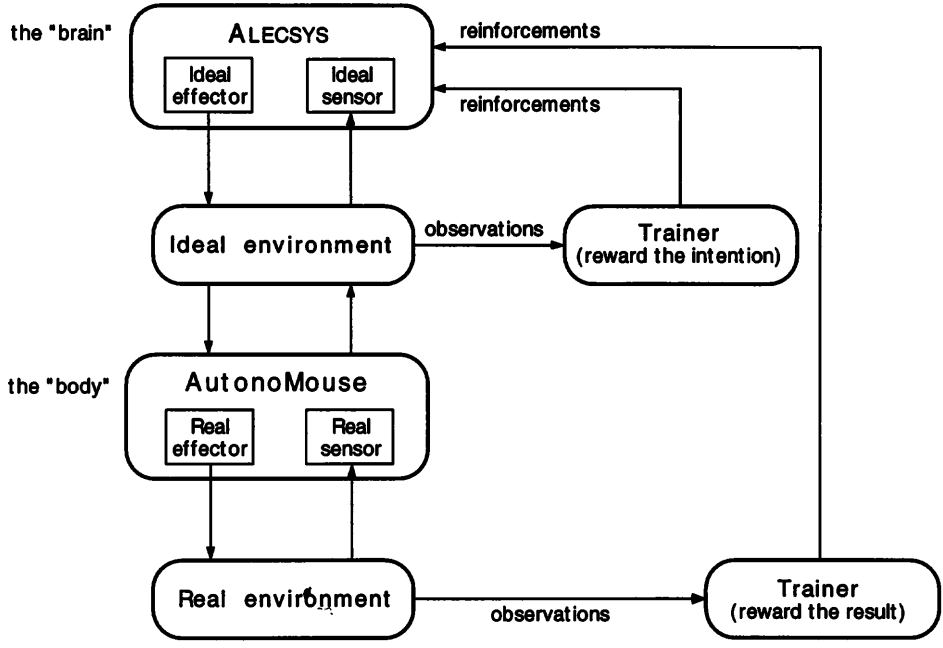


Figure 5.3
Reward-the-intention and reward-the-result trainers

diminishes the distance from the goal. (In the light-approaching example the rewarded action is a move that causes AutonoMouse II 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.

With the reward-the-intention policy, the observed behavior can be the desired one only if there is a correct mapping between the trainer's interpretation of sensor (motor) messages and the real world significance of AutonoMouse's sensory input (motor commands).[1] For example, if the two AutonoMouse's eyes are inverted the reward-the-intention trainer will reward ALECSYS for actions that an external observer would judge as wrong: while the trainer intends to reward the "Turn toward 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 section, we present the results of some experiments designed to test ALECSYS's learning capability under the two different reinforcement policies. We also introduce the following handicaps to test ALECSYS's adaptive capabilities: inverted eyes, blindness in one eye, and incorrect calibration of motors. All graphs presented are relative to a single typical experiment.

5.2.4 An Experimental Study of Training Policies

In this section, we investigate what effect the choice of a training policy has on noisy sensors and motors, and on a set of lesions[2] we deliberately inflicted on AutonoMouse II to study the robustness of learning. The experiment was set in a room in which a lamp had been arbitrarily positioned. The robot was allowed to wander freely, sensing the environment with only its two frontal directional eyes. ALECSYS received a high reward whenever, in the case of reward-the-result policy, the light intensity perceived by the frontal central sensor increased, or, in the case of reward-the-intention policy, the proposed move was the correct one with respect to the received sensory input (according to the mapping on which the reward-the-intention trainer based its evaluations). Sometimes, especially in the early phases of the learning process, AutonoMouse II lost contact with the light source (i.e., it ceased to see the lamp). In these cases, ALECSYS received a reward if the robot turned twice in the same direction. This favored sequences of turns in the same direction, assuring that the robot would, sooner or later, see the light again. In the following experiments, we report and comment on the results of experiments with the standard AutonoMouse II and with versions using noisy or faulty sensors or motors. In all the experiments, ALECSYS was initialized with a set of randomly generated classifiers.

5.2.4.1 Noisy Sensors and Motors

Figure 5.4 compares the performance obtained with the two different reinforcement policies using well-calibrated sensory or motor devices.[3] Note that the performance is better for the reward-the-result policy. Drops in performance at cycle 230 for the reward-the-intention policy and at cycle 250 for the reward-the-result policy are due to sudden changes in the light position caused by the experimenter moving the lamp far away from the robot. Figure 5.5 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

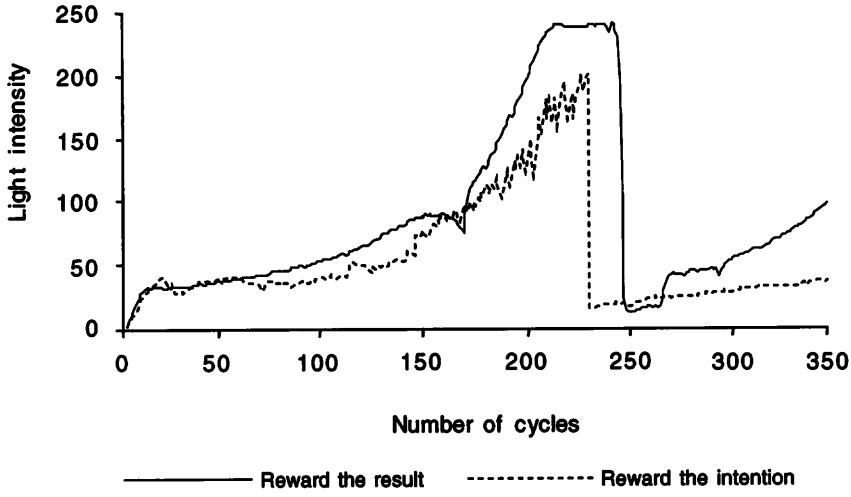


Figure 5.4
Difference in performance between reward-the-result and reward-the-intention training policies

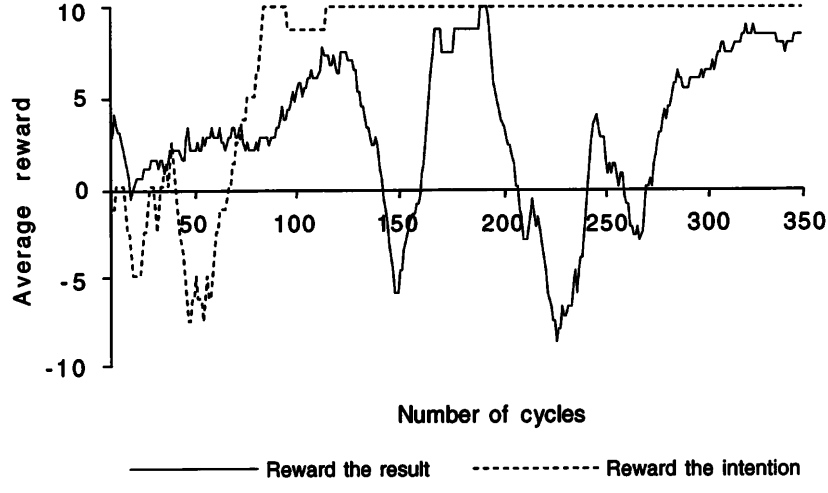


Figure 5.5
Difference in average reward between reward-the-result and reward-the-intention training policies

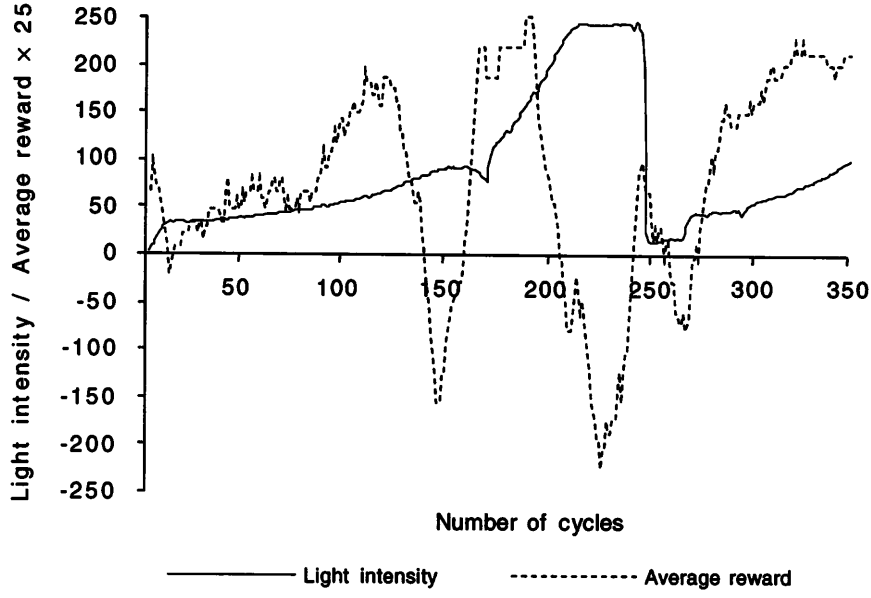


Figure 5.6
System performance (light intensity) and average reward (multiplied by 25 to ease visual comparison) for reward-the-result training policy

perfect information, ALECSYS learns to do the right thing. In the case of the reward-the-result policy, however, ALECSYS does not always get the expected reward. This is because we did not completely succeed despite our efforts to build good, noise-free sensors and motors, and to calibrate them appropriately. Figure 5.6 directly compares light intensity and average reward for the reward-the-result policy (average reward is multiplied by 25 to ease visual comparison).

These results confirm our expectation, as discussed in section 2.3, that a trainer using the reward-the-intention policy, in order to be a good trainer, needs 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 nonreasonable requirement because 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 it is teaching.

It is interesting that the number of cycles required by the real robot to reach the light is significantly lower than the number of cycles required by

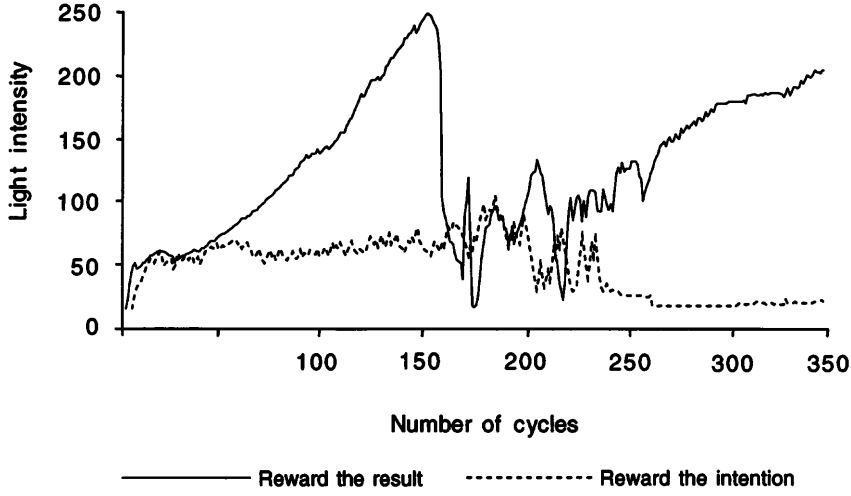


Figure 5.7
Difference in performance between reward-the-result and reward-the-intention training policies for AutonoMouse II with inverted eyes

the simulated robot to reach a high performance in the previous experiments. This is easily explained if you think that the correct behavior is more frequent than the wrong one as soon as performance is higher than 50 percent. The real robot starts therefore to approach the light source much before it has reached a high frequency of correct moves.

5.2.4.2 Lesions Study

Lesions differ from noise in that they cause a sensor or motor to systematically deviate from its design specifications. In this section we study the robustness of our system when altered by 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 AutonoMouse II are shown in figures 5.7 and 5.8 (these graphs are analogous to those regarding the standard eye configuration of figures 5.4 and 5.5). 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 figures 5.4 and 5.7). As discussed before

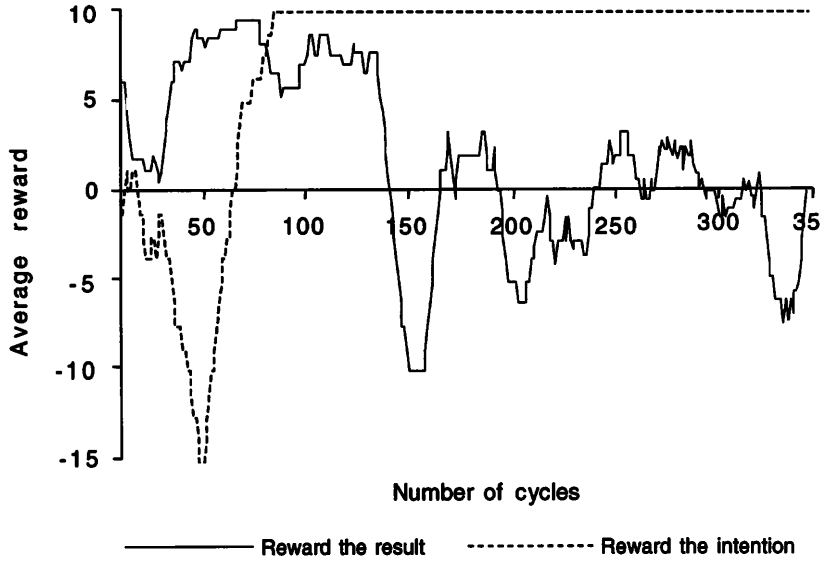


Figure 5.8
Difference in average reward between reward-the-result and reward-the-intention training policies for AutonoMouse II with inverted eyes

(see section 2.3), with inverted eyes the reward-the-intention policy fails. (Note that in this experiment the light source was moved after 150 cycles.)

Similar qualitative results were obtained using only one of the frontal directional eyes (one blind eye experiment). Figure 5.9 shows that the reward-the-intention policy is not capable of letting the robot learn to chase 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 robot to approach the light, although it requires more cycles than with two working eyes. (At cycle 135, there is a drop in performance where AutonoMouse II lost sight of the light and turned right until it saw the light source again.)

As a final experiment, the robot 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.

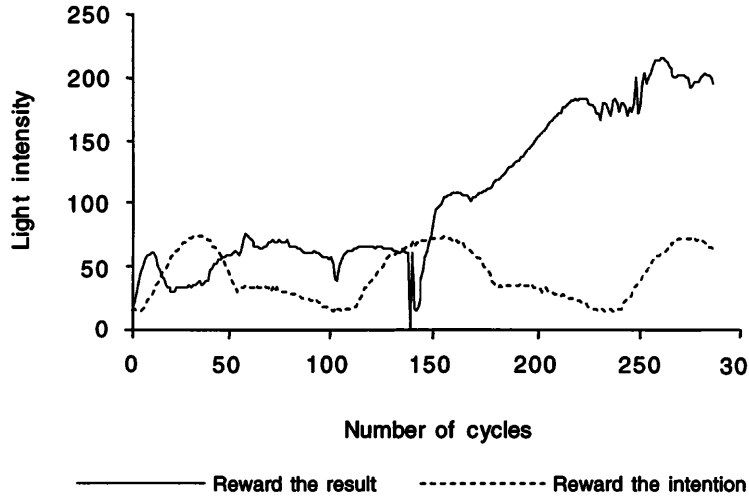


Figure 5.9
Difference in performance between reward-the-result and reward-the-intention training policies for AutonoMouse II with one blind eye

The net effect was an AutonoMouse II 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). Because the average speed of the robot was higher, however, it took fewer cycles to reach the light. Figure 5.10, the result of a typical experiment, shows that the reward-the-result training policy was better also in this case than the reward-the-intention training policy.

5.2.4.3 Learning to Chase a Moving Light Source

Learning to chase a moving light source introduces a major problem; the reinforcement program cannot use distance (or light intensity) changes to decide whether to give a reward or a punishment. Indeed, there are situations in which AutonoMouse II goes toward the light source while the light source moves in the same direction. If the light source speed is higher than the robot's, then the distance increases even though the robot made the right move. This is not a problem if we use the reinforcement program based on the intention, but it is if the reinforcement program is based on the result. Nevertheless, it is clear that the reward-the-intention policy is

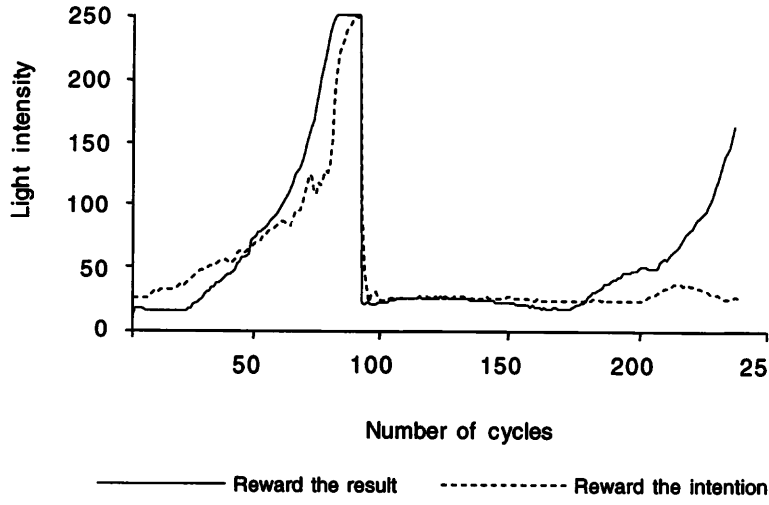


Figure 5.10
Difference in performance between reward-the-result and reward-the-intention training policies for AutonoMouse II with incorrect regulation of motors

less appealing because it is not adaptive with respect to calibration of sensors and effectors. A possible solution is to let ALECSYS learn to approach a stationary light, then to freeze the rule set (i.e., stop the learning algorithm) and use it to control the robot. With this procedure one should be aware that, once learning has been stopped, the robot can use only rules developed during the learning phase, which must therefore be as complete as possible. To ensure complete learning, we need to give AutonoMouse II the opportunity to learn the right behavior in every possible environmental situation (in our case, every possible relative position of the light source). Figure 5.11 reports the result of the following experiment. At first the robot is left free to move and to learn the approaching behavior. After 600 cycles we stop the learning algorithm, start to move the light source, and let the robot chase it. Unfortunately, during the first learning phase AutonoMouse II does not learn all the relevant rules (in particular, it does not learn to turn right when the light is seen by the right eye), and the resulting performance is not satisfactory. At cycle 1,300, we therefore start the learning algorithm again and, during the cycles between 1,700 and 2,200, we start to present the lamp to AutonoMouse II on one side, and wait until the robot starts to do the right thing. This procedure is repeated many times, presenting the light alternately in front,

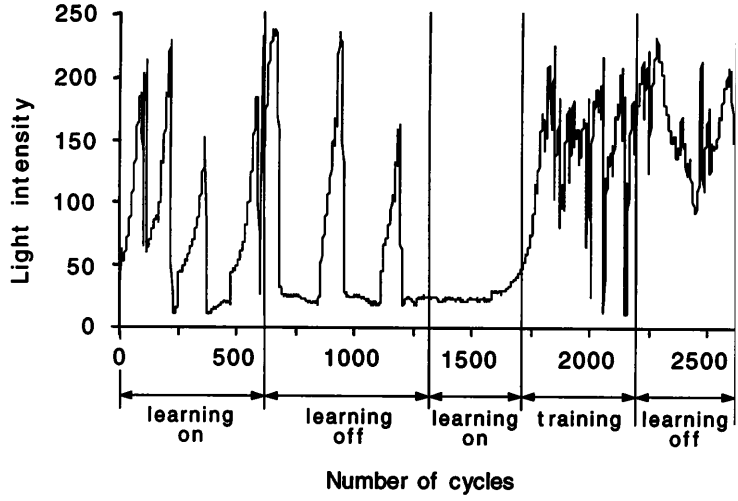


Figure 5.11
AutonoMouse II performance in learning to chase moving light

on the right, on the left, and in back. The exact procedure is difficult to describe, but very easy and natural to implement. What happens during the experiment, which we find very exciting, is that the reactive system learns in real time. The experimenter has only to repeat the procedure[4] until visual feedback tells him that ALECSYS has learned, as manifested by the robot beginning to chase the light. After this training phase, learning is stopped again (at cycle 2,200) and the light source is steadily moved. Figure 5.11 shows that this time the observed behavior is much better (performance is far from maximum, however, because the light source is moving and the robot therefore never reaches it).

5.2.4.4 Learning to Switch between Two Behaviors

In these two experiments, the learning task was made more complicated by requiring AutonoMouse II to learn to change to an alternative behavior in the presence of a particular noise. Experiments were run using the reward-the-result policy.

In the first experiment, we used the two directional frontal eyes and the two behaviors `Chase` and `Escape`. During the `Chase` phase the experimental setting was the same as in the first experiment with standard capabilities (that is, the experiment in section 2.4.1), except for input messages, which were one bit longer due to the noise bit (see figure 5.2b).

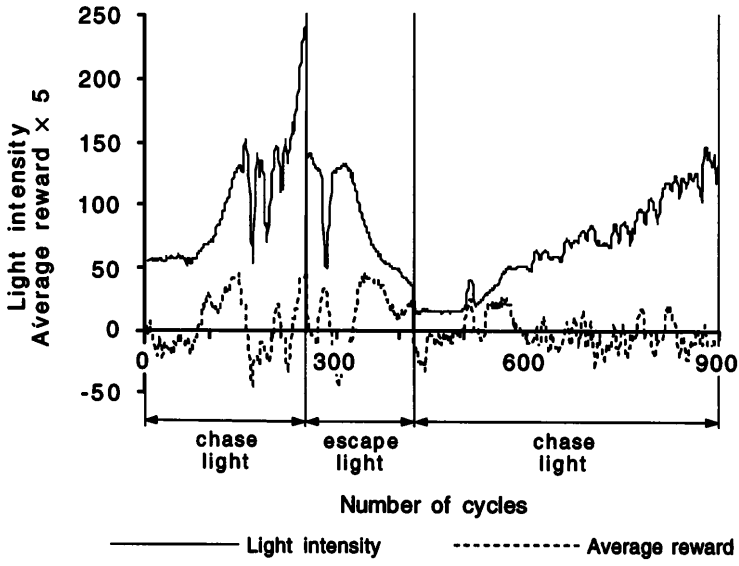


Figure 5.12
AutonoMouse II performance in learning to switch between two behaviors: chasing light and escaping light (robot uses two eyes)

When the noise started, the reinforcement program changed according to the behavior that was to be taught, that is, escaping from the light. Again, as can be seen from figure 5.12, the observed behavior was exactly the desired one.

In the second experiment, we used four directional eyes and the two behaviors `Chase` moving forward and `Chase` moving backward. During the `Chase` moving forward phase, the experimental setting was the same as in the first experiment with standard capabilities (that is, the experiment in section 2.4.1), except for input messages, which were three bits longer due to the noise bit and to the two extra bits for the two rear eyes (see figure 5.2c). When the noise started, the reinforcement program changed according to the behavior that was to be taught. From figure 5.13, it is apparent that after the noise event (at cycle 152) performance fell very quickly;[5] the robot had to make a 180-degree turn before it could again see the maximum intensity of light from the light source.

