DEVELOPMENT OF AN EMBEDDED EVOLUTIONARY CONTROLLER TO ENABLE COLLISION-FREE NAVIGATION OF A POPULATION OF AUTONOMOUS MOBILE ROBOTS

A thesis submitted to
The University of Kent at Canterbury
In the subject of Electronic Engineering
for the degree of
Doctor of Philosophy

By

Eduardo do Valle Simões

November 2000

To Liane with love

Abstract

This work studies evolutionary computation applied to robotics. It has produced a genetic system where the population exists in a real environment, where they exchange genetic material and reconfigure themselves as new individuals to form the next generations, providing the means of running genetic evolutions in a real physical platform. This thesis describes the techniques that could be adapted from the literature as well as the novel techniques developed to allow the design of the hardware and software necessary to embedding the distributed evolutionary system. It also describes the environment where the experiments were carried out in real time and in simulation. These experiments test the influence of different parameters, such as different mutation rates and partner selection, crossover, and reproduction strategies. This work reports comparative studies between different methods of embedding an evolutionary control circuit in a population of six autonomous mobile robots. The reviewed architectures are: evolvable hardware; dynamic state machine; condition-behaviour mapping; pulse stream neural systems; and the chosen one: RAM neural networks.

This work proposes and implements a fully embedded distributed evolutionary system that is able to achieve collision free-navigation in a few hundreds of trials. Evolution can manipulate the morphology of the robot: the configuration of the sensors and the motor speed levels. This thesis presents the first experimental proofs of the embedded evolution concept applied to the evolution of the morphology and control circuit of a population of real robots in real time. It proposes the *Predation strategy* that not only can improve the performance of the system but also prevents the population from being stuck in local optima. It is demonstrated that evolution can help in the traditional design of robotic platforms, since it can suggest the best features a robot should incorporate to perform specific tasks. Finally, this work provides understanding on the implementation of real evolutionary systems and inspiring insights that have potential of application in the areas of automation and cybernetics.

Acknowledgments

Many people supported and helped during the completion of this work, and now it is time to find the right words to thank all of them.

I would like to express my deep gratitude to my very kind friend and supervisor Dr. Keith Dimond for his guidance, advice, and encouragement during the whole period of this work.

I am sincerely thankful for the help and assistance provided by the members of staff in the Electronic Engineering Laboratory. In particular to Harvey who has generously supplied his deep and diverse pool of knowledge in 68HC11 design and technologies; and to Clive and Terry, for all the guidance in manufacturing the robots; to Dave Smith and Gill for being so friendly every time I showed up with a list of hundreds of components to be ordered.

My sincere thanks also to my friends Patrícia and Flávio Zigelmann from the Department of Statistics for the inspiration and endless explanations on how to analyse my experimental data.

During the development of my Ph.D., three colleagues stood out and marked this period with a special shine. They are that kind of people you do not even need to thank, because they already know… Still, I would like to express my deepest affection to Anne, Dimitrios, and Rick, for all the unconstrained help, productive discussions, and friendship that I hope will last far beyond this Ph.D.

I am also very grateful to the CNPq, the Brazilian Council for Research, for providing the opportunity to study at the University of Kent.

I would also like to thank all my friends in the Electronic Engineering Laboratory, especially Alex, Shanta, Fuadd, Osama, Hossam, Samuel, Elina, Julia, and William for providing support and a friendly environment. Particularly to Jose for <u>not</u> touching the electrostatic sensitive robots, and last, but by no means least, to Dr. Ng for helping so promptly.

Finally, I would like to thank my family for giving me the initial encouragement and tools to pursue my ambitions. In particular, I wish to thank my wife, Liane who, being a scientist herself, has contributed directly and indirectly to my work. Without her loving support and encouragement over the years I could never have managed to get this work done.

Publications arising from this work

- Simões, E.D.V.; Dimond, K.R. *An evolutionary controller for autonomous multi-robot systems*. In IEEE International Conference on Systems, Man and Cybernetics, October, 1999, Tokyo, Japan, v.6, pp. VI596-VI601, 1999, *Invited Paper*.
- Simões, E.D.V.; Dimond, K.R. *Embedding a distributed evolutionary system into a population of autonomous mobile robots*. To be published at the IEEE International Conference on Systems, Man and Cybernetics, October, 2001, Tuscon, USA.
- Simões, E.D.V.; Dimond, K.R. *Predation: an approach to improving the evolution of real robots with an embedded evolutionary controller*. Submitted to IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2001), October 2001, Maui, Hawaii, USA.
- Simões, E.D.V.; Dimond, K.R. *Inheritance Selection: inheriting fitness values in an embedded evolutionary system*. In preparation to be submitted to IEE Proceedings: Computers and Digital Techniques.

Table of Contents

Table of Figures

List of Tables

Glossary

AE	Artificial Evolution
AI	Artificial Intelligence
ALife	Artificial Life
ALU	Arithmetic Logic Unit
AM	Amplitude Modulation
ANN	Artificial Neural Network
ANS	Artificial Neural Systems
ASCII	American Standard Code for Information Interchange
CD-ROM	Compact Disk Read-only Memory
CMOS	Complementary Metal Oxide Silicon
CPWM	Coherent Pulse Width Modulation
DC	Direct Current
DIP	Dual Inline Package
DSM	Dynamic State Machine
EA	Evolutionary Algorithm
EC	Evolutionary Computation
EE	Embedded Evolution
EEPROM	Electronically-Erasable Programmable Read-only Memory
EHW	Evolvable Hardware
ER	Evolutionary Robotics
ES	Evolutionary Systems
FF	Front Fast
FM	Front Medium
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GAs	Genetic Algorithms
GP	Genetic Programming
GSN	Goal-Seeking Neuron
IE	Incremental Evolution

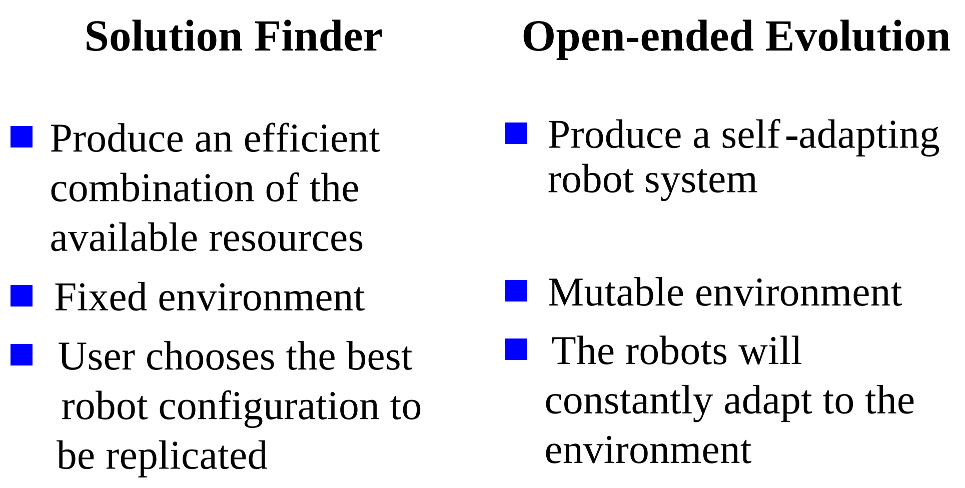
IR	Infrared
ISA	Industry Standard Architecture
LC	Logic Cell
LED	Light-emitting Diode
NN	Neural Network
PC	Personal Computer
PCB	Printed Circuit Board
Ph.D.	Doctor of Philosophy
PID	Proportional Integral Derivative
PLN	Probabilistic Logic Neuron
pRAM	Probabilistic Random Access Memory
PROM	Programmable Read-only Memory
PS	Pulse Stream
PWM	Pulse-width Modulation
RAM	Random Access Memory
ROM	Read-only Memory
S/C	Signal Conversion
SAGA	Species Adaptation Genetic Algorithm
SCI	Serial Communication Interface
TLL	Turn Left Long
TLS1	Turn Left Short1
TLS2	Turn Left Short2
TPU	Time Processing Unit
TRL	Turn Right Long
TRS1	Turn Right Short1
TRS2	Turn Right Short2
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VIA	Versatile Interface Adaptor

1 INTRODUCTION

This chapter introduces this Ph.D. thesis, explaining what is proposed and the contributions to the state-of-the-art. It presents the perspectives of Evolutionary Robotics (ER) for multi-robot control and overviews its main approaches, unanswered problems, and some promising directions. This chapter also introduces the workspace, the chosen application, and the delimitation of the Ph.D. project. It also shows a synopsis of the thesis organisation, chapter by chapter. The concepts mentioned here will be thoroughly explained in Chapters 2 and 3.

This work is concerned with automated synthesis of robotic embedded controllers using Evolutionary Computation [Par96]. This is a new research field also known as: Artificial Evolution, Evolvable Systems, or Evolutionary Electronics [Ale66]. Evolutionary methods have been employed for developing robot controllers automatically in simulation [Cli96], on physical systems [Hig96b], and combinations. Specifically, this work concentrates in embedding an evolutionary algorithm within a population of physical robots.

A review of the state-of-the-art of robot control strategies showed the need for contributions in the areas of self-training [Cli92] [Gra96], autonomy and auto-adaptation to changes in the environment where the robots are operating [Hus93] [Per96b] [Tom96]. Therefore, the main interest of this work will be the development of a strategy for controlling a group of robots that combines these approaches into a promising solution involving Evolutionary Computation [Yao97]. It is intended to be applied to control a group (population) of robots in the field, instead of just using the evolving group to develop an optimum controller for a single robot, what has been the main purpose of EC until now [Flo98b]. The proposed evolutionary system innovates for it can produce not only a trained robot but also an open-ended evolution, continuously adapting the robot controllers to cope with a variable environment. Figure 1.1 shows the difference between applying evolution as a solution finder (the traditional approach) and as an open-ended evolutionary system. The first concentrates on producing an efficient combination of the available resources in a fixed environment until some threshold level is achieved. The user then chooses the best robot configuration to replicate. If the environment conditions change, the robot controller is doomed to fail [Jak95] [Jak96]. The proposed system is able to face a mutable environment, since the robots are constantly being modified by evolution to fit these variations.



Solution Finder

- Produce an efficient combination of the available resources
- Fixed environment
- User chooses the best robot configuration to be replicated

Open-ended Evolution

- Produce a self-adapting robot system
- Mutable environment
- The robots will constantly adapt to the environment

Figure 1.1 – Two different ways of applying evolution to a robotic system.

Instead of using Artificial Evolution (AE) [Flo94] [Fun00a], such as Genetic Algorithms (GAs) [Mit95], as an optimisation technique for a conventional Neural Network (NN) architecture [Gar94b] [Nor94] [Per96a], the proposed solution means using it to design robot control circuits automatically [Cli94] [Flo96b]. The evolutionary technique can operate by modifying the configuration of a predefined architecture (such as a Neural Network), or by building the whole control circuit from scratch using a programmable device (such as an FPGA). Figure 1.2 shows how the robot control circuit interfaces a sensor module, from which it receives the sensor readings, and commands the motor drive module on how to drive the motors. Evolution can work with basic low-level primitives, together with noise and uncertainty, to synthesise a solution that could be of a very different nature to the way electronic circuits are normally designed [Tho94c] [Key97d]. AE can build/evolve systems that are too complex to understand, but functional, nevertheless. This work consists of building an Artificial Genetic System, where the population will actually exist as real robots and exchange genetic information in order to adapt to solve a particular problem.

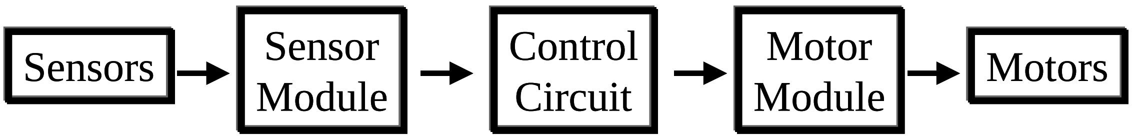


Figure 1.2 – How the evolvable control circuit fits in the robot architecture.

Most of the work to date has applied genetic algorithms to produce controllers for a simulated (Figure 1.3-a) population of robots [Cli96] [Fic00] [Fun00b]. Limited numbers of researches have evolved the controllers in simulation and then transferred them onto physical robots [Mig95] [Smi98]. An even smaller number of researchers have conducted experiments where the genetic algorithms run in an external computer and the configuration pattern for each individual is downloaded to one single physical robot (Figure 1.3-b) to be evaluated one at a time [Mon95] [Tho95] [Flo96c] [Flo97b]. This still cannot be considered a real evolutionary system, because the population cannot interact in parallel and most of the uncertainty of a truly physical evolutionary system where all individuals exist and interact in real time will not be present [Jak95] [Jak97a] [Jak97b]. Finally, just a handful of experiments, reported by Floreano and Mondada [Flo01], makes use of two or more physical robots, but still tethered to external computation (Figure 1.3-c) where the genetic code resides. Therefore, this work is novel in implementing for the first time a fully embedded evolutionary system.

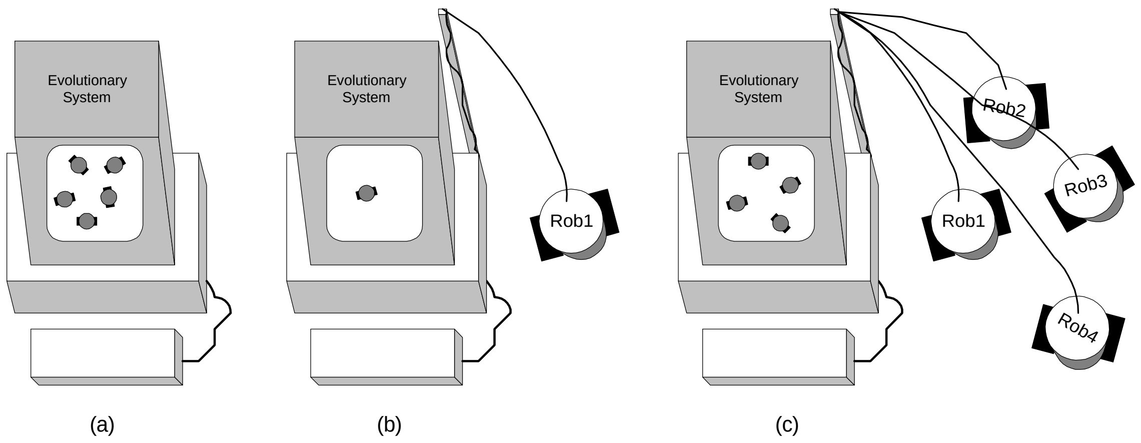


Figure 1.3 – Different ways to apply evolutionary techniques to robotics: simulated environment (a); simulated evolution with real robot evaluation (b); and real population tethered to an external computer where the evolutionary system resides (c).

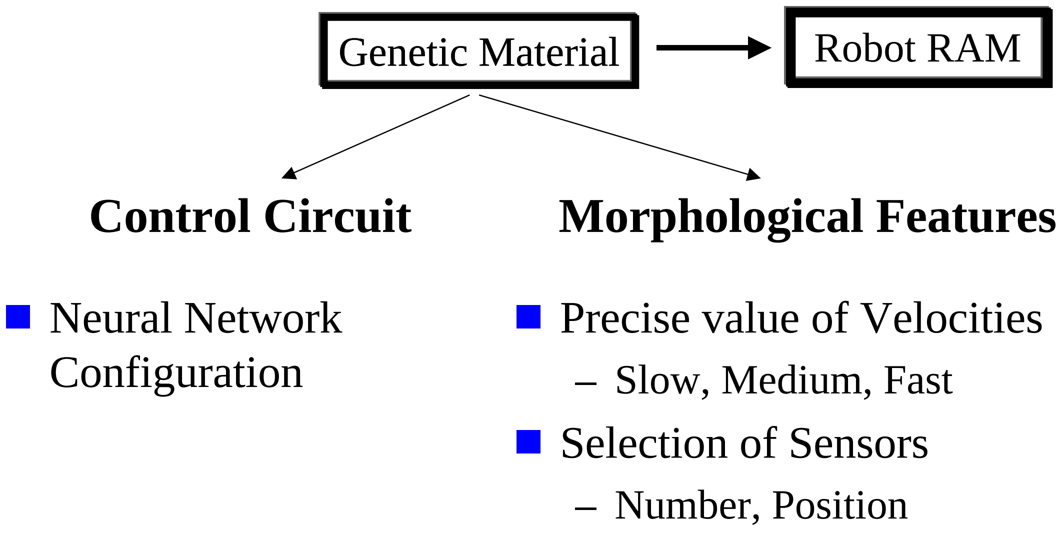


Figure 1.4 – The genetic material stored in the robot RAM defines not only the control circuit, but also the morphological features of the robot.

Differently from other authors that involve simulation in some of the evolutionary phases [Har93b] [Mil94] [Ste94] [Bul95] [Tho96a], the term *Evolutionary System* will be applied in this work to describe an environment where the individuals physically exist and artificially breed and die, to give place to the next generation. For that reason, it is not an Evolutionary Algorithm, but a real Evolutionary System. The initial literature review has demonstrated that an evolutionary system has never been implemented fully on-board of a population of real robots, working completely independent of external computation [Har97b] [Yao97] [Lay99a] [Teu99] [Wer99]. It was not until 1999 that Watson et al. [Wat99a] (evolving the controller) and the publication of the preliminary results of this work [Sim99] (evolving the controller and morphology) claimed to provide the first experimental proofs of the Embedded Evolution concept. Embedded Evolution is defined in this work as evolution that takes place in a population of real robots, completely independent from external computation or human intervention to evaluate, reproduce, and reposition the robots for new trials. Therefore, what is proposed is the physical implementation of a genetic system containing the robots as the individuals and a genetic code (bits stored in the RAM memory of the robots) that specifies the configuration of their control device, and their speed and sensor organisation (morphology). In the scope of this work, the term morphology is defined as the physical, embodied characteristics of the robot, such as its mechanics and sensor organisation [Mat96]. So far, the few examples of evolving both morphology and control exist only in simulation [Lip00]. Therefore, this thesis will show for the first time an embedded system that evolves both the control circuit and the morphology of the robots. Figure 1.4 shows that the genetic material of the robots can define their control circuit (the configuration of a Neural Network), the position of the sensors and the precise value of motor velocities.

To allow a clear investigation of the implicit interactive features within the evolutionary approach, a relatively simple task that does not involve explicit robot interaction was chosen: collision-free navigation [Mor96] [Rao96] [Key97a]. Therefore, the robots are encouraged to explore the environment while avoiding collisions into the walls, obstacles, or other robots [Set97]. Although trivial for traditional robotic computation domains, collision-free navigation provides a non-trivial search space for an evolutionary system, mainly where both robot control and morphology are evolved [Wat99a]. Furthermore, evolutionary methodologies for collective tasks in multi-robot interactive domains have not yet been reported [Lip00]. Hence, the insights that this research will bring about intend to be basic steps towards the developing of evolutionary techniques to a level where they can seriously be considered for designing industrial robots.

The main issues normally addressed by Evolutionary Computation are: *i)* to synthesise automatically more complex behaviours than could be produced by hand [Ang94a] [Bul95]; *ii)* to exploit all of the available features, considering that some of them may even be opaque to the designer [Pol00]; *iii)* to produce the desired behaviour specifying *what* the robot should do, but not *how* the controller works [Tho96b]; *iv)* to show that evolutionary techniques can reduce the human effort required to develop control systems as compared to traditional manual methods [Tho97a]. The main issues to be addressed in this work lie under *ii* and *iii*. Because of simplicity of the task, issues *i* and *iv* do not have significant impact (i.e., the behaviour aimed by the evolutionary controller is relatively simple, and could have been designed by hand with less effort).

1.1 Problem Delimitation

To adapt the proposed solution to the conditions of the Electronic Engineering Laboratories, the theoretical work was evaluated over a set of experiments,

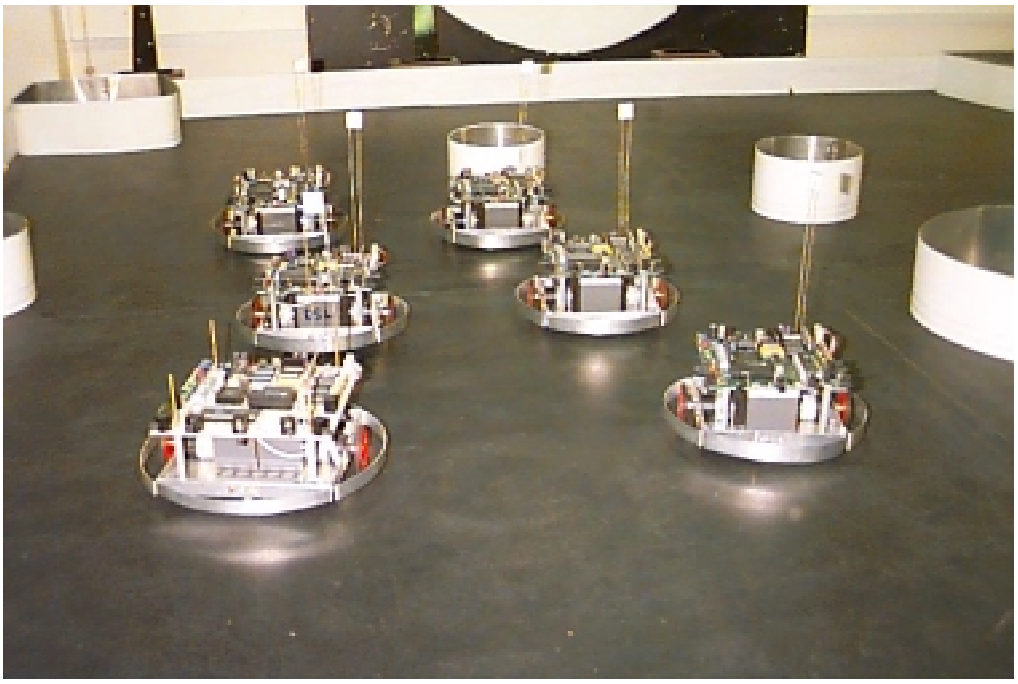


Figure 1.5 – General view of the six robots and their working domain.

performed within a workspace that can be adapted to reproduce simple robotic applications.

- **The Workspace**

The workspace consists of six autonomous mobile robots working in a 2.50m × 2.50m domain, where they can navigate, avoid obstacles, and perform specific tasks. The robots have eight two-bit infrared proximity sensors and 32 speed levels for each motor. Both the robots and their working domain were specially built for the experiments in this work. Figure 1.5 shows the six robots navigating in their working domain. Because the workspace contains various robots, the environment also includes some robot-to-robot interference [Sch96]. Such interference in this work is defined as collisions between robots and reflection of the infrared signals by approaching robots.

All robots have a binary bit string, the "chromosome", containing the genetic code that specifies their control device and physical features, such as speed and position of the sensors [Flo94]. The internal architecture is specified by this genetic code at the control circuit level. This control circuit is implemented within a microprocessor [Nol94] [Mon96], able to be reconfigured to produce new generations of more adapted robots. The robot individual capacity is quite simple, but presents the necessary

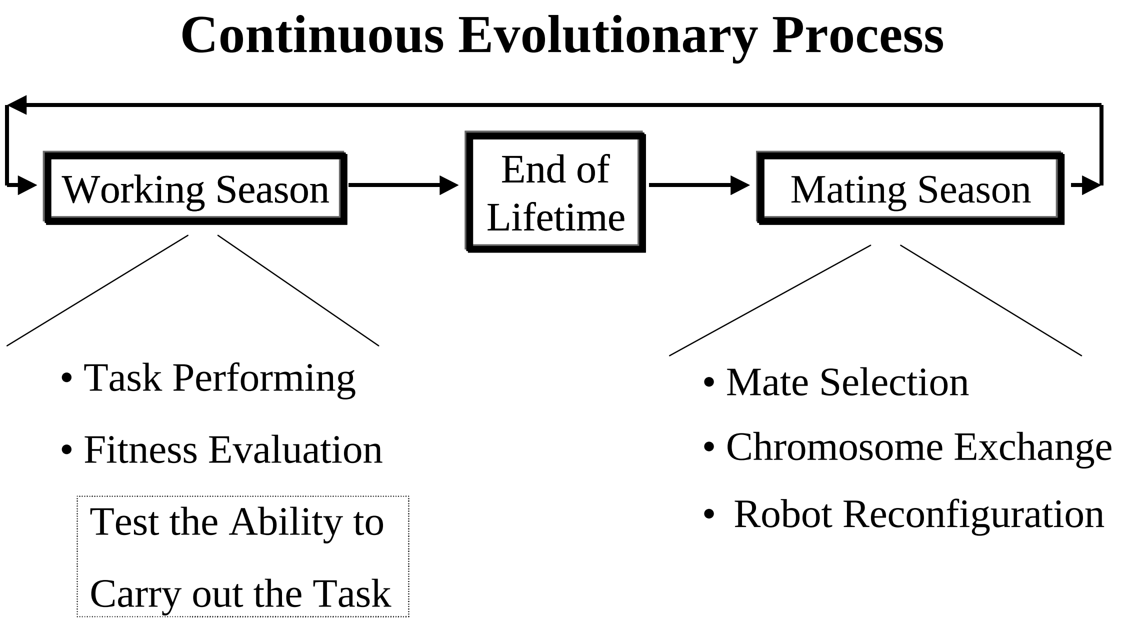


Figure 1.6 – The continuous evolutionary process of the robot population.

evolutionary capabilities. The evolutionary system is not based in an external computer, but is distributed between the robots and coexists with their evolvable controller inside the microprocessor. The distributed embedded evolutionary system of each robot communicates to the others through an embedded radio [Mat98b].

- **The Expected Behaviour**

The robots will navigate and try to perform the obstacle avoidance task totally independent of human intervention or external computation [Fic99]. The robots work in a cyclic procedure, where they have a working phase, where they try to perform the selected task, and a mating phase, where they reproduce. Figure 1.6 shows the continuous evolutionary process where the robots are constantly adapting to changes at the environment. Once a robot completes a working phase, it will attempt to find a “partner” with which to breed. If a partner satisfies the selection criteria, the two robots can exchange genetic code, and transform themselves into different robots (two parents mate and reconfigure themselves as two new offspring) [Bac91] [Bed97]. According to the selection criteria, the robots that are more adapted to perform the specified task will have more chance of generating descendants. Therefore, the best-adapted robots survive and spread their characteristics [Mon93]. After many generations (after some threshold level is achieved), the evolutionary system should produce a majority of well-adapted robots, qualified to work in the environment, and a few unfit robots.

1.2 Objective

The objective of this work is to develop a self-training process, where a group (“population”) of six real robots can be taught to perform specific tasks in a closed mutable environment, completely independent of human intervention, external computation or power supply (see Figure 1.7). The robots should be linked by radio, forming a decentralised evolutionary system. That aim brings substantial technological demands and considerable algorithm detail must be added before it is workable. The robots must be able to adapt constantly their control circuit and morphology to changes of the surroundings, continuously modifying their internal configuration to work properly in that environment. Therefore, the robots must evolve while deployed “in the field”, using the real world to act as “its best model” [Bro91b].

This work studied the main techniques for evolutionary robotics and tested the most relevant ideas that could be employed to produce a fully embedded evolutionary controller to navigate a population of physical robots in real time [Rey95]. Therefore, during the outcome of the work, many different concepts were proposed and investigated while looking for inspiration to develop the neural network architecture, and the evaluation, selection and reproduction strategies (e.g., developing the fitness function, or the crossover and mutation operators). However, some of this research could not be applied to embedding evolution within the robots and only the optimal results were reported for the simple reason of space. Nevertheless, the generated insights and understanding are always present in the discussions and justifications presented in this thesis.

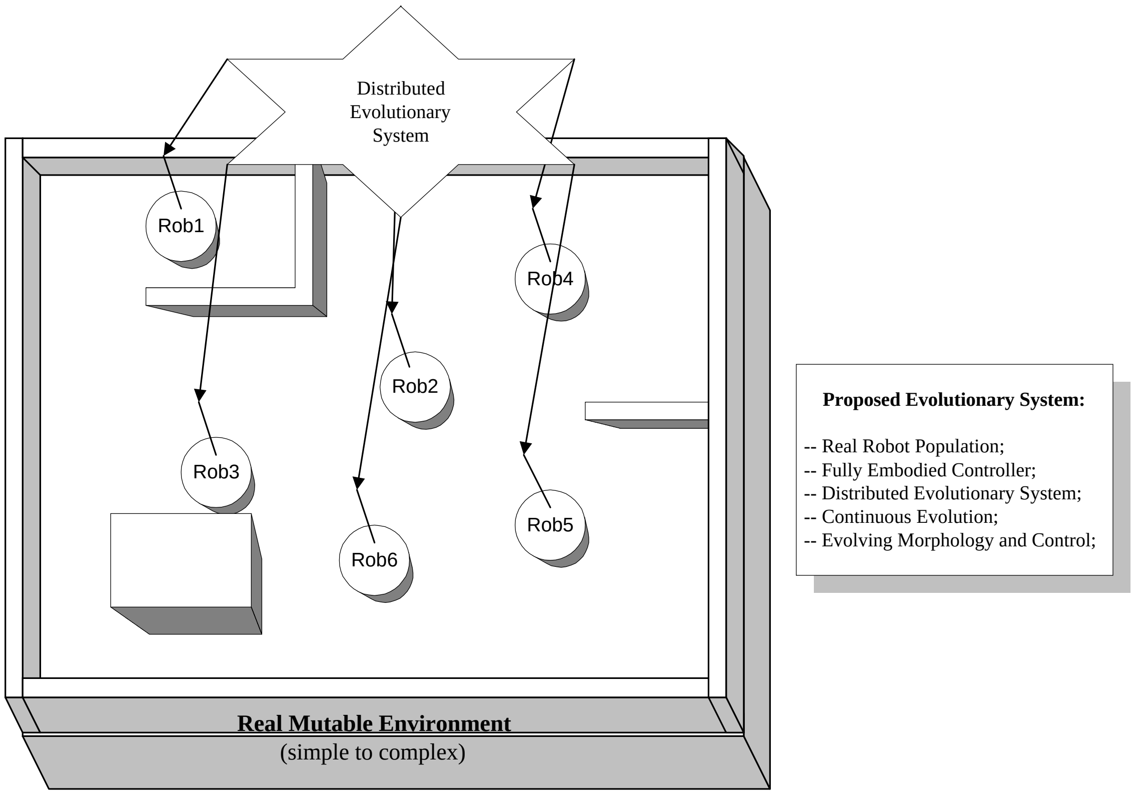

Distributed Evolutionary System
Rob1
Rob2
Rob3
Rob4
Rob5
Rob6
Proposed Evolutionary System:
-- Real Robot Population;
-- Fully Embodied Controller;
-- Distributed Evolutionary System;
-- Continuous Evolution;
-- Evolving Morphology and Control;
Real Mutable Environment
(simple to complex)

Figure 1.7 – The proposed system consists of a distributed evolutionary controller embedded within a population of real robots. All robots are linked by radio, forming a decentralised system.

Goals:

- To implement a workspace with a group of six autonomous mobile robots and the environment containing the structures that are necessary for navigation and task performing, such as walls and obstacles, as well as a monitor computer to produce a data record of the evolutionary experiments;
- To produce a novel distributed fully-embedded evolutionary controller for a population of autonomous mobile robots applied to collision-free navigation in the suggested working environment;
- To show that the proposed evolutionary system can produce not only a trained robot but also an open-ended evolution, continuously adapting the robot controllers to cope with a variable environment;
- To produce the first experimental proofs of the Embedded Evolution concept, where both control circuit and morphology are evolved.

1.3 Thesis Organisation

Chapter 2 is devoted to the study of evolutionary computation applied to robotics. The chapter reviews evolutionary robotics and more specifically genetic algorithms and presents the most relevant strategies that can be applied to the development of evolutionary robotic systems. In addition, the principles underlying evolutionary computation are addressed together with a review of the literature to explore the weak points and show where improvements are required. It overviews the most important topics and concepts that will be referred to in the other chapters, so that it provides vital information to the reader and can be used as a reference of the terms employed in this thesis. The main addressed topics are: artificial life, embedded evolution, incremental evolution, species adaptation genetic algorithms, co-evolution of different behaviours, and co-adaptation of controller and morphology. It also considers the issues on evolving the robot population in simulation and the problems that arise when the final result is transferred to a real robot.

Chapter 3 reports comparative studies between different alternatives to embedding an evolutionary control circuit in a population of autonomous mobile robots that is able to achieve the task of collision-free navigation. The chapter debates if the evolvable control circuit should be structured or not; if it should be simulated or integrated; and if it should be trained at first and then just refined by evolution. It discusses the main architectures that can be applied to the design of a navigation control circuit that can be evolved. It also attempts to apply different strategies to implement the embedded controller according to the project specifications and limitations and provides a comparative analysis of these strategies that selected a weightless neural network as the best option. The reviewed strategies are: evolvable hardware; dynamic state machine; condition-behaviour mapping; pulse stream neural systems; and Boolean neural networks.

Chapter 4 introduces the formulation of the proposed evolutionary system and what is required to allow its implementation in the robots. The chapter describes how the evolutionary system was developed, the techniques that could be adapted from the literature to be applied to the proposed system, and the novel techniques developed to allow the design of every circuit and program necessary to embedding a distributed evolutionary system in a population of autonomous mobile robots. It presents the robots and the environment where the experiments were carried out and specifies what is necessary to perform the evolution of the population towards the desired task-behaviour: collision-free navigation. It also defines the employed genetic operators such as crossover, mutation, partner selection, and reproduction. The architecture of the chosen Boolean neural network is developed and described as well as the strategy of communication that allows the distributed evolutionary system to control the evolving robots.

Chapter 5 describes the robot hardware and software, showing how a population of autonomous mobile robots was built to accommodate the distributed evolutionary system as they were specified in Chapter 4. The chapter describes the robot architecture, explaining how its internal circuits were conceived and work. It discusses the alternatives to build the computing system, the sensor circuit, the motor drive circuit, and the communication circuit, which allow the robots to communicate and coordinate their activities. It also describes a stand-alone radio board that was developed to allow a monitor computer to supervise the evolutionary experiments and produce a data record containing all the relevant information for each generation.

Chapter 6 presents the preliminary experiments performed with the real robots that helped to develop and refine, by trial and error, the configuration of the evolutionary system, the navigation control circuit, the sensor module, the motor drive module, and the monitor computer. It tests the influence of different parameters in the performance of the system. The chapter puts to test the ability of the proposed evolutionary system to evolve the part of the robot morphology that corresponds to the sensor configuration. It determines the best sensor configuration that could be used as a reference to the other experiments. It also investigates if the evolutionary system is able to evolve an unstructured controller. A different selection strategy, called inheritance selection, which chooses the robots to reproduce by calculating their average performance in the previous generations is also proposed in Chapter 6.

Chapter 7 presents the experiments run in simulation that tested the influence of different parameters of the evolutionary system, such as different mutation rates, strategies to select the partners to breed, crossover strategies, and reproduction techniques. Simulation allowed an efficient and fast way to test novel strategies that improved considerably the performance of the evolutionary system. The chapter introduces the developed simulator program and how it was used to evaluate the robots in a virtual world. It investigates different ways to evolve the robot configuring bit string and proposed the use of *predation* to bring new genetic material to the population, introducing more diversity and preventing it from being stuck in local optima.

Chapter 8 brings together all the insights and understanding achieved in the previous two chapters to build a fully embedded distributed evolutionary system that evolves both the robot navigation control circuit and its morphology and is able to achieve collision free-navigation in a few hundreds of trials. The chapter presents the first experimental proofs of the embedded evolution concept applied to the evolution of the morphology and control circuit of a population of real robots in real time. It attempts to evolve the configuration of the sensors around the robot and the speed levels of the motors. It also compares the performances of an unstructured controller and a structured neural network.

Finally, Chapter 9 presents the conclusions and the contributions of this research. The outlined scope and assumptions of this work along with its limitations are compared to the obtained results to determine if it succeeded in producing a powerful embedded evolutionary system that is able to achieve obstacle avoidance. The chapter shows not only what was achieved, but also what was learned while executing the research and offers some possible opportunities for further investigation to extend the scope of the work reported in this thesis.

Appendix A describes the contents of the attached CD-ROM, which contains the software developed in this work, as well as some photographs and videos of the robots performing their task. Appendix B contains the schematic diagrams of the electronic circuit of the robots and the radio board. Finally, Appendix C contains some tables summarising the many experiments that for reason of space could not be included in the body of the thesis.

2 EVOLUTIONARY ROBOTICS

This chapter presents methods of applying Evolutionary Computation (EC) to design controllers automatically for autonomous mobile robots. It introduces EC and overviews the state-of-the-art in this area. It also presents several different subdomains of this research field, such as Genetic Algorithms (GAs), Genetic Programming (GP), Evolutionary Systems (ES), and Artificial Life (ALife). This chapter will also expand the Evolutionary System domain into several interesting techniques that can be applied to improve the evolution of controllers for the robots. Such techniques are: co-evolution; co-adaptation; incremental evolution; and species adaptation genetic algorithms.

2.1 Evolutionary Computation

Evolutionary Algorithms (EAs) are an emergent computing strategy because of their ability to solve difficult optimisation problems and their versatility in applications involving the machine-learning field [Tom95]. EA methodologies have been applied not only in academic research, but also in industrial problems [Cal95]. EAs are adaptation, search, and optimisation procedures inspired by biology, more specifically in the Darwinian Theory of Evolution [Kni48] [Cam99]. They abstract and mimic some of the traits of natural evolution to produce functional artificial adaptive processes [Flo97a]. Though evolution-inspired, EA is free to use whatever strategy works well for a given class of problems, even if it has no direct biological counterpart. EA is particularly suited to a physical robotic system because, even though it is frequently seen as a sequential execution of genetic routines in simulation, it is intrinsically parallel, and can control a *population* of individuals (robots) that work simultaneously in spatially-extended domains.

Evolutionary Computation is a research field that involves nature-inspired techniques such as Genetic Algorithms and Genetic Programming, which are explained below.

2.1.1 Genetic Algorithms

Genetic Algorithms (GAs) were introduced in the sixties when the fundamental biological mechanisms were abstracted away and expressed mathematically in a form that can be simulated on a computer and applied to a wide range of problems [Hol62]. GAs are search engines applied to large search spaces of well-defined finite dimensionality [Shi00]. GAs consist of mapping feasible solutions in some problem space to *individuals* throughout a string of binary digits, also called the *chromosome*. The characteristics or features of each feasible solution are also called the *phenotype*, which is represented by a binary code, called the *genotype*. Therefore, each individual, through a suitable coding, represents a point (a feasible solution) in the search space of a given problem.

A GA is an iterative procedure where a *population* of individuals is randomly generated [Mit95]. Each individual of the population is evaluated in each iteration step, also called a *generation*, and given a score, or *fitness value*, that relates to its performance in solving the target problem [Tom95]. The following generations are formed by evolution so that, in time, the population comes to consist of better (fitter) individuals. Each subsequent generation is created, traditionally, by applying two genetic operators, known in biology as *crossover* and *mutation*, to a selected group of individuals from the previous generation [Wat00b]. In a fitness-proportional selection [May96], the individuals are selected with a probability proportional to their relative fitness, so that good individuals have a greater chance of being selected to *reproduce*. The parents reproduce, creating a resultant individual consisting of a mixture of the parental genetic material (*crossover*) along with a small amount of copying errors (*mutation*). Reproduction may be *sexual*, when crossover between two parents occurs, or *asexual*, in which case just mutation is used to produce offspring from a single parent [Bro00].

The crossover operator usually works by exchanging substrings of the parent chromosomes after a randomly-selected crossover point. The mutation operator adds background noise to the process to prevent premature convergence into local optima. Mutation is applied to the chromosome by flipping bits at random with a certain probability called the *mutation rate* [Ang94b]. If the natural metaphor is to be followed, a given generation consists of different creatures (individuals) whose chances of survival are in relation to their fitness. The better (fitter) a creature, the higher its probability of survival, and in time, the population will be comprised of better creatures. However, GAs are stochastic and convergence is not guaranteed [Koz98].

There are several possibilities to select individuals for reproduction as a function of their fitness [Bar98]. Fitness-proportional selection is one of the best-known methods [Fun98]. Once the fitness *fi* of each individual *i* in a given generation is obtained, it is possible to calculate the total population fitness *TF*:

$$TF = \sum_{i=1}^{Popsize} fi$$

Then, a probability *pi* is assigned to each individual as follows:

$$pi = \frac{fi}{TF}$$

Finally, a cumulative probability *Cpi* is obtained for each individual by adding up the fitness of the individual to the fitness of the preceding population members:

$$Cpi = \sum_{k=1}^{i} pk, \qquad i = 1, 2, \ldots, Popsize$$

A random number *r* uniformly distributed in [0,1] is generated *Popsize* times and the individual *i* is selected for reproduction each time:

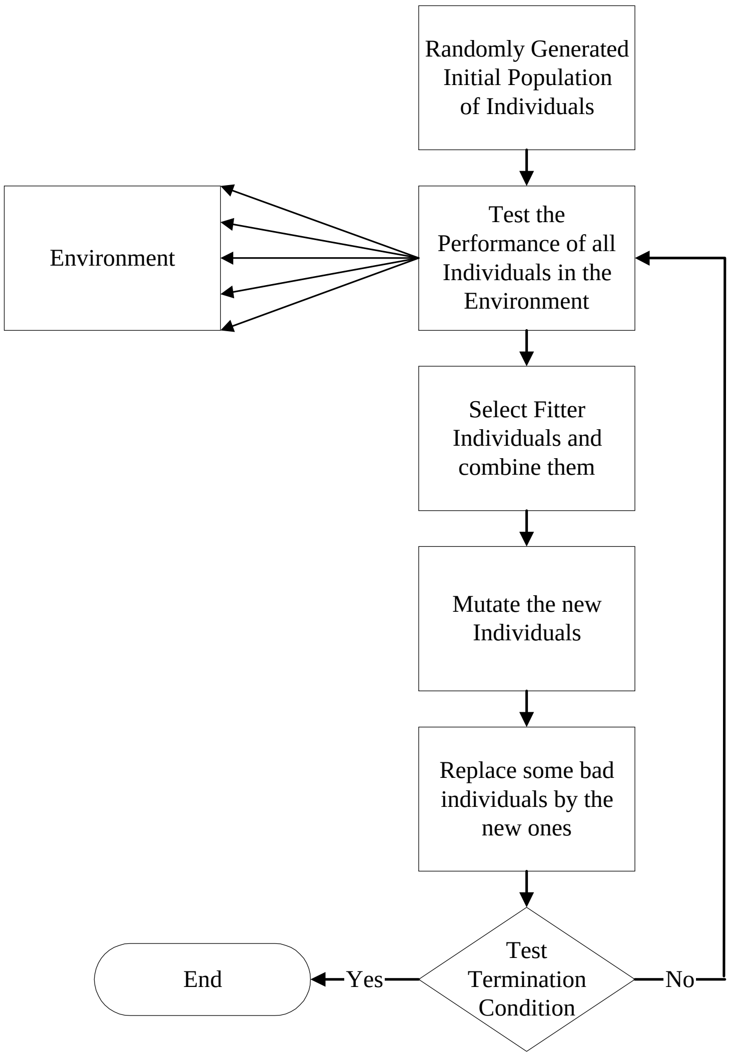
$$Cp(i\text{-}1) < r \leq Cpi$$

If $r < Cpi$, the first individual is selected for reproduction. This method can be compared to the spinning of a biased roulette wheel [Tom95] divided into *Popsize* slots, each with a size proportional to the fitness of the respective individual. If an individual has a high fitness score, it has a high probability of being selected many times and will produce many descendants. One problem with this approach is that, after the population converges and the individuals get better, the differences in fitness between individuals become small, which renders selection ineffective [Tom96]. Other methods that do not allocate trials proportionally to fitness try to solve this problem. Examples of such methods are Ranking selection and Tournament selection.

In Ranking selection, the individuals are ordered by fitness and the best individual is selected a predetermined multiple of the number of times the worst one is selected [Tod97]. In tournament selection, a number of individuals (the tournament size) are selected at random with uniform probability and only the best one among them is selected to reproduce. The selection pressure is proportional to the tournament size [Har93a]. This method is not global so that local tournaments can be held simultaneously in spatially isolated populations.

Figure 2.1 illustrates how a standard GA works. The population is randomly initialised and tested in the environment. The best individuals reproduce to create the next generation, which will be tested again. The process only stops when a termination condition is satisfied [Set98a]. Therefore, evolution continues until a satisfying solution is found. In some cases, there is an implicit fitness evaluation as the individuals battle for virtual resources necessary for survival. In other cases, an explicit fitness function is applied to each generation of creatures, forcing evolution in a desired direction [Bro91a] [Bro92].

In classical GA, crossover is the fundamental operator and mutation is seen as a supplementary function [Tom96]. Crossover combines parents selected from beneficial trials, thus combining strings that have already been proven relatively good, and so having more chance of generating fitter individuals [Bro00]. Selection and crossover alone tend to cause rapid convergence, with the danger of losing potentially useful genetic material. The original version of the crossover operation is *one-point crossover*, where, once two bit strings have been selected to produce offspring, a position *p* is selected at random between one and the length of the strings minus one [Tom95]. Both strings are then divided at this position, generating four substrings: *substring1a*; *substring1b*; *substring2a*; and



Randomly Generated Initial Population of Individuals
Environment
Test the Performance of all Individuals in the Environment
Select Fitter Individuals and combine them
Mutate the new Individuals
Replace some bad individuals by the new ones
End
Yes
Test Termination Condition
No

Figure 2.1 – General scheme of a standard genetic algorithm.

substring2b. Two new strings are produced by swapping all bits of these substrings: offspring1 = *substring1a* + *substring2b*; offspring2 = *substring1b* + *substring2a* (here "+" means concatenation). These two new offspring will enter the new population for the next generation in place of their parents. This strategy was inspired by biology and is very simple and efficient [Bro00]. The idea was expanded into *multi-point crossover*. It applies the same strategy, but divides the original strings in more than one cut point and the substrings are swapped among these points.

Another strategy is *uniform crossover*, which forms two offspring from two parent bit strings. For each bit in the first offspring, a bit in the corresponding position is copied randomly with some probability from one of the parents. The second offspring gets the corresponding bit from the remaining parent. This strategy is less likely to preserve good building blocks, but provides a more uniform distribution of the bits [Wat00b].

Mutation is necessary to avoid search stagnation by introducing new diversity. Without mutation, evolution can only combine the individuals of the population to form the next generations. As only the best individuals are selected to reproduce, the population tends to converge to a single better configuration. Therefore, after some generations, the configuring bit strings of the population will be all the same and the population will cease to improve, since it cannot be modified anymore. Mutation introduces new diversity by randomly flipping some bits of the chromosome every time the individuals reproduce. Hence, modifying their configuring bit string and allowing evolution to proceed. Nonetheless, mutation rates need to be low, otherwise the search tends to degenerate into a random walk [Har96].

Generally, applying mutation to a fit individual often produces negative results, deteriorating its fitness, and is only occasionally positive [Tho94c]. *Neutral* mutations are the ones that modify the genotype, but leave the phenotype unchanged (and fitness) [Bar98]. Nevertheless, they open the possibility of a further mutation making some difference. In this case, the mapping from genotype to phenotype contains redundancy such that a phenotype is represented by many genotypes [Shi00]. This allows genetic changes to be made while maintaining the current phenotype and thus may reduce the chance of becoming trapped in sub-optimal regions of genotype space.

Adaptive mutation schemes vary the rate or the form of mutation, or both, during the GA run [Tho97b]. Therefore, the search space can be explored uniformly at first and more locally towards the end, locally improving the best candidates. A small mutation rate will tend to converge the population into a local optima [Har93a]. If the mutation rate increases, the population can spread around this local optima to search the neighbourhood towards ridges of new hills, but if the mutation rate becomes too big, then the fitness of the population may disperse completely. Usually, GAs are not strong enough to demonstrate conclusively that the global optimum has been reached [Cli97] [Har97a].

A variant of GA, called Genetic Programming (GP), was proposed by J. R. Koza [Koz98]. While in a GA evolution takes place at the genotypic level (i.e., at the level of coding sequences), GP emphasises phenotypic adaptation (i.e., the behavioural expression of a genotype in a specific environment). GP is an attractive approach to evolving controllers in simulation since it manipulates higher-level primitives, such as Lisp programs, for example. This abstraction leads to a reduction of the search space, making evolution work much faster [Ban93].

Since it was proposed, GP began to conquer some space from GA applications. De Garis applied GP to configure neural networks [Gar94a]; and Sen applied GP to multi-robot systems [Hay95]; Later, Koza and Rice were successful in evolving box-pushing behaviour with the same approach, but still in simulation [Koz92]. Nordin applied GP to evolve a real robot controller [Nor96] and Koza *et al.* also employed FPGAs to accelerate the evolutionary process [Koz97]. However, the effectiveness of these approaches is dependent on the proper design of the behaviour primitives.

As the complexity of applications increases, the foresight needed to design it by hand will soon be outstripped, making progress beyond relatively simple domains infeasible [Cli92]. Therefore, the possibility of an automatic synthesis without explicit design offered by evolution becomes very attractive [Har93a]. Natural evolution is the existence proof for the viability of this approach, given that relatively complex designs like the human eye were produced in this way [Kep94] [Tod97].

2.2 Evolutionary Robotic Systems

Since John von Neumann, in the early 1950s, posed the question "*Can a machine reproduce?*", many researchers have attempted to investigate the logic necessary for reproduction. In analogy with nature, this was an attempt to allow an artificial machine to create a copy of itself, which in turn could create more copies [Mic95] [Mey97]. Ever since, many attempts to create such self-replicating machines succeeded in computer simulations. A good example is Chris Langton's *Cellular Automaton* [Maz98]. The effort culminated when Langton and his colleagues defined the term Artificial Life (ALife) [Lan89]. They attempted to abstract the fundamental dynamical principles underlying biological phenomena, and recreate these dynamics in other physical media [Sip95] [Dau98]. Differently from biological research, which is essentially analytic, ALife is synthetic, attempting to construct phenomena from their elemental units [Mic95] [Mit95]. The reproductive process proposed by von Neumann is as real as that carried out in nature [Bro92] [Ste95].

ALife studies not only improve the understanding of nature, but also bring insights into artificial models such as Evolutionary Systems (ES). ES offers the ability of adaptation to a dynamic environment [Nol94]. Therefore, ES offers greater adaptability to deal with unforeseen events than traditional design. Brooks combined ALife and ES methods to design robots that had the necessary adaptability to function in a human environment [Bro91a] [Bro91b]. These methods work differently from the top-down methodology of traditional Artificial Intelligence (AI) [Ste95]. AI starts by identifying a complex behaviour and then tries to build a system that presents all the details of the behaviour [Neb96]. ALife starts from simple elemental units, gradually building its way upwards through evolution, in a bottom-up manner [Sip95]. Whereas AI has traditionally concentrated on complex human functions, ALife concentrates on basic natural behaviours, emphasizing survivability in complex environments [Dau98].

In a recent article [Lip00], Lipson and Pollack attempted to bring computer models closer to physical reality. They described a system that evolves locomotive machines in simulation, but can also employ a rapid-prototyping device to build them automatically. They claimed to have achieved automatic reproduction of robotic life forms. However, their work consists of a simulated evolutionary process, where some individuals can be automatically prototyped. As evolution goes on in the computer, there is no connection to the physical world, such as fitness evaluation. What their work really innovates, therefore, is the ability of automatically prototyping some individuals to display physically what is going on in simulation. Their robotic population does not exist or reproduce in the real world. In evolutionary terms, there is no difference from their previous work [Fun99], where they implement the structures by hand out of real Lego blocks. However, they have only used this to show the robustness of the structures generated by evolution in the simulation. Hence, there is some way to go before self-reproducing robots can exist in the real world [Bro00].

What a truly evolutionary robot system needs is a strategy that allows the “robot creatures” to self-replicate [Kan97]. The only feasible solution so far is to build the next generation by hand, using some modular blocks that can be attached to the bodies of the robots, enabling them to modify their configuration from one generation to the next [Tem95]. These blocks can contain different sensors, motors, or actuators, so that robots containing different features, or characteristics, can be produced [Har93c] [Har94]. Figure 2.2 illustrates how different blocks containing actuators, sensors, and motors with different capacities can be mounted on the robot body. However, there is an alternative to automate

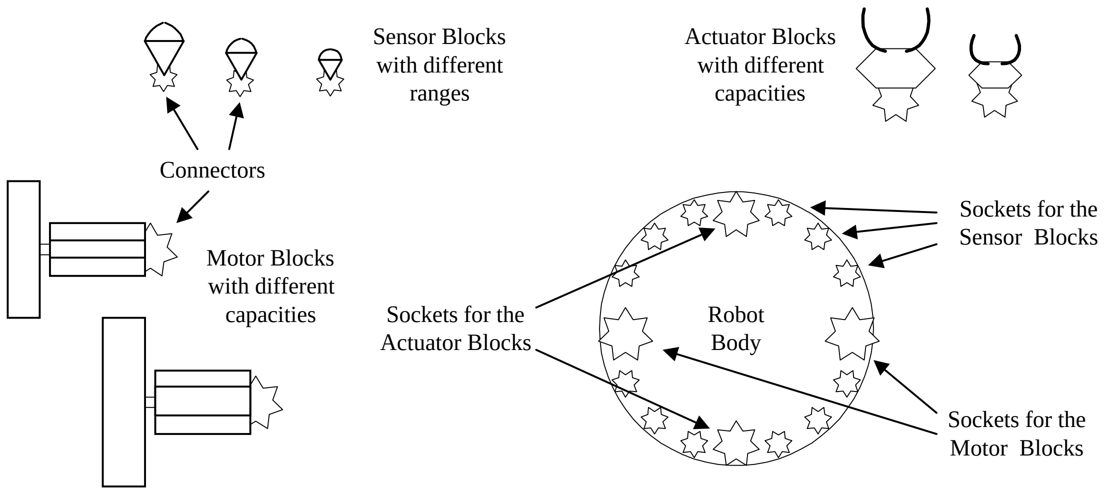

Sensor Blocks with different ranges
Actuator Blocks with different capacities
Connectors
Motor Blocks with different capacities
Sockets for the Sensor Blocks
Sockets for the Actuator Blocks
Robot Body
Sockets for the Motor Blocks

Figure 2.2 – Robot body containing different sockets used to connect different blocks that contain motors, sensors, and actuators with different capacities.

this strategy. Consider an evolutionary system where different sensor configurations are to be evolved together with the robot controller. If all the available sensors are placed around the robot, and can be enabled or disabled by the genotype, different physical characteristics can be produced, so that the robots can have different sensor configurations. In comparison to natural evolution, where humans lost their tail because it was not practical anymore, this strategy consists of giving the robots many “tails” (redundant sensors) and letting evolution decide which ones to pick [Har97c]. Therefore, new robots with different physical characteristics can be produced without the necessity of rebuilding their mechanical bodies. This strategy is used in this work to produce the evolutionary robotic system that will be described in Chapter 4.