As after the turn, ALECSYS did not have the right rules (remember that the rules developed in the moving forward phase were different from the rules necessary during the moving backward phase because of the bit that

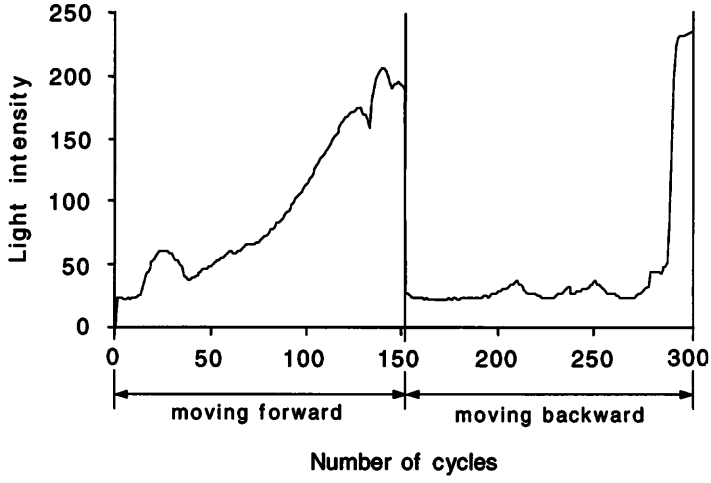


Figure 5.13
AutonoMouse II performance in learning to switch between two behaviors: chasing light moving forward and chasing light moving backward (robot uses four eyes)

indicated presence or absence of noise), AutonoMouse II was in a state very similar to the one it was in at the beginning of the experiment. Therefore, for a while it moved randomly and its distance from the light source could increase (as happened in the reported experiment). At the end, it again learned the correct set of rules and, as can be seen in the last part of the graph, started to approach the light source moving backward.

These two experiments have shown that increasing the complexity of the task did not cause the performance of the system to drop to unacceptable levels. Still, the increase in complexity was very small, and further investigation will therefore be necessary to understand the behavior of the real AutonoMouse in environments like those used in simulations.

5.3 EXPERIMENTS WITH AUTONOMOUSE IV

5.3.1 AutonoMouse IV Hardware

AutonoMouse IV (see figure 5.14) has two directional eyes, a sonar, front and side bumpers, and two motors. Each directional eye can sense a light source within a cone of about 180 degrees. The two eyes together cover a 270-degree zone, with an overlapping of 90 degrees in front of the robot. The sonar is highly directional and can sense an object as far away as 10

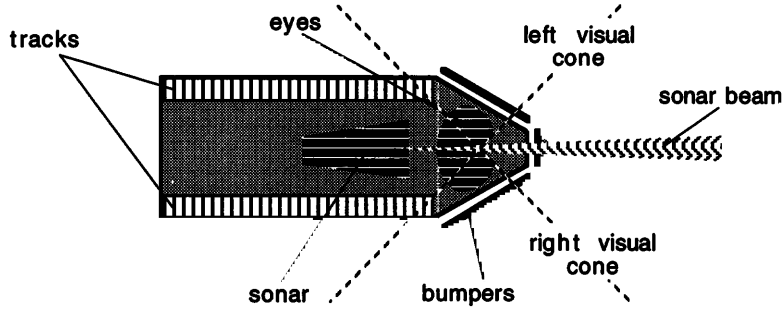


Figure 5.14
Functional schematic of AutonoMouse IV

meters. The output of the sonar can assume two values, either "I sense an object," or "I do not sense an object." Each motor can stay still or move the connected track one or two steps forward, or one step backward. AutonoMouse IV is connected to a transputer board on a PC via a 4,800-baud infrared link.

5.3.2 Experimental Settings

The aim of the experiments presented in this section was to compare the results obtained in simulation with those obtained with the real robot. The task chosen was `FindHidden`, already discussed in section 4.5. The environment in which we ran the real robot experiments consisted of a large room containing an opaque wall, about 50 cm × 50 cm, and an ordinary lamp (50 W). The wall's surface was pleated, in order to reflect the sonar's beam from a wide range of directions. The input and output interfaces were exactly the same as in the simulation, and so was the RP. The input from the sonar was defined in such a way that a front obstacle could be detected within about 1.5 m from the robot. The first experiment presents the results obtained on the complete task, while the second experiment discusses results on the relation between sensor granularity and learning speed.

Although we designed the simulated environment and robot so as to make them very similar to their real counterparts, there were three main differences between the real and the simulated experiments. First, in the real environment the light was moved by hand and hidden behind the wall when AutonoMouse IV got very close to it (10–15 cm), as compared to the simulated environment, where the light was moved by the

simulation program in a systematic way. The manual procedure introduced an element of irregularity, although, from the results of the experiments, it is not clear whether this irregularity affected the learning process.

Second, the distances of AutonoMouse IV from the light and from the wall were estimated on the basis of the outputs of the light sensors and of the sonar, respectively. More precisely, each of the two eyes and the sonar output an eight-bit number from 0 to 255, which coded, respectively, the light intensity and the distance from an obstacle. To estimate whether the robot got closer to the light, the total light intensity (that is, the sum of the outputs of both eyes) at cycle t was compared with the total light intensity at cycle $t - 1$. The sonar's output was used in a similar way to estimate whether the robot got closer to the wall.

The eyes and the sonar were used in different ways by the agent and by the RP: from the point of view of the agent, all these sensors behaved as on/off devices; for the RP, the eyes and the sonar produced an output with higher discriminative power. Therefore, the same hardware devices were used as the trainer's sensors and, through a transformation of their outputs, as the sensors of the agent (the main reason for providing the agent with a simplified binary input was to reduce the size of the learning system's search space, thus speeding up learning).

By exploiting the full binary output of the eyes and of the sonar, it is possible to estimate the actual effect of a movement toward the light or the wall. However, given the sensory apparatus of AutonoMouse IV, we cannot establish the real effect of a left or right turn; for these cases, the RP based its reward on the expected move, that is, on the output message sent to the effectors, and not on the actual move. In other words, we used a reward-the-intention reinforcement policy. To overcome this limitation, we have added to AutonoMouse IV a tail which can sense rotations of the robot. The resulting robot, which we call "AutonoMouse V," will be the object of experimentation in chapter 7.

Third and finally, for practical considerations, the experiments with AutonoMouse IV were run for about four hours, covering only 5,000 cycles, while the simulated experiments were run for 50,000 cycles.

5.3.3 `FindHidden` Task: Experimental Results

The graph of figure 5.15 shows that AutonoMouse IV learned the target behavior reasonably well, as was intuitively clear by direct observation during the experiment. There is, however, a principal discrepancy between

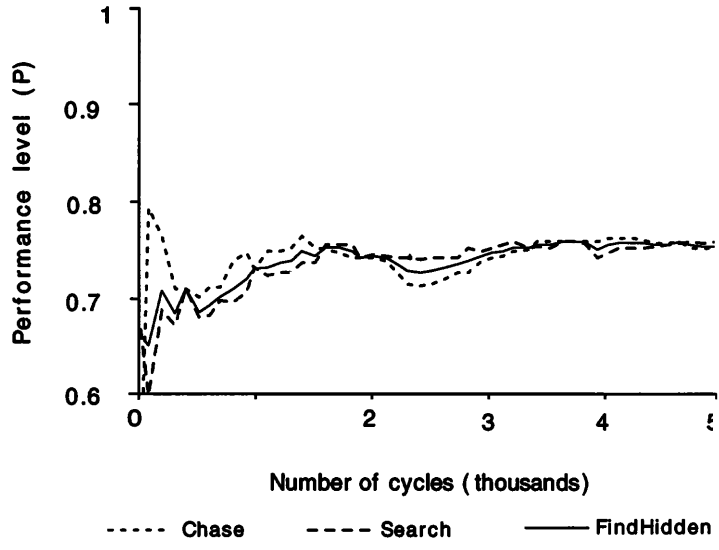


Figure 5.15
`FindHidden` experiment with AutonoMouse IV

the results of the real and those of the simulated experiments (see figure 4.20). In the simulation, after 5,000 cycles the `Chase`, `Search`, and `FindHidden` performances had reached, respectively, the approximate values of 0.92, 0.76, and 0.81; the three corresponding values in the real experiment are lower (about 0.75) and very close to each other. To put it differently, although the real and the simulated light searching performances are very similar, in the simulated experiment the chasing behavior is much more effective than the searching behavior, while in the real experiment they are about the same.

We interpret this discrepancy between the real and simulated experiments as an effect of the different ways in which the distance between the robot and the light were estimated. In fact, the total light intensity did not allow for a very accurate discrimination of such a distance. A move toward the light often did not result in an increase of total light intensity large enough to be detected; therefore, a correct move was not rewarded, because the RP did not understand that the robot had gotten closer to the light. As a consequence, the rewards given by the RP with respect to the light-chasing behavior were not as consistent as in the simulated experiments.

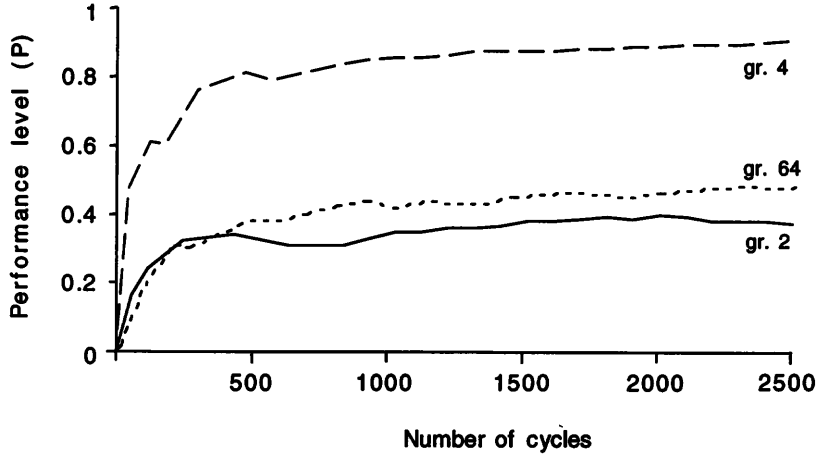


Figure 5.16
Sensor virtualization with AutonoMouse IV

Table 5.1
Comparison of real and simulated AutonoMouse IV performances when changing sensor's granularity

Sonar granularity	Performance of simulated robot after 5,000 cycles	Performance of real robot after 5,000 cycles
2	0.456	0.378
4	0.926	0.906
64	0.770	0.496

5.3.4 Sensor Granularity: Experimental Results

To further explore the relation between sonar sensor granularity and learning performance, we repeated with the real robot the same experiment previously run in simulation (see section 4.5.2). We set the granularity of the real robot sonar sensor to 2, 4, and 64 (see figure 5.16). As was the case in the simulation experiment, the best performance was obtained with granularity 4, that is, when the granularity matched the number of levels used by the trainer to provide reinforcements. It is interesting to note (see table 5.1) that the overall performance level was lower in the real robot case. This is due to the difference between the simulated and the real sonar: the real sonar was able to detect a distant

obstacle only if the sonar beam hit the obstacle with an angle very close to 90 degrees.

5.4 POINTS TO REMEMBER

- Results obtained in simulation carry over smoothly to our real robots, which suggests that we can use simulation to develop the control systems and fine-tune them on real robots.
- The kind of data used by the trainer can greatly influence the learning capabilities of the learning system. While this can seem a trivial point, we have shown how two reasonable ways to use data on a robot's behavior determine a different robustness of the learning system to the robot's malfunctioning.
- As was the case in simulation experiments, sensor granularity is at best when it matches the number of levels used by the trainer to provide reinforcements.

Chapter 6

Beyond Reactive Behavior

6.1 INTRODUCTION

In the experiments presented in the two previous chapters, our robots were viewed as *reactive systems*, that is, as systems whose actions are completely determined by current sensory input. Although our results show that reactive systems can learn to carry out fairly complex tasks, there are interesting behavioral patterns not determined by current perceptions alone that they simply cannot learn. In one important class of nonreactive tasks, *sequential behavior patterns*, the decision of what action to perform at time t is influenced by the actions performed in the past.

This chapter presents an approach to the problem of learning sequential behavior patterns based on the coordination of separate, previously learned basic behaviors. Section 6.2 defines some technical terminology and situates the problem of sequential behavior in the context of a general approach to the development of autonomous agents. Section 6.3 defines the agents, the environment, and the behavior patterns on which we carried out our experimentation, and specifies our experimental plan. Section 6.4 describes the two different training strategies we used in our experiments, while section 6.5 reports some of our results.

6.2 REACTIVE AND DYNAMIC BEHAVIORS

As the simplest class of agents, *reactive systems* are agents that react to their current perceptions (Wilson 1990; Littman 1993). In a reactive

This chapter is based on the article "Training Agents to Perform Sequential Behavior" by M. Colombetti and M. Dorigo, which appeared in *Adaptive Behavior* 2 (3), © 1994 The MIT Press.

system, the action $a(t)$ produced at time t is a function of the sensory input $s(t)$ at time t:

$a(t) = f(s(t))$.

As argued by Whitehead and Lin (1995), reactive systems are perfectly adequate to *Markov environments*, more specifically, when

- the *greedy control strategy* is globally optimal, that is, choosing the locally optimal action in each environmental situation leads to a course of actions that is globally optimal; and
- the agent has knowledge about both the effects and the costs (gains) of each possible action in each possible environmental situation.

Although fairly complex behaviors can be carried out in Markov environments, very often an agent cannot be assumed to have complete knowledge about the effects or the costs of its own actions. Non-Markov situations are basically of two different types:

1. *Hidden state environments.* A *hidden state* is a part of the environmental situation not accessible to the agent but relevant to the effects or to the costs of actions. If the environment includes hidden states, a reactive agent cannot select an optimal action; for example, a reactive agent cannot choose an optimal movement to reach an object that it does not see.
2. *Sequential behaviors.* Suppose that at time t an agent has to choose an action as a function of the action performed at time $t-1$. A reactive agent can perform an optimal choice only if the action performed at time $t-1$ has some characteristic and observable effect at time t, that is, only if the agent can infer which action it performed at time $t-1$ by inspecting the environment at time t. For example, suppose that the agent has to put an object in a given position, and then to remove the object. If the agent is able to perceive that the object is in the given position, it will be able to appropriately sequence the placing and the removing actions. On the other hand, a reactive agent will not be able to act properly at time t if
(i) the effects of the action performed at time $t-1$ cannot be perceived by the agent at time t (this is a subcase of the hidden state problem); or
(ii) no effect of the action performed at time $t-1$ persists in the environment at time t.

To develop an agent able to deal with non-Markov environments, one must go beyond the simple reactive model. We say that an agent is "dynamic" if the action $a(t)$ it performs at time t depends not only on its

current sensory input $s(t)$ but also on its *state* $x(t)$ at time t; in turn, such state and the current sensory input determine the state at time $t+1$:

$a(t) = f(s(t), x(t))$;

$x(t+1) = g(s(t), x(t))$.

In this way, the current action can depend on the past history.

An agent's states can be called "internal states," to distinguish them from the states of the environment. They are often regarded as memories of the agent's past or as representations of the environment. In spite (or because) of their rather intuitive meaning, however, terms like *memory* or *representation* can easily be used in a confusing way. Take the packing task proposed by Lin and Mitchell (1992, p. 1) as an example of non-Markov problem:

> [...] Consider a packing task which involves 4 steps: open a box, put a gift into it, close it, and seal it. An agent driven only by its current visual percepts cannot accomplish this task, because when facing a closed box the agent does not know if a gift is already in the box and therefore cannot decide whether to seal or open the box.

It seems that the agent needs to remember that it has already put a gift into the box. In fact, the agent must be able to assume one of two distinct internal states, say 0 and 1, so that its controller can choose different actions when the agent is facing a closed box. We can associate state 0 with "The box is empty," and state 1 with "The gift is in the box." Clearly, the state must switch from 0 to 1 when the agent puts the gift into the box. But now, the agent's state can be regarded:

1. as a *memory* of the past action "Put the gift into the box;" or
2. as a *representation* of the hidden environmental state "The gift is in the box."

Probably, the choice of one of these views is a matter of personal taste. But consider a different problem. There are two distinct objects in the environment, say A and B. The agent has to reach A, touch it, then reach B, touch it, then reach A again, touch it, and so on. In this case, provided that touching an object does not leave any trace on it, there is no hidden state in the environment to distinguish the situations in which the agent should reach A from those in which it should reach B. We say that this environment is "forgetful" in that it does not keep track of the past actions of the agent. Again, the agent must be able to assume two distinct internal states, 0 and 1, so that its task is to reach A when in state 0, and

to reach *B* when in state 1. Such internal states cannot be viewed as representations of hidden states because there are no such states in this environment. However, we still have two possible interpretations:

1. Internal states are *memories* of past actions: 0 means that *B* has just been touched, and 1 means that *A* has just been touched; or
2. Internal states are *goals*, determining the agent's current task: state 0 means that the current task is to reach *A*, state 1 means that the current task is to reach *B*.

The conclusion we draw is that terms like *memory*, *representation*, and *goal*, which are very commonly used for example in artificial intelligence, often involve a subjective interpretation of what is going on in an artificial agent. The term *internal state*, borrowed from systems theory, seems to be neutral in this respect, and it describes more faithfully what is actually going on in the agent.

In this chapter, we will be concerned with internal states that keep track of past actions, so that the agent's behavior can follow a sequential pattern. In particular, we are interested in dynamic agents possessing internal states by design, and learning to use them to produce sequential behavior. Then, if we interpret internal states as goals, this amounts to learning an action plan, able to enforce the correct sequencing of actions, although this intuitive idea must be taken with some care.

Let us observe that not all behavior patterns that at first sight appear to be based on an action plan are necessarily dynamic. Consider an example of *hoarding behavior*: an agent leaves its nest, chases and grasps a prey, brings it to its nest, goes out for a new prey, and so on. This sequential behavior can be produced by a reactive system, whose stimulus-response associations are described by the following production rules (where only the most specific production whose conditions are satisfied is assumed to fire at each cycle):

Rule 1: $\rightarrow$ move randomly.

Rule 2: not grasped & prey ahead $\rightarrow$ move ahead.

Rule 3: not grasped & prey at contact $\rightarrow$ grasp.

Rule 4: grasped & nest ahead $\rightarrow$ move ahead.

Rule 5: grasped & in nest $\rightarrow$ drop.

In fact, the experiments reported in chapters 4 and 5 show that a reactive agent implemented with ALECSYS can easily learn to perform tasks similar to the one above.

It is interesting to see why the behavior pattern described above, while merely reactive, appears as sequential to an external observer. If, instead of the agent's behavior, we consider the behavior of the global dynamic system constituted by the agent and the environment, the task is actually dynamic. The relevant states are the states of the environment, which keeps track of the effects of the agent's moves; for example, the effects of a grasping action are stored by the environment in the form of a grasped prey, which can then be perceived by the agent. We call *pseudo-sequences* tasks performed by a reactive agent that are sequential in virtue of the dynamic nature of the environment, and use the term *proper sequences* for tasks that can be executed only by dynamic agents, in virtue of their internal states.

Let us look at these considerations from another perspective. As it has already been suggested in the literature (see, for example, Rosenschein and Kaelbling 1986; Beer 1995), the agent and the environment can be viewed as a global system, made up of two coupled subsystems. For the interactions of the two subsystems to be sequential, at least one of them must be properly dynamic in the sense that its actions depend on the subsystem's state. The two subsystems are not equivalent, however. Even though the agent can be shaped to produce a given target behavior, the dynamics of the environment are taken as given and cannot be trained. It is therefore interesting to see whether the only subsystem that can be trained, namely, the agent, can contribute to a sequential interaction with states of its own: this is what we have called a "proper sequence."

In this chapter, instead of experimenting with sequences of single actions, we have focused on tasks made up of a sequence of phases, where a phase is a subtask that may involve an arbitrary number of single actions. Again, the problem to be solved is, how can we train an agent to switch from the current phase to the next one on the basis of both current sensory input and knowledge of the current phase?

One important thing to be decided is when the phase transition should occur. The most obvious assumption is that a transition signal is produced by the trainer or by the environment, and is perceived by the agent. Clearly, if we want to experiment on the agent's capacity to produce proper behavioral sequences, the transition signal must not itself convey information about which should be the next phase.

6.3 EXPERIMENTAL SETTINGS

This section presents the environments, the agent, and the target behavior for our experiments.

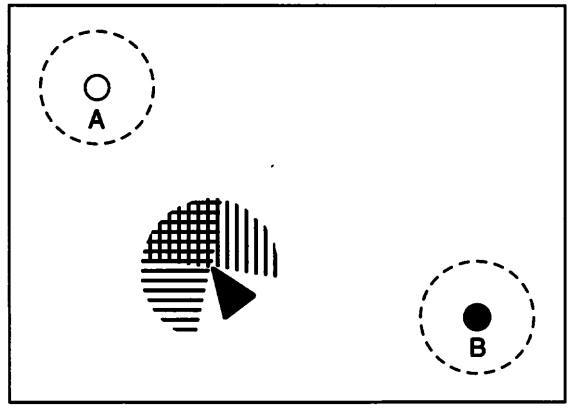


Figure 6.1
Forgetful environment for sequential behavior

6.3.1 The Simulation Environment

What is a good experimental setting to show that proper sequences can emerge? Clearly, one in which (1) agent-environment interactions are sequential, and (2) the sequential nature of the interactions is not due to states of the environment. Indeed, under these conditions we have the guarantee that the relevant states are those of the agent. Therefore, we carried out our initial experiments on sequential behavior in forgetful environments—environments that keep no track of the effects of the agent's move.

Our experimental environment is basically an empty space containing two objects, *A* and *B* (figure 6.1). The objects lie on a bidimensional plane in which the agent can move freely; the distance between them is approximately 100 forward steps of the agent. In some of the experiments, both objects emit a signal when the agent enters a circular area of predefined radius around the object (shown by the dashed circles in the figure).

6.3.2 The Agent's "Body"

The AutonoMouse used in these experiments is very similar to simulated AutonoMice used for experiments in chapter 4. Its sensors are two on/off eyes with limited visual field of 180 degrees and an on/off microphone. The eyes are able to detect the presence of an object in their visual fields, and can discriminate between the two objects *A* and *B*. The visual fields of

the two eyes overlap by 90 degrees, so that the total angle covered by the two eyes is 270 degrees, partitioned into three areas of 90 degrees each (see figure 6.1). The AutonoMouse's effectors are two independent wheels that can either stay still, move forward one or two steps, or move backward one step.

6.3.3 Target Behavior

The target behavior is the following: the AutonoMouse has to approach object A, then approach object B, then approach object A, and so on. This target sequence can be represented by the regular expression $\{\alpha\beta\}^*$, where α denotes the behavioral phase in which the AutonoMouse has to approach object A, and β the behavioral phase in which the AutonoMouse has to approach object B. We assume that the transition from one phase to the next occurs when the AutonoMouse senses a transition signal. This signal tells the AutonoMouse that it is time to switch to the next phase, but does not tell it which phase should be the next.

Concerning the production of the transition signal, there are basically two possibilities:

1. *External-based transition*, where the transition signal is produced by an external source (e.g., the trainer) independently of the current interaction between the agent and its environment; or
2. *Result-based transition*, where the transition signal is produced when a given situation occurs as a result of the agent's behavior; for example, a transition signal is generated when the AutonoMouse has come close enough to an object.

The choice between these two variants corresponds to two different intuitive conceptions of the overall task. If we choose external-based transitions, what we actually want from the AutonoMouse is that it learn to switch phase each time we tell it to. If, instead, we choose result-based transitions, we want the AutonoMouse to achieve a given result, and then to switch to the next phase. Suppose that the transition signal is generated when the agent reaches a given threshold distance from A or from B. This means that we want the agent to reach object A, then to reach object B, and so on. As we shall see, the different conceptions of the task underlying this choice influence the way in which the AutonoMouse can be trained.

It would be easy to turn the environment described above into a Markov environment so that a reactive agent could learn the target behavior. For example, we could assume that A and B are two lights, alternatively

switched on and off, exactly one light being on at each moment. In this case, a reactive AutonoMouse could learn to approach the only visible light, and a pseudo-sequential behavior would emerge as an effect of the dynamic nature of the environment.

6.3.4 The Agent's Controller and Sensorimotor Interfaces

For the $\{\alpha\beta\}^*$ behavior, we used a two-level hierarchical architecture (see chapter 3) that was organized as follows. Basic modules consisted of two independent LCSs, LCS_α and LCS_β, in charge of learning the two basic behaviors α and β. The coordinator consisted of one LCS, in charge of learning the sequential coordination of the lower-level modules.

The input of each basic module represents the relative direction in which the relevant object is perceived. Given that the AutonoMouse's eyes partition the environment into four angular areas, both modules have a two-bit sensory word as input. At any cycle, each basic module proposes a motor action, which is represented by four bits coding the movement of each independent wheel (two bits code the four possible movements of the left wheel, and two bits those of the right wheel).