2.2.1 Issues on Using Simulators

When EC is used to design automatically a robot controller, a software simulation of the robot will make it easier to be manipulated by evolution [Har94] [Hus95]. Even when a real robot is used, the actual controller undergoing evolution is usually simulated in software [Tho97a]. When developing EAs using simulation, to achieve realistic results the simulation must take account of the imperfection of the real world, such as noise, and non-ideal performance of sensors [Jak95]. However, a significant drop in performance may occur when the results of the algorithm optimisation are loaded into the robot [Jak96] [Jak97a]. Another approach is to use the real robots themselves [Jak97b] [Jak98b]. However, the problem with this approach is the prohibitively long time taken, as demonstrated by experiments on Khepera robots [Nor97]. Nevertheless, the experiments in Chapter 8 will show that fully-embedded evolution of simple behaviours can succeed in less than two hours, because it allows the evaluation of the whole robot population in parallel [Fic99] [Sim99] [Wat99a].

Most of the reported work with evolutionary systems makes use of computer simulations of the robots [Cli96] [Cha98]. Some authors used faithful representations of the environment, including some effects of noise [Nol94] [Jak95] [Mat95]. However, many concluded that if noise levels in the simulator differ significantly from those in the real robots, the obtained solutions are less likely to be transferred to the real system [Mat96]. Brooks advertised in [Bro92] that when building a simulator it would be very difficult to estimate the uncertainty that the real world presents. In addition, there is a danger of confusing the global world-view, where sensors, for example, return perfect information, with the robot view of the world. Hence, it is likely that the evolved controllers will rely on unrealistically accurate actuators and sensor responses. Any abstraction made in a simulation may be exploited by the EA and result in behaviour that is badly adapted to the real world.

The simulators are limited by the experience of the programmer on the relevant factors to be included [Cli92]. That is, the simulation will only include the levels of noise that the programmer expects to happen. For complex robots, a faithful simulation may take more time than trials on real robot architectures [Mig95]. According to Harvey *et al.* in [Har94], producing sufficiently accurate simulations of visual sensing can be prohibitively long. Thus, when applying simulated evolution to robot vision, using a real video input system proved a more attractive option [Bro91b].

There can be no substitute for experience with real robots, but simulations, though, do have some benefit if the appropriate care is taken [Cli96] [Smi98]. In an open-ended evolution, for example, simulation can be used to speed up the process by training basic skills to simulated robots in an abstracted environment, and then the best solutions can be transferred to the robots to be refined later, while the robot continues evolving in the real world. This strategy may accelerate evolution, even though performances may significantly deteriorate when the best simulation solutions are transferred to the physical robots [Nol94] [Nol95].

Another good use of simulators is in abstracting complexity from the robots and environment [Mig95]. This strategy will be used in the work reported in Chapter 7 to allow different characteristics of the system to be analysed individually in simulation. It is possible in simulation, for example, to isolate the robots from the uncertainty of the real world, and analyse the effects of genetic operations like the crossover and mutation strategies. Therefore, simulation is acknowledged as a valuable tool, used to gain insight and understanding of complex systems. However, the developed simulator was not intended to provide a high-fidelity simulation of the robots and their environment. Instead, it served only as a testbed to set up new experiments.

2.2.2 Embedded Evolution

Floreano and Mondada in [Flo96a] took the first steps towards bridging the gap between computer models and physical reality. They used a physical Khepera robot to provide physical sensor readings, tethered to an off-board workstation, hosting the evolutionary software. They were followed by many researches, such as Harvey [Har97b], Jakoby *et al.* [Jak98a], Husbands [Hus98], Mataric *et al.* [Sch96], and Thompson [Tho97a], among others. Nevertheless, all these authors lack parallelism or independence from external computation, or both. Watson et al. [Wat99a] and the preliminary results of this work [Sim99] described the first experiments with the Embedded Evolution (EE) concept.

EE is a methodology for the automatic design of physical robot controllers that takes place in a population of real robots [Wat99a]. It avoids the problems of evolving in simulation and transferring the design to physical robots and speeds up evolution by evaluating each robot in parallel. The evolutionary algorithm is decentralised [Mat97c], distributed among and embedded within the robot population so that there is no need for human intervention to evaluate, breed, reconfigure, or reposition the robots for new trials [Och99a]. Fitness evaluation, partner selection, and reproduction are carried out autonomously and individually by the robots [Sim99]. The robots are not only autonomous in their task behaviour, but in the evolutionary algorithm as well [Fic99].

When designing control systems for physical robots, it is not always clear how a robot control system should be decomposed and the sub-systems often include interactions mediated via the environment [Har97b]. Another major problem is that the interactions among sub-parts tend to grow exponentially as the systems become more complex [Tho97a]. When the complexity of the interactions grows, a primary decomposition into perception, planning, and action becomes difficult to obtain [Pol00]. An alternative to this is to allow evolution to deal with the unexpected interactions between sub-systems and design a controller based on the behaviour of the complete system (i.e., seeing the robot as a whole: body; sensors; motors; and controller) [Flo94].

In EE, evolution is manipulating a physical object that exists in real-time and space, and behaves according to the mechanics of the robots [Tho97a]. To determine the interactions between the robot components, it is crucial to consider their size, shape, and location. This fact makes those interactions richer, but, in some ways, more constrained, since the characteristics of the components and their interactions are not exactly predictable or constant over time. Evolution should be able to exploit freely the collective behaviour of the components without the necessity of predicting it from knowledge of their individual properties [Tho96b]. It can take advantage of the rich structures and dynamical behaviours that are natural to the robot hardware, far beyond the scope of conventional design, which is only a sub-set of the possibilities. Even stochastic noise is not always a problem and can present advantageous effects [Tho94c]. However, it alters the behaviour of the robot hardware, modifying fitness and obscuring the fitness landscape. This effect decreases the difference in fitness between two robots that are neighbours in genotype space and may lead to imprecision in selecting the fittest robot [Set98b]. These effects will be analysed in Chapter 6 where this problem will be approached.

A noticeable disadvantage of EE is that it is difficult to collect experimental data from the robots, since there are no centralised mechanisms for evolution [Wat99a]. EE is also susceptible to failure if the robots become reproductively isolated, which can easily happen in communication systems based on a local-range infrared link [Fic99]. EE adds even more difficulty to the inherent robotic problem of replicating entire experiments [Tho97a]. Noise in the sensors, effectors, and the environment result in a large variance across trials [Mat96]. This uncertainty was already difficult to reproduce in simulation, even by preserving the same random seeds, but it is practically impossible in a physical controller, embedded within a population of real robots. Any stochastic components in the algorithm compound the problem. In simulated evolution, the programmer can manipulate the rules to eliminate non-determinism and produce repeatable trials [Jak96].

Much of the analysis of expected behaviours in EE is provided by the user (i.e., it is qualitative and based on human judgment) [Mat95]. Typically, statistical tests are not significant as insufficient data are available. Because genetic algorithms are stochastic, the average result over several runs is a more useful indicator of their performance [Set97]. However, because of the uncertainty and variability of physical experiments, an average performance is also difficult to establish as trials may vary significantly.

The design of a fitness function in EE is more difficult than in simulation, because not all sensory information taken by simulation (such as the position of the robot in the environment) is actually available from the point of view of the robot in the real world [Nol94]. Towards the end of the evolutionary experiment, the differences between individual fitness become smaller, resulting in a less effective selection [Tom95]. In EE, a possible alternative is to increase the duration of a generation at the end of the experiment, to introduce more selection pressure in the environment and differentiate the better-adapted individuals [Sim99] [Pol00].

2.2.3 Encoding the Characteristics of the Robot

When an evolutionary algorithm is distributed among a population of physical robots, some parameters must be selected to codify their control circuit and embedded characteristics [Har93c]. These parameters are usually represented by a binary string of bits (*alleles*), also called the robot *genotype*. The *chromosome* contains the binary string. Then, the genotype is applied to configure the robot features, also called the robot *phenotype*, in such a way that determined groups of bits (*genes*) in specific locations in the chromosome configure specific features [Nol94]. If the encoding is robust enough, that is, if it presents some redundancy, small modifications (mutation) in the genotype should not produce radical changes in the phenotype [Cam99]. This is a concept referred to in biology as Neutrality, Neutral Mutations, or Neutral Variation [Ler76]. Therefore, some genetic variations observed in populations are probably trivial in their impact on reproductive success. Moreover, even if only one fraction of the extensive variation in a genotype significantly affects the organisms, it is still a reservoir of raw material for evolution.

Neutrality can be obtained by using an artificial neural network (ANN) to implement the robot controller [Bee91] [Nol92]. ANNs have usually an intrinsic redundancy, tolerance to noise, and low degradation with respect to component failure [Ale90]. They typically have a smooth fitness landscape and present good neutrality, allowing some neuron contents or weights to be modified without radically changing their fitness (their response for the same set of stimulus) [Mat96]. Some care must be taken when mapping phenotype to genotype, because the size of the chromosome exponentially affects evolution performance. For a chromosome of length *n*, and *b* possible alleles at each position, the search space is calculated as b^n possibilities. Typically, the bigger the search space, the longer the time that the evolutionary process spends evaluating complex designs to find an appropriate configuration of primitives. Then, successful evolution becomes a practical impossibility [Bac91].

Although crossover is viewed as the fundamental operator in classical GAs [Tom95], in EE, the population size tends to be comparatively very small (e.g., ten individuals or less) [Sim99] [Wat99a]. In these cases, the search can rapidly converge to stagnation in few generations and mutation alone will be the only source of diversity in evolution. Therefore, after an initial phase where crossover boasts evolution to a rapid convergence, EE will rely on selection and more sophisticated versions of mutation as the principal evolutionary operators [Sch95]. In general, a compromise must be found between exploitation of good fitness regions (local improvement by crossover) and further exploration of the search space (use of mutation to avoid missing better regions further away) [Har96] [Cli97].

There are several strategies to encoding the characteristics of the robot architecture. The most relevant ones will be presented below:

- **Incremental Evolution**

Incremental Evolution (IE) was suggested by Harvey in [Har92] [Har93a] and states that rather than randomly initialising the population to evolve controllers to a challenging task, it is better to evolve from a population that has been selected for a similar but less challenging task [Cli92]. IE works with no predetermined number of components to be used in the design [Mee96]. A sequence of increasingly complex tasks is presented, involving the evolution in succession of progressively more complex systems. Brooks' approach is an example of "wiring in" new behaviours one at a time, and waiting until current behaviours are thoroughly debugged before "wiring in" the next one [Bro91b]. Evolutionary robotics typically needs adaptive improvement techniques rather than the optimisation skills of the GAs on a fixed-dimensional search space [Mit95]. The number of components required to produce the expected behaviour may be unknown *a priori*. Incremental evolution will progressively increase the number of components through successively more difficult tasks. This approach takes GAs as adaptive improvers rather than optimisers [Bro92].

- **Species Adaptation Genetic Algorithms**

To allow for a wider variety of possible control architectures, the length of the genotype may be variable [Har93b]. Species Adaptation Genetic Algorithms (SAGA) was proposed by Harvey [Har92] explicitly for dealing with variable-length genotypes. SAGA makes use of incremental evolution starting with a short, simple genotype, which produces elementary behaviour. It lets evolution synthesise the elementary behaviours at first and then progressively increases the genotype size, allowing evolution to synthesise again from this better-adapted population [Mat96]. Therefore, complex individuals can be evolved from simple ones, with associated increase in genotype lengths [Har93a]. These two phases (genotype alteration and synthesis) repeat until a sufficiently complex design is obtained. The variations in the genotype length should be achieved, on the average, without significant impact on fitness [Har94]. In SAGA, crossover between individuals is only allowed for parents within a certain genotype distance from each other. Reproduction can be problematic with genotypes of different length and domain-specific GA programs may be necessary [Tho94c].

- **Robot Symmetry**

To reduce the search space for evolution, the physics of the robot and its controller can have some degree of symmetry [Bro92]. For example, with bilateral symmetry, both sides of the controller and morphology can be specified by the same genome [Mat96]. If the encoding scheme allow some repeatability of modular structures, a genotype that encodes only the type and number of predefined substructures will be much more compact [Ban94].

- **Co-evolution of different behaviours**

The co-evolution of two or more behaviours is a delicate process that has been achieved mostly in simulation [Bul95] [Bla98] [Och99b]. Co-evolution, as for example in the pursuit-evasion context [Flo01], occurs when one kind of pursuing robot has its fitness determined by the current behaviour of another kind of evader, and *vice versa* [Har97b]. GAs typically work on static fitness landscapes. Alternatively, co-evolution works on dynamic fitness landscapes that change over time [Mil94] [Flo98a]. To achieve this, co-evolution makes use of the relationships among the systems that are being evolved in such a way that they depend on each other. Rather than evolving to solve a fixed problem, the organisms are constantly adapting to each other and their surroundings, including food, mates, competitors, and predators. For example, stealth can be the response of a predator for countering a more agile prey, and so on [Flo97a]. Competitive fitness functions that are dependent on the constituents of the population can provide a more robust training environment than independent fitness functions [Nol98a].

Co-evolutionary strategies proved successful alternatives in many works involving simulation, such as parental imprinting, aggressive signalling, sexual selection, and evolved communication. Embedded co-evolution in robotics involves the construction of two or more robotic populations (predators and prey for example) that are interrelated to each other [Bul95]. This adds complexity and cost to the already difficult task of embedding an evolutionary algorithm within physical hardware.

- **Co-adaptation of controller and morphology**

A truly general encoding scheme should smoothly integrate specifications of both controller and morphological features [Mat96]. This scheme may include configuration codes that define and permit co-adaptation of the control architecture and parameters specifying sensory and motor characteristics [Nol98b]. Distinctively from the interactions between competitive behaviours of co-evolutionary algorithms, co-adaptation involves parallel evolution of different features of the same individual [Wat00a]. Therefore, the morphology of a robot, such as sensor distribution and range can be selected by evolution while the controller progressively learns how to deal with the new features [Lun97]. Using the natural metaphor, it is like both nervous system and body shape evolved together in the natural world of a species, as the genetic code can configure the shape of the body as well as the instinct in the brain. In this way, the natural skills that an individual is born with as well as the characteristics of its body are manipulated by evolution from generation to generation, until an efficient combination emerges to solve the target task.

3 Robot Control Circuit

This chapter discusses alternative strategies to the traditional implementation of robotic control systems based on artificial intelligence (AI) and evolutionary computation (EC). It describes some of the main approaches, pointing out the key challenges, unanswered problems, and some promising directions. An analysis of the state-of-the-art will select the best technique for the implementation of an embedded evolvable control scheme for a group of autonomous mobile robots.

3.1 Embodying an Evolvable Control Circuit

From the study of the Evolutionary Robotics field presented in Chapter 2, it was possible to identify the necessary characteristics of a control strategy for the adaptive/evolvable behaviour-based robot controller. The working environment attributes and the number of robots that will carry out the chosen evolutionary scheme were introduced in Chapter 1. Once the environment and the tasks have been determined, the next step before drawing the robot architecture is to define the technique that will implement the robot controller.

The strategy for the implementation of an embedded evolvable controller has to provide the appropriate building blocks for evolution to work with [Mat96]. The Evolutionary System (ES) has to be able to manipulate the controller primitives, adjusting them to perform the desired behaviour. It has to allow the designer (the evolutionary technique) to see the robot as a whole (body, sensors, motors, and “nervous system”), as a dynamic system coupled with a dynamic environment [Lun97]. To do so, there are several possibilities. Consequently, it is necessary to consider some classification and concepts:

- **Should the Control be Structured or not?**

One of the main problems in evolving robotic controllers is how to guarantee the possibility, at any time, of understanding what is really implemented inside the chosen adaptable (evolvable) device [Gru94] [Tho94a]. This can be achieved by using some high level semantic groupings. However, this approach inevitably incorporates the human designer’s prejudices (and limitations) [Har97b]. So the controller primitives manipulated by the evolutionary process should be at the lowest possible level, but still permit some kind of structured arrangement that can be deduced from the circuit [Tho96]. This restriction practically eliminates any unconstrained use of low-level semiconductor physics.

Without restrictions, evolution can go on by taking note of the overall behavioural effect of variations made on real circuits. This is very different from conventional structured design techniques, which proceed by modelling, abstraction, or analysis [Wag95]. These are simplifying constraints normally imposed to make design tractable by humans, a decomposition of a complex system into separate parts of manageable size [Tho94c]. Evolution does not need a prior analysis of the problem and such constraints can be relaxed. If some objective fitness function can be written to tell how well an individual can carry out some specific task, evolution should allow the gradual emergence of a solution for a complex system without explicit design. However, all this freedom generally costs a large number of trials of a real circuit working in a physical environment [Hig94]. In addition, sometimes evolution cannot evolve hardware unsupervised, because of the danger of damaging expensive parts or injuring people [Hig96b]. There is the risk, for example, of short-circuiting outputs, over stimulating drivers and actuators, or repetitive mechanical collisions of movable parts [Iba97] [Key97a]. Adequate simulation, normally used to speed up evolutionary processes, may take much longer than using real hardware, or may not even be feasible as when vision modelling or detailed semiconductor physics are involved [Har97b].

The use of abstract models or structures simplifies design and allows the result of an evolutionary process to be extracted from the circuit and subsequently analysed [Tho99]. However, useful properties of the real hardware that have been abstracted away will not influence the behaviour of the designed control circuit [Tho00]. Evolvable Hardware (EHW) avoids the need of design abstractions and the accompanying constraints, and works by observing variations made into the real hardware [Har96] [Tho97a]. It does not need to consider the spatial structure (modularity) or the temporal structure (synchronisation) of the circuit. Furthermore, there is no need to constrain artificially the dynamics of the configurable hardware being used [Tho98]. Physical electronic circuits can display a broad range of dynamic behaviours, of which discrete time systems, digital systems, and even computational systems are just subsets [Tho96c]. Nevertheless, the evolutionary process can also benefit from a kind of modularity, such that different phenotype characteristics can be improved semi-independently by genetic operations [Wag95]. It must be observed that any kind of modularity that is appropriate to the future identification of the designed circuit could be very different from that which benefits evolution [Kod98].

Although human and evolutionary decomposition may sometimes coincide, evolution, according to Thompson [Tho94b], should ideally "be allowed to explore the full range of possibilities without inheriting limitations from humans". Unconstrained evolution can also integrate into the design an ability to function in the presence of faults [Tho96a]. Therefore, it can produce a working system from faulty parts, integrating fault tolerance and automatic design [Key97c]. To make it possible, it is important that evolution operates at the same level of abstraction as the faults, imposing very low-levels of abstractions to generate a fault tolerant integrated control circuit [Key98].

Certainly, unconstrained evolution has a considerable potential for many applications [Har94] [Tho96c], but as this work concentrates in studying embedded evolution, it is more important to identify and exploit generic structures that can produce the desired phenotype for the expected behaviour [Jak96]. Therefore, it is important to take complexity out from the controller and the task behaviour, concentrating the effort in designing and analysing a distributed embedded system [Key97a] [Key97b]. Hence, a modular higher-level architecture will be selected.

- **Should the Control be Simulated or Integrated?**

To evaluate the fitness of an individual, its genotype is expressed to produce a setting of the evolvable control circuit configuration [Har93b]. In the so called "intrinsic hardware evolution", that data set is then applied to the chip, which instantiates the corresponding circuit, and the behaviour of this real circuit is then evaluated to give a fitness score. In the "extrinsic" case, evolution is carried out using a simulated model, with only the final score being downloaded onto the chip [Cha98]. When using simulations, it is important to decide how realistic the model should be; that is how timing, communication, and interactions among control, sensors, and motors should be handled [Tho97a]. In simulations, evolution cannot proceed by analysing the overall behaviour of the real circuit, and has to manipulate abstract models, as described in Chapter 2.

Since the goal of this work is to produce a population of self-training real robots that can control their own evolution, only the issues associated with intrinsic hardware will be addressed. Hence, the question here is: should this programmable control circuit be integrated into a distinct hardware circuit or implemented in software and executed by the robot on-board microprocessor? If the genetically specified piece of hardware is tested *in situ*, the low-level physics of the hardware can be used, and the components can behave at their natural timescales, without the necessity of global clocking or other design constraints [Chi93]. The embedded evolutionary algorithm will execute within the robot on-board microprocessor. This means that a processor and memory will be already available on the robots and, therefore, implementing the evolvable control in software is the simplest alternative.

- **Should the Control be Trained at First and Then Refined by Evolution?**

The major problems for evolving a robot controller for a specific environment were explained in Chapter 2. It pointed out the risk of choosing an Evolutionary Algorithm (EA) that cannot produce the expected behaviour [Har97b]. In Evolutionary Robotics (ER), a genotype specifies the characteristics of a control system, and depending on the application, it may not be a fixed-dimensional search space of known size and the fitness landscape is not always smooth [May96] [Bar98]. Moreover, the number of components required to produce an appropriate behaviour may be unknown at first, and the number of needed components will increase over time when Incremental Evolution (IE) is applied through successively more difficult tasks [Har93a].

An initial training phase where the control system can be taught to work properly with samples of the most common situations that can be found in the real environment can be run in simulation [Cha98]. This possibility facilitates the specification of the crossover and selection parameters, and the mutation rate, in a fitness-proportional selection [Bar98]. The idea is to generate a training set to be used by the chosen evolutionary technique and use it to produce an initial population of distinct trained control circuits, able to behave adequately in some standard situations. This population can then be downloaded in to the real robots and continue to evolve in the real noisy environment [Cli92]. This solution can be more complex, but should reduce the amount of time that is necessary to achieve a reasonable adapted behaviour. It is also a safe way to evolve real hardware, since the circuit will be pre-defined in simulation where trials involving dangerous situations can be performed. Once a safe, stable configuration is found; evolution can be applied to tune its parameters.

Although applying an initial training phase may sound attractive to embedded evolution, it may bias evolution to early convergence into a local optima and include fallible designer’s prejudice [Mee96]. As the chosen task-behaviour of collision-free navigation is relatively simple and can be safely executed with a population of real autonomous mobile robots, a complete evolution of the global system is preferred. To give evolution more freedom to explore uniformly the search space in the initial phase of the experiment, it is a better idea to start with a randomly initialised population, positioned into random places in the environment [Tho97a].

3.2 Alternatives for the Controller Architecture

The possibilities presented below for implementing the evolvable control circuit of the robots were chosen according to their relevance to the state-of-the-art and the viability of evolving them on hardware. This section reviews some alternatives to implementing the robot embedded controller, presenting insights obtained while considering their viability to this work. This analysis is not a comprehensive study leading to the implementation and test of all alternatives, but a brief review of their relevant characteristics and how they can be adapted to implement an embedded evolutionary controller.

The alternatives considered for implementing the evolutionary controller are:

a) Evolvable Hardware;
b) Dynamic State Machine;
c) Condition-Behaviour Mapping;
d) Pulse Stream Neural System;
e) Boolean Neural Network;

Even though most of the suggested alternatives were inspired by the work of other authors, such as Thompson [Tho94a] [Tho96b] (alternatives a and b), Mataric [Mat97b] (alternative c), Reyneri *et al.* [Rey93] [Chi95] (alternative d), and Simões et al. [Bot96] [Sim97] (alternative e), this section presents insights and new solutions for adapting them to fit the evolutionary controller and the robot application suggested in Chapter 1.

3.2.1 Evolvable Hardware

Evolvable Hardware (EHW) is a recently introduced concept [Tho97a] that allows artificial evolution to manipulate directly the configuration of a silicon chip. Intrinsic EHW deals with evolving electronic circuits in real time, measuring performance directly from the hardware to calculate the fitness value [Hig96b].

A Field-Programmable Gate Array (FPGA) was considered to be evolved by the genetic system to perform the robot control. In an FPGA, the functioning of the chip is not determined in the factory, but in the field by the users [Lay99b]. The use of a real chip has some problems that must be considered: the limit on the number of times it could be reconfigured; the ability to reconfigure while basic operations are still being performed; the presence of a layer of secret proprietary software between the user and the chip; the possibility of easily damaging the device with an invalid configuration; and the dependency of the internal components on the temperature and singular physical properties of the chip

[Hig94]. It is also possible to build custom configurable hardware systems out of separate components [Tho95], but each architecture will have its own characteristics.

There are many interesting possibilities arising from the use of evolvable techniques together with FPGAs [Tho96c], such as fault-tolerant design or design under low power/area constraints. If the evolvable algorithm uses no modelling or abstraction, then the physical properties of the chip can be explored to generate the desired behaviour. This approach also allows physical faults to be used as positive characteristics (phenotype) to be evolved into a useful controller [Tho94a]. However, evolving circuits in lower levels of abstraction that could work over the full range of conditions (both external, like temperature, and internal, like the properties peculiar to individual chips) which they could encounter in an industrial application is still a difficult problem with the present scientific state-of-the-art [Gar97]. Another problem is interfacing the FPGA to an IBM PC-compatible ISA bus or a serial port, to provide a communication link between the robot main processor and the programmable controller; hence, the configuration bits can be downloaded to the FPGA. These problems are specific to the application and depend on the FPGA being used. Xilinx is aware of these problems and produced the XC6200 Series that overcame many of these difficulties [Xil96]. Before this series, FPGA architectures were proprietary and researchers could not manipulate their internal structures, nor understand the meaning of their configuring bit string. The architecture of the XC6200 is open and the configuration bits can be written in small sections that change local circuitry.

To guarantee that each generated configuration will have the same phenotype on all controller chips, one simple solution is the use of a clock signal for synchronisation [Tho97a]. This temporal co-ordination (phase control) can also be manipulated by evolution and be determined in real time according to the problem at hand. Such strategy can add a powerful new dimension to electronic systems: *time*. Then, the physical/analogue characteristics of the chip under evolutionary control will not alter the phenotype of the controller. However, they are not useful to the algorithm anymore [Tho00].

At the initial phase of this work, where FPGAs were being investigated to implement the robot controller, real time evolution of a physical FPGA was only beginning to be proved experimentally by Thompson [Tho94c] [Tho96c]. To simplify the robot architecture, a software implementation of an array of logic gates can be run by the robot on-board microprocessor, instead of using a real FPGA. This approach reduces complexity and cost and allows a larger population to be built. It also permits to concentrate in embedding the evolutionary strategy, instead of dealing with problems associated with the controller real hardware physics. With this technique, many problems pointed out above (physical properties, interfacing...) do not need to be considered anymore.

For a software implementation of an evolvable hardware controller, an initial solution consists of an array of 256 logic cells, where each cell has three inputs and its output can produce all Boolean functions of three inputs. Figure 3.1 represents how the output of each cell can be connected through reconfigurable switches to any input of its four neighbouring cells only, to form a two-dimensional array. This limitation is necessary to prevent short circuits among the outputs of the cells if this strategy is to be transferred to a real FPGA in future work. Consequently, the variety of wires for the interconnection of the logic cells, so common in most FPGAs [Tho96c], had to be suppressed. A configuration memory holds the genotype containing the settings (two bits) of the switches that control the routing of the cell inputs and outputs, and a set of eight bits for each cell that compose the performed Boolean function.

The contents of the configuration memory are then codified on to a genotype and an evolutionary algorithm can automatically evolve these bits to obtain a robot control circuit to achieve a given task. Therefore, a genotype of 2560 bits (256×10 bits per cell) can encode the functions and the interconnection patterns of the 256 logic cells. Figure 3.2 shows an example of how the logic cell array can be connected to eight 2-bit proximity sensors to calculate the corresponding speed levels for two motors. The population of robot controllers (depending on the number of available robots) is initialised randomly. For each generation, every genotype is used to configure the software implementation of the logic cell array that coexists with the evolutionary algorithm within the robot microprocessor. The architectures presenting the best performances are selected and their configuration bit strings are combined to produce daughter architectures (resulting configuration strings) that are used to reconfigure the robot control circuit before starting another evaluation phase.

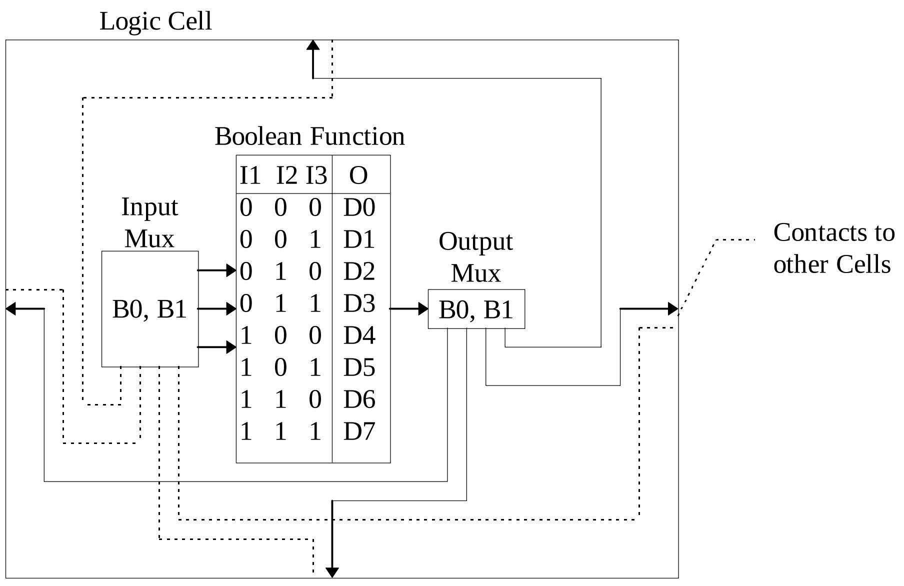


Figure 3.1 – The programmable logic cell. The settings of the switches that control the cell are specified by 10 bits; Bits *B0* and *B1* control the input/output multiplexers and bits *D0* - *D7* specify the desired Boolean function.

Once problems related to intrinsic EHW are solved, such as how to automate the crossover process having a commercial software between the FPGA and the supervisor system, a real FPGA can be used in the robots with few modifications in the evolutionary system. Because of this, the evolutionary software can evaluate, select, and reproduce the individuals, which are FPGA configurations, but can only generate an output file that needs to be manually loaded into the commercial software that burns the FPGA circuit. The commercial software cannot be automatically controlled by the evolutionary system, imposing human interference. Each generated circuit (phenotype) can be previously evaluated, to check if there is a path between the controller inputs and outputs, so that trials will not be wasted with infeasible solutions [Tho96c]. The parent genotypes will be combined until a group of adequate individuals is generated.

An evolvable hardware as the presented FPGA is an alternative to implement the robot evolutionary controller, but some problems can be anticipated: this is a complex architecture that is difficult to simulate and demands computation power that a simple microcontroller will have problems to provide [Mot96]; the number of components (cells, interconnections) necessary to produce the robot task behaviour is still unknown, so the architecture may not be able to achieve a good performance; despite the modularity of the array, the connectivity of the cells depends on their position in the array, making it

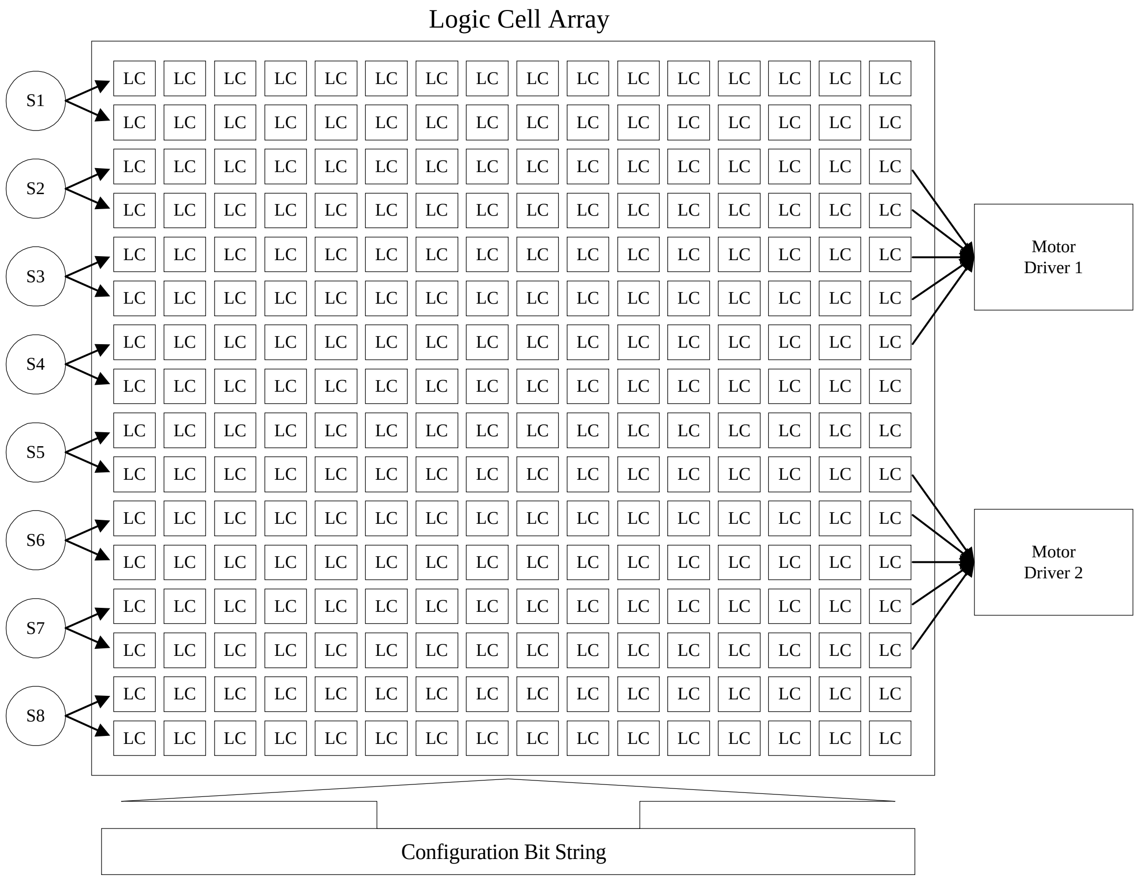


Figure 3.2 – The evolvable hardware controller: a 256 logic-cell (*LC*) array. Eight 2-bit proximity sensors (*S1* to *S8*) connect to specific cells on the first column. Two groups with five specific cells each produce the output signal to the motor drivers (the 5-bit outputs allow 32 speed levels to each motor). All 256 logic cells are configured with 2560 bits (10 bits per cell) stored into the *configuration bit string*.

difficult to add more cells to expand functionality; and in spite of being inspired in a real FPGA architecture, their similarities are not enough to guarantee a direct conversion to a real circuit [Per96b]. Therefore, the use of EHW was discarded.

3.2.2 Dynamic State Machine

Another alternative to the implementation of the robot evolvable controller is the use of a random-access memory (RAM) to implement a dynamic state machine (DSM) [Tho95], instead of the well-known “direct-addressed ROM” implementation of a

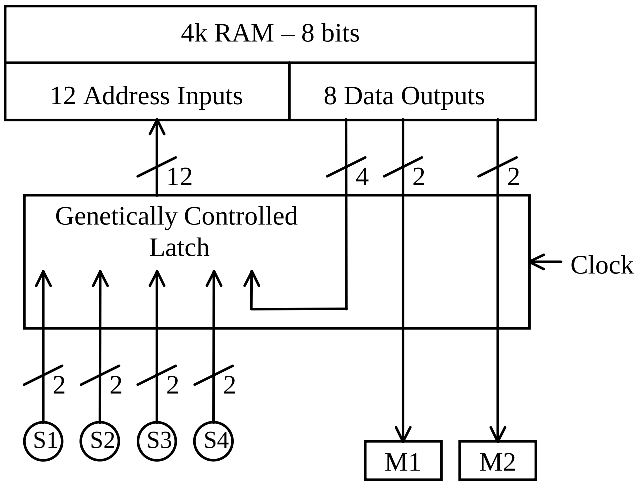


Figure 3.3 – The evolvable Dynamic State Machine: *S1* to *S4* are 2-bit infrared proximity sensors; and *M1* and *M2* are the motor driver modules. This solution was inspired by [Tho97a].

finite state machine (FSM). It can be obtained by placing the contents of the RAM under evolutionary control [Tho99]. Each signal can be latched according to the clock or feed directly the memory address inputs. Figure 3.3 shows a DSM applied to control a robot, where the sensor signals are converted to digital inputs and held by a clocked register as the current state for a central DSM control with evolvable contents. This configuration was inspired by the solution proposed by Thompson in [Tho97a]. The memory outputs contain the speed commands to drive the motors in the next state. Each present-state and input combination form the address that select a specific output. These binary codes are then converted by the motor drive modules *M1* and *M2*, and used to control both motors.

If a fixed clock is applied to the system, the sensory/control/motor functional decomposition cannot be removed. Therefore, evolution cannot manipulate the dynamics of the signals and environment to generate the control system [Tho95]. However, it makes possible that complex digital sensors and motor drivers are used to allow well-elaborated tasks. Nevertheless, the system still produces a rich range of possible dynamic behaviours. The evolutionary system can have control of the latch and decides which of the input lines are clocked, and which are free running. Therefore, it can decide from which lines it will keep a trace of previous stimuli and actions. If no clock is applied, it will not be possible to simulate the control machine in software, since the effects of the asynchronous variables and their interaction with the clocked ones depend upon physical properties of the hardware.

The size of the RAM as well as the number of state variables depend on the characteristics of the sensors and motors used in the robot and the expected behaviour. A 4Kbyte memory with four state variables can be applied to control a two-wheeled robot with four infrared sensors. More state variables can be introduced incrementally as the difficulty of the task is increased [Mee96]. The genetically controlled latch needs 12 more bits to select which address input will be clocked. The contents of the RAM and the clocked/unclocked condition of each variable are directly encoded onto a linear bit-string genotype (32,768 + 12 bits) [Wat99b]. This very simple controller can drive the robot around based in the information supplied by its four proximity sensors. Nevertheless, it constitutes an enormous search space (2^{32780} possibilities) and the evolution towards a good solution may take too long and seems impracticable [Shi00]. Even making use of incremental evolution to reduce the search space, the dimensionality of this approach may be untreatable in a real time embedded evolutionary system [Har93c]. Considering that a more complex robot architecture containing eight proximity sensors and motor drivers with 32 speed levels was proposed, the use of this approach to implement the control circuit would get even more complex and had to be discarded.

3.2.3 Condition-Behaviour Mapping

The “state and action” representation of the world of a mobile robot can be described in a higher level of condition and behaviour abstraction [Mat97a]. Behaviours are control laws that achieve and/or maintain particular goals. They are designed (or learned) so as to provide the desired outputs while abstracting away the low-level details of control [Mat97b]. Behaviours are triggered by conditions predicated on sensor readings and mapped into a proper subset of the state space, which is necessary and sufficient for activating a particular behaviour [Mat94]. This subset is typically much smaller than the complete robot state space. Behaviours abstract away the details of the low-level sub-controllers driving the robot, while conditions abstract away the details of the robot state space.

The learning strategy consists of finding (evolving) a mapping from conditions to behaviours into some effective procedure for a specified task [Mat96]. Each

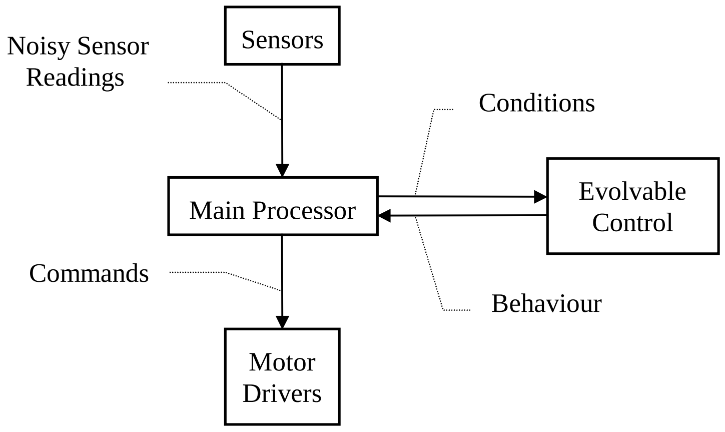


Figure 3.4 – The implementation of an evolvable controller based onto a Condition/Behaviour mapping scheme. The controller asks the main processor about the conditions faced by the robot, and then selects the most adequate behaviour to be executed by the processor. This solution was inspired by [Mat97b].

robot controller can be evolved to select the most appropriate conditions for triggering each specified behaviour. This strategy requires pre-processing and post-processing to deal with the abstraction of noisy sensor readings into an input state and the low-level details of controlling the robot according to the selected behaviour. Therefore, the evolvable controller has to ask the main processor about the conditions faced by the robot, and then selects and tells the processor the most appropriate behaviour to be executed [Mic97]. Figure 3.4 shows a controller inspired by the work presented by Mataric in [Mat97b].

Some examples of possible conditions encountered by the robots are:

- *near wall;*
- *have object;*
- *low battery;*
- *obstacle on the left;*

These conditions are in general easily detected internally by the robot and will trigger appropriate behaviours according to the chosen procedure [Wer96]. Some expected behaviours are:

- *disperse;*
- *go home;*
- *look for recharge;*
- *turn right;*

Table 3.1 – An example of condition-behaviour mapping.

Condition				**Behaviour**			
near wall	***have object***	***low battery***	***obstacle on the left***	***disperse***	***go home***	***look for recharge***	***turn right***
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	0	1
0	1	0	0	0	1	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	0	1
1	0	0	0	1	0	0	0
1	0	0	1	0	0	0	1
1	0	1	0	0	0	1	0
1	0	1	1	0	0	0	1
1	1	0	0	1	0	0	0
1	1	0	1	0	0	0	1
1	1	1	0	0	0	1	0
1	1	1	1	0	0	0	1

The fitness function can be calculated using some reinforcement learning concepts, such as *reward* and *punishment* [Mat93]. Immediate reward and progress estimating functions can be applied to evaluate a robot fitness. It can be analysed externally (e.g., the relative position of the robot to targets, obstacles, other robots…) or internally (e.g., sensor readings, distance to obstacles, collisions, low battery indication…) [Mat97a]. When implementing an embedded evolutionary controller, the fitness function can only be analysed internally. The learning process consists of allowing evolution to adjust the values of a table containing the mapping of each condition to the corresponding behaviour, as it is exemplified in Table 3.1.

The controller is an ordering table implemented in the robot RAM memory. In the example given by Table 3.1, this memory (a mapping table) only needs 16×4 bits to implement the evolvable controller. If more than one behaviour are selected (by writing one in its corresponding position) in the table, the ones to the left have priority to be chosen. Therefore, the order of the behaviours in the table rows is also important. In a real situation, however, the memory will be considerably bigger to accommodate all possible conditions and behaviours necessary to navigate the robot safely. The evolutionary algorithm can control the bits stored in this memory and adjust them to a proper mapping between the conditions and behaviours. The memory bits can then be arranged in sequence to form a bit string: the robot genotype (or chromosome) [Set99]. The execution of the task

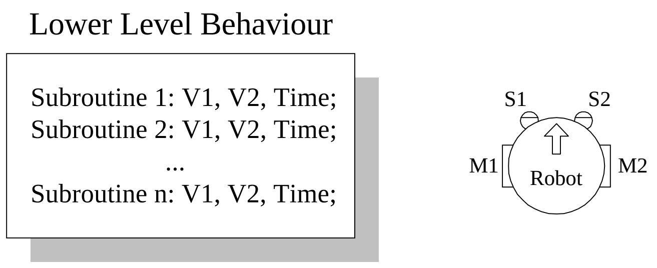
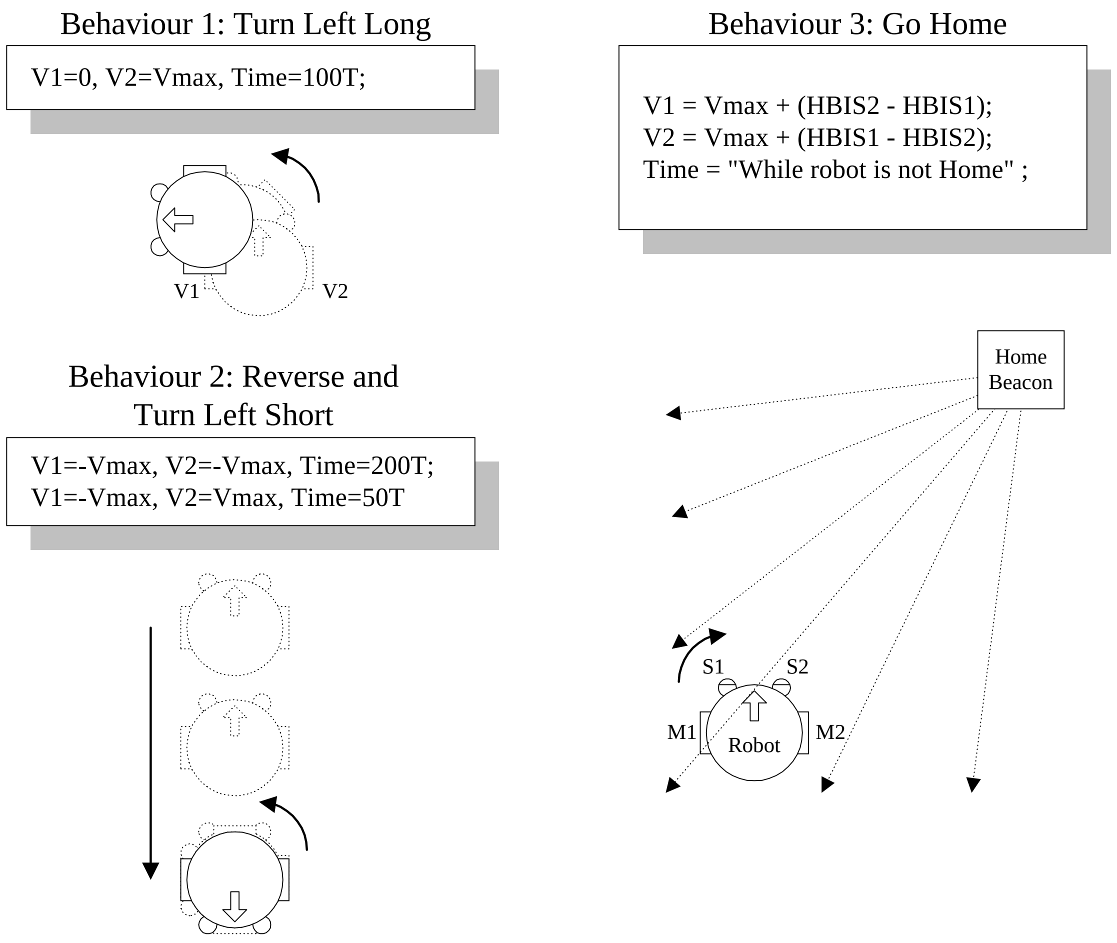

Lower Level Behaviour
Subroutine 1: V1, V2, Time;
Subroutine 2: V1, V2, Time;
...
Subroutine n: V1, V2, Time;
S1
S2
M1
Robot
M2

Figure 3.5 – Example of a lower-level behaviour containing a set of subroutines that are chosen and adjusted by evolution. *M1* and *M2* are the left and right motors of the robot, and *S1* and *S2* are the left and right sensors. *V1* and *V2* are the speed levels of the corresponding motor and *Time* is the time delay while the subroutine is active.

may appear quite simple, since its learning space has been appropriately minimised. In practice, however, a uniform exploration and learning of the optimal policy can present a considerable challenge for an evolutionary technique in a highly dynamic, uncertain environment, where many conditions and behaviours are present [Mat98a].

A mixed strategy consisting of refining (or evolving) each behaviour separately at first, using an evolutionary algorithm to control their parameters, was considered to be used to implement the robot control. The idea consists of considering each behaviour as a lower-level task and evolving a distinct sub-controller to achieve each one of them [Cli92]. Each sub-controller is a reconfigurable circuit that can be implemented in software in the robot microprocessor and memory, having a limited set of subroutines containing speed values and time delays for each motor driver. Figure 3.5 shows an example of a lower-level behaviour containing a set of chosen subroutines. The values of *V1*, *V2***,** and *Time* are adjusted by evolution.

Once the set of subroutines is chosen for each behaviour, evolution can adjust the speed and time variables until the robot is able to perform it. Figure 3.6 shows an example of three lower-level behaviours and their corresponding set of subroutines. In the figure, *T* is a time constant that represents the delay of one iteration. If for example, the robot executes one iteration each 10ms, *T* = 10ms. If behaviour 1 is selected, the robot will adjust the speed of motor 1 (*V1*) to zero, and motor 2 (*V2*) to Vmax, the pre-defined value for maximum speed, and keep these settings for 100 iterations (1s). In doing so, the robot stops the left wheel and turns around it as demonstrated in the figure. The first subroutine of behaviour 2 sets negative speeds for both motors, making the robot reverse for 200 iterations (2s) before turning around its centre for 0.5s. If the “GO HOME” behaviour


Behaviour 1: Turn Left Long
V1
V2
Behaviour 2: Reverse and Turn Left Short
Behaviour 3: Go Home
Home Beacon
S1
S2
M1
M2
Robot

```
V1=0, V2=Vmax, Time=100T;
```

```
V1=-Vmax, V2=-Vmax, Time=200T;
V1=-Vmax, V2=Vmax, Time=50T
```

```
V1 = Vmax + (HBIS2 - HBIS1);
V2 = Vmax + (HBIS1 - HBIS2);
Time = "While robot is not Home" ;
```

Figure 3.6 – Example of three lower-level behaviours and their corresponding set of subroutines. *S1* and *S2* are the left and right sensors, *V1* and *V2* are the speed levels of the corresponding motor, and *T* is the delay of one iteration. *HBIS1* and *HBIS2* are the light intensity of the home beacon received by the left and right sensors respectively.

(behaviour 3) is selected, the sensor information will be used to drive the robot towards an infrared beacon that indicates the position of "home". *HBIS1* and *HBIS2* represent the light intensity of the home beacon received by the left and right sensors respectively. The difference of intensity is added to the speed, turning the robot until it is facing the beacon when the difference becomes zero.

The sub-controllers can be then evolved individually until each necessary behaviour is capable of driving the robot accordingly. When this first step can produce satisfactory results, all the sub-controllers (performing specific behaviours) are attached to a central condition-behaviour-mapping controller, which receives the present conditions of the robot as inputs. Then, the complete controller can be evolved in the dynamic environment to select the appropriate behaviours according to each situation the robot

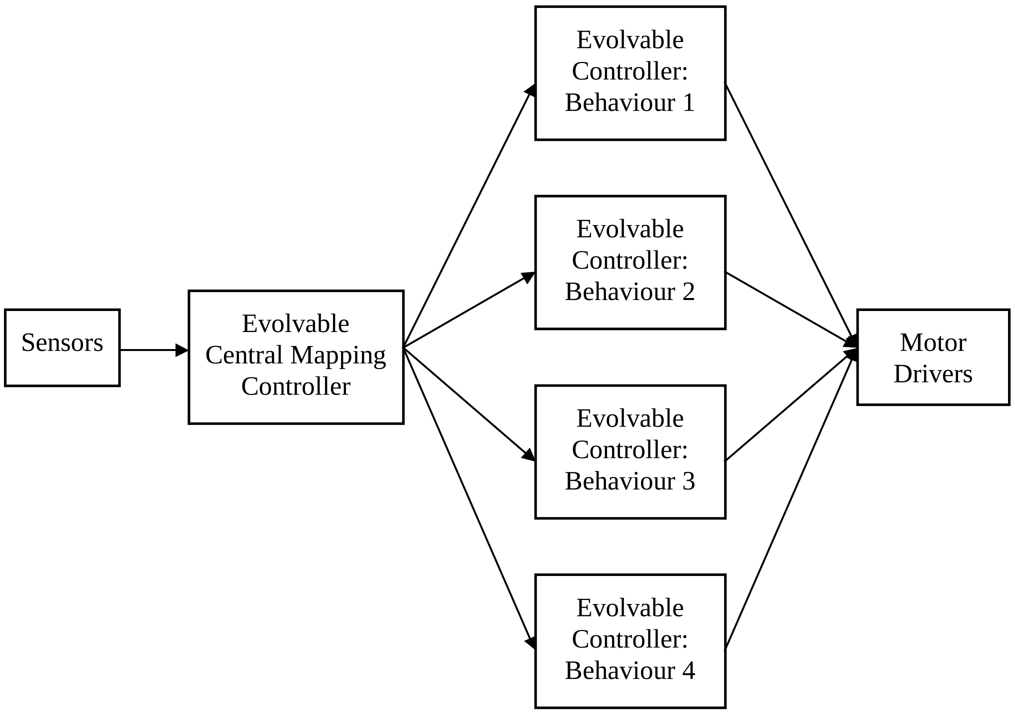

Evolvable Controller: Behaviour 1
Evolvable Controller: Behaviour 2
Sensors
Evolvable Central Mapping Controller
Motor Drivers
Evolvable Controller: Behaviour 3
Evolvable Controller: Behaviour 4

Figure 3.7 – The incremental control system: each behaviour is selected by a central mapping of condition to behaviour that translates the sensor readings into conditions and chooses the appropriate evolvable behaviour to control the motor drivers.

faces. Each behaviour as well as the central mapping of condition-behaviour can still be continuously refined by the genetic algorithm. Figure 3.7 shows a general view of the incremental control system.