Coordination is achieved by choosing for execution exactly one of the actions proposed by the lower-level modules. This choice is based on the value of a one-bit word, which represents the internal state of the agent, and which we therefore call the "state word." The effect of the state word is hardwired: when its value is 0, the action proposed by LCS_α is executed; when the value is 1, it is LCS_β that wins.

The coordinator receives as input the current value of the state word and one bit representing the state of the transition signal sensor; this bit is set to 1 at the rising edge of the transition signal and to 0 otherwise. The possible actions for the coordinator are (1) set the state word to 0, and (2) set the state word to 1. The task that the coordinator has to learn is "Maintain the same phase" if no transition signal is perceived, and "Switch phase" each time a transition signal is perceived.

The controller architecture, described in figure 6.2, is a dynamic system. At each cycle t, the sensory input at t and the value of the state word at t jointly determine both the action performed at t and the value of the state word at $t+1$.

6.3.5 Experimental Methodology

For each experiment reported in this chapter we ran twelve independent trials, starting from random initial conditions. Each trial included

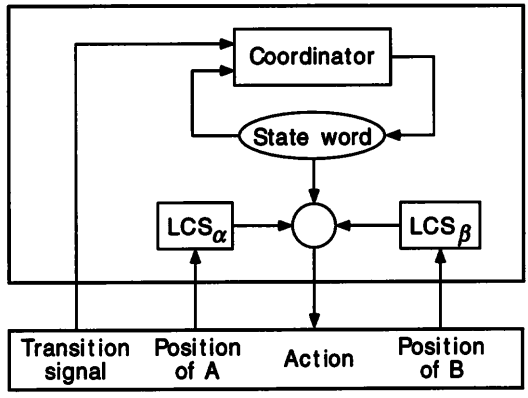


Figure 6.2
Controller architecture of AutonoMouse

- a *basic learning session* of 4,000 cycles, in which the two basic behaviors α and β were learned;
- a *coordinator learning session* of 12,000 cycles, in which learning of basic behaviors was switched off and only the coordinator was allowed to learn; and
- a *test session* of 4,000 cycles, where all learning was switched off and the performance of the agent evaluated.

In the learning sessions, the agent's performance $P_{\text{learn}}(t)$ at cycle t was computed for each trial as

$$P_{\text{learn}}(t) = \frac{\text{Number of correct actions performed from cycle 1 to cycle } t}{t},$$

where an action was considered as correct if it was positively reinforced. We called the graph of $P_{\text{learn}}(t)$ for a single trial a "learning curve." In the test session, the agent's performance P_{test} was measured for each trial as a single number:

$$P_{\text{test}} = \frac{\text{Number of correct actions performed in the test session}}{4{,}000}.$$

For each experiment we show the coordinator learning curve of a single, typical trial, and report the mean and standard deviation of the twelve P_{test} values for (1) the two basic behaviors (α and β); and (2) the two coordinator's tasks ("maintain" and "switch"). It is important to remark that the performance of the coordinator was judged from the

overall behavior of the agent. That is, the only information available to evaluate such performance was whether the agent was actually approaching A or approaching B; no direct access to the coordinator's state word was allowed. Instead, to evaluate the performance of the basic behaviors, it was also necessary to know at each cycle whether the action performed was suggested by LCS_α or by LCS_β; this fact was established by directly inspecting the internal state of the agent.

Finally, to establish whether different experiments resulted in significantly different performances, we computed the probability, p, that the sample of performances produced by the different experiments were drawn from the same population. To compute p, we used the Mann-Whitney test (see section 4.1.1).

6.4 TRAINING POLICIES

As we said in chapter 1, we view training by reinforcement learning as a mechanism to translate a specification of the agent's target behavior, embodied in the reinforcement policy, into a control program that realizes the behavior. As the learning mechanism carries out the translation in the context of agent-environment interactions, the resulting control program can be highly sensitive to features of the environment that would be difficult to model explicitly in a handwritten control program.

As usual, our trainer was implemented as an RP, which embodied the specification of the target behavior. We believe that an important quality an RP should possess is to be highly *agent-independent*. In other words, we want the RP to base its judgments on high-level features of the agent's behavior, without bothering too much about the details of such behavior. In particular, we want the RP to be as independent as possible from internal features of the agent, which are unobservable to an external observer.

Let us consider the $\{\alpha\beta\}^*$ behavior, where

α = approach object A;

β = approach object B.

The transitions from α to β and from β to α should occur whenever a transition signal is perceived.

The first step is to train the AutonoMouse to perform the two basic behaviors α and β. This is a fairly easy task, given that the basic behaviors are instances of approaching responses that can be produced by a simple

reactive agent. After the basic behaviors have been learned, the next step is to train the AutonoMouse's coordinator to generate the target sequence. Before doing so, we have to decide how the transition signal is to be generated. We experimented with both external-based and result-based transitions.

6.4.1 External-Based Transitions

External-based transition signals are generated by the trainer. Let us assume that coordinator training starts with phase α. The trainer rewards the AutonoMouse if it approaches object A, and punishes it otherwise. At random intervals, the trainer generates a transition signal, which lasts one cycle. After the first transition signal is generated, the AutonoMouse is rewarded if it approaches object B, and punished otherwise, and so on.

We can represent this strategy as a set of rules, shown in table 6.1. Columns 2 and 5 report the "correct" phase (i.e., the phase from the trainer's point of view) at times $t-1$ and t. Clearly, the correct phase at t is the same as the correct phase at $t-1$, when the transition signal at $t-1$ (reported in column 4) is OFF, and it is the opposite phase when the transition signal at $t-1$ is ON. Columns 3 and 6 report the phase from the point of view of the AutonoMouse. Note that the reinforcement given at t (column 7) depends only on a comparison between the trainer's and the AutonoMouse's points of view at time t, independently of the transition signal at $t-1$. In other words, reinforcements can be given according to the much simpler table 6.2.

However, this strategy is going to lead us into troubles. To understand why, remember that the AutonoMouse cannot read the trainer's mind, and that it can only try to correlate the reinforcement received with its own behavior and perception of the transition signal. If we extract from table 6.1 only the information available to the AutonoMouse, we obtain table 6.3. Unfortunately, this table is incoherent: the upper and the lower halves of the table are opposite with respect to the reinforcements.

To give a concrete example, let us suppose that in phase α the AutonoMouse is correctly approaching A, but does not change behavior in presence of a transition signal (i.e., it makes an error). Given that the AutonoMouse continues to approach A while it should now approach B, the trainer will give a punishment, applying rule 3 of table 6.1. But then, suppose that the AutonoMouse goes on approaching A. What should the trainer do? A *rigid reinforcement program* (RP_{rig}) would continue to punish the AutonoMouse, applying rule 9 of table 6.1. Unfortunately,

Table 6.1
Rigid RP: Trainer's strategy for reinforcing AutonoMouse's behavior at time t

Rule no.	Phase at $t-1$ according to trainer	Phase at $t-1$ according to AutonoMouse	Transition signal at $t-1$		Phase at t according to trainer	Phase at t according to AutonoMouse	Reinforcement at t
1	α	α	OFF	$\rightarrow$	α	α	+
2	α	α	OFF	$\rightarrow$	α	β	−
3	α	α	ON	$\rightarrow$	β	α	−
4	α	α	ON	$\rightarrow$	β	β	+
5	α	β	OFF	$\rightarrow$	α	α	+
6	α	β	OFF	$\rightarrow$	α	β	−
7	α	β	ON	$\rightarrow$	β	α	+
8	α	β	ON	$\rightarrow$	β	β	+
9	β	α	OFF	$\rightarrow$	β	α	+
10	β	α	OFF	$\rightarrow$	β	β	+
11	β	α	ON	$\rightarrow$	α	α	+
12	β	α	ON	$\rightarrow$	α	β	−
13	β	β	OFF	$\rightarrow$	β	α	−
14	β	β	OFF	$\rightarrow$	β	β	+
15	β	β	ON	$\rightarrow$	α	α	+
16	β	β	ON	$\rightarrow$	α	β	+

Table 6.2
Rigid RP: Trainer's strategy actually depends only on AutonoMouse's present behavior

Phase at t according to trainer	Phase at t according to AutonoMouse	Reinforcement at t
α	α	+
α	β	−
β	α	−
β	β	+

Table 6.3
Rigid RP: From AutonoMouse's point of view, trainer's strategy appears as incoherent

Rule no.	Phase at $t-1$ according to AutonoMouse	Transition signal at $t-1$		Phase at t according to AutonoMouse	Reinforcement at t
1	α	OFF	→	α	+
2	α	OFF	→	β	−
3	α	ON	→	α	−
4	α	ON	→	β	+
5	β	OFF	→	α	+
6	β	OFF	→	β	−
7	β	ON	→	α	−
8	β	ON	→	β	+
9	α	OFF	→	α	−
10	α	OFF	→	β	+
11	α	ON	→	α	+
12	α	ON	→	β	−
13	β	OFF	→	α	−
14	β	OFF	→	β	+
15	β	ON	→	α	+
16	β	ON	→	β	−

such training strategy is seen as incoherent by the AutonoMouse, which is simply maintaining the same behavior in absence of a transition signal, a conduct that is normally rewarded. Under these conditions, it would be impossible for the AutonoMouse to learn any meaningful policy. This inability has been observed experimentally.

We have therefore applied what we call a "flexible reinforcement program" (RP_{flex}), that is, after the AutonoMouse makes an error, the trainer changes phase. It is the trainer who adapts to the AutonoMouse behavior applying a flexible training method. In this way there will never be incoherence in the trainer behavior: in the previous case, after the AutonoMouse has applied the incorrect rule 3 of table 6.1, the trainer goes back to phase α. If now the AutonoMouse goes on approaching A, then the situation described by rule 1 of table 6.1 will be applied, which

now results in a positive reinforcement. In other words, the trainer's conduct with RP_{flex} can be described as follows:

- Start in phase α;
- When in phase α, reward the AutonoMouse if it approaches punish it otherwise; when in phase β, reward the AutonoMouse if it approaches B, and punish it otherwise;
- Change phase when a transition signal is produced;
- If the AutonoMouse appears not to change behavior in presence of a transition signal, punish it and restore the previous phase; and
- If the AutonoMouse appears to change behavior in absence of a transition signal, punish it and change phase.

The rationale of this reinforcement program is that the trainer punishes an inadequate treatment of the transition signal, but rewards coherency of behavior. Experiments 1, 2, and 3 (next section) have been run using RP_{flex}.

6.4.2 Result-Based Transitions

Let us now suppose that the target sequential behavior is understood as follows: the agent should approach and reach object A, then approach and reach object B, and so on. A major difference with respect to the previous case is that a transition signal is now generated each time the agent comes close enough to an object (see the dashed circles in figure 6.1). If we apply the rigid reinforcement program, the AutonoMouse does not learn the target behavior, for the reasons already pointed out in section 6.4.1. But in this case, it no longer makes sense for the trainer to flexibly change phase when the agent switches behavior: a phase is completed only when a given result is achieved, that is, when the relevant object is reached.

It is easy to see that incoherent situations can be eliminated by adding an extra sensor to the AutonoMouse that records information about the sign of the reinforcement received. This extra sensor, which we call a "reinforcement sensor," is a one-bit field in the sensory interface that tells the AutonoMouse whether the previous action had been rewarded or punished. Table 6.4 shows the rationale of RP_{rig}, including the reinforcement given at $t-1$. Table 6.5 reports the information available to the AutonoMouse in presence of a reinforcement sensor, assuming this is set to 1 when an action is rewarded, and to 0 when an action is punished. Contrary to table 6.3, table 6.5 is no longer incoherent, in that the corre-

Table 6.4
Rigid RP: Trainer's strategy for reinforcing AutonoMouse's behavior, including reinforcement given at $t-1$

Rule no.	Phase at $t-1$ according to trainer	Phase at $t-1$ according to AutonoMouse	Reinforcement at $t-1$	Transition signal at $t-1$		Phase at t according to trainer	Phase at t according to AutonoMouse	Reinforcement at t
1	α	α	+	OFF	→	α	α	+
2	α	α	+	OFF	→	α	β	−
3	α	α	+	ON	→	β	α	−
4	α	α	+	ON	→	β	β	+
5	α	β	−	OFF	→	α	α	+
6	α	β	−	OFF	→	α	β	−
7	α	β	−	ON	→	β	α	−
8	α	β	−	ON	→	β	β	+
9	β	α	−	OFF	→	β	α	−
10	β	α	−	OFF	→	β	β	+
11	β	α	−	ON	→	α	α	+
12	β	α	−	ON	→	α	β	−
13	β	β	+	OFF	→	β	α	−
14	β	β	+	OFF	→	β	β	+
15	β	β	+	ON	→	α	α	+
16	β	β	+	ON	→	α	β	−

Table 6.5
Rigid RP: Adding reinforcement sensor, trainer's strategy for reinforcing AutonoMouse's behavior no longer appears as incoherent

Rule no.	Phase at $t-1$ according to AutonoMouse	Reinforcement sensor at $t-1$	Transition signal at $t-1$		Phase at t according to AutonoMouse	Reinforcement at t
1	α	1	OFF	→	α	+
2	α	1	OFF	→	β	−
3	α	1	ON	→	α	−
4	α	1	ON	→	β	+
5	β	0	OFF	→	α	+
6	β	0	OFF	→	β	−
7	β	0	ON	→	α	−
8	β	0	ON	→	β	+
9	α	0	OFF	→	α	−
10	α	0	OFF	→	β	+
11	α	0	ON	→	α	+
12	α	0	ON	→	β	−
13	β	1	OFF	→	α	−
14	β	1	OFF	→	β	+
15	β	1	ON	→	α	+
16	β	1	ON	→	β	−

sponding lines of its upper and lower halves are now distinguished by the state of the reinforcement sensor.

Experiments 4, 5, and 6 (next section) show that the AutonoMouse is able to learn the target behavior when trained with the RP_{rig} if its sensory interface includes the reinforcement sensor. We think that the notion of reinforcement sensor is not trivial and therefore needs to be discussed in some detail.

6.4.3 Meaning and Use of the Reinforcement Sensor

At each moment, the reinforcement sensor stores information about what happened in the previous cycle; as such, it contributes to the agent's dynamic behavior. Its characteristic feature is that it stores information about the behavior of the trainer, not of the physical environment. It may seem that such information is available to the AutonoMouse even without

the reinforcement sensor because it is received and processed by the credit apportionment module of ALECSYS. The point is that no information about reinforcement is available to the AutonoMouse's controller unless it is coded into the sensory interface. In a sense, an agent endowed with the reinforcement sensor not only *receives* reinforcements but also *perceives* them.

It is interesting to see how the information stored by the reinforcement sensor is exploited by the learning process. When trained with the RP_{rig}, the AutonoMouse will develop behavior rules that, when the reinforcement sensor is set to 0, change phase in absence of a transition signal (rules 5 and 10 of tables 6.4 and 6.5), and do not change phase in presence of a transition signal (rules 8 and 11 of tables 6.4 and 6.5). Such rules can be viewed as *error recovery rules* in that they tell the agent what to do in order to fix a previous error in phase sequencing. Rules matching messages with the reinforcement sensor set to 1 will be called "normal rules," to distinguish them from error recovery rules.

Without a reinforcement sensor, punishments are exploited by the system only to decrease the strength of a rule that leads to an error (i.e., to an incorrect action). With the reinforcement sensor, punishments are used for one extra purpose, to enable error recovery rules at the next cycle. In general, as learning proceeds, fewer and fewer errors are made by the AutonoMouse, and the error recovery rules become increasingly weaker, so that sooner or later they are removed by the genetic algorithm.

Error recovery rules presuppose a reinforcement, and thus can be used only as far as the trainer is on. If the trainer is switched off to test the acquired behavior, the reinforcement sensor must be clamped to 1 so that normal rules can be activated. This means that error recovery rules will remain silent after the trainer is switched off; it is therefore advisable to do so only after all recovery rules have been eliminated by the genetic algorithm.

In the experiments reported in this chapter, error recovery rules were either eliminated before we switched off the trainer, or they became so weak that they were practically no longer activated. With more complex tasks, however, one can easily imagine situations where some error recovery rules could maintain a strength high enough to survive and to contribute to the final behavior, and where switching off the trainer would actually impoverish the final performance. On the other hand, one could still switch off the learning algorithm: the use of error recovery rules presupposes that an external system gives positive or negative "judgments"

about the AutonoMouse's actions but does not require the learning algorithm to be active. After switching off learning, the trainer actually turns into an *advisor*, that is, an external observer in charge of telling the agent, which is no longer learning anything, whether it is doing well or not.

We still do not know whether the use of an advisor has interesting practical applications; it seems to us that it could be useful in situations where the environment is so unpredictable that even the application of the most reasonable control strategy would frequently lead to errors. In such situations, it would not be possible to avoid errors through further learning; error recovery would therefore seem to be an appealing alternative.

6.5 EXPERIMENTAL RESULTS

This section reports the results of the experiments on the $\{\alpha\beta\}^*$ behavior. The experiments described are the following:

- Experiments 1–3: sequential behavior with external-based transitions and flexible reinforcement program.
- Experiments 4–6: sequential behavior with result-based transitions, rigid reinforcement program and reinforcement sensor.

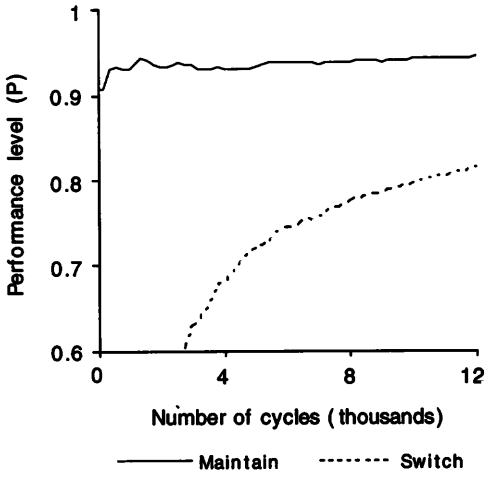
The performances of the coordinator tasks ("maintain" and "switch") are then compared by the Mann-Whitney test.

6.5.1 Experiment 1: External-Based Transitions and Flexible Reinforcement Program

In this experiment, transition signals are produced randomly, with an average of one signal every 50 cycles. The AutonoMouse is trained with the flexible reinforcement program, RP_{flex}. Figure 6.3 shows a typical learning curve for the coordinator learning session and reports the mean and standard deviation of the performances obtained in the test session of twelve trials.

It appears that the AutonoMouse learns to maintain the current phase (in absence of a transition signal) better than to switch phase (when it perceives a transition signal). This result is easy to interpret: as transition signals are relatively rare, the AutonoMouse learns to maintain the current phase faster than it learns to switch phase.

As a whole, however, the performance of the coordinator is not fully satisfactory, at least as far as the switch task is concerned. One factor that



Performance in test session: basic behavior

	Task α	Task β
Mean	0.9155	0.8850
Std. dev.	0.0656	0.1192

Performance in test session: coordination

	Maintain	Switch
Mean	0.9337	0.8276
Std. dev.	0.0471	0.0872

Figure 6.3
Experiment 1: Learning sequential behavior with external-based transitions

keeps the performance of the coordinator well below 1 is that the performances of the two basic behaviors are not close enough to 1. Indeed, during the training of the coordinator, an action may be punished even if the coordinator has acted correctly, if a wrong move is proposed by the relevant basic LCS.

There is, however, another reason why the learning of the coordination tasks is not satisfactory: RP_{flex} cannot teach perfect coordination because there are ambiguous situations, that is, situations where it is not clear whether the reinforcement program should reward or punish the agent. Suppose, for example, that the AutonoMouse perceives a transition signal at cycle t when it is approaching A on a curvilinear trajectory like the one shown in figure 6.4, and that at cycle $t+1$ it keeps on following the same trajectory. By observing this behavior, one cannot tell whether the AutonoMouse decided to keep on approaching A or whether it changed phase and is now turning to approach object B. Because the agent's behavior is ambiguous, any reinforcement actually runs the risk of saying the opposite of what it is intended to.

Ambiguous situations of the type described earlier arise because the agent's internal state is hidden from the point of view of the trainer. One possible solution is to make the relevant part of this state known to RP_{flex}. This was implemented in the next experiment.

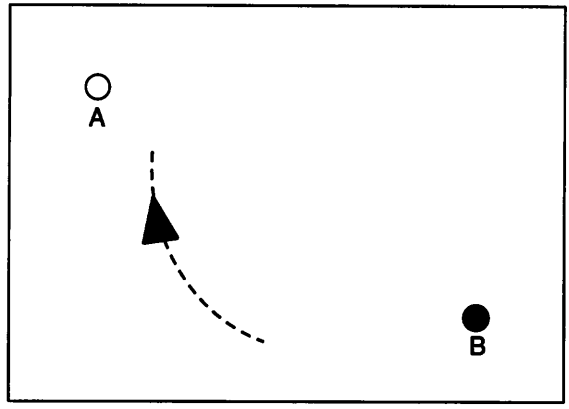


Figure 6.4
Ambiguous situation

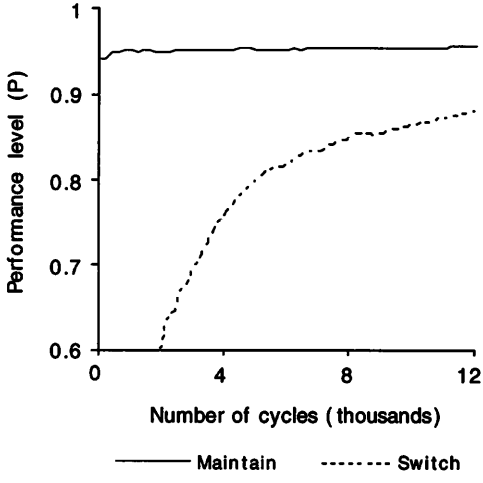
6.5.2 Experiment 2: External-Based Transitions, Flexible Reinforcement Program, and Agent-to-Trainer Communication

To eliminate ambiguous situations, we simulate a communication process from the agent to the trainer: better reinforcements can be generated if the agent communicates its state to the reinforcement program, because situations like the one described earlier are no longer ambiguous.

To achieve this result, we added to the AutonoMouse the ability to assume two externally observable states, which we conventionally call "colors." The AutonoMouse can be either "white" or "black," and can assume either color as the result of a specific "chromatic" action. The trainer can observe the AutonoMouse's color at any time. The basic modules are now able to perform one more action at each cycle, that is, to set a color bit to 0 (white) or to 1 (black); for this purpose, an extra bit is added to the action part of classifiers. In the basic learning session, the AutonoMouse is trained not only to perform the approaching behaviors α and β but also to associate a single color to each of them. During the coordinator learning session, RP_{flex} exploits information about the color to disambiguate the AutonoMouse's internal state. The results of this experiment are reported in figure 6.5.

6.5.3 Experiment 3: External-Based Transitions, Flexible Reinforcement Program, and Transfer of the Coordinator

Another interesting solution to the problem of ambiguous situations is based on the notion of transferring behavioral coordination. The idea is



Performance in test session: basic behavior

	Task α	Task β
Mean	0.8893	0.9442
Std. dev.	0.0650	0.0509

Performance in test session: coordination

	Maintain	Switch
Mean	0.9493	0.8876
Std. dev.	0.0290	0.0809

Figure 6.5
Experiment 2: External-based transitions and agent-to-trainer communication

that the $\{\alpha\beta\}^*$ behavior is based on two components: the ability to perform the basic behaviors α and β; and the ability to coordinate them in order to achieve the required sequence. While the basic behaviors are strongly linked to the environment, coordination is abstract enough to be learned in one environment, and then transferred to another. Therefore, we proceed as follows:

1. The AutonoMouse learns the complete $\{\alpha\beta\}^*$ behavior in a simpler environment, where the ambiguity problem does not arise;
2. The AutonoMouse is then trained to perform the two basic behaviors in the target environment; and
3. Finally, the coordinator rules learned in the simpler environment are copied into the coordinator for the target task. For this purpose, we select the rules that have the highest performance in the simpler environment; therefore, all twelve experiments in the target environment are run with the same coordinator.