This is a powerful solution that can be implemented on-board the robot microprocessor. It abstracts away from evolution the complexity of the sensory structures and motor drivers, allowing a significant reduction of the search space [Mee96]. The behaviours can become progressively more complex to accommodate other robot applications. The only reason this solution was not chosen is the necessity of human intervention to evaluate when each behaviour is obtained or to reposition the robots for new trials.

3.2.4 Pulse Stream Neural System

An interesting alternative to implement the evolvable controller is the use of Artificial Neural Systems (ANS), which are computing systems based on cognitive approaches [Rey94]. In PS neural systems, information is contained in waveform timing, not in the amplitude. Coherent Pulse Width Modulation (CPWM) [Rey93] is a PS technique which presents good performance for control applications [Chi95]. There are many ANS techniques already prototyped in hardware that can be used for robot control, but a fair comparison among them is difficult to find. Nevertheless, the CPWM implemented by Chiamberge, described in [Rey95], is a good solution for robot navigation problems. It was chosen for it can be easily evolved by a genetic algorithm, since the weights of the synapses are controlled by the voltage on specific capacitors, periodically refreshed from an external digital memory [Chi96]. The memory contents can be dealt with as a genotype and evolved by the chosen evolvable technique.

The CPWM chip has a self-standing prototype board to interface it to a controller plant [Rey95]. This board can be used as described in Figure 3.8 to perform the programmable control of the robots [Chi93]. The system array has 16 inputs coming from the external world (eight digital for direct CPWM signals and eight analogue, internally converted by on-chip converters). The chip has 32 digital outputs, including the 16 hidden ones, which are also available outside.

The inconvenience of applying the CPWM chip to control the chosen robots is that it requires an interfacing board, which may consume too much power to fit into a small-sized robot. However, if this technique is to be used, a specific interface for controlling the configuration memory and the CPWM chip from the robot microprocessor needs to be designed. This approach would involve a complex design of the robot architecture and the overall cost would exceed this project budged. Therefore, it had to be discarded.

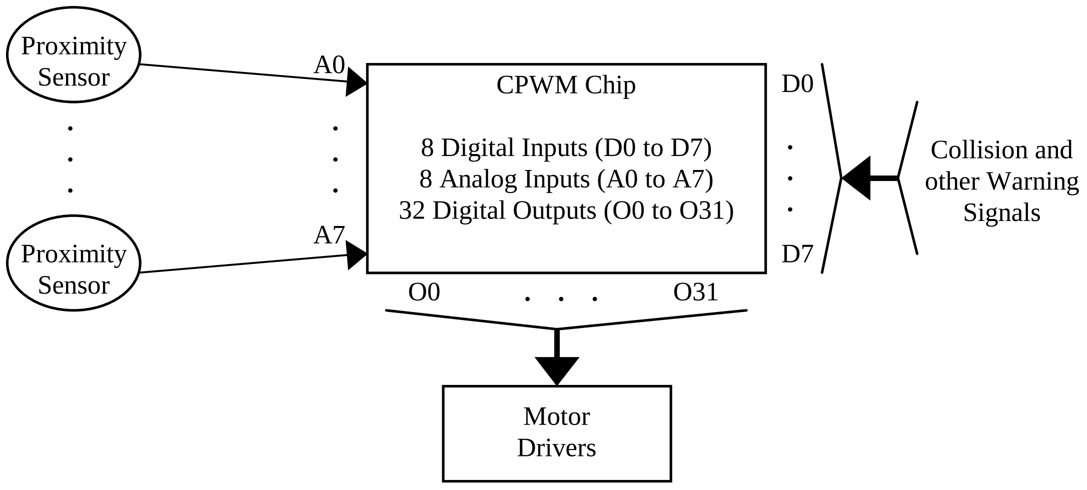

Proximity Sensor
Proximity Sensor
A0
A7
CPWM Chip
8 Digital Inputs (D0 to D7)
8 Analog Inputs (A0 to A7)
32 Digital Outputs (O0 to O31)
D0
D7
O0
O31
Collision and other Warning Signals
Motor Drivers

Figure 3.8 – A self-standing CPWM prototype board controlling a robot as illustrated in [Rey95].

3.2.5 Boolean Neural Network

Boolean Artificial Neural Networks have been used in many situations providing fast training, processing speed, and easy hardware implementation [Fil92] [Sim94] [Har97d] [Sim97]. Boolean neural networks can have weights between nodes or not. In the last case, they can also be described as “RAM-based”, “n-tuple based”, or Weightless neural networks [Lud99]. As the last name suggests, these models do not have weighted connections between neuron nodes, and work with binary inputs and outputs. The neuron functions are stored in look-up tables that can be implemented in software or using Random Access Memories (RAMs). The learning phase consists of directly changing the neuron contents in the look-up tables, instead of adjusting the weights between nodes. This characteristic generally allows a faster learning than weighted neural networks, since by modifying weights the previously learned information is also modified. In the more flexible Boolean neural networks, the neurons can learn new information by changing only the memory contents corresponding to the new input pattern, without modifying the information related to other inputs.

Since the original work by Bledsoe and Browning [Ble59] and Aleksander [Ale66] proposed the N-tuple sampling technique, many weightless models were proposed such as the PLN (Probabilistic Logic Neuron) [Ale90], *p*RAM (Probabilistic RAM) [Aus94], and GSN (Goal-Seeking Neuron) [Aus98]. They consist in the addition of new

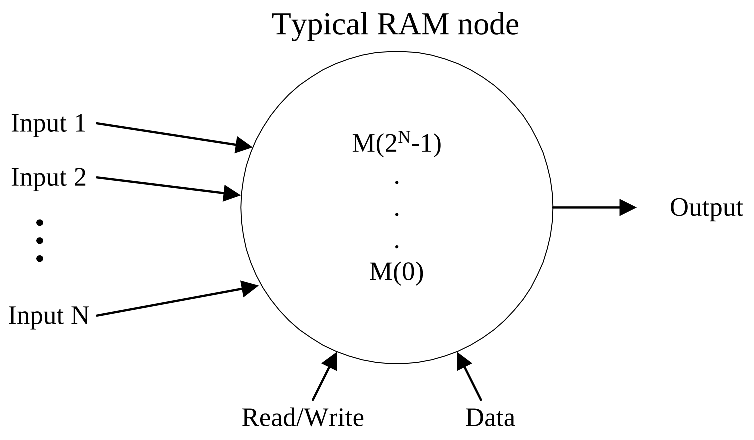


Figure 3.9 – Typical RAM node or neuron as illustrated in [Aus94].
Where: *N* is the connectivity (number of inputs);
Input 1 to *Input N* are the input terminals of the neuron;
M(0) to *M(2^N-1)* are the memory contents (neuron contents);
Output is the output terminal that transmits the addressed content;
Data is the input terminal to receive learned data;
Read/Write defines the working mode of the neuron: if it is in the learning or recognition phase;

features and learning schemes, resulting in more powerful neural nodes. In relation to robot implementations, the RAM model, or RAM node is very attractive, since it provides great flexibility, modularity, parallel implementation, and high speed of learning, what leads to less complex architectures that can easily be implemented with simple commercial circuits [Fil92]. Figure 3.9 shows a representation of a typical RAM neuron, as illustrated in [Aus94]. The RAM node is a random access memory addressed by its inputs. The connectivity of the neuron (*N*), or the number of inputs, defines the size of the memory: 2^N. The inputs are binary signals that compose the *N*-bit vector of the address that can access only one of the memory contents. This neuron consists of a memory that stores a look-up table that implements any Boolean function performed by the neuron. Therefore, a RAM neuron can compute any Boolean function of its inputs.

A RAM neural network is a single layer architecture where the number of neurons is typically enough to cover all of the network binary input vector and depends on the neuron connectivity. Simões *et al.* studied the main issues and constraints of the connectivity pattern in RAM networks in [Sim96]. Figure 3.10 shows the connectivity pattern of a RAM discriminator [Ale84]. For an input vector of size *K*, the number of necessary neurons *J* of connectivity *N* that should be used to cover all inputs of the input

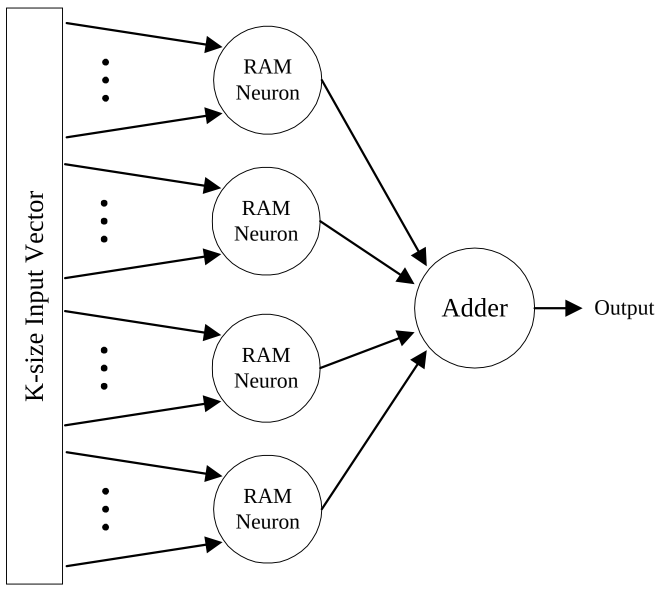


Figure 3.10 – The connectivity pattern of a WISARD discriminator, as illustrated in [Ale84].

vector should satisfy: $J \times N \geq K$. This neuron group is called a WISARD discriminator and its response is produced by connecting an adder that sums the neuron outputs, counting the number of active neurons (neurons outputting "1") in the group [Lud99]. A WISARD system is built by grouping together a set of discriminators, each one being responsible by recognising a different class of patterns. A winner-takes-all-block can be attached to the adder outputs to choose the discriminator containing the greater number of active neurons, pointing to the winning class. Although individual RAM neurons cannot generalise, a WISARD discriminator is able to classify unknown patterns based on the knowledge acquired by the learning phase.

Learning in a RAM neuron is much simpler than adjusting weights in weighted neural networks. It is performed by just writing new contents to the memory of the neurons [Aus88]. The neurons are initialised with "0" in all contents, and are taught to respond with "1" to the patterns that belong to the class corresponding to the neuron group, or discriminator. This is accomplished by writing "1" to the memory positions of all neurons of the corresponding class addressed by the input pattern.

Once a RAM architecture containing neurons grouped in discriminators for each class was chosen, a new strategy is necessary to allow its use to form an embedded evolvable controller for the robots. This strategy consists of replacing the learning phase by an evolutionary algorithm [Cli92], able to manipulate the neuron contents. To exemplify how this solution works, consider the same example of Section 3.2.2, where the robot has four infrared sensors and two motors. Each sensor reading is converted into a 2-bit signal and the network output consists of three classes of commands to each motor driver: *Stop*; *Front*; and *Reverse*. Different behaviours can be obtained by combining those three speed commands so that the robot can perform a long turn or a short one around its centre, stop, or move forward.

The implementation consists of two RAM neural networks, one controlling each motor. The size of the input vector (K), containing four sensors and two bits per sensor, for each neural net is 4×2, which equals to 8 bits. An n-tuple classifier [Aus88] with five two-input neurons is enough to cover all bits of the input vector. The neural net architecture is presented in Figure 3.11, where the neurons can be previously trained with the most common situations faced by the robot during normal navigation or just randomly initialised. The evolutionary algorithm will then try to adjust the neuron contents until the network behaviour is able to drive the robot properly. The neuron contents are directly encoded onto a linear bit-string genotype containing 120 bits (15 neurons for each motor; four bits per neuron). After randomly initialising this bit string, the genetic code can be evolved by the evolutionary algorithm in real time until the robots can perform the desired tasks, allowing the evolvable controller to face a real noisy environment.

This solution was inspired by a similar architecture, investigated before in [Bot96], that applied the fast training and simple hardware design of the RAM model to control a mobile Khepera robot. The article presented two weightless neural networks used to control the robot: the GSN [Sim94] and the RAM model [Sim96]. The implementation using the GSN model made use of two pyramids to control each wheel. The sensor values were directly connected to the pyramid inputs, and the network outputs were applied to control the wheel motors. The article presented a new concept that allowed each neuron, after the training phase, to be manipulated as a "black box". All possible combinations of neuron inputs were analysed to convert the complete network into a combinational circuit. The minimisation of the circuit logic was processed according to [Sim96]. There, a description of the minimised neuron network was converted into an Assembler language code that used simple basic instructions. Consequently, the network circuit could be directly executed by the microprocessor ALU (Arithmetic Logic Unit). This strategy decreased the execution time of the algorithm, because simple logic functions (*NOT*, *AND*, and *OR*) are faster to execute than complex floating-point operations used by a great number of robot control models [Pol00]. This strategy of mapping the behaviour of a

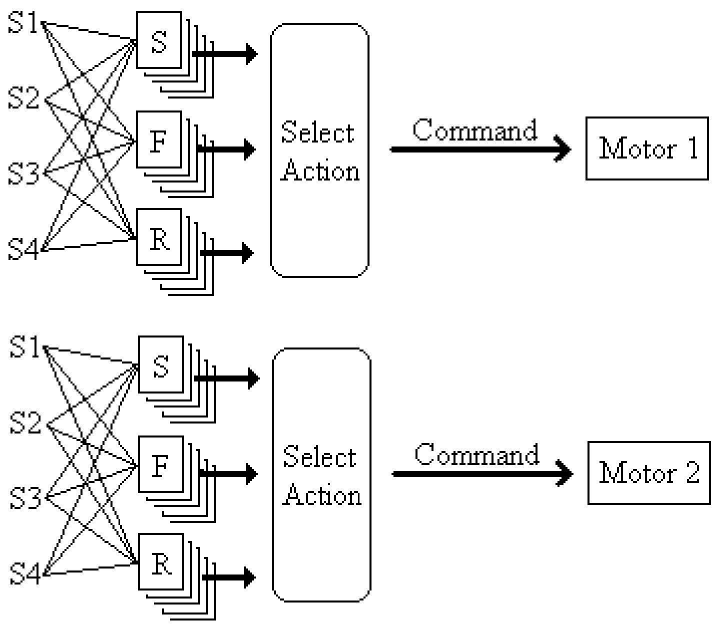


Figure 3.11 – RAM Neural architecture controlling a 2-wheeled Robot: *S1* to *S4* are infrared sensors; *S*, *F*, and *R* represent the groups of neurons of the classes *Stop, Front*, and *Reverse,* containing five neurons each.

complete circuit into a logic table will be applied to a RAM controller in the experiments described in Section 6.4 of this thesis.

The RAM neural network shown in Figure 3.11 has three behaviour classes with five neurons each that control the robot motors. The RAM model was implemented according to the same approach used for the GSN in [Bot96], with the inclusion of simple additions and comparisons, both necessary to the RAM model output evaluation, executed by the “Select Action” block in the figure. The RAM neural network simplicity and its implementation as elementary logic functions are responsible for its fast performance. The weightless neural networks, mainly the RAM and the GSN models, tend to occupy a small portion of memory. Therefore, memory allocation is not a great issue, increasing the possibilities for the robot computing system. The mapping of the RAM neural network into simple ALU logic functions and their direct execution in the microprocessor ALU can reduce even more the total memory required by the control algorithm. The faster speed provided by these simple implementations allows a faster controller, which can improve the decision rate in complex sensory systems, like artificial vision [Smi98]. This strategy also allows the use of low-cost microprocessors.

The strategy of previously training the neural net with examples of pre-defined situations and then leaving the neuron contents to be refined in real time manoeuvring can be faster than allowing evolution to work with a randomly initialised neural network [Tho99]. However, it can lead the population to become trapped into a local optima, and prevent evolution from exploring different controllers through the genotype space before climbing a fitness slope, reaching the expected behaviour [Har93a]. For all advantages mentioned above, a RAM neural network was chosen to implement the embedded evolutionary controller for the robot population. Chapter 4 presents more details of the chosen neural network architecture and shows how it is combined with the evolutionary system.

4 THE EVOLUTIONARY SYSTEM

This chapter presents the strategies chosen to implement the individual controller of each robot and the evolutionary system that controls the group of robots. It also shows an overview of the complete system and an introduction to the robot architecture. Although the strategies described in this chapter can be applied, in theory, to control any number of robots, in this work the global idea was adapted to control a group of six robots. Even though the suggested system was proven to work with such a small population, a larger population of robots would give greater diversity to evolution, improving the performance of the system [Fic00]. Thus, more individuals provide more genetic combinations and increase the chances of finding a good solution to the problem.

The goal of the implemented evolutionary system is to train automatically a team of six autonomous mobile robots to interact with an unforeseen environment in real time. The system is also able to continuously refine the generated solution during the whole working life of the robots, coping with modifications of the environment or in the robots. Although implemented into a specific group of 2-wheel differential-drive robots for a specific task, the evolutionary system described in Sections 4.1 and 4.2 can be adapted to control other kinds of robots performing different tasks. Section 4.3 and Chapter 5 describe how the system was specifically adapted to our team of robots. Nevertheless, these two sections are general enough to be used as guidelines to help the conversion of the system to other mobile or static platforms.

To test whether randomly initialised robots could really be trained by evolution to do something practical, a very simple task was chosen: exploration with obstacle avoidance [Mor96]. Such a simple task, that is also known as collision-free navigation [Rao96] [Key97a], facilitates the implementation of the system and allows its development in relatively low-cost robots. Therefore, more robots could be built and evolution can benefit from more diversity in the population. The main issue considering functional specification in an evolutionary system is to tell evolution *what* the robots have to do, without telling it *how* they are going to achieve that [Flo96a]. In our case, the robots are encouraged to explore the environment, going as fast as possible without colliding into the obstacles or each other. Because the workspace contains various robots, the environment also includes some robot-to-robot interference [Set97] (e.g., collisions between robots and reflection of the infrared signals by approaching robots). Though the presented task is simple and does not involve explicit robot interaction, the transparency of this domain allows a clear investigation of the implicit interactive nuances within the evolutionary approach. Chapter 6 will show how, based on a reward-punishment scheme, evolution can find unique, unexpected solutions for that problem.

4.1 Individual Control Strategy

The robot architecture can conceptually be seen as a central control module interfacing all other functional modules, which either supply or demand data required for autonomous processing (see Figure 4.1). The modules were implemented using a combination of dedicated hardware and software executed by the robot microprocessor. The robot architecture is configured by a set of parameters, a certain number of bits stored in RAM memory. In evolutionary terms, this set of parameters is called the robot chromosome [Koz98]. The Sensor Module is configured by a subset of the chromosome that indicates the number of sensors used and their position in the robot periphery. The motor drive module is configured by another subset of the chromosome that configures the speed levels of the robot.

The Motor Drive module receives and translates commands from the central control module and controls the direction of travel and speed of the two robot motors. The proximity of obstacles is obtained by the sensor module that decides which proximity sensors are connected to the central control module according to the parameters stored in the robot chromosome.

The subsystems of the Central Control Module are presented in Figure 4.2. Connected together via the communication module, the Evolutionary Control circuits of all robots control the complete evolutionary process. They process the data stored in the chromosome and send the configuration parameters to the Navigation Control and the other

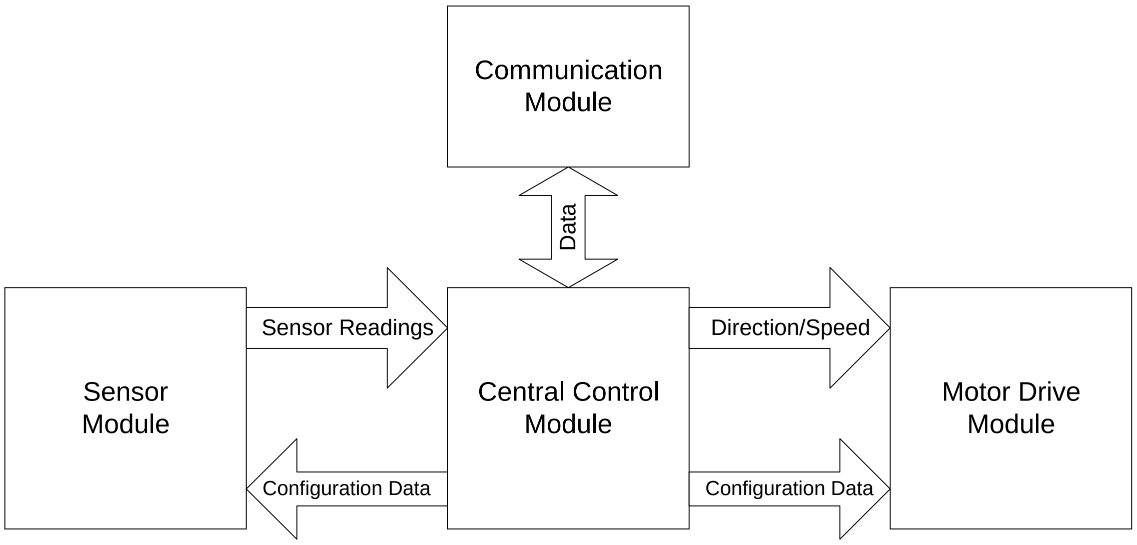


Figure 4.1 – Architecture of the robot control system: the Sensor Module and Motor Drive Module are configured by the Central Control Module, which processes data from the sensors and commands the motor drive module in how to drive the robot.

modules. The evolutionary control systems of all robots use communication to combine and form a global decentralised evolutionary system [Mat98b]. This global system controls the evolution of the robot population from generation to generation. It is responsible for selecting the fittest robots (the best-adapted to interact with the environment), mating them with the others by exchanging and crossing over their chromosomes, and finally reconfiguring the robots with the resultant data (the offspring) [Tom95].

A Supervisor Algorithm monitors the robot performance, informing the evolutionary control how well-adapted it is to the environment. According to events and tasks performed by the robot, perceived internally by special sensors, a score or fitness value is calculated and used by the global evolutionary system to select the best-adapted individuals to breed. The supervisor algorithm is responsible for activating a rescue routine, a built-in behaviour that is able to manoeuvre automatically the robot away from a dangerous situation once it is detected by the sensors. Contact sensors in the bumpers determine the occurrence and position of a collision. When activated, the rescue routine will take control of the robot until it is safely recovered. It can communicate directly to the motor drive module, by-passing the navigation control. When the rescue manoeuvre is completed, the supervisor algorithm allows the motor drive module to accept once more the commands of the navigation control circuit and the robot resumes on its way.

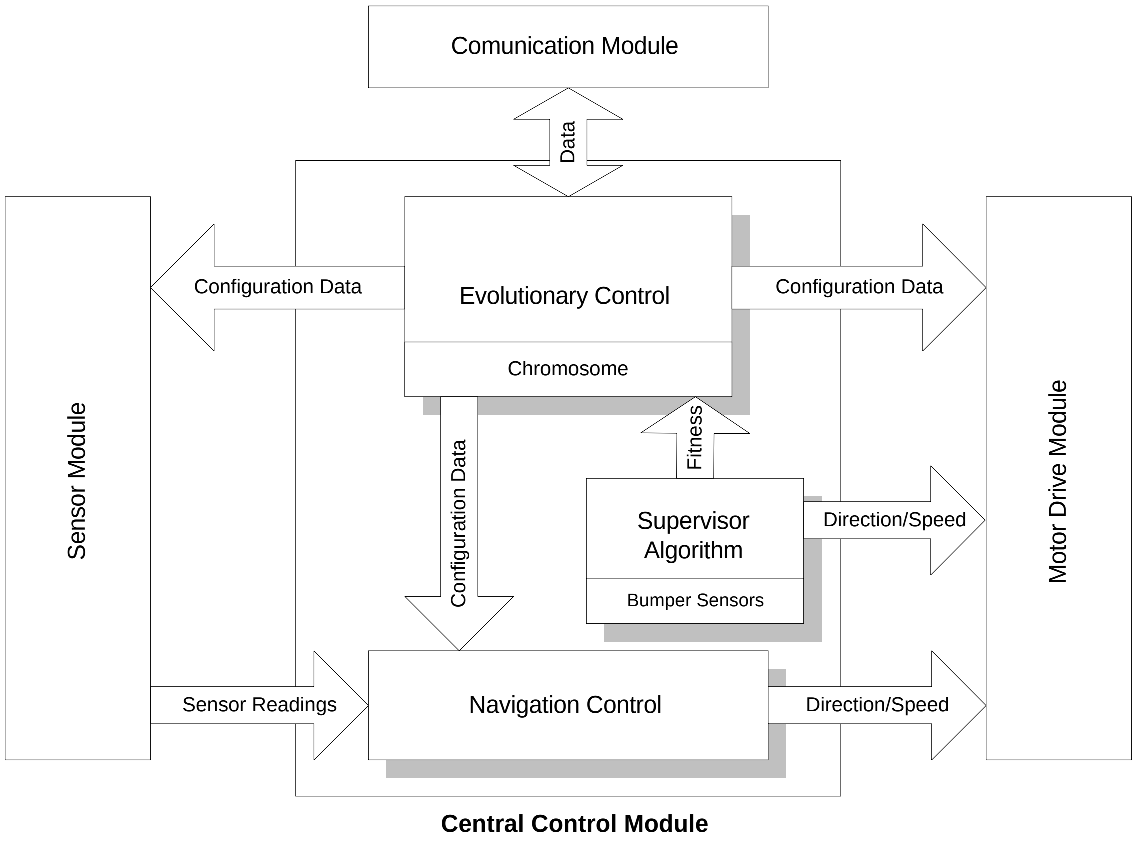


Figure 4.2 – The Robot Central Control comprises three main subsystems: the Evolutionary Control, the Supervisor Algorithm, and the Navigation Control.

It is the Navigation Control, configured by the evolutionary control, that commands the motor drive module according to the information provided by the sensor module. It processes the information of the sensors and decides what the robot has to do. Then, it sends a command to the motor drive module, which will control the speed of the motors to make the robot manoeuvre accordingly. The navigation control is the centre of the autonomous navigation of the robot. Configured by the parameters stored in the chromosome, it drives the robot independently. Evolution is responsible for adjusting these parameters so that the robot performs well in the environment. The implementation of the navigation control circuit is explained below in Section 4.1.1.

Section 4.2 explains how the developed evolutionary system works. All components of the robot architecture are presented in more detail in Section 4.3 and Chapter 5 describes them in terms of hardware and software.

4.1.1 The Navigation Control Circuit

A RAM neural network was chosen to implement the navigation control circuit basically for the reasons given in Section 3.2.5. RAM neural networks have unique features that facilitate their evolution by the system, simplify the implementation in the robot hardware, and allow small modifications to be carried out with minimum effort [Sim96] [Sim97]. They provide a robust architecture, with good stability to mutation and crossover. Most neural networks, like the chosen one, present redundancy between the genotype and the phenotype [Shi00]. In other words, a small change in the bits of the chromosome (the genotype) will not produce a radical change in the behaviour of the network (the phenotype) (see Section 2.1.1 for a detailed explanation of these terms). This characteristic is called Neutrality [Kni48]. Therefore, the selected neural network is stable enough to allow evolution to gradually refine the configuration parameters of the navigation control circuit, seeking a better performance. Its good neutrality makes it suited to be evolved by the system since a small mutation on a fit individual should, on the average, produce an individual of approximately the same fitness [Bar98]. Similarly, crossover between two parents of similar fitness, on the average, should produce offspring with similar fitness.

A Boolean neural network such as the RAM is a good solution for implementing with an embedded controller (see Section 3.2.5), because it is fast and small enough to cope with the speed and memory restrictions of the on-board microprocessor [Bot96]. It allows the utilisation of command outputs that can be interpreted as high-level routines or be employed as low-level incremental commands executed in each iteration [Sim96].

Figure 4.3 shows the sensor module processing the information of the sensors and feeding the neural network inside the navigation control circuit. The output of the neural network is a command that tells the motor drive module how to control the motors. The evolutionary control reads the information contained in the chromosome and sends the parameters to configure the sensor and motor drive modules. It also reads the contents of the neurons from the chromosome and transfers them to the neural network in the navigation control circuit. The motor drive module intercepts the command and activates the corresponding routine that generates the signals for the motors.

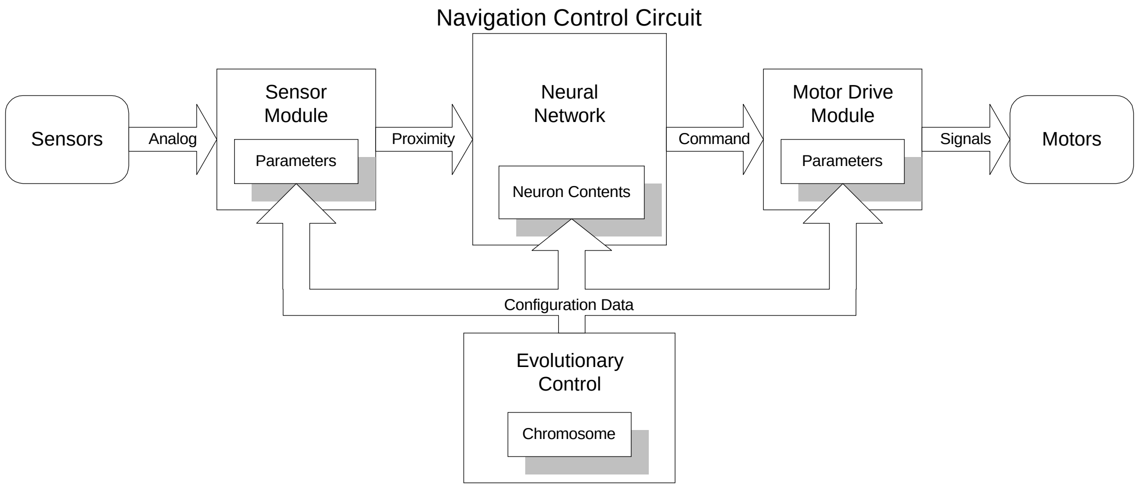
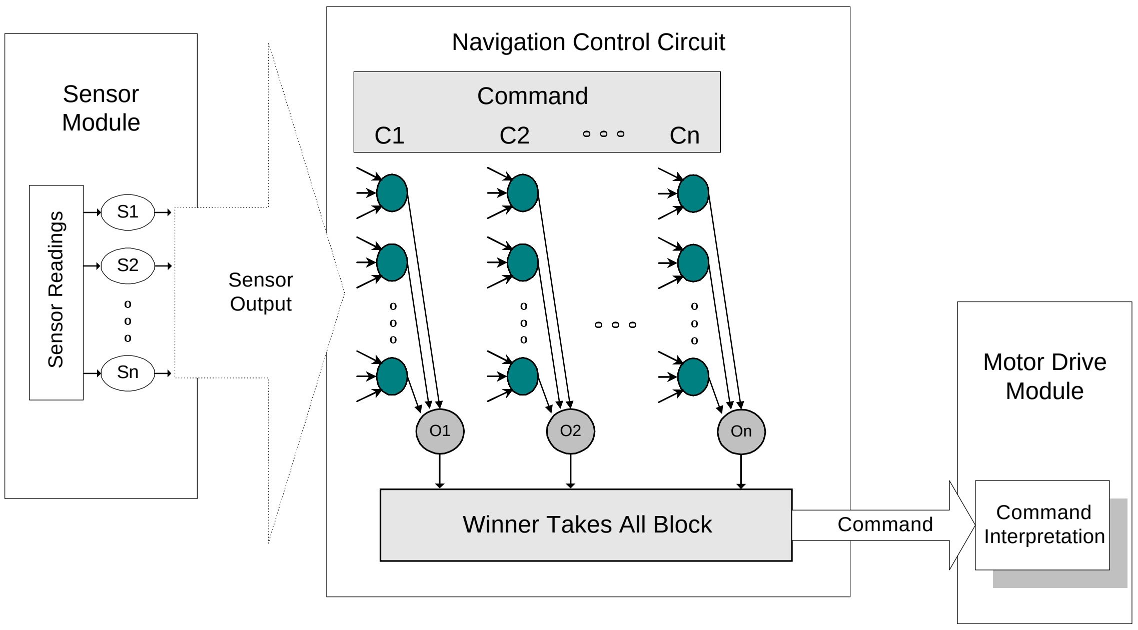


Figure 4.3 – The navigation control circuit interfacing the sensor and motor drive modules, and the evolutionary control.

Figure 4.4 shows more details on how the navigation control circuit interfaces the sensor and motor drive modules. Section 3.2.5 presented an introduction to the RAM neural network architecture. This section will show in more detail how it can be configured to implement the navigation control circuit. The neurons are connected in groups (discriminators) that correspond to one of the possible classes of commands (*C1*, *C2*, … *Cn*) the neural network can choose. The groups are connected to an Output Adder (*O1*, *O2*, … *On*) that counts the number of active neurons in the group. The Winner-takes-all block receives these counting from the output adders, chooses the group with more active neurons, and sends the corresponding Command to the motor drive module. The sensor module converts the analogue readings of the infrared proximity sensors into 2-bit signals that can be connected to the neuron inputs.

Figure 4.5 presents an example of eight possible commands. The commands *FS*, *FM*, *FF*, and *S* tell the command interpreter inside the motor drive module the level of speed it should signal the motors to go. The robot can *Stop*; or move to the ahead with three different speed levels: *Fast*, *Medium*, and *Slow*. The commands *TRS*, *TRL*, *TLS*, and *TLL* specify how the robot is going to turn. It can *Turn* to the *Left* or *Right* in a *Short* way (turning around its centre with one wheel going forward and the other backwards) or *Long* way (stopping one wheel to do an arch).



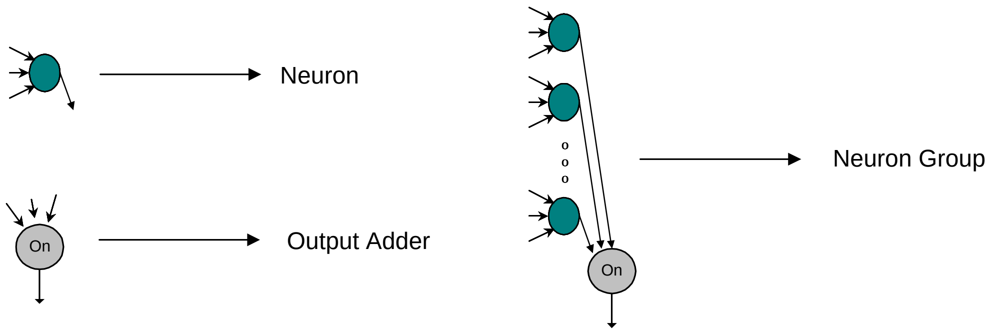


Figure 4.4 – The neural network in the navigation control circuit. *S1* to *Sn* are the binary sensor readings. The Output Adders (*O1* to *On*) count the number of active neurons in the group. *C1* to *Cn* are the classes of commands to the motor drive module.

Table 4.1 – Encoding of the Neural Network Commands.

Command	**Denomination**	**Hex Code**	**Binary Code**	**Description**
Command 1	*FS*	00	00000000	*Front Slow*
Command 2	*FM*	01	00000001	*Front Medium*
Command 3	*FF*	02	00000010	*Front Fast*
Command 4	*S*	03	00000011	*Stop*
Command 5	*TRS*	04	00000100	*Turns Right Short*
Command 6	*TRL*	05	00000101	*Turns Right Long*
Command 7	*TLS*	06	00000110	*Turns Left Short*
Command 8	*TLL*	07	00000111	*Turns Left Long*

Table 4.1 shows the encoding of the commands chosen by the neural network to control the motor drive module. Each command is represented by a hexadecimal number (00 to 07) and converted into eight bits that are transmitted to the

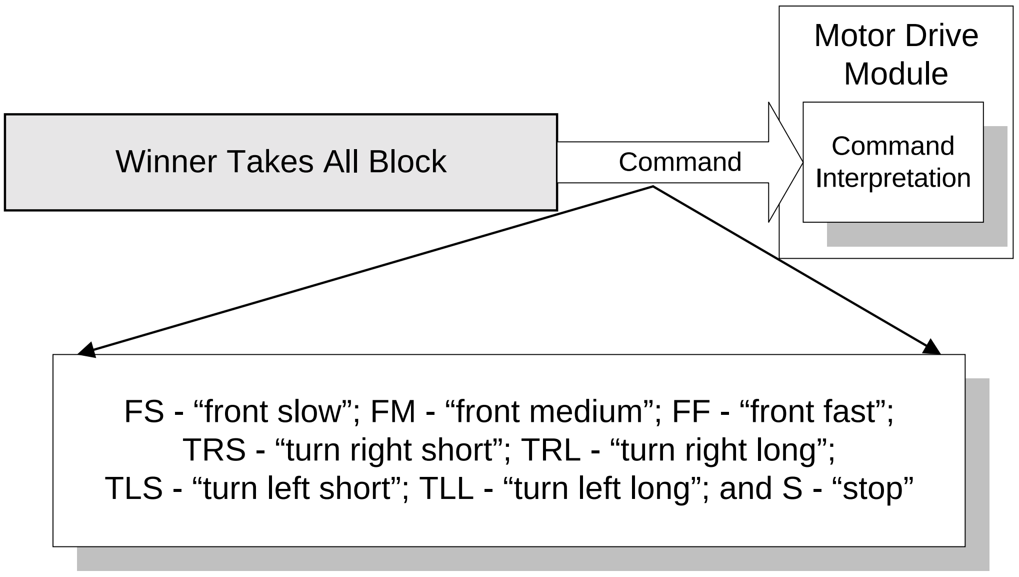

Motor Drive Module
Winner Takes All Block
Command
Command Interpretation
FS - “front slow”; FM - “front medium”; FF - “front fast”;
TRS - “turn right short”; TRL - “turn right long”;
TLS - “turn left short”; TLL - “turn left long”; and S - “stop”

Figure 4.5 – Example of eight possible commands to the motor drive module. *FS*, *FM*, *FF*, and *S* tell the command interpreter the level of speed of the motors and *TRS*, *TRL*, *TLS*, and *TLL* specify how the robot is going to turn.

motor drive module. The commands can then be interpreted by the motor drive module and the corresponding speed levels are selected to drive the motors. As it can be seen in the figure, only the first three bits are being used. The other five bits can be used to accommodate more codes in future applications.

In the selected approach, the inputs of the RAM neurons are connected to the sensor outputs provided by the sensor module. All neurons of the network have the same number of inputs, although that number may vary according to the application [Lud99]. In the implemented network, all neurons in the same position in the groups are connected to the same inputs (i.e., the first neuron of the first group will have the same inputs as the first neuron of the second group and so on…). Figure 4.6 shows a 3-input neuron where *i0*, *i1*, and *i2* are different neuron inputs connected to the input lines that bring in the signals from the sensors. $N_{m,n}$ is the neuron designator where *m* is the number of the group to which the neuron belongs and *n* is its position in the group [Fil92]. All neurons with the same position *n* are connected to the same input lines L_i, L_j, and L_k, and consequently, to the same sensor outputs.

The connectivity between the input lines and the sensor outputs is controlled by a Connectivity Matrix that defines which sensor outputs are connected to L_i, L_j, and L_k. Figure 4.7 exemplifies how the groups of neurons are connected in the neural network architecture. The connectivity matrix is randomly initialised at the beginning of a new

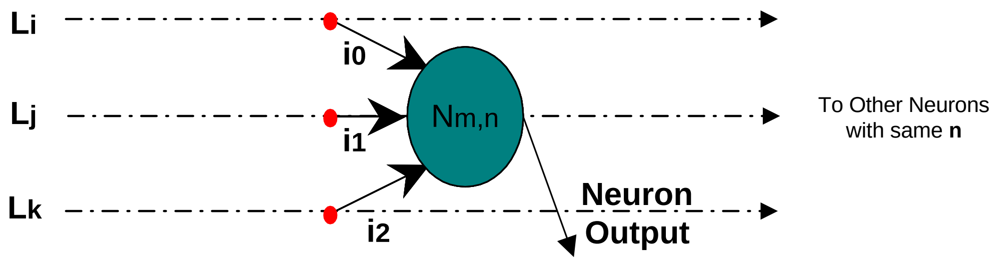


Figure 4.6 – Example showing a 3-input neuron: *Nm,n*.
Where: *m* – is the group the neuron belongs;
n – is the position of the neuron within the its group;
i0, *i1*, and *i2* are the neuron inputs;
Li, *Lj*, and *Lk* – are input lines connected to the sensor outputs.

evolutionary experiment. In the figure, the dotted lines between the sensor outputs and the input lines exemplify one randomly-generated possibility of how the connectivity matrix can connect the sensors to the network. In Chapters 6, 7, and 8, different experiments will be described with the same architecture, but with a different number of neurons and neuron inputs. In this way, an efficient configuration that is not too complex to be evolved by the system, but can still drive the robots properly, could be found.

One other advantage of RAM neural networks is their modularity [Ale90]. This characteristic simplifies the modification of the architecture. The number of neuron inputs can be modified by rearranging the connectivity to the sensors alone. Sensors can also be added or removed in this way. New commands are easily included by inserting more neuron groups [Aus98]. Figure 4.7 shows how to insert a new command (e.g., *Reverse – R*) in the network architecture by connecting a new neuron group (neuron group *m+1*) to the input lines that run horizontally through the architecture. This flexibility helped in carrying out many experiments where different configurations of the basic architecture were tested (see Chapter 6).

The RAM neural network can be evolved by simply storing sequentially the neuron contents into the robot chromosome and allowing the evolutionary algorithm to manipulate these bits. Section 3.2.5 explained how the architecture was chosen, and its configuration depends on insights gained through a long trial-and-error process that will be presented in the experiments in Chapters 6, 7, and 8. Basically, the neural network must have enough inputs to cover all the sensors, although some of the sensors may be connected to more than one input line [Sim97]. To avoid saturation, enough neurons must be placed in the groups so that the network can learn all the different input configurations

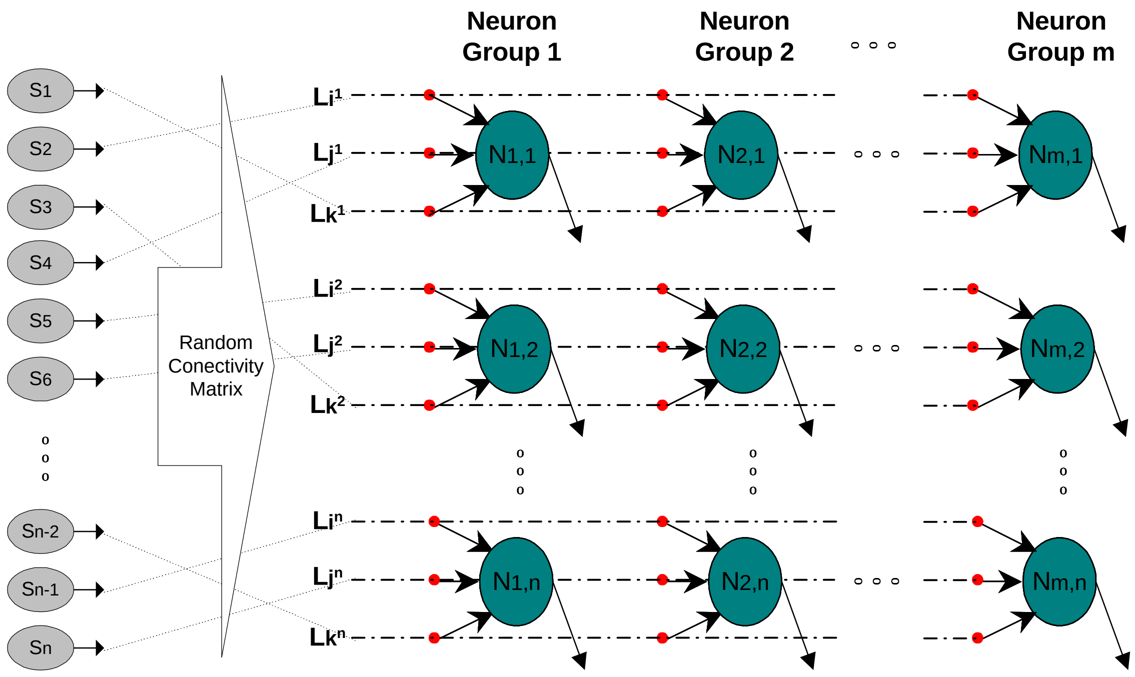
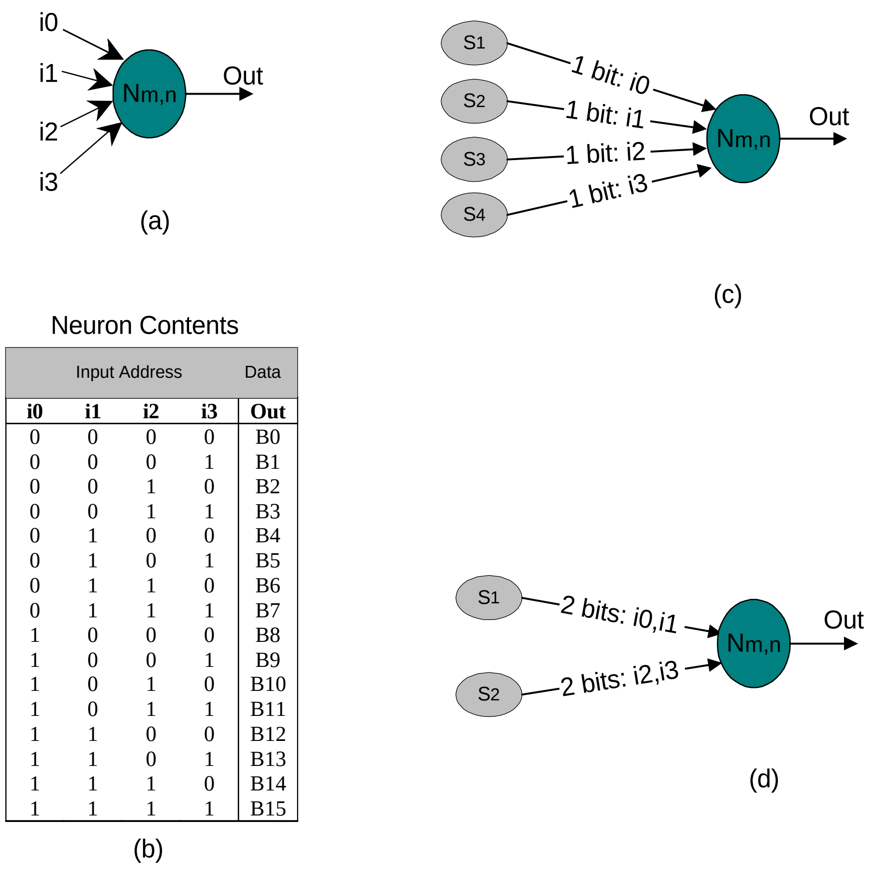


Figure 4.7 – Example showing the connectivity of 3-input neurons in the neural network architecture, containing *m* groups with *n* neurons each. Randomly initialised, the Connectivity Matrix defines which sensors are connected to the input lines *Li*, *Lj*, and *Lk*.

that correspond to the correct output commands [Fil92]. If the network is having difficulty learning a different configuration, more neurons should be added. Different architectures were implemented and simulated in software until the developed solution was obtained.

Figure 4.8 shows a 4-input neuron with capacity in its memory to store 16 bits. It presents a neuron (a) and its four inputs *i0*, *i1*, *i2*, and *i3*. The neuron contents are stored in the 16 bits of memory (b), addressed by the inputs. *B0* to *B15* are the binary contents of the neuron and can be “0” or “1”. The four inputs (*i0*, *i1*, *i2*, and *i3*) form the address that points to a single bit in memory. This figure also shows how a neuron can be connected to four sensors (c) working with just 1-bit signals: these sensors can detect the presence of an obstacle, but are unable to tell how far away it is. Figure 4.8 (d) shows how the inputs *i0* and *i1*, and *i2* and *i3* are combined to connect the neuron to two sensors that work with two-bit signals: they are able to tell the distance from obstacles with a range of four levels. This illustrates once more the flexibility of the RAM neural network. In a simple application, the sensor module can make use of 1-bit sensors to find the direction of obstacles. For a more elaborate navigation task, where the robot needs to discriminate how far it is from the obstacles, the sensor modules can be modified to supply the neural



Input Address				Data
i0	**i1**	**i2**	**i3**	**Out**
0	0	0	0	B0
0	0	0	1	B1
0	0	1	0	B2
0	0	1	1	B3
0	1	0	0	B4
0	1	0	1	B5
0	1	1	0	B6
0	1	1	1	B7
1	0	0	0	B8
1	0	0	1	B9
1	0	1	0	B10
1	0	1	1	B11
1	1	0	0	B12
1	1	0	1	B13
1	1	1	0	B14
1	1	1	1	B15

Figure 4.8 – A 4-input neuron (a) and its 16 bits of memory (b). *B0* to *B15* are the binary contents of the neuron. The four inputs (*i0*, *i1*, *i2*, and *i3*) form the address that points to a single bit in memory. *S1*, *S2*, *S3*, and *S4* are the sensor outputs from the sensor module. The figure shows how to connect the 4-input neuron to four 1-bit sensors (c) or to two sensors with 2 bits (d).

network with 2-bit signals from each sensor output. Both variations can be connected to the neural network without modifications of the architecture, allowing different experiments to be performed with the same navigation control circuit.

Figure 4.9 shows a different configuration of the RAM neural network used in some of the experiments to implement the navigation control circuit of the robot. It consists of an n-tuple classifier that provides eight commands for the motor drive module [Aus88]. The network is formed by 64 neurons containing four inputs (eight groups with eight neurons per group) connected to eight 2-bit sensors controlled by the sensor module [Sim99]. This is one variation of the configuration of the RAM neural network architecture studied in this work, with a different number of neurons, neuron inputs, commands, and sensor outputs. Chapters 6, 7, and 8 present a series of experiments that explored different configurations of the same basic architecture.

Each sensor reading is converted into a 2-bit signal as shown in Figure 4.8(d). For each sensor, the sensor module can enable or disable this 2-bit signal, preventing the sensor to be connected to the input lines. Sections 4.3.2 and 4.3.5 describe in more detail how the sensor module performs this. The connectivity matrix is initialised randomly every time a new evolutionary experiment begins. The network output consists of eight commands to the motor drive module: *S - Stop*; *FS - Front Slow*; *FM - Front Medium*; *FF - Front Fast*; *TRS - Turn Right Short*; *TRL - Turn Right Long*; *TLS - turn Left Short*; and *TLL - Turn Left Long*. The *Turn Short* command means that the robot will turn with one wheel going forward and the other backwards. The *Turn Long* command makes the robot turn by stopping one wheel. The winner-takes-all block chooses the group of neurons containing more active neurons, according to the sensor inputs in each iteration, and selects the corresponding command. The resultant manoeuvre and speed values are determined by the motor drive module, that gradually increments or decrements the speed of the motors in each iteration, until reaching the selected level (i.e., *stop*; *slow*; *medium*; or *fast*).

Figure 4.9 presents only one of many possibilities of implementing the navigation control circuit of the robot. It is not the final version and was modified many times in the experiments reported in Chapters 6, 7, and 8, in addition, other alternatives will be presented when each experiment is introduced. Chapter 5 explains in more detail the hardware and software that were used to implement the navigation control circuit.