The simpler, unambiguous environment used for coordination training is sketched in figure 6.6. It is a one-dimensional counterpart of the target environment: the AutonoMouse can only move to the left or to the right on a fixed rail. At each instant, the AutonoMouse is either approaching *A* or approaching *B*: no ambiguous situations arise.

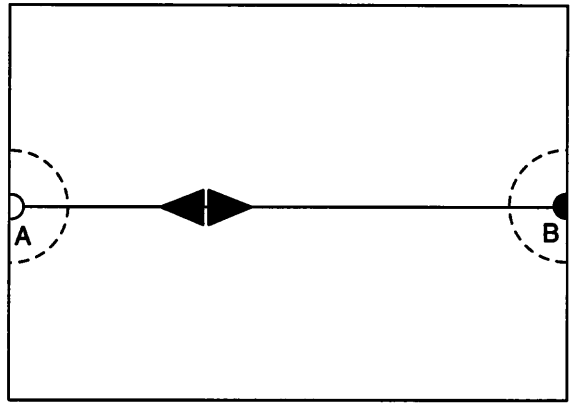


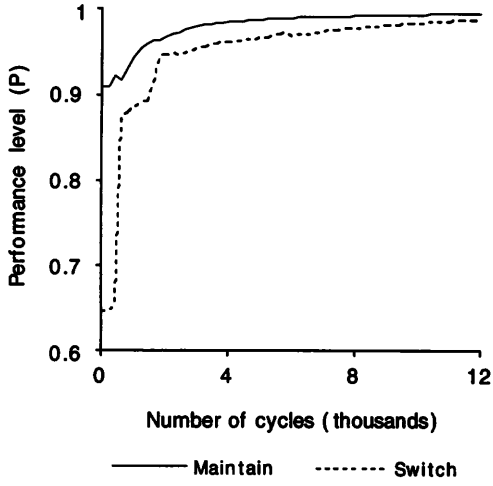
Figure 6.6
1-D environment

As reported in figure 6.7, due to the simplicity of the task, the performance achieved in the 1-D environment is almost perfect. Figure 6.8 shows the results obtained by transferring the coordinator to an Autono-Mouse that had previously learned the basic behaviors in the 2-D environment. The reason why the performance of the coordination task is higher than in the case of agent-to-trainer communication is that with agent-to-trainer communication the significance of the communication bit had to be learned.

6.5.4 Comparison of Experiments 1–3

In experiments 1–3, three pairwise comparisons are possible, only two of which are statistically independent. We decided to use experiment 1 as a control and to compare the other two experiments with it. The Mann-Whitney test tells us that

- performance of the maintain task in experiment 2 is not significantly different ($p = 0.4529$, adjusted for ties) from that in experiment 1;
- performance of the maintain task in experiment 3 is higher than in experiment 1, but the statistical significance of the difference is low ($p = 0.1059$, adjusted for ties);
- performance of the switch task in experiment 2 is higher than in experiment 1, but the statistical significance of the difference is low ($p = 0.0996$, adjusted for ties);



Performance in test session: basic behaviors

	Task α	Task β
Mean	0.9999	0.9999
Std. dev.	0.0003	0.0002

Performance in test session: coordination

	Maintain	Switch
Mean	0.9996	0.9963
Std. dev.	0.0012	0.0066

Figure 6.7
Experiment 3: External-based transitions in 1-D environment

Performance in test session: basic behaviors

	Task α	Task β
Mean	0.9646	0.9649
Std. dev.	0.0367	0.0315

Performance in test session: coordination

	Maintain	Switch
Mean	0.9615	0.9501
Std. dev.	0.0286	0.0364

Figure 6.8
Experiment 3: Transferring external-based coordinator from 1-D to 2-D environment

- performance of the switch task in experiment 3 is higher than in experiment 1, with high statistical significance ($p = 0.0018$, adjusted for ties).

Taking into account the mean performances of the switch task in the three experiments (see figures 6.3, 6.5, and 6.8), we conclude that, with external-based transitions,

1. The maintain task is learned reasonably well with all the strategies tested; transferring the coordinator appears to be the best approach, even if the difference with direct learning (experiment 1) cannot be considered statistically significant; and
2. For the switch task, transfer of the coordinator clearly appears to be the best approach, although agent-to-trainer communication also seems to overcome the problem of learning in ambiguous situations, albeit with low statistical significance.

6.5.5 Experiment 4: Result-Based Transitions and Rigid Reinforcement Program with Reinforcement Sensor

This and the next two experiments are run with result-based transitions, that is, a transition signal is generated each time the AutonoMouse reaches an object. The target behavior is therefore conceived as: reach object *A*, then reach object *B*, and so on. Consistent with this view of the target behavior, we adopt the rigid reinforcement program RP_{rig} and give the AutonoMouse a one-bit reinforcement sensor.

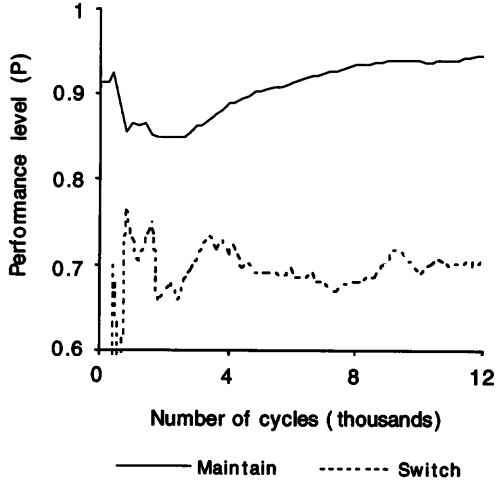
The results, reported in figure 6.9, show that the target behavior is learned, although the performance of the switch task is rather poor in comparison with experiment 1, possibly because the presence of new input information (the reinforcement sensor) makes the task more complex.

6.5.6 Experiment 5: Result-Based Transitions, Rigid Reinforcement Program with Reinforcement Sensor, and Agent-to-Trainer Communication

This experiment is the result-based analogue of experiment 2: the AutonoMouse is trained to assume a color, thus revealing its internal state. The results are reported in figure 6.10. As was the case for experiment 2, the coordinator performance increases noticeably.

6.5.7 Experiment 6: Result-Based Transitions, Rigid Reinforcement Program, and Transfer of the Coordinator

This experiment is the result-based analogue of experiment 3. Figure 6.11 shows the results obtained in the 1-D environment, and figure 6.12 gives



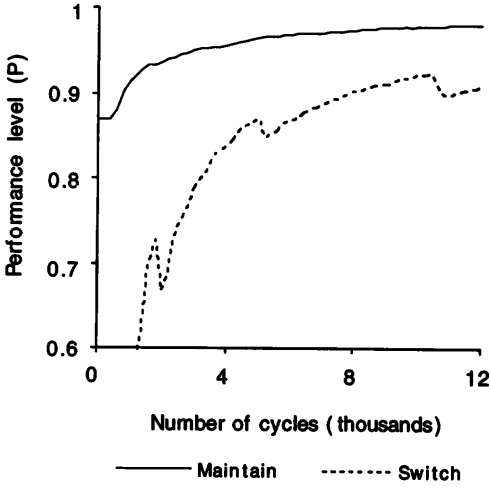
Performance in test session: basic behaviors

	Task α	Task β
Mean	0.9451	0.9743
Std. dev.	0.0426	0.0400

Performance in test session: coordination

	Maintain	Switch
Mean	0.9313	0.7166
Std. dev.	0.0528	0.1559

Figure 6.9
Experiment 4: Learning sequential behavior with result-based transitions



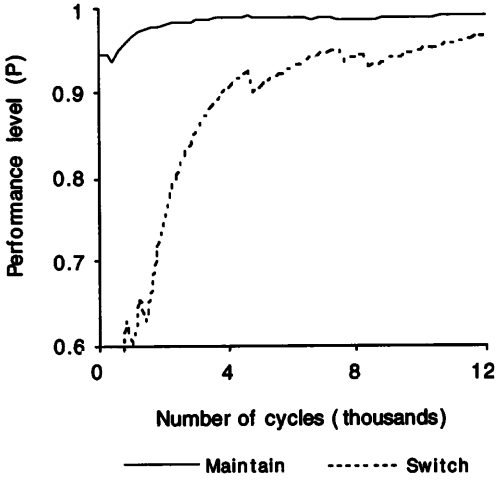
Performance in test session: basic behaviors

	Task α	Task β
Mean	0.9838	0.9879
Std. dev.	0.0201	0.0144

Performance in test session: coordination

	Maintain	Switch
Mean	0.9789	0.8885
Std. dev.	0.0350	0.0788

Figure 6.10
Experiment 5: Result-based transitions and agent-to-trainer communication



Performance in test session: basic behaviors

	Task α	Task β
Mean	0.9999	0.9999
Std. dev.	0.0002	0.0001

Performance in test session: coordination

	Maintain	Switch
Mean	0.9903	0.9670
Std. dev.	0.0317	0.0565

Figure 6.11
Experiment 6: Result-based transitions in 1-D environment

Performance in test session: basic behaviors

	Task α	Task β
Mean	0.9935	0.9895
Std. dev.	0.0067	0.0112

Performance in test session: coordination

	Maintain	Switch
Mean	0.9903	0.9372
Std. dev.	0.0078	0.0559

Figure 6.12
Experiment 6: Transferring result-based coordinator from 1-D to 2-D environment

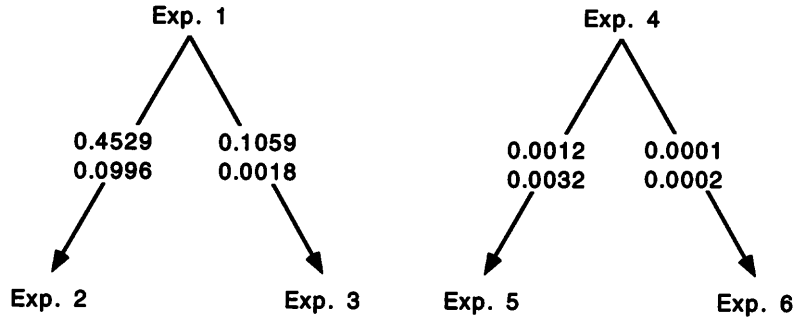


Figure 6.13
Result of Mann-Whitney tests for the "maintain" (above) and "switch" (below) tasks. Arrows indicate the direction of increasing value of performance (e.g., the performance of the maintain task in experiment 4 is lower than in both experiment 5 and experiment 6).

the performances of the AutonoMouse in the 2-D environment, after transferring the best coordinator obtained in the 1-D environment. As with external-based transitions, the performance of the coordination task is higher in the case of transfer of the coordinator than in the case of agent-to-trainer communication. Again, this is because the significance of the communication bit has to be learned.

6.5.8 Comparison of Experiments 4–6

In experiments 4–6, three pairwise comparisons are possible, two of which are independent. We decided to use experiment 4 as a control and to compare the other two experiments with it. In both cases, the Mann-Whitney test tells us that there are highly significant differences:

- performance in experiment 5 is higher than in experiment 4, with high statistical significance for both the maintain task ($p = 0.0012$) and the switch task ($p = 0.0032$);
- performance in experiment 6 is higher than in experiment 4, with high statistical significance for both the maintain task ($p = 0.0001$) and the switch task ($p = 0.0002$).

Taking into account the mean performances of the two tasks in these experiments (see figures 6.9, 6.10 and 6.12), we conclude that, with result-based transitions, coordinator transfer and agent-to-trainer communication significantly overcome the problem of learning in ambiguous situations, both for the maintain and for the switch task. Figure 6.13

summarizes the results of the Mann-Whitney tests for all relevant pairs of this set of experiments.

6.6 POINTS TO REMEMBER

- In the context of artificial systems, the term *internal state* can be profitably used to replace terms like *memory*, *representation*, and *goal*.
- Pseudo-sequences are those tasks performed by a reactive agent that are sequential in virtue of the dynamic nature of the environment.
- Proper sequences are sequential tasks performed by a dynamic agent, that is, an agent endowed with internal state.
- The best policy to train an agent to perform a proper sequence depends on the way in which transition signals are generated: external-based transition signals call for a flexible reinforcement program; result-based transition signals call for a rigid reinforcement program.
- To improve the performance of systems learning under the guidance of a rigid reinforcement program, it is useful to enrich the AutonoMouse with a reinforcement sensor (this can be seen as a kind of trainer-to-agent communication). The use of a reinforcement sensor determines as a side effect the generation of error recovery rules, which are activated only when the system makes an error.
- For both flexible and rigid reinforcement programs, the use of agent-to-trainer communication through some observable behavior improves performance. Performance can be further improved by developing the module in charge of controlling the sequence in a simpler environment where the basic sequential structure of the task, that is, $\{\alpha\beta\}^*$ is maintained, but where ambiguities in judging the AutonoMouse behavior are removed.

Chapter 7

The Behavior Analysis and Training Methodology

7.1 INTRODUCTION

As we suggested in chapter 1, we view our own work as a contribution to a unified discipline of agent development that we call "behavior engineering." Engineering of any kind is a human activity aimed to the development of artificial products and based on a technical apparatus including fundamental concepts, mathematical models, techniques for the application of such models, hardware and software tools, and methodologies. We think that behavior engineering should be no exception.

Our main concern in this book has been to show that learning can have an important role in behavior engineering. As regards robot learning, today there is a wide repertoire of reasonably well understood concepts and models, and techniques for applying them to concrete problems. Moreover, several software tools are available to develop learning systems; in particular, the previous chapters have concentrated mainly on the learning classifier system (LCS) as a model and on ALECSYS as a tool. By contrast, there is no widespread concern about the methodological issues involved in the use of learning as a component of robot development. While the previous chapters have presented various pieces of the whole picture, in this chapter we propose a methodology that summarizes our experiences as designers and builders of learning autonomous robots.

Section 7.2 presents a methodology for behavior engineering that we call BAT (Behavior Analysis and Training). Sections 7.3, 7.4, and 7.5 support the methodology with the description of three practical cases. The

This chapter is based on the article "Behavior Analysis and Training: A Methodology for Behavior Engineering" by M. Colombetti, M. Dorigo, and G. Borghi, which appeared in *IEEE Transactions on Systems, Man, and Cybernetics Part B 26* (3), © 1996 IEEE.

first case, regarding AutonoMouse V, builds upon experiments already presented (however sparsely) in the previous chapters. The other two cases are examples of an application of the BAT methodology to commercially available robots.

7.2 THE BAT METHODOLOGY

The goal of any engineering methodology is to help engineers to develop high-quality products. This involves clarifying various aspects connected with the specification, design, implementation, and assessment of the product.

The first important point to be settled is, how is the quality of the product going to be defined? Product quality is a complex function of many different components and cannot be measured by any simple scalar index. In general, some components of quality can be measured in quantitative terms, while others can only be qualitatively evaluated. In any case, a clear definition of all important aspects of quality may have an important impact on the development of the product.

Before being designed, a product has to be clearly specified. Specifications are important for two reasons: first, they guide the design process; and second, they define a reference against which the final product will be assessed. Moreover, proper specifications are critical because errors in specifications are often discovered only in the later stages of development, when correcting them can be very costly.

The core of the development process is design. At a fairly abstract level, a design process can be viewed as a sequence of decisions. Even though it is of course not possible to define an "algorithm" leading to perfect design, it is possible to point out what are the most critical decisions to be made, in what order they should be taken, and on what grounds. This is precisely what a designer wants from a "methodology."

Finally, the developer should know how to assess the quality of a product, that is, how to practically evaluate the different components of quality. In this section, we shall discuss all these facets from the point of view of agent development.

In the following sections, we shall have to distinguish clearly between different components of the robot, and in particular between its "body" and its "mind." We shall describe a robot as made up of a "shell" (the inert part of its "body"), a "sensorimotor interface" (including sensors and effectors), and a "controller" (its "mind").

To keep our treatment close to common robotic practice, we assume that the problem faced by the engineer is to develop an autonomous robot in a situation where both robot shell and environment are essentially predefined. Of course, we are interested in the cases where the robot controller includes a learned component.

In our experience, the main stages in the development of a robotic application are

1. To describe the robot shell and the initial environment, and to define the robot's *target behavior*, that is, the desired pattern of interaction between the robot and its environment;
2. To analyze the target behavior and decompose it into a structured collection of simple behaviors;
3. To completely specify the various components of the robot, in particular:
(i) the *sensors* and *effectors* interfacing the robot with its environment, possibly together with some artificial extension of the environment to make perception and action possible (for example, the "preys" we often dealt with actively produce a signal that allows the robot to locate them);
(ii) the *controller architecture*, that is, the overall structure of the robot's control program; and
(iii) a *training strategy*, that is, a systematic procedure for training the robot to perform the target behavior;
4. To design, implement, and verify the *nascent robot*, that is, the robot prior to training;
5. To carry out *training* until the target behavior is learned; and
6. To assess the robot's *learning process* and its *final behavior* with respect to the specified target behavior.

These stages can be arranged in a *logical sequence* (fig. 7.1), in which each stage exploits at least part of the outputs of previous stages. It is important to note, however, that the *temporal sequence* in which activities are carried out will in general not be a simple traversal of the logical sequence from the beginning to the end. Some activities may start before the previous one has been terminated; often, inadequacies will be detected that will force the developer to cycle back to a previous stage; and sometimes it will be even necessary to go through the whole logical sequence more than once, thus organizing the development process according to the *spiral model* (see, for example, Ghezzi, Jazayeri, and Mandrioli 1991). In this chapter we shall be concerned only with the logical relationships among stages.

Figure 7.1
Logical sequence of stages in BAT methodology

7.2.1 Application Description and Behavior Requirements

The outputs of this stage are

- a description of the robot shell;
- a description of the initial environment; and
- a set of requirements on the target behavior.

Clearly, a specification of the target behavior presupposes a clear description of both the robot and its physical environment. However, neither of the two can be thoroughly described before completing the specification stage (see section 7.2.3). The reason is that the sensorimotor apparatus of the robot can be fully designed only after the robot's behavior has been analyzed in detail; furthermore, the use of certain types of sensors or effectors may require a modification of the environment to achieve a satisfactory coupling between the robot and the environment. Therefore, in this initial stage we can only describe the robot shell (with incomplete or no sensorimotor apparatus) and the initial environment (prior to possible modification).

The requirements for the target behavior are often stated in qualitative terms; for example, we may state that the robot will have to inspect an area, collecting objects of a given type. Such an informal description of the target behavior, even when clear and unambiguous, is not sufficient as a basis for a later quantitative assessment of the robot's behavior. Therefore, a complete specification of the target behavior should also include

a formal, quantitative component. It is up to the subsequent behavior assessment stage to compare data from robot testing with such specification (see section 7.2.6).

7.2.2 Behavior Analysis

This stage takes as inputs the outputs of the previous stage, and produces as output a description of the structured behavior. The analysis of behavior and the definition of its structure are key aspects of our methodology. The target behavior is first decomposed into simpler component behaviors, which can often be be further decomposed into simpler ones, and so on, producing a hierarchical structure. The product of this decomposition is what we call "structured behavior." Again, it is desirable that the component behaviors be specified in a rigorous, quantitative way so that such specification can be compared with data coming from behavior assessment (see section 7.2.6).

There is no simple rule about how to carve simple behaviors out of more complex ones. For example, although "exploring the environment" can only be established as a single basic behavior on the basis of general knowledge about robot engineering and past design experience, once the basic behaviors have been singled out, their interactions can and should be completely defined.

What types of interactions among behaviors should one take into account? Again, there is no general answer because this issue is strictly connected to the kind of controller architecture one is going to design. In section 3.1, we singled out the following types of interaction: independent sum, combination, suppression, and sequence. As we shall see, a clear description of the interactions between simple behaviors is essential for the specification stage.

7.2.3 Specification

This stage takes as input the output of the previous stage, and produces as outputs the specifications of

- the robot's sensors and effectors;
- the extended environment;
- the controller architecture; and
- the training strategy.

As we have already said, both the robot shell and the initial environment are assumed to be given. In general, however, the sensorimotor

interface of the robot has to be at least partially redesigned for each specific behavior. For example, we may find out that objects in the environment can be roughly located through a sonar belt, and identified through a bar code reader. Analogously, requirements on motor control lead to decisions on the kind of output that the robot controller will have to send to the effectors. For example, suppose we need to control a platform moving on two tracks, connected to two independent motors. Assuming that the controller sends a message to each motor at every control cycle, we may decide that the controller's output can be one of the following: "Move forward"; "Do not move"; "Move backward."

Given the state of the art in robotics, robots are often unable to deal directly with the initial environment. For example, it may be necessary to modify the objects with which the robot has to interact so that they can be detected, identified, and located. It is part of the specification stage to find out what must be added to the environment to make the target behavior possible. The environment thus equipped will be called "extended environment."

Another task of the specification stage is to establish the controller architecture. In our work, we have experimented with various kinds of architecture, implemented via ALECSYS. In section 3.3, we contended that the architecture of the controller should parallel the organization of the structured behavior, as established by behavior analysis. This result was obtained by allocating a *behavioral module* to each simple behavior.

Architectures are classified as discussed in section 3.3. Table 7.1 summarizes the best correspondences between architecture types and mechanisms for building complex behaviors. Examples of the use of such architectures can be found in the previous chapters.

Table 7.1
Best architectural choices for different kinds of complex interactions

	Independent sum	Combination	Suppression	Sequence
Monolithic				
Flat with independent outputs	×			
Flat with integrated outputs		×		
Hierarchical		×	×	×

Finally, there is one more goal for the specification stage. As we have already suggested, the BAT methodology assumes that behaviors are allocated to behavioral modules by design, but the function of each module is developed by machine learning techniques, and we have assumed that the learning system is chosen in advance. However, one should not believe that having a learning system at hand solves all the problems. Even if the agent is able to learn, one still has to *teach* the right thing in the right way; in other words, one has to establish a *training strategy*.

Once more, there is no simple, general rule that can always lead to the right decisions. Moreover, even the *type* of decisions to be made will depend on the kind of learning system adopted. Developers who rely on some kind of reinforcement learning have at least to establish

- the *reinforcement program*, that is, the information that will be provided to the learning system to make the robot converge to the target behavior; and
- the *shaping policy*, that is, whether the structured behavior should be learned in one shot or by a sequence of learning sessions, and if the latter, in which order the various behaviors and their interactions will have to be learned.

Such decisions are critical because training is strongly responsible for the final performance of the robot.

7.2.4 Design, Implementation, and Verification of the Nascent Robot

The documents produced by the previous stages provide sufficient information to design, implement, and verify the *nascent robot*, that is, the robot endowed with all its hardware and software components before it is trained. We do not suggest any specific procedure to be followed regarding this stage: designers should exploit known methodologies for hardware and software engineering. The output of the stage is a physical agent that has to be trained to perform the target behavior.

7.2.5 Training

Training acts on the nascent robot according to the established training strategy. Its output will be the complete robot.

The main problem of training is that it can be an extremely expensive task: even the most efficient learning systems converge slowly, and therefore training may require numerous lengthy sessions. In many cases of practical interest, it is possible to speed up training through the use of

simulators, with only a fraction of the effort needed to train real robots. Another advantage of simulation is that it allows one to carry out training in environments that are more manageable than the real one in some important respect; we have exploited this aspect in the case presented in Section 7.4. In general, however, simulation is not sufficient because much of the richness and unpredictability of the real world is lost. A reasonable compromise involves using a simulator to develop a first approximation of the final controller, which can then be refined through direct training of the real robot.