4.2 Evolutionary Control System

It is the evolutionary control system, located inside the central control module of the robots (see Figure 4.1 and Figure 4.2), that performs the evolutionary processes of evaluation, selection, and reproduction [Tod97]. All robots are linked by radio, forming a decentralised evolutionary system. The evolutionary algorithm is distributed among and embedded within the robot population. Figure 4.10 exemplifies a cyclic evolutionary process where the individuals are evaluated according to their capacity to perform the tasks in the environment. If they perform well, it can be said that they are

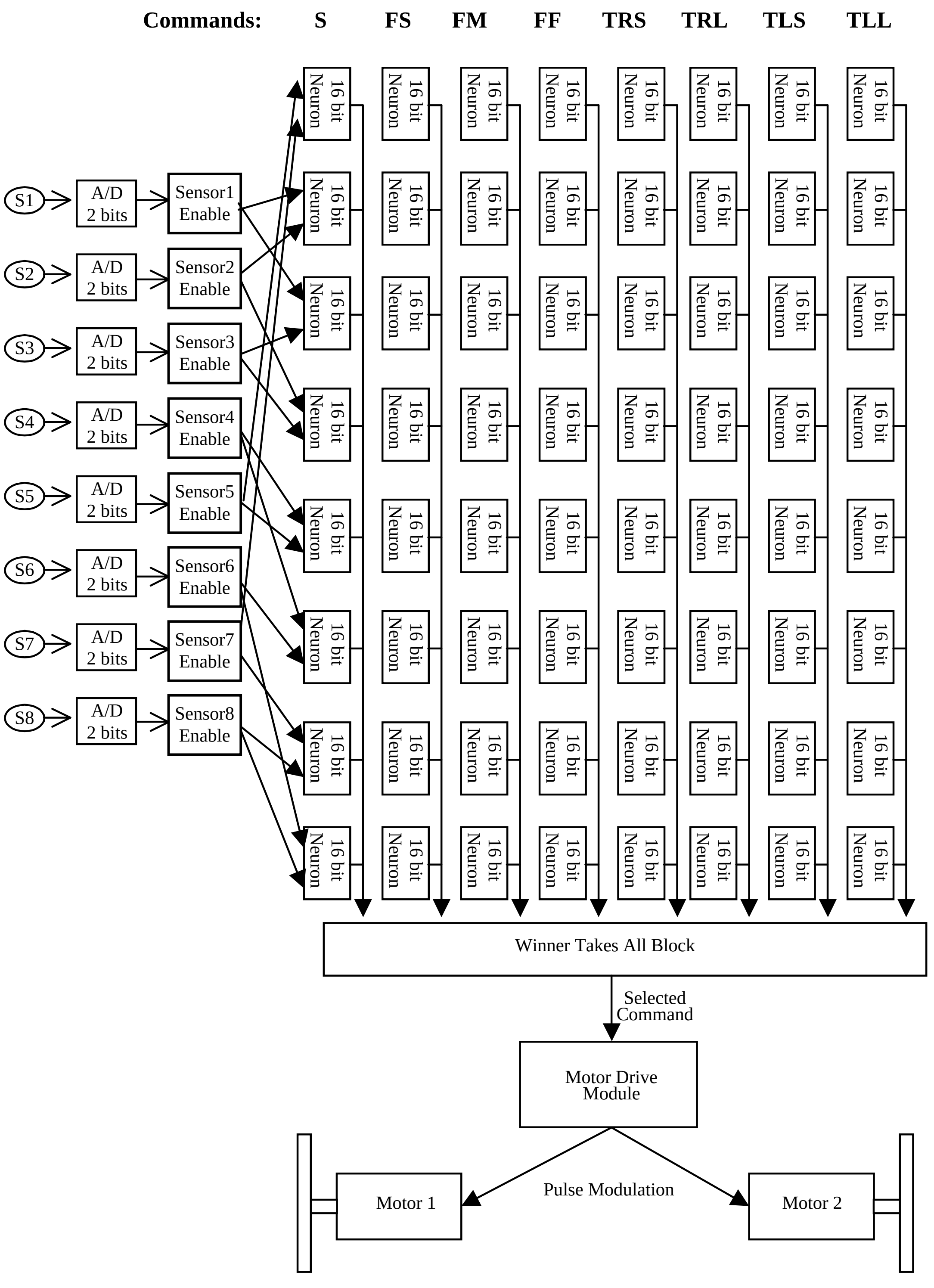


Figure 4.9 – A RAM Neural Network architecture controlling a 2-wheeled Robot: *S1* to *S8* are infrared sensors; *S*, *FS*, *FM*, *FF*, *TRS*, *TRL*, *TLS*, and *TLL* represent the groups of neurons of the classes *Stop*, *Front Slow*, *Front Medium*, *Front Fast*, *Turn Right Short*, *Turn Right Long*, *Turn Left Short*, and *Turn Left Long*.

well-adapted to (or fit for) the environment [Mit95]. The robots are assigned a score, or fitness value, that tells how fit they are. When the evaluation period is over, the individuals

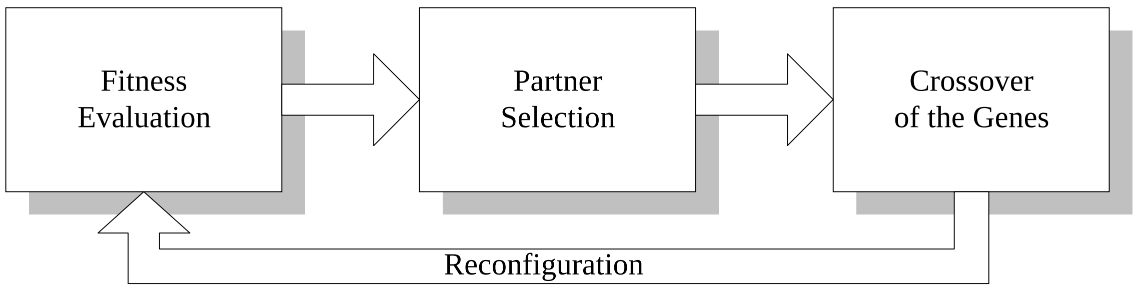


Figure 4.10 – An evolutionary process of evaluation, selection, and reproduction (or crossover).

select a partner to mate with according to their fitness value. The best individuals have more chance of being selected to breed. Next, they exchange their chromosomes, crossing over their genes to form the new combinations. The resultant chromosomes are then used to reconfigure the old individuals, originating new ones, or the offspring. Then, a new evaluation phase starts again. Assuming that new robots cannot really be created spontaneously, the offspring must be implemented by reconfiguring selected old individuals.

An evolutionary process, in the context of this work, is the procedure necessary for the development of suitable controllers for the population of robots. The process can stop when the average fitness value of the population reaches a specified threshold or continue indefinitely while the robots execute a certain task. In the developed evolutionary system, the robots work in a cyclic procedure, differently from a traditional design technique, where the controller is designed or trained at first and then transferred to the robot that is put to work. This cyclic procedure is inspired by the natural world where animals, like some birds for example, have a working or foraging season and a mating season, where they concentrate their attention in finding a mate and reproducing [Mit95].

The cyclic procedure of the robots, a *generation* in evolutionary computation terms [Bac91], is exemplified in Figure 4.11. The robots do not pursue reproductive activities concurrently with their task behaviour. Instead, they perform a working season, where they execute the selected task in the environment (or working domain) and are evaluated according to their performance. The internal timer of Robot 1 indicates the beginning of the mating season. It is important to observe that the evolutionary scheme is decentralised and distributed amongst all six robots. Robot 1 is by no means dominant in this process. The internal timer of Robot 1 is just used to signal the others, indicating the beginning of the mating season. It was necessary to avoid

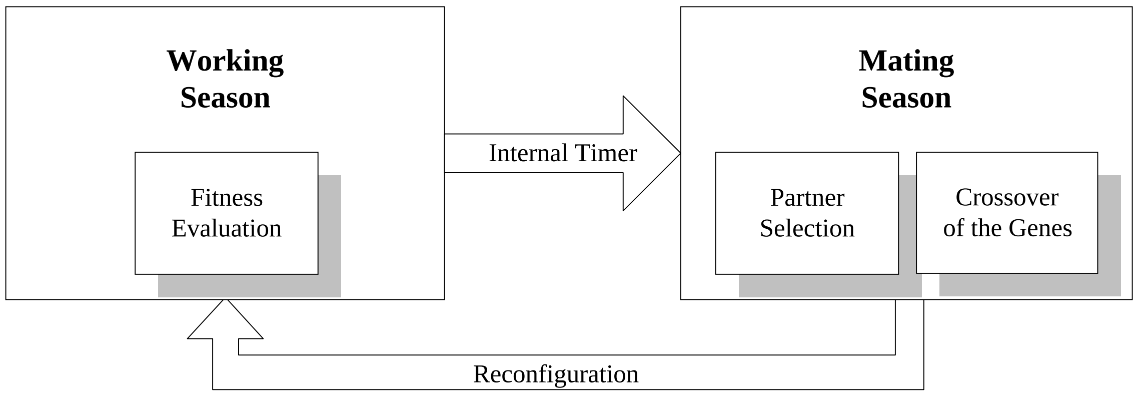


Figure 4.11 – The cyclic procedure of the robots, or a generation.

synchronisation problems, since it was impossible to guarantee that all robots would begin the mating season at the same time. In the mating season, the robots communicate to let the others know their fitness value. They start emitting a “mating call”, where they “shout” their identification, their fitness values, and chromosomes. The best robots survive to the next generation, breeding to become the “parents” of the new individuals [Nol94]. The less well-adapted robots recombine their chromosomes with the better-adapted ones, reconfiguring their parameters as a new robot before starting a new generation.

The robot recurring procedure for one generation, shown in Figure 4.11, works according to the following algorithm:

- **Working Season:**
    1. Avoid obstacles;
    2. Count collisions;
    3. The internal timer of Robot 1 indicates the beginning of the mating season;

- **Mating Season:**
    1. Robot 1 orders all robots to stop;
    2. Robot 1 sends a mating call via the radio (containing its identification, fitness value, and chromosome);
    3. Robot 2 then sends its mating call and so do all other robots, one after the other, until the last one;
    4. All robots listen for mating calls, receiving every fitness value, comparing with the others, and then selecting the partner to mate with (if own fitness is the highest, the robot does not breed);
    5. When all genes are received and partners chosen, start Crossover;

6. Begin reconfiguration with the resultant chromosome and wait until…
7. Robot 1 announces the end of the mating season and orders all robots to start another cycle.

The process begins with the random initialisation of the robot chromosomes. Then, the first generation starts with all robots performing their tasks in a working season. For the case of obstacle avoidance, they will navigate and have their fitness value calculated according to a function similar to the one presented in Figure 4.12. Robot 1 has control over the duration of the working season and uses the radio to stop the other robots when its internal timer reaches the end of the working season or the "lifetime" of the robots. That is important to synchronise the cycle and make sure that all robots will stop working at the same time. Starting with Robot 1, each robot transmits, one by one, a mating call via the radio, containing its identification, fitness value, and chromosome. When they are not transmitting, the robots listen for other mating calls, receiving the fitness value from the call, comparing it with the others, and then selecting the optimal partner with which to mate. If own fitness is the highest, the robot does not breed and "survives" to the next generation. The cycle will be completed when all robots find a partner to mate with and combine their genes in the crossover phase. The mating season lasts until all the six robots signal Robot 1 that they have found a partner, have mated, and have reconfigured themselves with the resultant chromosome. Robot 1 then orders them to restart another cycle (once more Robot 1 is used only to synchronise the next phase). In other words, the best-adapted robots "survive" to the next generation, while the others "die" after mating, to lend their bodies to their offspring.

4.2.1 Fitness Evaluation

A simple obstacle avoidance task was chosen. The limited complexity of this task allows a good evaluation of the fitness in a short period. A complex task requires more time so that the robots can be subjected to more challenges in order to show that they can perform well in more than specific situations. A smaller generation time means a faster evolution, because more combinations of solutions will be tried [Pol00].

A reward-punishment scheme is applied during the fitness evaluation process, executed by the supervisor algorithm (see Figure 4.2). Each robot is evaluated during the "*working season*", where its fitness function is calculated by penalising collisions and lack of movement (reducing the fitness value), encouraging the exploration of the environment (rewarding by increasing the fitness value for every second of movement). A major issue that must be addressed is how to detect a good (fit) robot. This question may be highly complex in nature [Har93a], but in the context of evolutionary programming, it can be simply defined by the programmer, in accordance with the particular problem at hand. Nevertheless, defining the rule for fitness evaluation is a crucial phase in evolutionary programming, since evolution proceeds without human intervention, relying on this rule to select the best individuals, and may produce a solution different from the expected [Ste94]. Furthermore, writing a fitness function depends on the targeted behaviour and the characteristics of the robot, and the necessary insights are gained through incremental augmentation over many trials in the environment.

For the obstacle-avoidance problem, a simple rule can be applied: a robot will increase its fitness each time it comes across an obstacle and successfully avoids it. Each time it collides, the fitness will be decreased. Figure 4.12 shows an example of a fitness function where the robot fitness is increased by one for every second the robot is in movement, encouraging exploration. It is punished by decreasing its fitness by ten when it collides. The fitness is also decreased by 100 to punish the robot for turning for more than five seconds. Therefore, this sub-function prevents a particular efficient solution that kept the robot spinning in a small circle within an obstacle-free area.

In a situation where an obstacle is close to the robot, but the proximity sensor readings are not interpreted correctly by the navigation control or are not enabled by the sensor module, a collision may occur and the fitness variable will be decreased. The bumper sensors will then be analysed by the supervisor algorithm to calculate where the collision took place in a total of 12 sectors with 30 degrees each (Section 4.3.5 explains this in more detail). Once the place of collision is detected, a rescue routine will drive the robot away from the obstacle, returning the navigation control to the neural network. When the robot is moving forward without colliding with obstacles, its fitness will be increased every second.

From the experience of a number of trial-and-error experiments, it was found that a simple fitness function usually produces the best results. This is because it

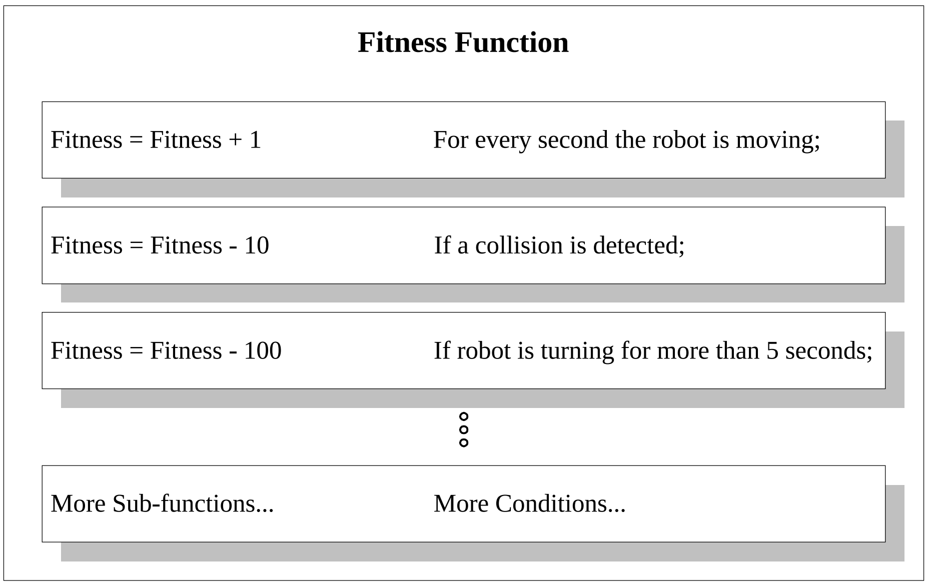


Figure 4.12 – An example of how a fitness function can be constructed.

does not eliminate the autonomy of evolution [Flo96a]. As more complex behaviours are evolved, the designer has a tendency to gradually add sub-goals to the fitness function. This strongly biases the possible solutions [Mat96]. This approach, although effective, reduces the search space of evolution and fails to facilitate the work of the designer.

4.2.2 Partner Selection

The procedure for partner selection is based on the robot fitness value or score. Whereas biologists try to analyse the selection mechanisms they believe exist in the natural world, artificial evolution seeks inspiration in nature to propose novel selection mechanisms [Har93a]. They can be very simple, like choosing the best robot to mate with all other ones, or more complex, such as the roulette-wheel technique (see Chapter 2) [Mit95]. In this work, the simple approaches are preferred because of the restrictions of an embedded controller (i.e., low processor speed and small memory size – there is a limit to how much memory the processor can address). These present a limit of what can be implemented on-board the robots. As the population is very small, a simple technique can deal with the robot selection without problems. Therefore, some simple, but efficient selection techniques were developed, and will be described in more detail when they are tested in the experiments in Chapter 6. These techniques are:

1. Select the robot with the highest fitness value in the generation to breed with all other robots and survive to the next generation. This tries to make sure that in the next generation the best fitness will be at least similar to the present one.
2. The fittest robot survives and the others choose their partners giving 80% chance of selecting the fittest robot, and 20% chance of selecting any other but itself.
3. An "*Inheritance*" scheme was developed: the score used to select the robot is the average of the robot fitness in the last five generations (i.e., inheriting the scores of its previous generations). The robot with the best average survives, but only breeds with the robots with the fitness in the present generation lower than its own fitness. This approach protects new robots that are actually better than the one with the highest average, but need more generations to be selected by their average.
4. Another very simple strategy that was effective in the experiments is to select the fittest robot, allow it to survive, and reconfigure all the others with a small variation (mutation) of its chromosome. This is a form of "asexual reproduction", where the robots do not cross over their chromosomes. All robots in the next generation will be a copy of the best one, but will suffer random changes (mutation) in a few genes. This has come to be called *Naive Evolution* by the GA community [Har93a]. This strategy has been changing the opinion of most GA researchers that emphasise the importance of crossover [Tom95].

All techniques suggested above are *elitist*. Elitism requires that the current fittest member (or members) of the population is never deleted and survives to the next generation [Tom95]. The developed inheritance scheme prevents a robot from being deleted even if it is not the fittest, but has the biggest accumulated average fitness value. It is the only selection technique where the fittest member of the population does not have the same number of offspring (or the same probability to have offspring) whether it is far better than the rest, or only slightly better. In the other ones, it will always have the same probability to have offspring; and in most of them, will breed with all other robots. This approach is often too severe in restricting exploration by the less fit robots [Bli96].

These techniques are better explained in Chapter 6, that shows many experiments performed with variations of the solutions presented above. A common problem with these techniques is the possible appearance of a super fit individual that can get many copies and rapidly come to dominate the population, causing premature convergence to a local optima [Tho94c]. This can be avoided by suitably scaling the evaluation function, or by choosing a selection method that does not allocate trials proportionally to fitness, such as ranking selection and tournament selection [Har93a]. This work, however, does not experiment with these methods. Instead, another solution will be developed to attack this problem (see Section 6.5).

4.2.3 Crossover Strategy

The crossover is the phase in the evolutionary algorithm where the chromosomes of both parents are combined to produce the offspring [Tom95]. Many techniques are proposed in the literature to implement the crossover phase [Hig96a] [Lan96] [Hem97] [Lan97]. Nevertheless, this work uses a very simple strategy, because of the restricted resources of the embedded controller.

In the developed evolutionary system, both morphological features and the controller circuit are evolved to respond to changes in the environment. The robots constantly adapt to changes in the surroundings by modifying their features and the contents of the RAM neural controller. The term “morphology” is defined as the physical, embodied characteristics of the robot, such as its mechanics and sensor organisation [Lun97]. In the experiments described in Chapter 8, the morphological features modified by evolution are the number and position of sensors, as well as the speed levels of the drive motors. Therefore, the genetic material specifies the configuration of the robot control device and morphological features, as shown in Figure 4.13. The control device is implemented within the robot microprocessor (a neural network for navigation control) and two programmable modules control the robot features, which are the sensor module and the motor drive module (see Figure 4.1).

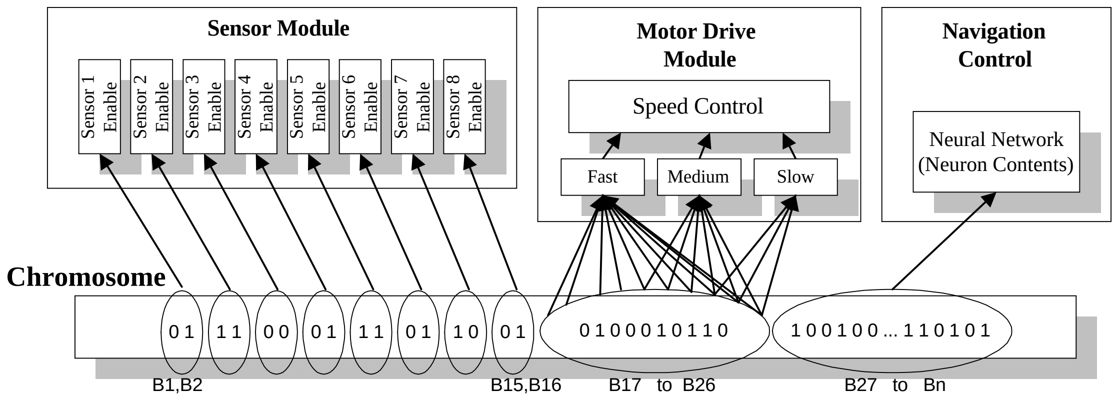
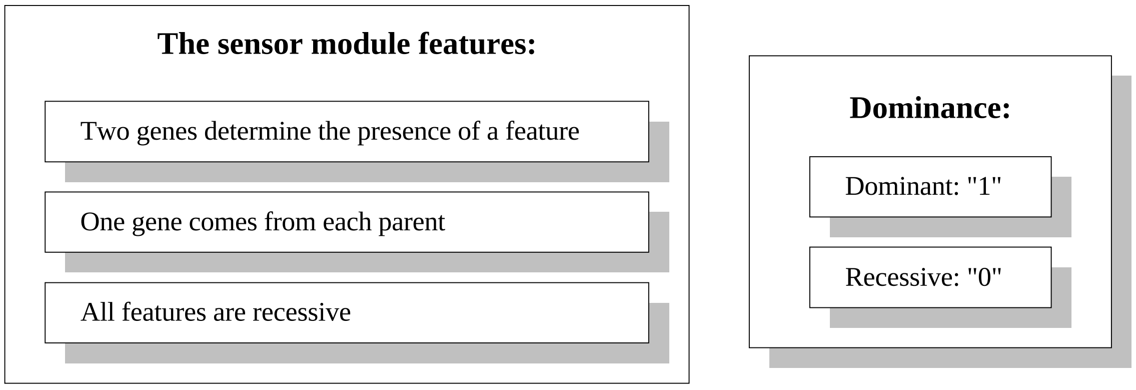


Figure 4.13 – An example of how the genes in the chromosome are used to configure the sensor module (eight pairs of genes: *B1*, *B2* to *B15*, *B16*), the motor drive module (ten genes: *B17* to *B26*), and the navigation control (neuron size × number of neurons: *B27* to *Bn*).

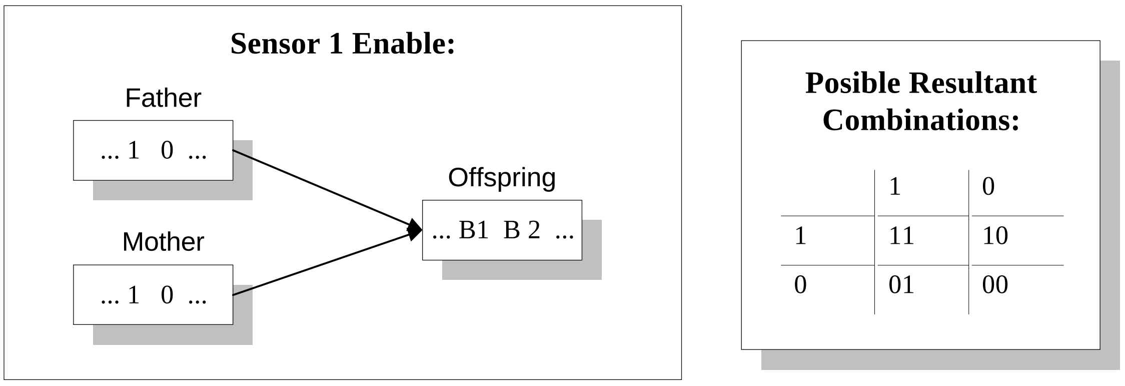
For the selection of the robot features controlled by the sensor module, a more complex "dominance approach" was implemented to combine the eight pair of genes [Ler76]. Each sensor in the sensor module is configured by two genes (two bits – *B1*, *B2* to *B15*, *B16* in Figure 4.13) in the chromosome: *i)* two genes will determine the presence of a feature ("enable the sensor"); *ii)* one gene comes from each parent; and *iii)* all features are recessive [Cam99]. The two genes are coded using bits in such a way that the combinations "1,1", "0,1", and "1,0" disable the sensor, and "0,0" enables it (see Figure 4.14).



The sensor module features:
Two genes determine the presence of a feature
One gene comes from each parent
All features are recessive
Dominance:
Dominant: "1"
Recessive: "0"

Figure 4.14 – The strategy to select the sensor module features. As all features are recessive: only the robots containing "0,0" in their respective positions in the chromosome have the corresponding sensor enabled.

An example where two robots mate using the developed strategy to select the sensor module features is shown in Figure 4.15. Considering only the feature that enables or disables sensor 1, the father's chromosome has *B1* = "1" and *B2* = "0"; the mother’s chromosome has the same genes, *B1* = “1” and *B2* = “0”. The strategy combines the chromosomes to produce the new *B1* and *B2* for the offspring. One of the two genes of the father and mother is randomly selected. By crossing such individuals, four possible combinations can be produced in the genotype, each with a 25% probability of occurring. Table 4.2 shows the probability of the phenotype, where the resultant robot has only a 25% chance of having the feature enabled. This is a way of introducing neutrality to the genes that select the sensors [Ler76] [Shi00] (i.e., different genotype configurations produce the same phenotype).



	1	0
1	11	10
0	01	00

Figure 4.15 – An example of the possible combinations that can result from the crossing of two parents containing both the dominant and the recessive genes.

Table 4.2 – Probability of a Phenotype to be present.

Phenotype	**Probability**
Do not have the feature	75%
Have the feature	25%

The motor drive module has ten bits associated with it in the chromosome. The features selected by the module are the three different speed levels for the motors. Figure 4.13 shows that the three speed levels, *Fast, Medium,* and *Slow* are controlled by these ten bits. The contents of these ten bits (B_{17}, B_{18}, B_{19}, B_{20}, B_{21}, B_{22}, B_{23}, B_{24}, B_{25}, and B_{26}) are added together to form a decimal number that indicates the speed level *Fast*. The result is a number between zero and ten that indicate the level of the *Fast* speed. The contents of the first six bits (B_{17}, B_{18}, B_{19}, B_{20}, B_{21}, and B_{22}) are added together to form a decimal number that indicates the speed level *Medium*. The result is a number between zero and six. In the same way, the contents of the first three bits (B_{17}, B_{18}, and B_{19}) are added together to form the decimal number that indicates the speed level *Slow*. The result is a number between zero and three. Therefore, as they use the same bits, this guarantees that the level *Fast* is always greater than or equal to *Medium*, which is always greater than or equal to *Slow*.

The resultant speed for the three levels is converted by the motor drive module to a value (an internal parameter) between 1 and 32, because the robot motor can have 32 speed levels. This conversion is not linear, because of the way the motors are controlled by pulse modulation, as it will be explained in Chapter 5, and the corresponding values are shown in

Table 4.3. This approach permits co-adaptation where the chromosome integrates specifications for both controller and morphological features [Lun97]. Evolution can select not only the number of sensors to use, but, if the number of sensors is fixed, it can select which ones to pick (i.e., the sensor position on the robot).

Table 4.3 – Conversion of the speed levels of the robot.

Speed Level:	0	1	2	3	4	5	6	7	8	9	10
Value (internal parameter):	1	7	9	11	14	17	20	23	26	29	32
Velocity (m/s):	0	0.02	0.05	0.08	0.1	0.13	0.15	0.17	0.2	0.23	0.26

For the genes that control the neural network and the motor drive module, a random exchange of the genes from the parents is used to form the resultant chromosome. This strategy is called *uniform crossover* [Lan96], although here only one offspring is produced. Therefore, a gene is selected from the father or the mother to occupy the corresponding position in the offspring chromosome. Considering the configuration given by Figure 4.9, a random exchange of the 1034 genes (64 neurons with 16 bits each, plus 10 bits to select the motor drive module) from both parents occurs after a dominance selection of the 16 bits that enable the sensors.

After the crossover is completed, a mutation phase starts. Applying mutation to a chromosome means that a small number of copying errors may occur when copying the genes from the parent chromosomes to the offspring [Och99a]. In this work, a mutation rate of *M*% means that each gene in the chromosome has a probability of *M*% of being selected and binary inverted (e.g., new gene = *NOT*(gene)). Therefore, for each bit in the chromosome a random real number *r* is generated between 0.0 and 100.0, if $r < M$ flip the bit. Small mutation rates, usually between 1 and 3%, are the ones that produced the best results in the experiments. Higher mutation is only useful in the beginning of the evolutionary experiment. A high mutation rate does not help to evolve faster, and did not prove to be a good strategy. Although it may help in the beginning of the process to bring more variety into such a small population (e.g., six robots), it slows down the process after the initial genetic material is combined. Therefore, in the long term, it produces a very uneven population, where the fittest individual is considerably distant from (better than) the average fitness of the population. Generally, only a few robots will be well adapted to the environment.

A low mutation rate makes the whole evolutionary experiment much faster [Och99a]. It produces a population with limited variation where the average fitness is almost as high as the fitness of the fittest individual. There are two reasons for this: *i)* late in the evolutionary experiment, the population that is combined to produce the next generation will present a high fitness value if the mutation rate is small. When such a converged population mate, it is more likely to produce a fit individual than a less fit one; *ii)* by applying a small mutation to a chromosome containing a majority of good genes, there is a chance that the few bad genes alone will be modified, thus producing a better offspring (i.e., if a high mutation modifies many genes in a chromosome that has more than 50% of good genes, the chance of changing good genes to bad ones is greater than changing bad to good ones). A low mutation rate is also important if the robots are working as a group and a bad performance can severely affect the overall operation [Har93c].