7.2.6 Behavior Assessment

Before introducing an assessment procedure, we need to clarify our notion of the quality of behavior. Because, broadly speaking, a robot is a hardware/software system, in principle we might simply borrow the quality components identified by hardware and software engineering: robustness, reliability, usability, portability, and so on. However, it seems to us that robots are special enough to deserve an independent treatment. For example, in classical software engineering one would sharply distinguish between correctness, robustness, and performance of a program. In particular:

- *Correctness* is a yes/no property. A program is said to be "correct" (with respect to an input-output specification) when it yields the specified outputs for all allowable inputs.
- *Robustness*, defined as the ability of the program to behave acceptably even when its input data lie outside the specified input set, is generally evaluated in qualitative terms.
- *Performance* is defined over a collection of metrics measuring the amount of resources (including memory and CPU time) necessary to produce an output.

A reasonable goal for a software designer is to maximize performance, given that correctness is achieved together with an acceptable level of robustness.[1] It is easy to see that the concepts previously defined are not really meaningful for an agent:

- In general, there is no clear yes/no notion of correctness of behavior. Take, for example, an exploration behavior. It would be awkward to give a Boolean definition of exploration; in practical cases, one is interested in the amount of territory explored in a given amount of time—a concept akin more to performance than to correctness.

- The robustness of behavior has not so much to do with input data lying outside a predefined range as it does with the variability of the environment: intuitively, an agent is robust if it is able to carry out its task even in unexpected conditions.
- Behavioral performance has to do not only with required resources but also with the degree of accomplishment of the agent's task (as remarked above).

Other interesting indices of behavioral quality are *flexibility* (defined as the richness of the agent's behavioral repertoire) and *adaptiveness* (defined as the ability to adapt in real time to changes in the structure or in the dynamics of the environment).[2]

Thus far, our approach has been the following. First, we define performance in such a way that a high level of performance implies the execution of the target behavior. Second, we define an RP that is monotonically related to performance (in such a way that higher rewards imply higher performance). Then we use reinforcement learning to develop a controller that maximizes reward, and thus performance.

It is interesting to note that even if we have not explicitly pursued components of quality different from performance, our agents show a good level of robustness and adaptiveness, thanks to the properties of LCSs and of reinforcement learning (see section 5.2.4). Indeed, the performance of LCSs tends to degrade fairly gracefully, and learning is a powerful way of coping with structural changes in the agent-environment interaction. In principle, however, there is no reason why the direct concern about quality should be confined to performance. It is feasible to include in the RP an evaluation not only of performance but also of other quality components, although more work is needed to see whether this approach is practically viable.

To evaluate performance, we have used two types of performance indexes:

1. *Local performance indexes* (or *learning indexes*) measure the effectiveness of the learning process (that is, the correspondence between what is taught and what is learned). In general, local performance is defined as

$$P(t) = \frac{R(t)}{t},$$

where t is the number of robot moves from the beginning of the experiment and $R(t)$ is the number of moves that have been positively reinforced from

the beginning of the experiment. This is the same index, first defined in section 4.1.1, that we used to evaluate the results of simulation experiments presented in the previous chapters. Note that local performance indexes allow one to evaluate the effectiveness of all kinds of behavior, either basic or coordination ones.

2. *Global performance indexes* measure the correspondence of the robot's behaviors with the target behavior (as defined by the corresponding requirements). Typically, global performance at t is computed as

$$G(t) = \frac{t}{A(t)},$$

where $A(t)$ is the number of achievements from the beginning of the experiment (i.e., the number of times the agent has reached the goal). This measure is used when $A(t)$ is not fixed in advance, as in the AutonoMouse V experiments reported in section 7.3. When the number of achievements is predefined, as in the HAMSTER experiments in section 7.4, G is computed as the total number of moves necessary to obtain them. Note that, contrary to the previous index $P(t)$, a low value of $G(t)$ denotes a high performance, and vice versa.

Given that we use immediate reinforcements to train the robot to perform its global task, local indexes are suitable for the assessment of the learning process, and global indexes are suitable for the assessment of the robot's global behavior. Note that a robot might reach a high local performance without even starting to perform the required task. Consider a robot whose target behavior is to reach a moving light. Typically, we would train the robot to move in the direction of the light, with the underlying belief that the robot will eventually reach its goal. However, if the light runs too fast, the robot will never reach it (low global performance), even if it moves in the right direction (high local performance). Actually, the two indexes are related to different facts: local performance has only to do with *learning what one is taught to do*, while global performance has also to do with *being taught the right thing*. It is part of the specification of the training strategy to make sure that, given the particular environmental conditions the robot is going to face, a high local performance will imply a high global performance. In any case, it is up to the assessment stage to prove that this is the case.

To conclude this section, figure 7.2 summarizes all the stages previously described. The two large gray arrows are there to remind us that assess-

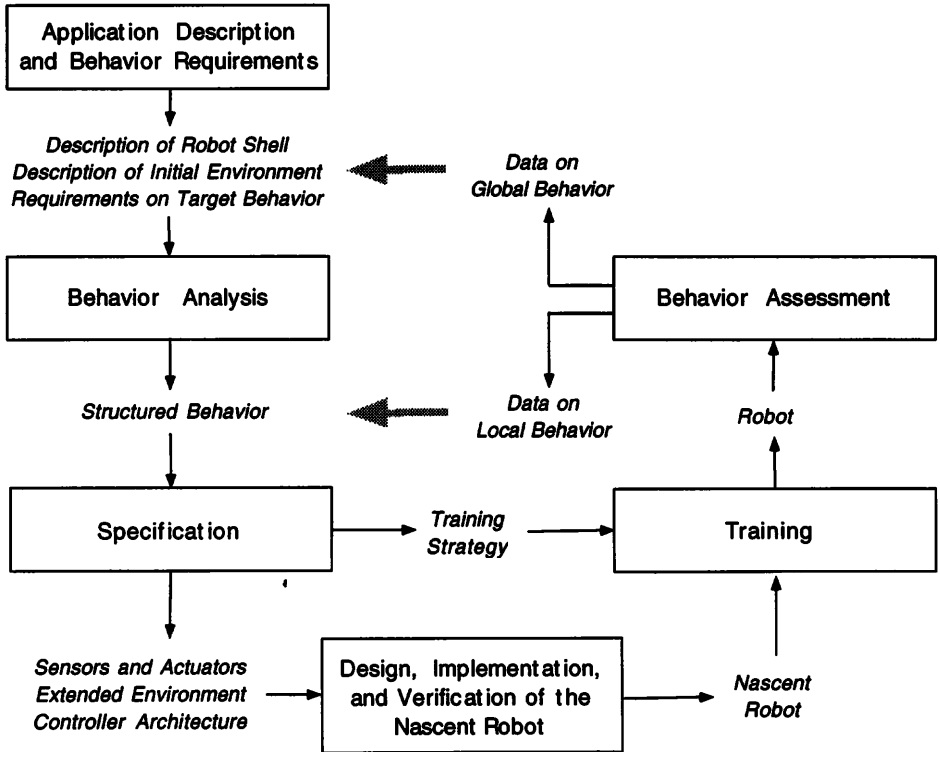


Figure 7.2
Global scheme of BAT methodology

ment essentially involves a comparison between experimental data and a specification. As we have already suggested, data on global behavior will be compared with the requirements on the target behavior, and local performance data with the specification of the structured behavior.

7.3 CASE 1: AUTONOMOUSE V

This section instantiates the BAT methodology using a simple robot, AutonoMouse V, as a running example. The application chosen is very similar to the `FindHidden` task briefly discussed in section 5.3. Here we give a much more detailed description of the whole experiment, which was run using AutonoMouse V, a version of AutonoMouse IV enriched with a rotation sensor.

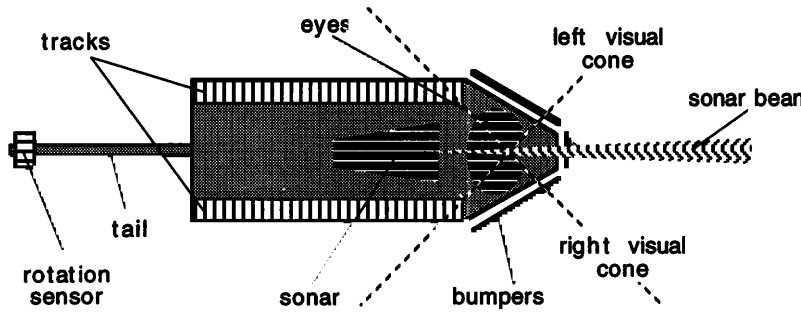


Figure 7.3
Functional schematic of AutonoMouse V

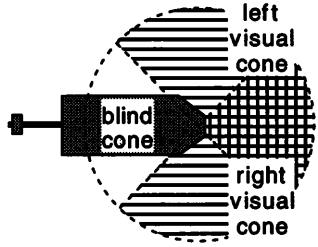


Figure 7.4
AutonoMouse V light sensors

7.3.1 Application Description and Behavior Requirements

AutonoMouse V is a small mobile robot, 35 cm long (plus 26 cm for the tail), 15 cm wide, and 28 cm high. Its sensory apparatus is two light sensors, one sonar, three bumpers, and a rotation sensor able to detect and measure a change of direction (see figure 7.3). It is also provided with two motors that control two tracks. The robot carries a battery that provides it with three hours of autonomy.

The light sensors are two photodiodes, positioned within a structure that makes them partially directional devices. The two eyes together cover a 270-degree zone, with an overlapping of 90 degrees in front of the robot (see figure 7.4). They can distinguish 256 levels of light intensity.

The sonar is highly directional (it detects obstacles in front of the robot) and can sense an object as far away as 10 meters. It outputs an eight-bit number between 0 and 256, coding its estimate of the distance to the obstacle.

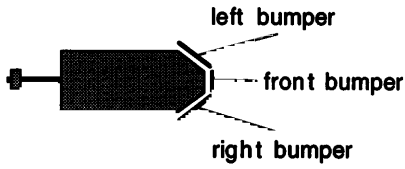


Figure 7.5
AutonoMouse V bumpers

The three bumpers, placed on the front of AutonoMouse V (see figure 7.5), are devices that change state when the robot bumps into an obstacle. The directional sensor is a rod, which we call a "tail," with a wheel at its end that can rotate around the rod axis. It therefore rotates when the robot turns, but not when the robot moves forward without turning. Its resolution is about 1.2 degrees.

Each motor has nineteen activation speeds: nine forward, nine backward, and one for not moving.

AutonoMouse V has some onboard computing capabilities to transform sensory input into digital messages in the format used by the learning algorithms. The learning algorithms run on a transputer board in a host computer connected to the robot via a 4,800-baud infrared link.

AutonoMouse V moves in a officelike environment lightened by artificial lights. The terrain is smooth, people do not interfere with the robot's movements, and there are no obstacles other than walls and the obstacle specifically used in experiments.

In this simple application, we want our robot to learn to search and follow a light moving at a speed comparable to the speed of AutonoMouse V. Now and then, the light disappears behind an obstacle; on these occasions, AutonoMouse V has to go around the obstacle to see whether the light is on the other side and then start to follow it again (see figure 7.6). We call the target behavior the "`FindHidden` behavior."

7.3.2 Behavior Analysis

The target behavior of our robot, described informally in the previous section, can be expressed as the following structured behavior:

$$\texttt{FindHidden} = \frac{\texttt{Chase}}{\texttt{Search}}.$$

That is, `Chase` and `Search` are the two basic behaviors constituting the structured behavior, and the interaction among them is suppression.

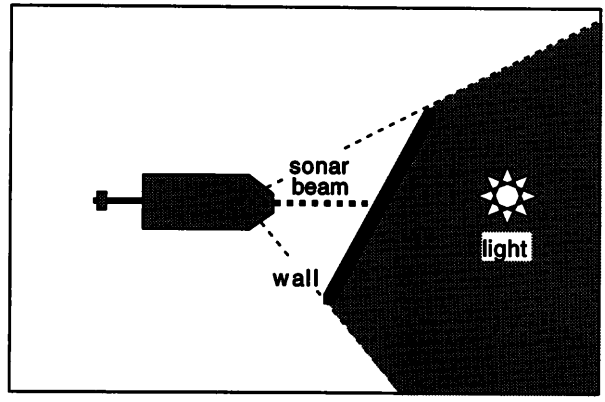


Figure 7.6
Experimental environment for `FindHidden` behavior

`Chase` is active when the light is seen. `Search` is a rather complex behavior that includes turning around to look for the light or the wall, approaching the wall when it is sensed by the sonar, and trailing around the wall when it is sensed by the bumpers.

7.3.3 Specification

The sensors defined in section 7.3.1 are too fine-grained for our application. When using a learning algorithm, it should be remembered that the greater the amount of sensory information, the greater the dimension of the search space. It is therefore advisable to choose the coarsest sensorial granularity that still allows the learning algorithm to distinguish among different environmental states from the task point of view. This was experimentally demonstrated in section 5.3.4.

We chose to put a threshold on both light sensors and the sonar sensor, transforming them into on/off devices. The output of the light sensors is therefore either "I see a light" or "I do not see a light," while the output of the sonar is either "I sense an object" or "I do not sense an object" (the threshold for the sonar was chosen such that objects are sensed if they are closer than 1.5 meters away).

The direction sensor, that is, the tail, was thresholded so that AutonoMouse V knows that it has changed its direction whenever it turns by an angle greater than 1.2 degrees. In order not to burden the learning system with useless extra search space given by too detailed control on movements, motor moves have been thresholded so that tracks are moved by

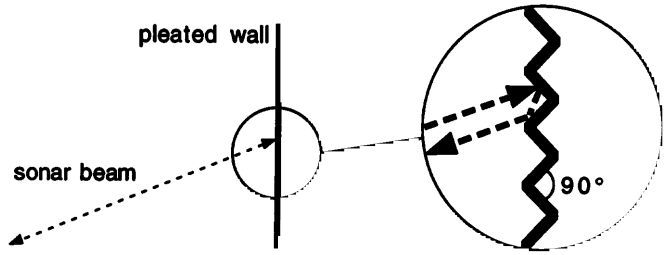


Figure 7.7
Pleated wall

sending two-bit messages that correspond to the following possible movements of each track: "Move backward at speed one", "Stay still", "Move forward at speed one", and "Move forward at speed two." Speed two corresponds to the maximum velocity of the robot, and speed one, to one half of maximum velocity.

The initial environment was extended by a 50 W lithgbulb moved around by one of the experimenters, and by a pleated wall obstacle (see figure 7.7). A pleated wall was chosen because our single and highly directional sonar lacked the capacity to perceive smooth surfaces under all the possible angles of incidence of the sonar signal.

Given the kind of structured behavior described in section 7.3.2, the switch architecture is the most appropriate and was used for all AutonoMouse V experiments (this architecture was compared with other possible solutions both in real-world experiments and in simulation; see Dorigo and Maesani 1993). In this architecture, the light approaching and the light searching behaviors (basic behaviors) are coordinated by a switch module that learns to choose which basic behavior to give priority to. As regards the training strategy, there are two decisions to be made: the reinforcement program and the shaping policy.

7.3.3.1 Reinforcement Program

The reinforcement program is in charge of giving reinforcements after each move of AutonoMouse V. It is usually composed of a different subprogram for each basic behavior. For AutonoMouse V, we defined the following reinforcement programs for each of the two behaviors identified in section 7.3.2:

1. *Light-approaching reinforcement program.* After each move, the robot is rewarded if its distance from the light decreases; otherwise, it is

punished. Changes in distance are evaluated by measuring the change in intensity of light as seen by the two eyes. AutonoMouse V eyes have therefore a double use: as agent sensors (in which case, they are thresholded), and as trainer sensors (in which case, their full granularity is used to evaluate changes in light intensity).

2. *Light-searching reinforcement program.* Light searching is not as easy to define as light approaching in that it allows the definition of a few different searching strategies. The main goal of this behavior is to search behind obstacles to see whether the light is there. For example, the agent could first use sonars to locate the edge of the obstacle, and then move directly in that direction. Another possible strategy could be to use the sonar to locate an obstacle, then approach it, and finally turn around it using bumpers to maintain contact with the obstacle (this could be done with a reinforcement program that rewards a right turn when the left bumper is on, and a left turn when it is off; in this way AutonoMouse V will learn to move along the obstacle with a zigzag kind of motion). These and a few other strategies were investigated by Dorigo and Maesani (1993). In the experiments presented here, we used the first strategy, which works as follows: if the sonar senses an obstacle, then the robot is rewarded if it moves forward and at the same time it turns in the same direction of the previous move. As soon as the sonar no longer senses the obstacle, the robot is rewarded if, while still moving forward, it turns in the opposite direction ("Move forward while turning" moves are obtained by setting one motor to speed two and the other to speed one). This reinforcement policy generates an obstacle-approaching trajectory like the one presented in figure 7.8.

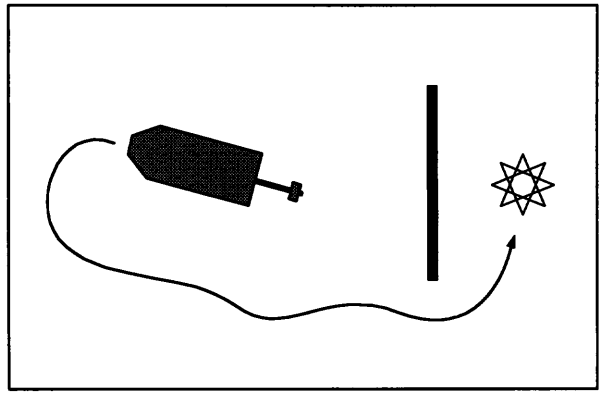


Figure 7.8
Light-searching behavior

7.3.3.2 Shaping Policy

The shaping policy used was modular. First we trained the two basic behaviors, then froze them (that is, their learning algorithms were switched off) and let the coordination behavior learn.

7.3.4 Design, Implementation, and Verification

This step exploits standard hardware and software engineering methodologies. All the design and implementation details regarding this stage of the BAT methodology are documented in Dorigo and Maesani 1993.

7.3.5 Training

AutonoMouse V was trained using the training strategy defined in section 7.3.3. The results obtained using the switch architecture with modular shaping were compared with those obtained training the two basic behaviors in a simulated environment and then copying the two rule sets into the two basic behavioral modules used to control the real robot (we call this the "switch architecture with initial knowledge"). After transferring the two rule sets, training continues using the same training and shaping strategies as in the previous experiment.

7.3.6 Behavior Assessment

This stage of the BAT methodology is devoted to assessing the degree of learning achieved by the learning agent. Two tools are useful for this assessment: the local and the global performance indexes. Figures 7.9 and 7.10 report the behavior of $P(t)$ and $G(t)$, as defined in section 7.2.6, for the switch architecture (SA) and the switch architecture with initial knowledge (SAIK), respectively. Graphs are averaged over five runs.

Both indexes show that, given the same amount of learning cycles, starting with some initial knowledge developed in simulation helps. None of the architectures achieves very high local performance. This is mainly due to limited learning time; we stopped the experiment after 6,000 AutonoMouse V moves because of time constraints (6,000 moves take about one hour of time). A longer experiment showed that after 12,000 AutonoMouse V moves, the system was still learning.

In this experiment, the assessment of global behavior is somewhat arbitrary, given the highly research-oriented nature of the task. $G(t)$ decreases as $P(t)$ increases, an indication that the training strategy is somewhat successful in driving the learning system toward better global performance. Whether the given training strategy is good can be evaluated

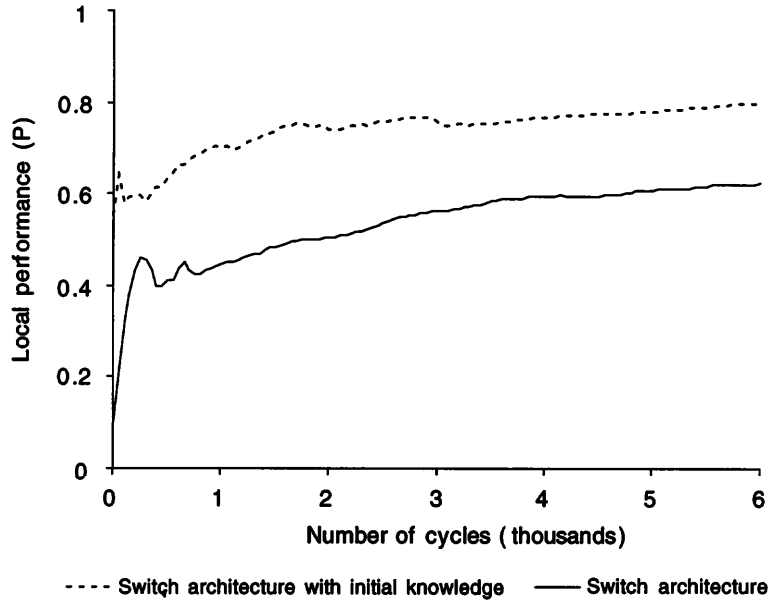


Figure 7.9
Local performance index: Comparison between SA and the SAIK architectures (with local index, higher value reflects a better performance). Average over 5 runs.

only by comparing it with other training procedures such as the one proposed in section 7.3.3. In a more application-oriented task, $G(t)$ would be compared against quantitative requirements on the target behavior.

7.4 CASE 2: HAMSTER

A second example of the application of the BAT methodology is HAMSTER, a mobile robot based on a commercial platform, whose task is to bring "food" to its "nest."[3] We shall confine ourselves to describing the main differences with respect to AutonoMouse V. The chief features of HAMSTER are that it combines "innate" (i.e., prewired) and learned behaviors, and that its training was carried out in a simulated environment and then transferred to the physical robot.

7.4.1 HAMSTER's Shell

HAMSTER's shell is Robuter, a commercial platform produced by Robo-Soft (see figure 1.5 for a picture). It is 102 cm long, 68 cm wide, and 44 cm

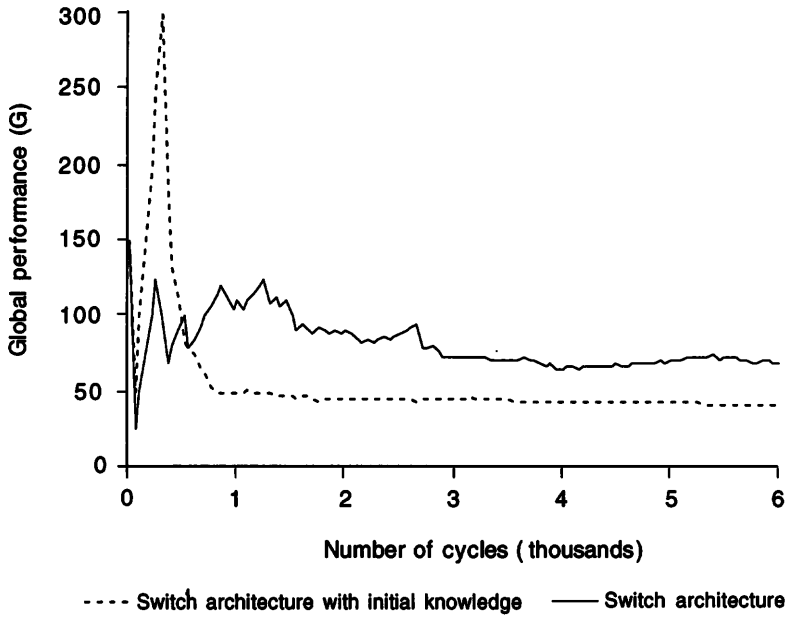


Figure 7.10
Global performance index: Comparison between SA and SAIK architectures (with global index, lower value reflects better performance). Average over 5 runs.

high. The configuration we used has a belt of twenty-four Polaroid sonars, surrounding the whole platform. Motion is produced by two motors acting on two independent wheels.

7.4.2 The Hoarding Behavior

The target behavior is food hoarding, that is, collecting food pieces and storing them in HAMSTER's nest. The environment is a room of size 14 m × 13.3 m, with various obstacles (see figure 7.11). Each piece of food is a cylinder (diameter 30 cm, height 70 cm) that slides on the floor when pushed by HAMSTER. The nest is located in a corner of the room. The target behavior can be decomposed as follows:

```
HoardFood = (LeaveNest · GetFood · ReachNest)*
            + AvoidObstacles.
```

7.4.3 Specification

Two rigid metal bars have been adapted to the Robuter's front so that food cylinders do not slip to either side when they are pushed. A frontal

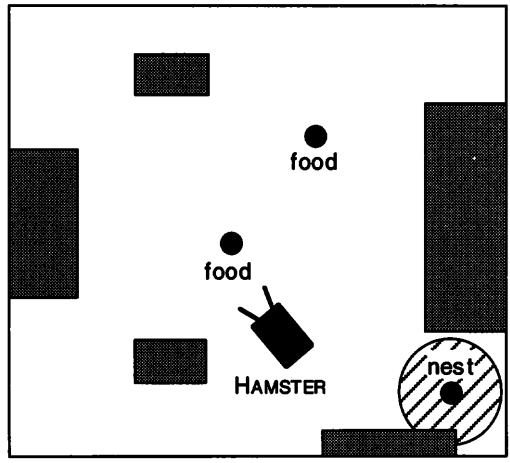


Figure 7.11
HAMSTER's environment

proximity sensor, based on the frontal sonars, allows HAMSTER to sense whether an object in front is far away, fairly close, very close, or at contact. The food cylinders are wrapped in violet paper, and the nest's position is marked by another cylinder (diameter 30 cm, height 130 cm) wrapped in pink paper. HAMSTER uses a frontal color camera to identify the position of food cylinders and of the nest, which are distinguished on the basis of color. Moreover, the nest sensor exploits an odometer (that is, a sensor that estimates the robot's position and heading) to identify the approximate position of the nest when this is not visible.

A main concern of the HAMSTER project was to combine learned and innate behavior modules. We chose to program `AvoidObstacles` directly, implementing a potential-based avoidance mechanism that exploits Robuter's sonars (see, for example, Latombe 1991).

The remaining part of the hoarding behavior, that is, the food-fetching cycle, can be analyzed as a *pseudo-sequence*, see section 6.2. This means that at each control cycle there is enough information coming from the sensors to decide which of the three subbehaviors should be active:

1. `LeaveNest`: When the robot is in the nest: leave previously captured food in the nest, and then leave the nest.
2. `GetFood`: When the robot is out of the nest and no food is captured: get a piece of food.

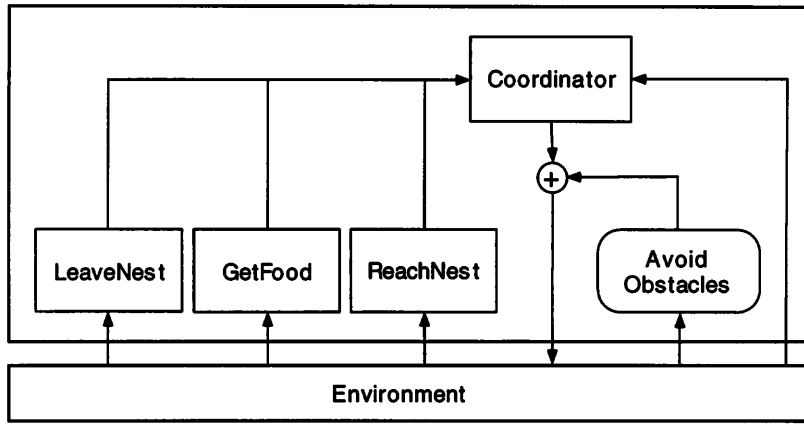


Figure 7.12
Controller architecture for HAMSTER

3. `ReachNest`: When the robot is out of the nest and a piece of food is captured: reach the nest, pushing the piece of food.

We adopted a hierarchical controller architecture, with four behavioral modules at level 1, and a coordinator module at level 2. Each level-1 module proposes a direction for the robot's movement. The coordinator chooses one of the moves proposed by `LeaveNest`, `GetFood`, and `ReachNest` on the basis of the current situation, which is then combined with the move proposed by `AvoidObstacles` (see figure 7.12). In general, this combination amounts to a weighted vector sum of the moves proposed by `AvoidObstacles` and by the rest of the system. There is, however, a more complex case. When HAMSTER is nearing a piece of food, the front part of the `AvoidObstacles` behavior has to be inhibited; otherwise, the food could never be captured. The `GetFood` and `ReachNest` behaviors learn to inhibit the front sonars as needed.

7.4.4 Training

HAMSTER was trained through a modular shaping policy; that is, each learned level-1 module was trained separately and frozen. The coordinator was then trained to achieve the target behavior.

We decided to use very simple reinforcement programs (RPs). For example, the `ReachNest` behavior was rewarded if and only if the distance from the nest decreased. Training the robot while obstacle avoidance was active would have forced us to embed a model of obstacle

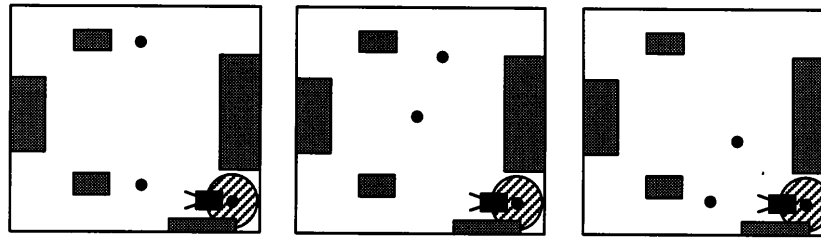


Figure 7.13
Three experimental situations. Rectangular shaded areas are obstacles; black circles are food pieces. HAMSTER is shown at its initial location, that is, in nest (shaded circle).

avoidance into the RP, which would be unable to evaluate HAMSTER's moves properly without knowing the effects of obstacles on the robot. But such an RP would have been very complex. To keep the RP simple yet eliminate the interference of the innate `AvoidObstacles` behavior, training therefore had to be carried out in an obstacle-free environment.

Because we had no such physical environment available, all training was carried out in a simulated environment, and the resulting modules were then transferred to the real robot. The transfer of the learned controller from a simulated environment turned out to be an interesting option in its own respect.

7.4.5 Assessment

The degree of learning was evaluated through the local performance indexes of the basic behaviors, computed in the simulated environment with no obstacles. As expected, we obtained reasonably high values (from 0.75 to 0.85, depending on the specific behavior and on different RPs we have experimented with). We did not compute the local performance index in the environment with obstacles because it would have been meaningless; for example, during the `ReachNest` behavior the robot would have been punished if it moved away from the nest in order to avoid an obstacle.

We then transferred the controller to the real robot and ran some experiments in the real environment (with obstacles). In this environment, it is meaningful to evaluate the global performance, computed as the number of moves necessary to accomplish the task, averaged over a set of sample situations. More precisely, we chose three different initial situations (see figure 7.13), all including two pieces of food, and ran ten trials

Table 7.2
Global performance index of HAMSTER in real environment

	Situation a	Situation b	Situation c
Mean	365.30	450.00	313.50
Std. dev.	27.77	127.30	34.33

for each of them, recording the total number of cycles necessary to complete the hoarding of both pieces of food.

Finally, we computed the mean and the standard deviation of such data (table 7.2). In situation *c*, HAMSTER was unable to accomplish the task five times out of ten due to the difficulty of getting the piece of food in front of the initial robot position and then avoiding the close obstacle. For this situation, the data reported relate to the five successful trials only.

7.4.6 Conclusions

The HAMSTER project shows that while it is feasible to implement a robot's controller starting from both innate and learned behavioral modules, to keep the RPs as simple as possible, it is necessary to carry out training in environments where the innate behaviors are not active. In our case, this required that we carry out training in a simulated environment, and then transfer the learned controller to the physical robot for the final assessment.

Whether the global performance of HAMSTER can be considered acceptable is difficult to say. In a real application, some minimum performance level would have been established in advance. In our case, we can only say that an informal observation of the behavior gave us the impression that HAMSTER was performing reasonably well.

7.5 CASE 3: THE CRAB ROBOTIC ARM

For our final example of the BAT methodology, we used an industrial manipulator, namely an IBM 7547 with a SCARA geometry (see figure 1.6 for a picture). In this section, we shall refer to this robot as "CRAB" (classifier-based robotic arm), a name that indeed fits the shape of its arm.

CRAB is a two-link robotic arm. The first link can rotate 200 degrees around the shoulder joint, and the second link can rotate 160 degrees, with respect to the first link, around the elbow joint. As a result, the end effector, attached to the wrist, can cover the gray and the dashed areas shown in figure 7.14. The effectors are two motors, acting on the shoulder

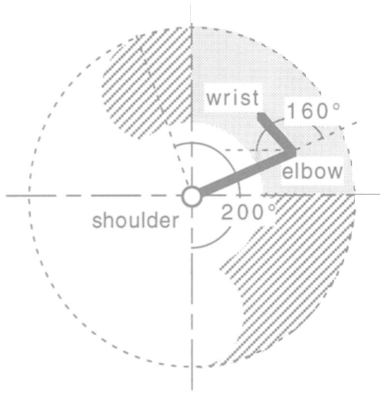


Figure 7.14
CRAB manipulator: Schematic drawing of action space

and the elbow joints, that rotate the first link (with respect to the fixed base) and the second link (with respect to the first link).

The target behavior we taught to CRAB is a very simple one, namely, to have the end effector reach a still object ("food") placed in the gray area of figure 7.14 (the "foraging area"). In this case, the target behavior itself is a simple behavior, and therefore behavior analysis is trivial.

The reason we were interested in such a simple behavior is that, given the polar geometry of CRAB, ALECSYS has to learn behavior rules that are very different from the ones controlling the mobile robots described in the previous sections. In particular, the relation between the actions (i.e., elementary rotations of the two links) and the resulting movement of the end effector involves complex trigonometric transformations. Therefore, it is not at all obvious that the results obtained with mobile robots can be made to bear on the present case.

In CRAB's environment, there is exactly one piece of food in the foraging area at any moment. Each time CRAB reaches it, the piece of food is moved to a new random position within the foraging area. To be sensed by the robot, the piece of food emits infrared light that can be received by an appropriate sensor. Deciding where to place such a sensor presents a simple, but interesting specification problem.

A natural agent using arms to grasp objects would be endowed with two types of sensory capacities: *proprioception*, the capacity to know the relative position of the limbs; and *vision*, the capacity to identify the relative direction and distance of the object to be grasped. The problem with

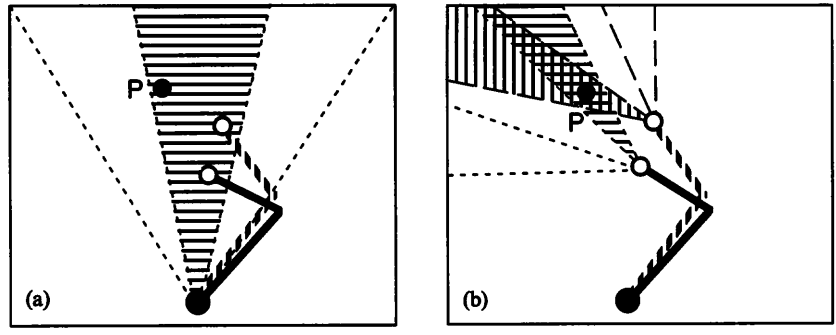


Figure 7.15
Why sector-based sensor must be placed on end effector: (a) position of object *P* cannot be distinguished from position of end effector; (b) object *P* can be approached with any degree of precision.

our artificial agents is that ALECSYS can only deal with a very limited amount of input information, largely insufficient to determine relative distances and angles of objects with the required degree of precision. Therefore, for both proprioception and infrared sensing we adopted sector-based sensors, as we did with our mobile robots.

However, because it cannot discriminate positions on the basis of distance, a sector-based sensor placed on the shoulder is structurally unable to locate an object precisely, (see figure 7.15a). To guide the end effector exactly to a given point, the sector-based sensor must be placed on the end effector (figure 7.15b). Even though the agent will still not be able to discriminate on the basis of distance, the end effector can be guided to the goal position simply by moving in its direction.

As regards proprioception, simulation experiments showed that it is enough to have information about the elbow angle, that is, the angle between the first and the second link (Dorigo, Patel, and Colombetti 1994; Patel and Dorigo 1994; Patel, Colombetti, and Dorigo 1995). The sensorimotor apparatus of CRAB was then specified as follows:

- A proprioceptive sensor provides information on whether the elbow angle is between 0 and 80 degrees, or between 81 and 160 degrees.
- An infrared sensor, placed on the end effector, tells the robot in which of eight equal sectors the food is located (all sectors are of 45 degrees).
- Two motor effectors, one each for the elbow and shoulder joints, rotate right (4 degrees for both joints), rotate left (4 degrees for both joints), or stay still.

For our controller, we chose a monolithic architecture, given that the target behavior was made up of a single, simple behavior.

The infrared sensor we implemented allowed us to locate the food piece in the correct sector with good precision, but gave us a rather noisy estimate of the distance between the food piece and the end effector. Note however, that information about this distance is not used by the learning agent but only by the RP to compute reinforcements. Each move was reinforced in proportion to the variation of the distance between the end effector and the food piece, as estimated by the infrared sensor.

To speed up learning in a typical experiment, we started from an initial rule base obtained from a training session of 25,000 cycles run in a simulated environment and ran a supplementary training session of 2,500 cycles with the real robot. At the end of this training session, we obtained a local performance index of 0.86, showing a limited increase in performance from the initial rule base obtained through simulated pretraining (a test run on the real robot with such an initial rule base gave us a performance of 0.83). The result suggests that the supplementary training session on the real robot adapted the rule base learned in the simulated environment to the more noisy physical sensor.

7.6 POINTS TO REMEMBER

- Behavior analysis and training (BAT) is proposed as a first example of a behavior engineering methodology based on the analysis of behavior, the integration of machine learning techniques with other aspects of robot design, and the independent assessment of learning and of global behavior.
- At least in some cases it is possible to use behavioral modules learned in simulated environments as a starting point for real robot training.
- Learned behavioral modules can be integrated with hand-coded modules.
- The results obtained from three practical cases show that the BAT methodology is sound, can be applied to robots of different types (like mobile platforms and robotic arms), and can lead to realizations of practical interest.

Chapter 8
Final Thoughts

8.1 A RETROSPECTIVE OVERVIEW

In this book we have presented a new approach to the development of learning autonomous robots through behavior analysis, design, and training. We view our work as a first step toward behavior engineering, an integrated discipline for the development of robotic agents. In particular, we have advocated a methodology centered on robot shaping, that is, the instruction of agents based on reinforcement learning that exploits trainer-produced reinforcements.

Our robots are strongly coupled with their environment through their sensorimotor apparatus, and are endowed with an "innate" (i.e., predesigned) controller architecture that has the ability to learn from experience. The learning activity of our agents can be viewed as a grounded (i.e., environment-sensitive) translation of an external specification of behavior coded into a reinforcement-producing program, the RP.

Although our experimental activity exploits learning classifier systems as a learning paradigm, we believe our results can be extended to any learning approach used to solve reinforcement learning problems. Indeed, the idea of using a trainer to provide step-by-step reinforcement is more basic than the specific learning model adopted.

For our experiments, we used both simulated and real agents. Simulated environments have proved very useful to test design options, and the results of simulations turned out to be robust enough to carry over to the real world without major problems. The experimental activity carried out so far allows us to make several claims:

• Animat-like interactions in simple environments can be practically developed through step-by-step reinforcement learning. In general, we

consistently obtained reasonably high learning rates across multiple experimental trials.

- The kind of Animat-like behaviors we developed in simulation can be developed also with real robots, provided the sensory and motor capacities of the robots are similar to those of their simulated counterparts (see the experiments in chapters 5 and 7). When the use of a real robot is too time-consuming, training in a simulated environment and then transferring the resulting controller to a real robot can be a viable alternative.
- Fairly complex interactions can be developed even with simple, stimulus-response (S-R) agents. In particular, behavior patterns that appear to follow a sequential plan can be realized by an S-R agent when there is enough information in the environment to determine the right sequencing of actions (see, for example, the `Chase/Feed/Escape` experiment in section 4.4). However, the addition of non-S-R elements, like a memory of past perceptions, can improve the level of adaptation to the dynamics of the environment.
- The use of internal state (i.e., a state of the *controller*) allows our agents to learn simple sequential behavior patterns, a subtype of the larger class of dynamic (i.e., non-S-R) behaviors. Experiments are reported in chapter 6.
- To develop an agent, both explicit design and machine learning have an important role. In our approach, the main design choices involve (1) sensors, effectors, and controller architecture of the agent; (2) the artificial objects in the environment; (3) the sensors and the logic of the trainer; and (4) the shaping policy. Learning is in charge of developing the functions implemented by the various modules of the agent controller. A comprehensive methodology is presented in chapter 7.

8.2 RELATED WORK

Our work situates itself in the recent line of research that concentrates on the realization of artificial agents strongly coupled with the physical world and usually called "embedded" or "situated agents." Paradigmatic examples of this trend include Agre and Chapman 1987, Kaelbling 1987, Brooks 1990, 1991, Kaelbling and Rosenschein 1991, and Whitehead and Ballard 1991.

While there are important differences among the various approaches, some common points seem to be well established. A first, fundamental requirement is that agents must be able to carry on their activity in the real world and in real time. Another important point is that adaptive be-

havior cannot be considered as a product of an agent in isolation from the world, but can only emerge from a strong coupling of the agent and its environment.

Our research has concentrated on a particular way to obtain such a coupling: the use of the LCS to dynamically develop a robot controller through interaction of the robot with the world. In this section we relate our work to other approaches to the same problem developed in adjacent research fields.

Most prominent is the work on learning classifier systems (Holland and Reitman 1978; Wilson 1987; Robertson and Riolo 1988; Booker 1988; Booker, Goldberg, and Holland 1989; Dorigo 1995). As seen in chapter 2, building on that work, we introduced the idea of using a set of communicating classifier systems, running in parallel on an MIMD architecture. We also modified the basic learning algorithms to make them more efficient.

More generally, our work is related to the whole field of reinforcement learning. Reinforcement learning has recently been studied in many different algorithmic frameworks. Besides LCSs, the most common frameworks are temporal difference reinforcement learning and related algorithms, like the adaptive critic heuristics (Sutton 1984) and Q-learning (Watkins 1989; Watkins and Dayan 1992), often implemented by connectionist systems (e.g., Barto, Sutton, and Anderson 1983; Williams 1992), and evolutionary reinforcement learning (e.g., Moriarty and Mikkulainen 1996). These different approaches to reinforcement learning are often overlapping. For example, the adaptive heuristics critic and Q-learning have been implemented through a connectionist system by Lin (1992); Compiani et al. (1989) have shown the existence of tight structural relations between classifier systems and neural networks; Dorigo and Bersini (1994) have shown the strong connections between LCSs and Q-learning; and Kitano (1990), Whitley, Starkweather, and Bogart (1990), Belew, McInerney, and Schraudolph (1991), Nolfi and Parisi (1992), Schaffer, Whitley, and Eshelman (1992), and Maniezzo (1994) have shown that neural nets can be learned by means of an evolutionary algorithm.

It happens very often that the reinforcement learning problems used to illustrate and compare proposed algorithms are taken from the realm of autonomous robotics, but only a few of these applications deal with real robots. For example, Singh (1992), Lin (1992), and Millán and Torras (1992) use a point robot moving in a two-dimensional simulated world. Grefenstette's SAMUEL, a learning system that uses genetic algorithms

(Grefenstette, Ramsey, and Schultz 1990), learns decision rules for a simulated task (a plane learns to avoid being hit by a missile). Booker's GOFER system (Booker 1988) deals with a simulated robot living in a two-dimensional environment, whose goal is to learn to find food and avoid poison.

The choice of using a real robot makes things very different because many aspects of the agent-environment interaction become truly unpredictable and the efficiency of the system becomes a major constraint. In the following sections, we will review applications of the above-mentioned classes of techniques to the learning of autonomous robot controllers. We will limit our attention to systems that, like ours, run on real robots.

8.2.1 Classifier System Reinforcement Learning

Donnart and Meyer (1996) have developed MonaLysa, an interesting learning architecture based on learning classifier systems, which they demonstrated in a real robot application. Although also based on a hierarchical control architecture, MonaLysa differs from ALECSYS in many respects. Its control system is composed of a greater variety of modules (reactive, planning, context manager, internal reinforcement, and auto-analysis). Whereas in ALECSYS tasks are switched on or off by a learned behavioral module acting as a coordinator, in MonaLysa tasks can be added or suppressed according to a criterion of internal satisfaction of the control system. MonaLysa has been demonstrated on a goal-finding-plus-obstacle-avoiding task; it is able to generate and store plans that help to avoid obstacles while reaching the goal, and to alter these plans in reaction to environment modifications. Also, MonaLysa uses a version of LCS in which the classifiers completely cover the possible input-output mappings: no genetic algorithm is used to search for useful rules.

A somewhat different approach is taken by Bonarini (1996), who uses ELF (a fuzzy version of LCS) to train a mobile robot to chase another mobile robot. In this application, ELF exploits a real-time simulation of the interaction to implement a version of "anytime learning" (Grefenstette and Ramsey 1992).

8.2.2 Temporal Difference Reinforcement Learning and Related Algorithms

Temporal difference methods, widely studied in the last few years, have also been applied to the control of real robots. Kaelbling used reinforcement learning to let a robot—called "Spanky"—learn to approach a light source and avoid obstacles (Kaelbling 1990; but see also Kaelbling 1991).

She used a statistical technique to store an estimate of the expected reinforcement for each action and input pair and some information of how precise that estimate was. Unfortunately, only a qualitative description of the experiments run in the real world has been reported. Kaelbling's robot took from 2 to 10 minutes to learn a good strategy, while AutonoMouse II was already pointing toward the light after at most one minute. Still, the experimental environments were not the same, and Spanky's task was slightly more complex. One of the strengths of our approach is that it allows for an incremental building of new capabilities, something it is not clear Kaelbling's approach allows.

Mahadevan and Connell (1992) have implemented a subsumption architecture (Brooks 1991) in which the behavioral modules learn by extended versions of the Q-learning algorithm. Besides using a different learning approach, their work also differs from ours in that their behavior coordination strategy is not learned but directly programmed.

Millán (1996) has proposed TESEO, a connectionist controller based on the actor-critic architecture (Barto, Sutton, and Anderson 1983). TESEO's main goal is to overcome three common limitations of systems trying to solve RL problems: slow convergence, unsafe process, and lack of incremental improvement. It achieves these results in a simple navigation problem by using a set of basic reflexes (i.e., predesigned reactive behaviors) and by learning new reactions from basic reflexes. It differs from ALECSYS in many respects, the most important being its use of prewired behaviors ("basic reflexes" in Millán's terminology) in a subsumption-like architecture. These basic reflexes allow the robot to start with an acceptable control policy, which is later refined or augmented by learning. In the HAMSTER experiment presented in section 7.4, ALECSYS was used to learn to coordinate prewired and learned basic behavioral modules.

8.2.3 Evolutionary Reinforcement Learning

In the last few years, behavior-based robotics has proved to be an excellent test bed for evolutionary computation methods. Beer and Gallagher (1992) have used genetic algorithms to let a neural net learn to coordinate the movements of their six-legged robot. Harvey, Husbands, and Cliff (1994) have evolved neural network controllers capable of producing a variety of visually guided behaviors. They also demonstrated the principle of incremental evolution, helped by selecting tasks of increasing complexity (i.e., a population developed to learn a given task is used as the

starting population to solve a slightly more difficult task). Floreano and Mondada (1996) has evolved a neural controller directly on the robot. They demonstrated the evolution of a collision-free navigation behavior and a homing behavior. All these approaches differ from ours in that they use a genetic algorithm to develop neural net controllers, instead of a set of behavioral rules.

Koza and Rice (1992) used the genetic programming paradigm (Koza 1992) to evolve Lisp programs to control an autonomous robot. Although their experiments are run in simulated environments, they are still interesting because they try to reproduce the results obtained by Mahadevan and Connell (1992); unfortunately, their use of a simulated robot makes a fair comparison very difficult. Although Koza and Rice use genetic algorithms to develop a computer program, their approach is very different from ours: their genetic algorithm searches a space of Lisp programs, while ours is cast into the classifier system framework.

Many studies of evolutionary reinforcement learning have taken the approach of developing the controller in simulation and then transferring it to the real robot. Examples are Nolfi et al. 1994, Grefenstette and Schultz 1994, and Jakobi, Husbands, and Harvey 1995. We also studied controller transfer from the simulated to the real robot (see section 7.4).

Another interesting piece of work, Meeden 1996, compares connectionist reinforcement learning (based on backpropagation) and evolutionary reinforcement learning (based on genetic algorithms), as a way to develop a neural network–based controller for a small toy car called "Carbot." (The interested reader can find a nice review of recent work on evolutionary reinforcement learning in Matarić and Cliff 1996.)

8.2.4 Work on Shaping and Teaching in Reinforcement Learning

The idea of shaping a robot is related to the work by Shepanski and Macy (1987), who propose to train a neural net manually by interaction with a human expert. In their work the human expert replaces our reinforcement program. Their approach, although very interesting, seems difficult to use in a nonsimulated environment; it is not clear therefore whether it can be adopted for real robot shaping.

Clouse and Utgoff (1992) propose a method in which a teacher observes a reinforcement learner performance and provides advice whenever the learner is judged not to progress well. Utgoff and Clouse (1991) propose a similar method, in which the teacher waits for the learner's request of help. Both approaches differ from ours in that the teacher says explicitly

which action should have been chosen, while our trainer only provides an immediate evaluation of the action chosen by the agent.

Singh (1992) applies a shaping method to help a simulated robot learn composite tasks. In his approach, shaping means letting a robot learn a succession of tasks, where each succeeding task requires a subset of the previously learned tasks plus a new elemental task. He ran experiments using compositional Q-learning, an extension of Q-learning that allows the construction of Q-values of a composite task from the Q-values of the elemental tasks in its decomposition. Singh's shaping differs from ours in that it focuses on the incremental nature of learning (the robot learns new tasks building on its previous experience): there is no explicit trainer, and the meaning of the modules that make up his architecture is not designed but learned on-line.

In Lin's teaching method (1992), the teacher has to know how to generate a solution ("lesson" in Lin's terminology) to the problem confronting the learner. Because the design of such a teacher can be equivalent to being able to find the solution of the learning problem (or at least a good approximation of the solution), Lin's teaching approach can be considered as a way to transfer knowledge from an expert trainer to an ignorant agent. But because our trainer does not have to produce a solution, only to evaluate moves, it is easier to design that Lin's.

Thrun and Mitchell (1993) have developed a method that allows an agent learning by reinforcement to use prior knowledge in the form of neural networks capable of predicting the results of actions. Gordon and Subramanian (1994) have proposed an evolutionary reinforcement learning method based on genetic algorithms. A teacher provides advice in the form of "if [conditions], then [achieve goal]." These rules are then made operational using the learning system background knowledge about goal achievement. Maclin and Shavlik (1996) have proposed a similar method, whose main difference lies in the use of connectionist Q-learning instead of a genetic algorithm. All these works differ from ours in that their goal is to use domain knowledge and to integrate it in the knowledge representation system of the learning agent.

Caironi and Dorigo (1994) have applied the same type of trainer (reinforcement program) proposed in this book to simulated robotic problems using Q-learning as the learning algorithm. Caironi and Dorigo's work improves on previous work by Whitehead (1991a, 1991b), who also studied the problem of integrating delayed and immediate reinforcements in the Q-learning framework. Both these works differ from ours in that

experiments were conducted in very simple simulated environments, where the robot had complete knowledge of the state.

Nehmzow and McGonigle (1994) have used a trainer to provide immediate reinforcement to their robot: they provide positive/negative reinforcements by covering/uncovering the robot's upward light sensors. Their work is a first example of shaping by a human trainer.

Finally, Asada et al. (1996) have developed a control system based on Q-learning for a robot that learns to shoot a ball into a goal. To reduce learning time, they use a method they call "learning from easy missions." The idea is to let the robot learn first in easy situations, then in incrementally more difficult ones. Obviously, this requires some sort of external supervisor, capable of ranking situations by difficulty and of choosing tasks of increasing difficulty.

8.3 TRAINING VERSUS PROGRAMMING

The basic idea fostered in this book is that training an agent can be better than programming it. As with all approaches having some innovative content, one can take an optimistic or a pessimistic attitude. Let us start with some optimism.

We have shown that a feasible way to build the controller of a robotic agent is to have it learn on the basis of direct interaction between the robot, its environment, and a trainer. We have approached the learning problem by evolutionary computation and reinforcement learning with the help of a trainer.

As far as evolutionary computation is concerned, the idea is that such an approach will eventually allow robot developers to obtain higher-quality behavior than through direct coding (Matarić and Cliff 1996). The natural history of animal species shows that a process of fitness maximization can lead to highly complex and organized behavioral structures. In principle, there is no reason why an artificial process should not be able to mimic the natural development process, provided enough computational resources are available. Of course, we all know that the amount of resources exploited by natural evolution to produce complex organisms is enormous. Thus engineers are likely to need a deus ex machina to shortcut the natural process by many orders of magnitude: in our work, we have exploited the design of the controller architecture and of the sensorimotor interface.

The amount of a priori design required by our approach is still extensive. This might be reduced by increasing the use of evolutionary compu-

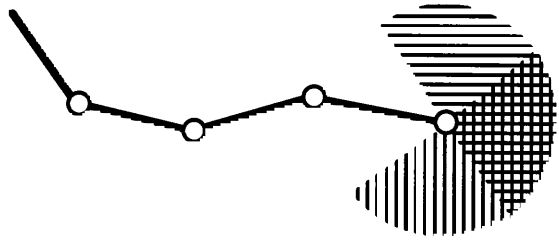


Figure 8.1
Snakelike mobile robot

tation techniques; at least in principle, the structure of the sensorimotor interface and the architecture of the controller could undergo a process of artificial evolution (see, for example, Cliff, Harvey, and Husbands 1993). However, we have not yet explored this possibility.

Regarding the reinforcement learning aspect of our approach, step-by-step reinforcement given by an RP might be regarded as a limitation because the designer needs to write the RP, which could at first sight appear as difficult as directly coding the control program. It is important to note, however, that the RP need not be a control program in disguise. To see this, consider the problem of developing a light-following behavior for a number of different robots: wheel or track-based vehicles (like the AutonoMice and HAMSTER), a robotic arm (like CRAB), and perhaps a snakelike robot like the one sketched in figure 8.1 (see, for example, Nilsson and Ojala 1995).

The interesting aspect of RP-based training is that, at least in principle, it is possible to train all such robots with the same RP, that is, one that gives reinforcements proportional to the decrease of distance between the robot's sensor and the goal. The resulting control programs will be sharply different, reflecting the different types of agent-environment interactions, which in turn depend on the different geometries of the agents. In this example, the effort required to invent the RP is virtually zero: that the distance between the robot's sensor and the goal should decrease is simply a formal statement of the *specification* of the light-following behavior. On the other hand, nobody has to worry about the more or less complex kinematics of the different robots: remarkably, this job is taken up by the learning process.

A problem with the process sketched above, and in general with our training approach, is that it relies on much more information than

processes based on delayed reinforcements, in which a reinforcement is given only when a goal is reached. Moreover, proponents of delayed reinforcement learning (RL) claim, and we agree with them, that most autonomous agent learning problems are naturally modeled as delayed RL problems.

Unfortunately, the current state of understanding makes it unfeasible to use delayed RL to develop the control system of a real robot in a reasonable amount of time. We have therefore adopted an approach that puts the burden of delayed reinforcement on the shoulders of a trainer program, which embodies human knowledge of the task, understood as a pattern of agent-environment interaction. With respect to delayed reinforcement methods, there is a difference in the type and amount of input information that has to be considered in order to evaluate the agent's behavior: in general, a step-by-step RP requires more input information than a delayed RP. However, the goal of behavior engineering is not to save on input information but rather to exploit what input information is actually available, or can be made available at a reasonably low cost. A possible extrå cost of step-by-step reinforcement might well be justified by the higher learning speed that is achieved.

In conclusion, we believe that our approach based on training is sound, useful, and compatible with the present directions of robotic research, which is strongly concerned with the development of more complex types of behavior, so complex that they probably cannot be achieved through delayed RL. At present, however, there are some limitations that must be overcome before training can become a standard robot development technique.

8.4 LIMITATIONS

A first limitation of our approach is that it restricts the application of learning to basic and coordination behaviors and presupposes traditional design of other important aspects, like the sensorimotor interface and the architecture of the controller. This limitation is very common in current learning systems for nonsimulated autonomous robots (e.g., Mahadevan and Connell 1992).

A further limitation is that our training technique relies on the availability of fairly sophisticated sensors. It is true that some sensory capacities are required only during training and are not needed for the performance of the learned behavior. For behavior to be adaptive, how-

ever, it is desirable that learning never be completely switched off, and this implies that full sensory capacity (agent's plus trainer's) should be available for the whole life span of the agent. The availability of sophisticated sensors at a reasonably low cost is therefore likely to be a precondition for the success of robot shaping as a development technique. On the other hand, because the availability of powerful and low-cost sensors is also likely to become a precondition for the success of robotics in general, at least for applications in natural or almost natural environments, we expect that our requirements for the sensory apparatus of robots will be compatible with future developments in applied robotics.

Another limitation has to do with the complexity of behaviors that can be dealt with by the present version of ALECSYS. We feel that we have already exploited a system like ALECSYS almost at the maximum of its potentialities. Given that the size of the search space grows exponentially with the amount of input information, there is no hope that this problem can be solved simply by using faster and faster hardware. We therefore need to develop a new tool capable of dealing efficiently with larger amounts of information. At present, our Animat-like tasks make only soft demands on the sensory capacities of the robot: the sensors we use (or assume in simulations) can do little more than give binary information about the presence or absence of simple objects in certain regions of the environment. Realistic applications in complex environments will require the robot to recognize and locate many types of "passive objects," that is, objects that are not specially engineered to be sensed by the robot.

A further problem has to do with the development of dynamic behavior. Most of our experiments were aimed at developing stimulus-response controllers. On two occasions, we went beyond this limit, but clearly the problem deserves a much wider and more systematic study. In particular, it is not clear to what extent more complex behaviors will have to rely on internal models of agent-environment interaction; in other words, we do not know how "cognitive" a robot should be to perform tasks of practical interest in realistic environments.

8.5 THE FUTURE

When thinking about the future developments of a line of research, it is reasonable to distinguish between what can and should be done in the near future and the more visionary intuitions of the far-off lands to which our efforts might finally bring us.

8.5.1 The Near Future

All the limitations discussed in the previous section deserve some research effort. One interesting direction for future research would be to extend the application of learning techniques to aspects of robot behavior other than behavioral learning, in particular, perception and controller architecture.

As far as perception is concerned, it would be desirable to use sensors that can physically adapt to the robot's environment. Given that research on evolvable hardware is still at an embryonic stage (but see Higuchi et al. 1996), we expect that in the near future sensor adaptation will have to be developed in simulated environments.

The architecture of the controller, on the other hand, can, at least in principle, be dealt with satisfactorily at the level of software. This means that the structure of behavioral modules might emerge from the learning process, instead of being predesigned. Another interesting extension to learning would be the integration of step-by-step training with delayed reinforcements (see Caironi and Dorigo 1994; Dorigo and Colombetti 1994b). While we believe that some form of step-by-step reinforcement will be necessary to speed up learning at least in the initial stages, there is no reason why the actual achievement of final goals should not be a source of reinforcement for the agent. It would also be interesting to investigate whether, at least in some applications, we can get completely rid of the step-by-step RP. Recent results in reinforcement learning and training (Shepanski and Macy 1987; Clouse and Utgoff 1992; Nehmzow and McGonigle 1994) suggest that the design of the reinforcement program, which currently requires a substantial effort, could be replaced by direct interaction with a human trainer. In the future, this possibility could be compared with another interesting option, that is, the description of the target behavior through some kind of high-level, symbolic language.

As a whole, we believe that our work shows the importance of learning to achieve a satisfactory level of adaptation between an artificial agent and its environment. Clearly, much further research is needed to understand whether our approach can scale up to a complexity comparable to the adaptive behavior of even simple living organisms and existing preprogrammed robots. In particular, we believe that it is time to face the problem of dealing with continuous input and output data, which would allow a much more satisfactory and reliable coupling between the agent and its environment.

As regards the BAT methodology, we expect it to remain fairly stable in the future. This will allow us to design and implement a number of soft-

ware tools to help designers in robot development. Tasks that strongly need such tools are behavior analysis and specification, in particular, the specification of sensors.

Finally, aspects of behavior engineering that deserve a deeper analysis are the concept of quality of behavior, and the related issue of behavior assessment (see, for example, Smithers 1995). An appropriate set of behavior metrics will have to be developed if this area is to find its way into the field of industrial applications.

8.5.2 Beyond the Horizon

It seems to us that artificial agents will more and more exhibit a double nature: that of artificial systems (i.e., systems designed by humans), and that of quasi-biological agents (see, for example, Steels 1995). The very notions of environment, behavior, and the like obviously derive from biology; the same is true for many computational models widely adopted in the realization of artificial agents, like neural networks and evolutionary computation. Most important, the key concept for agent design seems to us to be that of adaptiveness, again mutuated from biology.

Literally speaking, "adaptive" means "capable of adaptation." Contrary to the term *adaptable*, which has both active and passive acceptations (meaning both "able to adapt" and "able to be adapted"), *adaptive* has an inherent active flavor: for example, an adaptive controller would be one that can tune its parameters on line to control a time-variant process. Biologists distinguish among different kinds of adaptation.[1] The two kinds relevant for us here are *evolutionary* (or *phylogenetic*) *adaptation*, the process by which species adjust to environmental conditions through evolution, and *ontogenetic adaptation*, the process by which an individual adjusts to its environment during its lifetime. As far as behavior is concerned, ontogenetic adaptation is a result of *learning*.

Let us first analyze evolutionary adaptation. Many researchers, including us, have shown that evolutionary computation is a suitable paradigm to develop the behavior of artificial agents. In practical applications, artificial evolution mingles with individual learning in that reproduction only occurs at the software level; evolutionary procedures can thus be viewed as a class of machine learning methods. What characterizes evolutionary methods is the presence of discovery operators, like mutation and crossover, and a process of selection based on an evaluation of the agent's behavior. Given the analogy with natural evolution, the result of such evaluation is usually called "fitness," a term that in biology denotes an

organism's effectiveness in spreading its own genetic makeup. Depending on the approach adopted, fitness can apply to different entities: to the whole behavior exhibited by the agent, to a single behavioral module, or to a highly specific behavior rule. In all cases, however, fitness will always be a function of the actual behavior produced by the agent in its environment.

Given that an individual has high fitness if it produces successful offspring, it is almost tautological that the natural world is mainly inhabited by organisms with high fitness. The interesting problem for biologists is to find out which features of the individual essentially contribute to its overall fitness (or, in other words, which features have high *adaptive value*). In the realm of artificial agents, by contrast, the relationship between fitness and reproductive success is reversed: *first*, the fitness of an individual is computed as a function of its interaction with the environment; and *second*, the fittest individuals are caused to reproduce.

This pattern is not exclusive to artificial systems; it is also applied by breeders of cattle, horses, dogs, and other animals to produce breeds with predefined features. Indeed, we believe that the best metaphor of evolutionary computation is not biological evolution, but breeding. This change of perspective is not without consequences. For example, it is typical of breeding that highly disadaptive features can be selected: the survival of pure-bred animals is often more problematic than that of mongrels. Analogously, applying evolutionary methods to the development of task-oriented behavior may easily result into a robot with a low survival capacity. In both animal breeding and evolutionary robotics, the fault is not in the evolution process; rather, it stems from the definition of a fitness criterion that is not sufficiently comprehensive. This suggests that, as far as possible, the fitness function used to develop robot behavior should take into account the overall quality of the agent, and not performance alone—thus leading to the intuition that quality should be regarded as the artificial counterpart of biological fitness.

An important difference between the natural and the artificial is that, in animals, learning is sharply different from evolution. Moreover, the capacity of individual learning in organisms is itself a product of evolution; we must therefore expect to find such a capacity where it has adaptive value. In the realm of the artificial, on the other hand, there is no sharp conceptual separation between individual learning and evolution. In our experiments, for example, we have implemented individual learning of new behaviors in terms of the evolution of a population of behavior

rules: the ontogenetic adaptation of our agents is actually the result of a sort of phylogenetic adaptation taking place at a finer grain size.

Machine learning techniques can achieve adaptation in two basically different ways: *parameter setting* and *structural learning*. In ALECSYS, for example, the bucket brigade algorithm is used to compute classifier strength (parameter setting), and the genetic algorithm is used to create new classifiers (structural learning). Many well-known learning algorithms, like backpropagation and simulated annealing, can only be used for parameter setting, whereas evolutionary computation methods, like genetic algorithms, are widely used both for parameter setting and structural learning.

There is no doubt that automatic parameter-setting methods are going to be useful, if not essential, for developing artificial agents, because agents will have to adapt to a world that designers can model only with some approximation. For this purpose, a variety of known learning methods can be used, including evolutionary strategies.

The most stimulating applications of evolutionary strategies, however, are those intended to bring about new structures, such as behavior rules, neural network topologies, articulation of a system into modules, and the like. We shall therefore direct our attention to such applications.

From the point of view of structural design, the main interest of evolutionary computation is that it allows us to deal with a larger design space. If we call the conceptual space that is searched by human designers "rational design," we can say that evolutionary strategies search in the space of "nonrational design."[2] The relevant question is then, is there any practical justification to search such a space? Or, in other words, what are the limitation of rational design that we would like to overcome by enlarging the spectrum of possibilities? Before we try to answer this question, we need to better characterize human design.

We think that the main qualifying aspect of rationally designed systems is that they are *modular*. Modularity is what makes a complex system easy to conceive, understand, modify, and so on: divide and conquer is perhaps the most fundamental engineering principle.

A system is modular when it is built from subsystems that are both highly cohesive and loosely coupled (see, for example, Ghezzi, Yazayeri, and Mandrioli 1991). High cohesion is an intramodular property: it means that the subsystem has a well-identified function and contains only components that operate to realize such a function. Loose coupling is an intermodular property: it means that the complexity of the interfaces

between subsystems is small with respect to the internal complexity of the modules.

The virtues of modularity are manifold. Modular systems are easier to understand, maintain, and modify; in general, they are less prone to errors and easier to debug. It is important to note, however, that these properties are not directly related to the intrinsic performance of the system: in principle, a highly modular system may be less efficient than a less modular counterpart, although most nonmodular systems, because they have grown in a chaotic way, often behave rather poorly. The advantages of modularity concern the relationships between an artificial system and the human beings involved in its design, implementation, and maintenance.

According to classical engineering methodologies, lack of modularity is a shortcoming. As regards nature, there is often sharp disagreement as to how modular biological systems are (see, for example, Fodor 1985). While nobody would claim that natural systems show no articulation into subsystems, such subsystems do not seem to be structured in a modular way (at least under the definition of module that we gave above): it is almost a truism that living organisms tend to act as a whole, which implies very strong coupling among subsystems and thus a low degree of modularity.

Given our previous considerations, the lack of modularity of natural systems should come as no surprise. Indeed, observing natural systems, it seems reasonable to assume, at least as a working hypothesis, that their high degree of fitness might profit from their nonmodular organization. Clearly, nature optimizes fitness, not understandability by humans!

That a high degree of modularity is not always a desirable (or possible) choice can be clearly seen by analyzing artificial intelligence programs. Much of artificial intelligence, and in particular knowledge engineering, could be viewed as the engineering of low-modularity software systems. Consider the programming paradigm of production systems: the interactions among production rules in any large production system are strong and, as any artificial intelligence programmer knows, often difficult to control. That the royal road of modularity is abandoned mainly by those who aim at building "intelligent" systems is indeed an interesting fact.

Even if nonmodular systems can be very effective, to work properly they have to solve a number of hard problems, most notably:

- *Redundancy of input information.* While modular systems implement a rational preselection of useful input, nonmodular systems often have to deal with highly redundant input information.

- *Interference among subsystems.* In nonmodular systems, the interaction among subsystems can be very complex. This may lead to useful emergent properties, but also to improper behavior.
- *Difficulty of focusing action.* Nonmodular systems may find it difficult to focus their action in a coherent way.

We should be able to identify specific solutions to these problems in natural agents. In our opinion, this is precisely the role of such important biological mechanisms as *attention*, *inhibition*, and *motivation*: attention deals with highly redundant input information; inhibition solves conflicts arising from interference among subsystems; and motivation focuses the system's activity toward specific goals.

In modular systems, these mechanisms are relatively marginal. Indeed, attention, inhibition, and motivation are not recognized as important architectural concepts in traditional software design. On the other hand, complex concurrent systems do make use of mechanisms, like interrupts, that are clearly related to inhibition; moreover, inhibition plays a fundamental role in the subsumption architecture, proposed by Brooks (1991) as a basis for behavior-based robotics.

Again, it is interesting to look at production systems that have a low degree of modularity. Production systems of the OPS5-type (Brownston et al. 1985) indeed deal with the problems we have pointed out. For example, the "recency principle" can be viewed as a mechanism to focus the interpreter's attention toward recently acquired information; inhibition among production instances is virtually implemented in the selection strategies applied to the conflict set; and context-based metaprogramming is analogous to a motivation mechanism.

Examples of nonmodular artificial systems are the exception, however, not the rule. In spite of the many interesting nonmodular ideas from artificial intelligence, the divide-and-conquer strategy has strong and valid rationality motivations. Is there any concrete need to depart from such a principle?

Our opinion is that we cannot always be perfectly rational. Rationality presupposes knowledge: where we do not know enough, no rational methodology can lead us to the right decisions. And the development of an autonomous agent is a striking example of a case where we do not know enough: the real world is too complex and unpredictable to allow for exhaustive modeling. Thus far, we have experimented with very simple agents in fairly stable environments, which has allowed us to rely on rationality in the design of our robots. But the analysis carried out in this

book leads us to expect that such an approach cannot smoothly scale up to really complex agents interacting with natural environments.

Of course, we have also to keep in mind that a methodology for behavior engineering must be technologically feasible. Evolutionary computation is highly time-consuming; given that physical robots move rather slowly, such computation should be carried out in simulated environments as far as possible. On the other hand, strong adaptation to a real environment can only be achieved in the environment itself, given that simulation environments are too idealized. This dilemma must be resolved in order to work out a feasible engineering methodology.

One possible way out, and one that we have already started to follow (see the CRAB experiment in section 7.5), is to develop artificial agents in simulated environments as far as possible, and then to refine them through learning in the real world. How far this approach can be pursued, however, remains an open question.

Notes

Chapter 1

1. In this book, we take the term *reactive agent* to be a synonym of *stimulus-response* (S-R) *agent*. With this definition, we do not consider any agent whose controller is dynamic (i.e., has an internal state) to be reactive. It is also common to define a reactive agent as the opposite of a deliberative agent, which is able to plan its actions according to an explicit representations of its goals. Our use of the term *reactive* is more restrictive. For example, we do not consider an agent with some memory of the past to be reactive, even when the agent is not deliberative (see Section 4.2.3, and chapter 6).

Chapter 2

1. Other well-known implementations of Holland's LCS are Booker 1988, Booker, Goldberg, and Holland 1989, Robertson and Riolo 1988, and Wilson 1994.

2. The bucket brigade algorithm belongs to the class of temporal difference algorithms (Sutton 1988) and bears much resemblance to Watkins's Q-learning, as discussed, for example, by Dorigo and Bersini (1994) and by Wilson (1994). Most recent research on LCSs has shown that the use of a version of the bucket brigade closer to Q-learning bears beneficial effects to the LCS learning performance (see, for example, Wilson 1994).

3. A solution is also called an "individual," and in LCSs an individual is a "classifier" (see terminology box 2.2).

4. The GA used in LCS_0 is a panmictic GA, that is, parent classifiers are chosen among all the classifiers in the population. This approach has been recently criticized by Wilson (1996; but see also Booker 1982), who proposes to recombine only classifiers belonging to the same action set. The rationale here is that classifiers in a same action set are likely to be relevant to the same or similar problems, and therefore recombination among them could have a higher probability to generate new useful classifiers. Using this restrictive mating for the GA, together with a different way to evaluate classifiers usefulness based on accuracy instead of strength, Wilson has recently designed XCS, which is probably, up to now, the best-performing LCS (Wilson 1995, 1996).

5. It could seem strange to call a sum of strengths "energy." In point of fact, we adopted "energy" because the defined dimension has some of the properties of an energy (e.g., it is additive with respect to subsystems). On the other hand, we chose not to change the well-established usage "strength," to indicate the utility of a classifier.

6. 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).

Chapter 4

1. Questions about representation, rules format, and so on will be discussed in section 4.3.

2. Remember that we restricted the visual field of the AutonoMouse to two overlapping cones of 60 degrees, as opposed to the standard 180-degree overlapping cones, to make the light exit the visual field more frequently.

3. AutonoMouse's performance has been decomposed by computing the ratio (number of times AutonoMouse is rewarded when the light is not visible divided by number of times the light is not visible) for figure 4.5, and analogously for figure 4.6. The two performance graphs have then been interpolated and plotted with respect to the actual cycle number to obtain comparable graphs.

4. Because motor messages are six bits long, this implies that three bits in the action string are useless.

5. Rules in LCS-chase are 15 bits long, in LCS-hide 18 bits long, and in LCS-switch 6 bits long.

Chapter 5

1. If there is correct mapping, we say that ALECSYS and the AutonoMouse are "well calibrated."

2. We call "lesions" malfunctions that make the behavior of a sensor or motor systematically differ from design specifications, as distinguished from "noise," which in our experiments could be modeled as a Gaussian distribution around a mean behavior that meets the design specifications.

3. In this and in the following experiments, 100 cycles took about 60 seconds.

4. For example, the experimenter could keep the light at the right side of AutonoMouse II to teach the robot to turn right.

5. In this experiment the performance index was given by the light intensity seen by the frontal eyes when AutonoMouse II had to move forward, and by the light intensity seen by the rear eyes when the robot had to move backward.

Chapter 7

1. In general, further aspects of quality will be involved in any practical application.

2. The difference between *adaptiveness* and *robustness* can be stated as follows: an agent is *adaptive* if its controller is capable to change online its characteristics in such a way to make its behavior better matched to a changing environment, while an agent is *robust* when its controller is able to maintain a reasonable level of performance in a changing environment without changing its own structure.

3. The name HAMSTER (Highly Autonomous Mobile System trainED by Reinforcements) is mainly justified by the robot's target behavior.

Chapter 8

1. See, for example, the entry ADAPTATION in *The Oxford Companion to Animal Behaviour* (McFarland 1981).

2. We prefer the term *nonrational design* to *irrational design* because in everyday language the latter has a negative connotation, while the former is more neutral.

References

Agre, P. E., and Chapman, D. 1987. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*. Morgan Kaufmann.

Arkin, R. C. 1990. Integrating behavioral, perceptual, and world knowledge in reactive navigation. In P. Maes, ed., Special issue on designing autonomous agents. *Robotics and Autonomous Systems* 6 (1–2): 105–122.

Asada, M., Noda, S., Tawaratsumida, S., and Hosoda, K. 1996. Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Machine Learning* 23 (2–3): 279–303.

Barto, A. G., Bradtke, S. J., and Singh, S. P. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72 (1–2): 81–138.

Barto, A. G., Sutton, R. S., and Anderson, C. W. 1983. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics* 13 (5): 834–846.

Beer, R. D. 1995. A dynamical systems perspective on autonomous agents. *Artificial Intelligence* 72 (1–2): 173–215.

Beer, R. D., and Gallagher, J. C. 1992. Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior* 1 (1): 91–122.

Belew, R. K., McInerney, J., and Schraudolph, N. N. 1991. Evolving networks: Using the genetic algorithm with connectionist learning. In *Proceedings of the Second International Conference on Artificial Life*. Addison-Wesley.

Bellman, R. 1957. *Dynamic programming*. Princeton University Press.

Bertoni, A., and Dorigo, M. 1993. Implicit parallelism in genetic algorithms. *Artificial Intelligence* 61 (2): 307–314.

Bertsekas, D. 1987. *Dynamic programming: Deterministic and stochastic models*. Prentice-Hall.

Bonarini, A. 1996. Learning dynamic fuzzy behaviors from easy missions. In *Proceedings of IPMU '96*. Proyecto Sur de Ediciones, Granada.

Booker, L. B. 1982. Intelligent behavior as an adaptation to the task environment. Ph.D. diss., University of Michigan, Ann Arbor.

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 (1–3): 235–282.

Brooks, R. A. 1990. Elephants don't play chess. In P. Maes, ed., Special issue on designing autonomous agents. *Robotics and Autonomous Systems* 6 (1–2): 3–16.

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

Brownston, L., Farrell, F., Kant, E., and Martin, N. 1985. *Programming expert systems in OPS5: An introduction to rule-based programming*. Addison-Wesley.

Caironi, P. V. C., and Dorigo, M. 1994. Training Q-Agents. Technical Report IRIDIA/94-14. Université Libre de Bruxelles (Brussels).

Camilli, A., Di Meglio, R., Baiardi, F., Vanneschi, M., Montanari, D., and Serra, R. 1990. Parallelizzazione di Sistemi a Classificatori su architetture MIMD. Progetto finalizzato sistemi informatici e calcolo parallelo, Sottoprogetto 3 (architetture parallele), Technical Report 3-17, CNR, Italy.

Cliff, D., and Bullock, S. 1993. Adding "foveal vision" to Wilson's Animat. *Adaptive Behavior* 2 (1): 49–72.

Cliff, D., Harvey I., and Husbands, P. 1993. Explorations in evolutionary robotics. *Adaptive Behavior* 2 (1): 73–110.

Cliff, D., and Ross, S. 1994. Adding temporary memory to ZCS. *Adaptive Behavior* 3 (2): 101–150.

Clouse, J. A., and Utgoff, P. E. 1992. A teaching method for reinforcement learning. In *Proceedings of the Ninth Conference on Machine Learning*. Morgan Kaufmann.

Cohen, P. R. 1995. *Empirical methods for Artificial Intelligence*. MIT Press.

Colombetti, M., and Dorigo, M. 1993. Learning to control an autonomous robot by distributed genetic algorithms. In J. A. Meyer, H. Roitblat, and S. W. Wilson, eds., *Proceedings, From Animals to Animats 2: Second International Conference on Simulation of Adaptive Behavior (SAB92)*. 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.

Cramer, N. L. 1985. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*. Erlbaum.

Dickmanns, D., Schmidhuber, J., and Winklhofer, A. 1987. Der genetische Algorithmus: Eine Implementierung in Prolog. Fortgeschrittenenpraktikum, Institut für Informatik, Lehrstuhl Prof. Radig, Technische Universität, Munich.

Donnart, J.-Y., and Meyer, J.-A. 1996. Learning reactive and planning rules in a motivationally autonomous Animat. In M. Dorigo, ed., Special issue on learning

autonomous robots. *IEEE Transactions on Systems, Man, and Cybernetics* Part B 26 (3): 381–395.

Dorigo, F., and Maesani, A. 1993. Development of a mobile robot: Integration of design and machine learning techniques (in Italian). Master's thesis, Dipartimento di Elettronica e Informazione, Politecnico di Milano (Milan).

Dorigo, M. 1992. Using transputers to increase speed and flexibility of genetics-based machine learning systems. *Microprocessing and Microprogramming Journal* 34: 147–152.

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

Dorigo, M. 1995. ALECSYS and the AutonoMouse: Learning to control a real robot by distributed classifier systems. *Machine Learning* 19 (3): 209–240.

Dorigo, M., ed. 1996. Special issue on learning autonomous robots. *IEEE Transactions on Systems, Man, and Cybernetics* Part B 26 (3): 361–364.

Dorigo, M., and Bersini, H. 1994. A comparison of Q-learning and classifier systems. In D. Cliff, P. Husbands, J. B. Pollack, and S. W. Wilson, eds., *Proceedings, From Animals to Animats 3: Third International Conference on Simulation of Adaptive Behavior (SAB94)*. MIT Press.

Dorigo, M., and Colombetti, M. 1994a. Robot shaping: Developing autonomous agents through learning. *Artificial Intelligence* 71 (2): 321–370.

Dorigo, M., and Colombetti, M. 1994b. The role of the trainer in reinforcement learning. In *Proceedings of MLC-COLT '94 Workshop on Robot Learning*, New Brunswick, NJ. Available on Internet at http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/robot-learning.html

Dorigo, M., and Schnepf, U. 1991. Organisation of Robot Behaviour through Genetic Learning Processes. In *Proceedings of the Fifth IEEE International Conference on Advanced Robotics (ICAR '91)*. IEEE Press.

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. 1991. ALECSYS: A parallel laboratory for learning classifier systems. In *Proceedings of the Fourth International Conference on Genetic Algorithms*. Morgan Kaufmann.

Dorigo, M., Patel, M. J., and Colombetti, M. 1994. The effect of sensory information on reinforcement learning by a robot arm. In *Proceedings of the Fifth International Symposium on Robotics and Manufacturing (ISRAM '94)*. ASME Press.

Dress, W. B., 1987. Darwinian optimization of synthetic neural systems. In *Proceedings of the First IEEE International Conference on Neural Networks*. IEEE Press.

Floreano, D., and Mondada, F. 1996. Evolution of homing navigation in a real mobile robot. In M. Dorigo, ed., Special issue on learning autonomous robots. *IEEE Transactions on Systems, Man, and Cybernetics* Part B 26 (3): 396–407.

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

Fodor, J. 1985. The modularity of mind. Focal article with commentary. *Behavioral and Brain Sciences* 8: 1–42.

Fogel, D. B., Fogel, L. J., and Porto V. W. 1990. Evolving neural networks. *Biological Cybernetics* 63 (6): 487–493.

Franklin, J., Mitchell, T., and Thrun, S., eds. 1996. Special issue on robot learning. *Machine Learning* 23 (2–3).

Fujiki, C., and Dickinson, J. 1987. Using the genetic algorithm to generate Lisp source code to solve the Prisoner's dilemma. In *Proceedings of the Second International Conference on Genetic Algorithms*. Erlbaum.

Gaussier, P., ed. 1995. Special issue on moving the frontiers between robotics and biology. *Robotics and Autonomous Systems* 16 (2–4).

Ghezzi, C., Jazayeri, M., and Mandrioli, D. 1991. *Fundamentals of software engineering*. Prentice-Hall.

Goldberg, D. E. 1989. *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley.

Gordon, D., and Subramanian, D. 1994. A multistrategy learning scheme for agent knowledge acquisition. *Informatica* 17: 331–346.

Grefenstette, J. J., and Ramsey, C. L. 1992. An approach to anytime learning. In *Proceedings of the Ninth International Machine Learning Conference*. Morgan Kaufmann.

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., and Schultz, A. C. 1994. An evolutionary approach to learning in robots. In *Proceedings of MLC-COLT '94 Workshop on Robot Learning*. New Brunswick, NJ.

Harvey, I., Husbands, P., and Cliff D. 1994. Seeing the light: Artificial evolution, real vision. In D. Cliff, P. Husbands, J. B. Pollack, and S. W. Wilson, eds., *Proceedings, From Animals to Animats 3: Third International Conference on Simulation of Adaptive Behavior (SAB94)*. MIT Press.

Hicklin, J. F. 1986. Application of the genetic algorithm to automatic program generation. Master's Thesis, University of Idaho.

Higuchi, T., Iwata, M., Kajitani, I., Iba, H., Hirao, Y., Furuya, T., and Manderick, B. 1996. Evolvable hardware and its application to pattern recognition and fault-tolerant systems. In E. Sanchez and M. Tomassini, eds., *Towards Evolvable Hardware: An Evolutionary Engineering Approach*. Lecture Notes in Computer Science 1062. Springer.

Hillis, W. D. 1985. *The connection machine*. MIT Press.

Holland, J. H. 1975. *Adaptation in natural and artificial systems*. University of Michigan Press. Reprint, MIT Press, 1992.

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. 1986. Escaping brittleness: The possibilities of general purpose learning algorithms applied to parallel rule-based systems. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, eds., *Machine Learning II*. Morgan Kaufmann.

Holland, J. H., Holyoak, K. J., Nisbett, R. E., and Thagard, P. R. 1986. *Induction: Processes of inference, learning, and discovery*. MIT Press.

Holland, J. H., and Reitman, J. S. 1978. Cognitive systems based on adaptive algorithms. In D. A. Waterman and F. Hayes-Roth, eds., *Pattern-directed inference systems*. Academic Press.

INMOS. 1989. *The transputer handbook*. 2d ed.

Jacoby, N., Husbands, P., and Harvey, I. 1995. Noise and the reality gap: The use of simulation in evolutionary robotics. In *Proceedings of the Third International Conference on Artificial Life*. Springer.

Kaelbling, L. P. 1987. An architecture for intelligent reactive systems. In M. P. Georgef, and A. L. Lansky, eds., *Reasoning about actions and plans*. Morgan Kaufmann.

Kaelbling, L. P! 1990. Learning in embedded systems. Ph.D. diss., Stanford University, Stanford, CA.

Kaelbling, L. P., 1991. An adaptable mobile robot. In *Proceedings of the First European Conference on Artificial Life*. MIT Press.

Kaelbling, L. P., Littman, L. M., and Moore, A. W. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research* 4: 237–285.

Kaelbling, L. P., and Rosenschein, S. J. 1991. Action and planning in embedded agents. In P. Maes, ed., Special issue on designing autonomous agents. *Robotics and Autonomous Systems* 6 (1–2): 35–48.

Kitano, H. 1990. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems* 4: 461–476.

Koza, J. R. 1992. *Genetic programming*. MIT Press.

Koza, J. R., and Rice, J. P. 1992. Automatic programming of robots using genetic programming. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*. AAAI Press.

Kröse, B. J. A., ed. 1995. Special issue on reinforcement learning and robotics. *Robotics and Autonomous Systems* 15 (4).

Latombe, J. C. 1991. *Robot motion planning*. Kluwer.

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. In *Proceedings of the 1993 IEEE International Conference on Neural Networks*. IEEE Press.

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

Lin, L.-J., and Mitchell, T. M. 1992. Memory approaches to reinforcement learning in non-Markovian domains. Technical Report CMU-CS-92-138. School of Computer Science, Carnegie Mellon University, Pittsburgh.

Littman, M. L. 1993. An optimization-based categorization of reinforcement learning environments. In J. A. Meyer, H. Roitblat, and S. W. Wilson, eds., *Proceedings, From Animals to Animats 2: Second International Conference on Simulation of Adaptive Behavior (SAB92)*. MIT Press.

Maclin, R., and Shavlik, J. W. 1996. Creating advice-taking reinforcement learners. *Machine Learning* 22 (1–3): 251–281.

Maes, P. 1990. A bottom-up mechanism for behavior selection in an artificial creature. In *Proceedings, From Animals to Animats: First International Conference on Simulation of Adaptive Behavior (SAB90)*. MIT Press.

Maes, P. 1994. Modeling adaptive autonomous agents. *Artificial Life* 1 (1–2): 135–162.

Maes, P., and Brooks, R. A. 1990. Learning to coordinate behaviors. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*. AAAI Press.

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

Maniezzo, V. 1994. Genetic evolution of the topology and weight distribution of neural networks. *IEEE Transactions on Neural Networks* 5 (1): 39–53.

Matarić, M. J., and Cliff, D. 1996. Challenges in evolving controllers for physical robots. Special issue on evolutional robotics, *Robotics and Autonomous Systems* 19 (1): 67–83.

McFarland, D., ed. 1981. *The Oxford Companion to Animal Behaviour*. Oxford University Press.

Meeden, L. A. 1996. An incremental approach to developing intelligent neural network controllers for robots. In M. Dorigo, ed., Special issue on learning autonomous robots. *IEEE Transactions on Systems, Man, and Cybernetics* Part B 26 (3): 474–484.

Michalewicz, Z. 1992. *Genetic algorithms + data structures = evolution programs*. Springer.

Millán, J. del R. 1996. Rapid, safe, and incremental learning of navigation strategies. In M. Dorigo, ed., Special issue on learning autonomous robots. *IEEE Transactions on Systems, Man, and Cybernetics* Part B 26 (3): 408–420.

Millán, J. del R., and Torras, C. 1992. A reinforcement connectionist approach to robot path finding in non-maze-like environments. *Machine Learning* 8 (3–4): 363–395.

Mitchell, M. 1996. *An introduction to genetic algorithms*. MIT Press.

Moriarty, D. E., and Mikkulainen, R. 1996. Efficient reinforcement learning through symbiotic evolution. *Machine Learning* 22 (1–3): 11–32.

Nehmzow, U., and McGonigle, B. 1994. Achieving rapid adaptations in robots by means of external tuition. In D. Cliff, R. Husbands, J. B. Pollack, and S. W. Wilson, eds., *Proceedings, From Animals to Animats: Third International Conference on Simulation of Adaptive Behavior (SAB94)*. MIT Press.

Nilsson, M., and Ojala, J. 1995. "Self-awareness" in reinforcement learning of snake-like robot locomotion. In *Proceedings of IASTED 95*, Cancun. Acta Press.

Nolfi, S., Floreano, D., Miglino, O., and Mondada, F. 1994. How to evolve autonomous robots: Different approaches in evolutionary robotics. In *Proceedings of the Fourth International Workshop on Artificial Life*. MIT Press.

Nolfi, S., and Parisi, D. 1992. Growing neural networks. In *Proceedings of the Third International Conference on Artificial Life*. Addison-Wesley.

Nordin, P., and Banzhaf, W. 1996. A genetic programming system learning obstacle avoiding behavior and controlling a miniature robot in real time. Technical Report 4/95. Department of Computer Science, University of Dortmund.

Patel, M. J., Colombetti, M., and Dorigo, M. 1995. Evolutionary learning for intelligent automation: A case study. *Intelligent Automation and Soft Computing Journal* 1 (1): 29–42.

Patel, M. J., and Dorigo, M. 1994. Adaptive learning of a robot arm. In *Proceedings of Evolutionary Computing AISB Workshop (Selected Papers)*. Springer.

Piroddi, R., and Rusconi, R. 1992. A parallel classifier system to solve learning problems (in Italian). Master's thesis, Dipartimento di Elettronica e Informazione, Politecnico di Milano (Milan).

Puterman, M. 1994. *Markov decision processes: Discrete dynamic stochastic programming*. Wiley.

Riolo, R. L. 1987. Bucket brigade performance: 2. Default hierarchies. In *Proceedings of the Second International Conference on Genetic Algorithms*. Erlbaum.

Riolo, R. L. 1989. The emergence of default hierarchies in learning classifier systems. In *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann.

Robertson, G. G. 1987. Parallel implementation of genetic algorithms in a classifier system. In *Proceedings of the Second International Conference on Genetic Algorithms*. Erlbaum.

Robertson, G. G., and Riolo, R. L. 1988. A tale of two classifier systems. *Machine Learning* 3 (2–3): 139–160.

Rosenschein, S. J., and Kaelbling, L. P. 1986. The synthesis of digital machines with provable epistemic properties. In *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning about Knowledge*. Morgan Kaufmann.

Ross, S. 1983. *Introduction to stochastic dynamic programming*. Academic Press.

Saffiotti, A., Konolige, K., and Ruspini, E. H. 1995. A multivalued logic approach to integrating planning and control. *Artificial Intelligence* 76 (1–2): 481–526.

Saffiotti, A., Ruspini, E. H., and Konolige, K. 1993. Integrating reactivity and goal-directedness in a fuzzy controller. In *Proceedings of the Second Fuzzy-IEEE Conference*. IEEE Press.

Schaffer, J. D., Whitley, D., and Eshelman, L. J. 1992. Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92)*. IEEE Press.

Shepanski, J. F., and Macy, S. A. 1987. Manual training techniques of autonomous systems based on artificial neural networks. In *Proceedings of the First IEEE Annual International Conference on Neural Networks*. IEEE Press.

Siegel, S., and Castellan, N. J. 1988. *Nonparametric statistics for the behavioral sciences*. 2d ed. McGraw-Hill.

Singh, S. P. 1992. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning* 8 (3–4): 323–339.

Skinner, B. F. 1938. *The behavior of organisms: An experimental analysis*. Appleton Century.

Smithers, T. 1995. On quantitative performance measures of robot behaviour. In L. Steels, ed., Special issue on the biology and technology of intelligent and autonomous systems. *Robotics and Autonomous Systems* 15 (1–2): 107–134.

Sommerville, I. 1989. *Software engineering*. Addison-Wesley.

Steels, L. 1990. Cooperation between distributed agents through self-organization. In Y. Demazeau, and J.-P. Müller, eds., *Decentralized* AI. North-Holland.

Steels, L. 1994. The artificial life roots of artificial intelligence. *Artificial Life* 1 (1–2): 75–110.

Steels, L., ed. 1995. Special issue on the biology and technology of intelligent and autonomous systems. *Robotics and Autonomous Systems* 15 (1–2).

Sutton, R. S. 1984. Temporal credit assignment in reinforcement learning. Ph.D. diss., University of Massachusetts, Amherst.

Sutton, R. S. 1988. Learning to predict by the methods of temporal differences. *Machine Learning* 3 (1): 9–44.

Sutton, R. S. 1990. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*. Morgan Kaufmann.

Sutton, R. S. 1991. Dyna, an integrated architecture for learning, planning, and reacting. *SIGART Bulletin* 2: 160–163.

Sutton, R. S., Barto, A. G., and Williams, R. J. 1991. Reinforcement learning is direct adaptive optimal control. In *Proceedings of the 1991 American Control Conference*. American Control Council.

Tani, J. 1996. Model-based learning for mobile robot navigation from the dynamical systems perspective. In M. Dorigo, ed., Special issue on learning autonomous robots. *IEEE Transactions on Systems, Man, and Cybernetics* Part B 26 (3): 421–436.

Thrun, S., and Mitchell, T. 1993. Integrating inductive neural network learning and explanation-based learning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*. Morgan Kaufmann.

Utgoff, P., and Clouse, J. 1991. Two kinds of training information for evaluation function learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*. AAAI Press.

Watkins, C. J. C. H. 1989. Learning with delayed rewards. Ph.D. diss., University of Cambridge.

Watkins, C. J. C. H., and Dayan, P. 1992. Technical note: Q-learning. *Machine Learning* 8 (3–4): 279–292.

Whitehead, S. D., 1991a. A study of cooperative mechanisms for faster reinforcement learning. Technical Report TR 365. Computer Science Department, University of Rochester.

Whitehead, S. D., 1991b. A complexity analysis of cooperative mechanisms in reinforcement learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*. AAAI Press.

Whitehead, S. D., and Ballard, D. H. 1991. Learning to perceive and act by trial and error. *Machine Learning* 7 (1): 45–83.

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

Whitley, D., Starkweather, T., and Bogart, C. 1990. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel Computing* 14: 347–361.

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

Wilson, S. W. 1985. Knowledge growth in an artificial animal. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*. Erlbaum.

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

Wilson, S. W. 1990. The Animat path to AI. In *Proceedings, From Animals to Animats: First International Conference on the Simulation of Adaptive Behavior (SAB90)*. MIT Press.

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

Wilson, S. W. 1995. Classifier fitness based on accuracy. *Evolutionary Computation* 3 (2): 149–176.

Wilson, S. W. 1996. Generalization in XCS. Unpublished contribution to the ICML '96 Workshop on Evolutionary Computation and Machine Learning, Bari, Italy. Available on Internet at http://netq.rowland.org/sw/swhp.html

Index