A variable mutation rate would theoretically make use of a high rate to speed up evolution until a certain “fitness level” is achieved, and then reduce mutation in order to increase the average fitness and produce a more balanced population [Har93a]. However, it has not been possible to find a way to detect when that “fitness level” is achieved in the embedded controller from the robot point of view. Therefore, there is no way of knowing how close to the final result the robot is, because that result is unknown and may not be a unique solution for the problem. It was expected that a premature drop of the mutation rate could cause the population to be stuck in a local optima for a longer time [Har92], but that did not happen experimentally. Instead, fixing a low mutation rate from the beginning of the process actually made the whole evolution a lot faster (see Chapter 7). The ideal mutation rate turned out to be the one that has the greatest chance of modifying only one of the genes in the chromosome (e.g., 1% for a chromosome containing 100 genes, 0.1% for a chromosome with 1000 genes, and so on).

4.3 System Specification

This section presents in a general way how the robotic system should be implemented to work with the developed evolutionary system. Chapter 5 describes how the system was designed in terms of hardware and software in the lowest details. The ideas described here, in principle, can be adapted to work with other robots, in different environments.

4.3.1 Overview

The mobile autonomous robots form a decentralised embedded evolutionary system where no host computer is required. Nevertheless, an IBM PC is used to monitor all data exchanged via the radio link, producing a complete record of the chromosomes, parameters, and variables for every generation.

Figure 4.16 shows a perspective of the working domain containing the environment where the robots work and the monitor computer, connected to the radio link to monitor the communication among the robots. The robots can communicate with each other and the monitor computer via an asynchronous serial data link using their communication module. The computer monitors the robot internal variables without interfering in the system, but has the capacity to start or stop an evolutionary experiment.

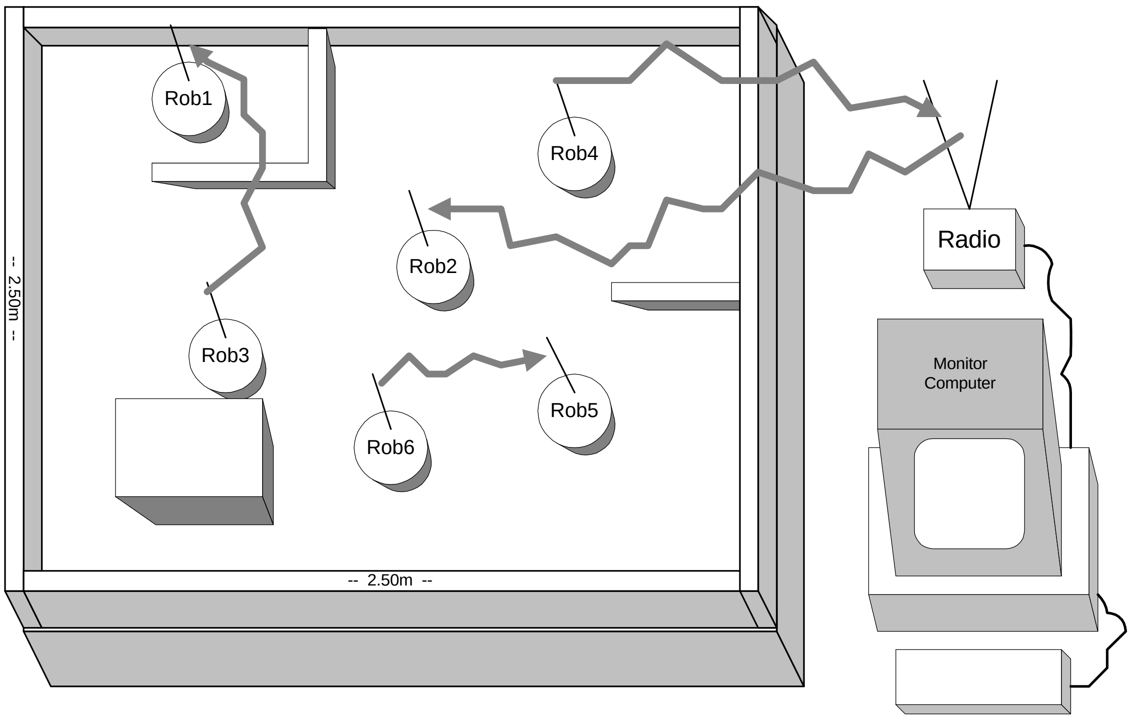


Figure 4.16 – Working domain containing the environment where the robots work, as well as the monitor computer connected to the radio link.

A radio board is connected to the monitor computer, which logs data on the evolutionary experiment. It is a multi-channel driver/receiver interfacing the IBM PC via its serial port. A software interface permits the downloading of software, data, and commands between the robots and the computer. Programming and low-level debugging are provided by the computer. When programming or debugging, bi-directional data between the robot processor and the computer can operate either via a wired link or via radio. Chapter 5 shows more details of the software and hardware involved. Figure 4.17 shows the monitor computer connected to the radio and Figure 4.18 shows a close-up of the radio board that is used by the monitor computer to communicate with the robots and monitor the evolutionary experiment.

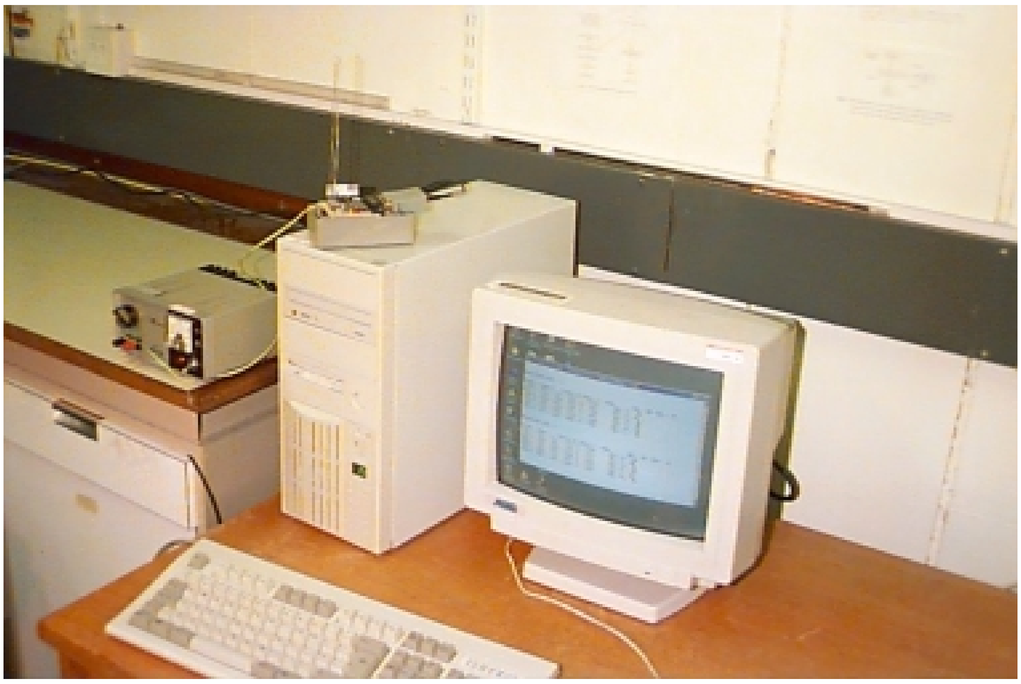


Figure 4.17 – The monitor computer connected to the radio.

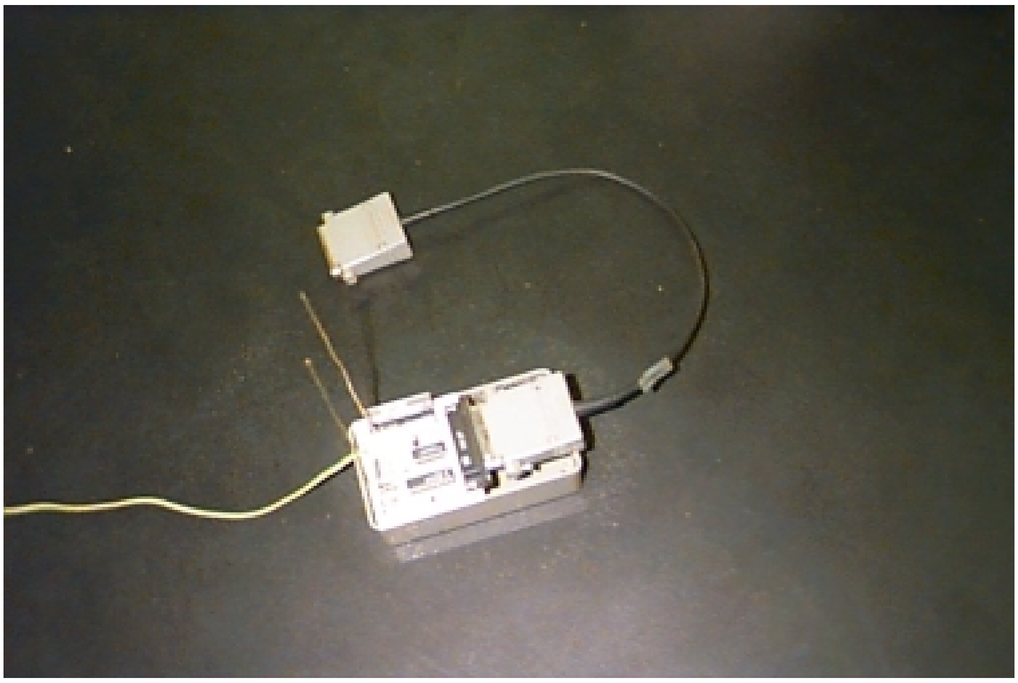


Figure 4.18 – A close-up of the radio board.

The experiments were performed within a 2.50m × 2.50m working domain, containing walls and obstacles of varied sizes, where the robots can explore the environment, avoiding collisions. Many movable obstacles and internal walls of different sizes are available to change the scope of the workspace where the robots navigate. Figure

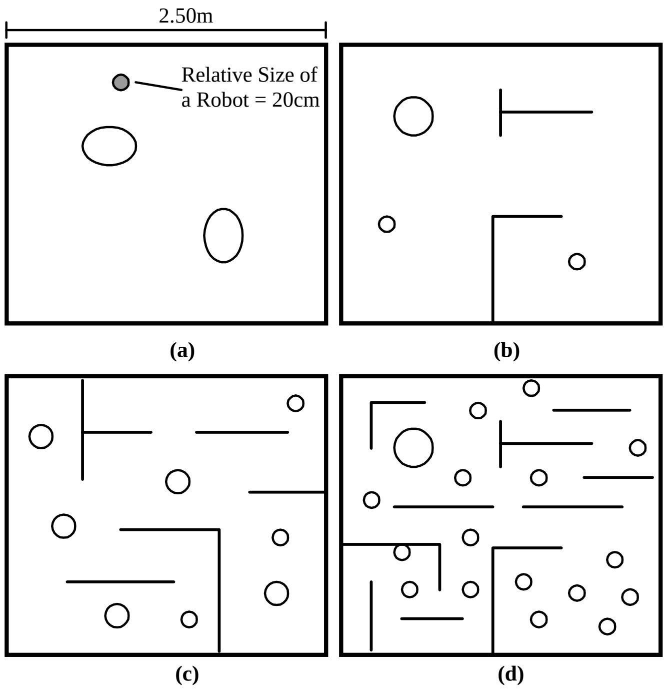
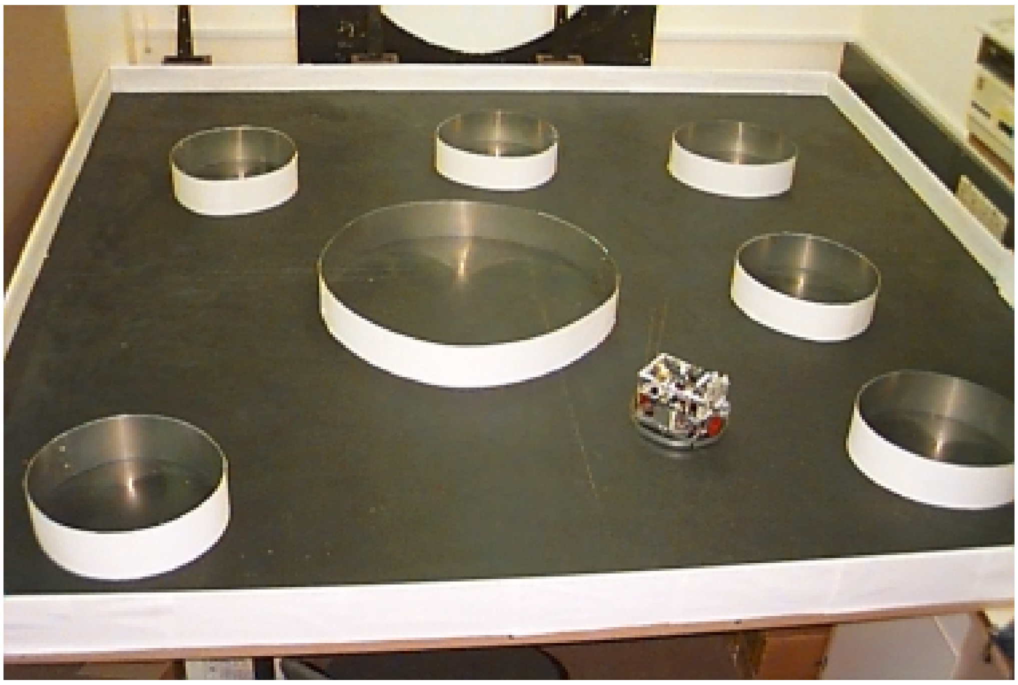


Figure 4.19 – Four different configurations of the 2.50m × 2.50m workspace showing how the environment can be modified for the experiments, from simple (a) to very complex (d).

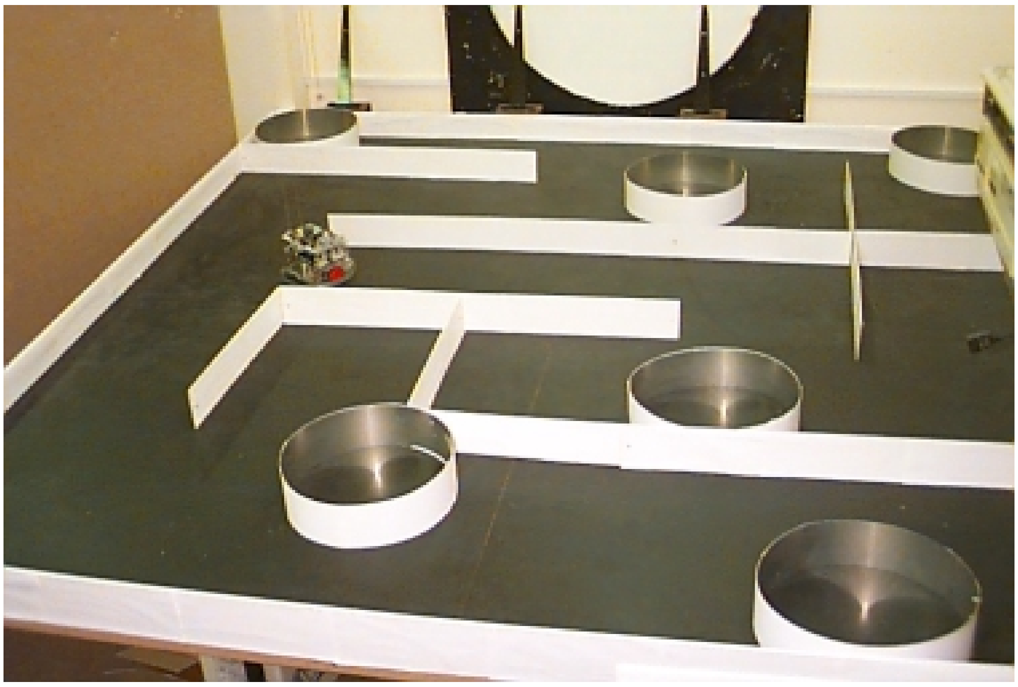
4.19 shows how the workspace can be modified into four different configurations by rearranging the obstacles and walls. This flexibility is necessary to allow different degrees of complexity of the environment during the experiments. Figure 4.20 shows different configurations of the workspace representing a simple (a) and a complex (b) environment.

4.3.2 Introduction to the Robot Architecture

The robot architecture consists of a two-wheel differential-drive platform (20cm diameter), containing a Motorola 68HC11 - 2MHz, 64Kb of RAM [Mot96], bumpers with eight collision sensors, and eight peripheral active infrared proximity sensors. It exchanges information with the other robots at 1.2Kbps by a 418MHz AM radio. Both robots and workspace were specially built for the experiments.



(a)



(b)

Figure 4.20 – An example on how simple (a) and complex (b) environments can be produced by rearranging the obstacles and walls.

Figure 4.21 shows a plan view of the base, where the position of the infrared proximity sensors, the four bumpers, and the wheels are illustrated. All eight proximity sensors are connected to the sensor module, which is configured by the chromosome. The module can individually enable or disable the sensors, changing the number of active

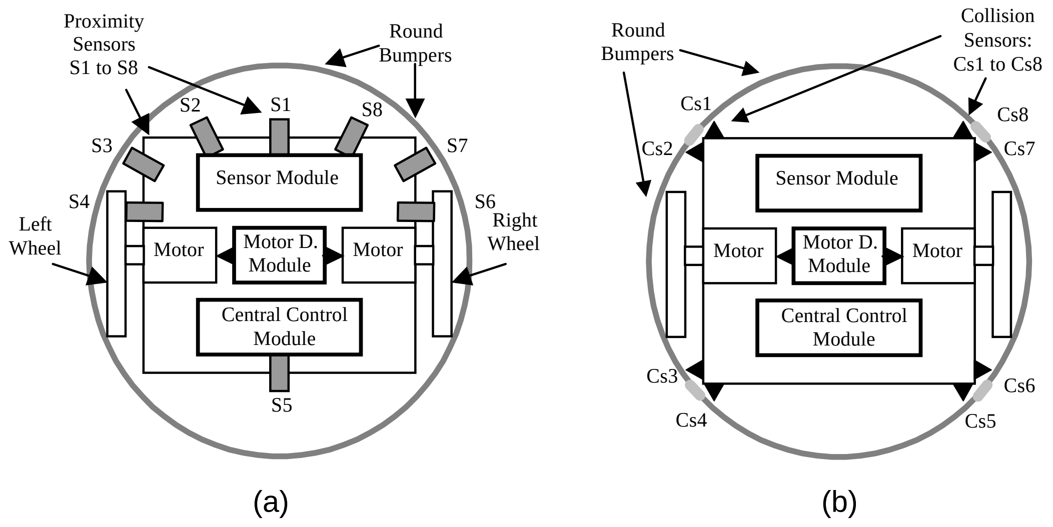
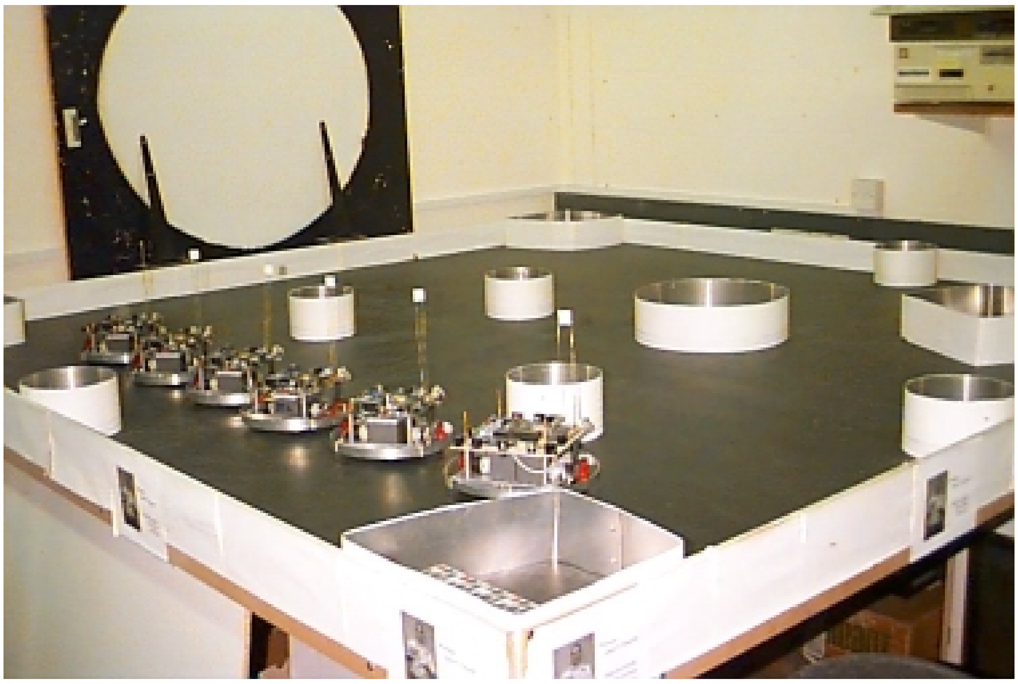


Figure 4.21 – Robot architecture (a) representing the position of the proximity sensors and motors, the Central Control, the Motor Drive, and the Sensor Modules. A round bumper (b) with collision sensors surrounds the robot.

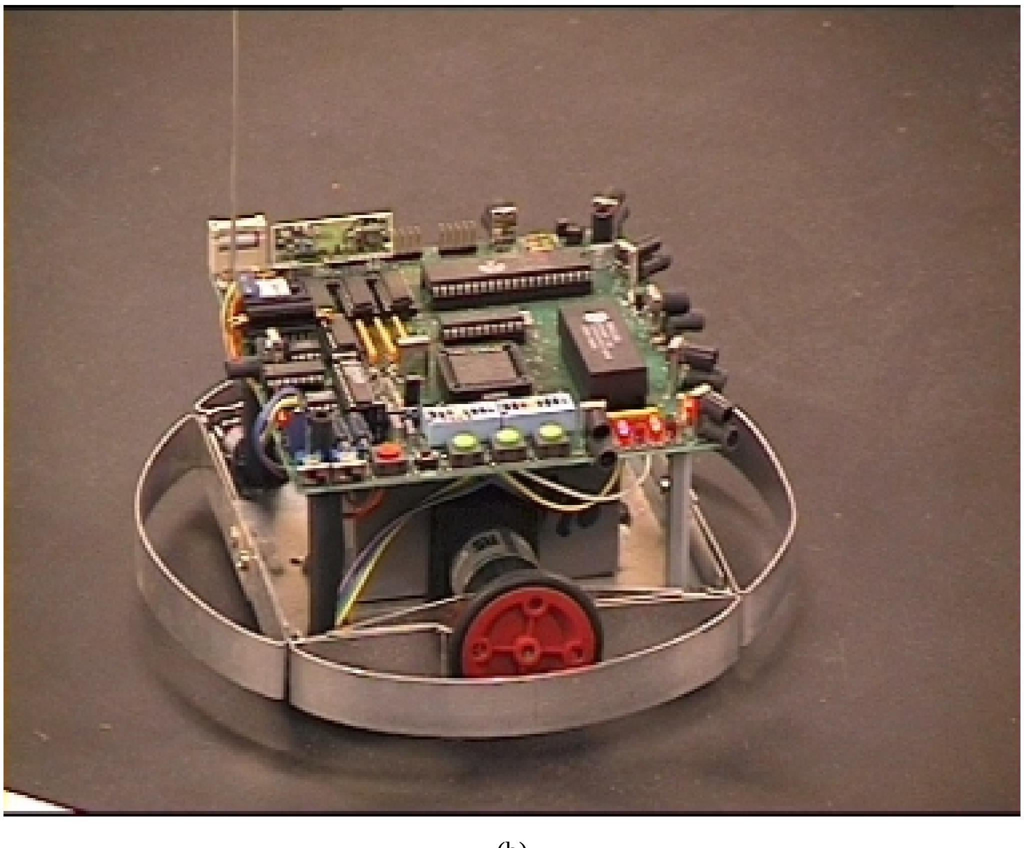
sensors and, consequently, their position in each generation. Therefore, the physical, embodied characteristics of the robot can be modified. In evolutionary terms, these constitute the morphology of the robot [Lun97]. The wheels are placed in the middle of the robot, allowing it to turn around its central axis. The robot has four round bumpers on its front, back, left, and right, which are attached to the base by eight contact sensors (*Cs1* to *Cs8*). These sensors permit the supervisor algorithm to pinpoint the location of a collision.

Figure 4.22 (a) shows the team of robots parading in their workspace. Robot 1 is a prototype and is larger than the others. Figure 4.22 (b) shows a top-view of Robot 2, displaying its infrared proximity sensors and round bumpers. Each robot has a banner identifying it by displaying its number. The three green keys on the right form a small keyboard used to enter commands to the robots manually.

The robot architecture is composed of modules that interface each other and can be combined with other circuits to allow new features, such as a different chassis, power supply, actuators, or even a new communication system to be installed. Figure 4.23 shows how the five units are packed together. The base contains the communication system, the computing system, the sensor pack, the power management pack, and the motor drive pack. The communication system contains the radio link, which allows serial



(a)



(b)

Figure 4.22 – A view of the robot team (a) and top-view of Robot 2 (b), showing the position of the infrared sensors and bumpers.

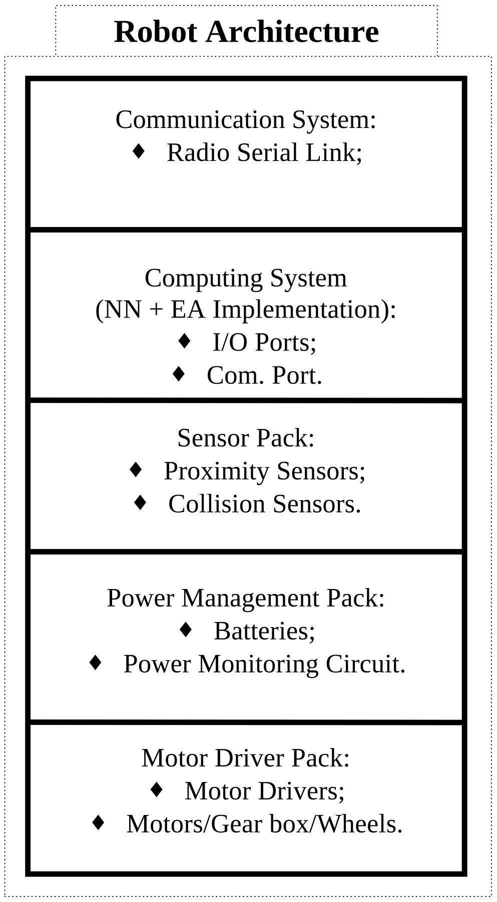


Figure 4.23 – The robot modular architecture, containing the communication system, the computing system, the sensor pack, the power management pack, and the motor drive pack.

communication among the robots and the monitor computer. The computing system contains the neural network of the navigation control and the evolutionary control system. It accesses the sensors through I/O ports and uses the COM port to interface the radio. The sensor pack groups the infrared proximity sensors around the periphery of the robot and the contact switches attached to the bumpers. The power management pack is basically composed of the voltage monitoring circuit, which can switch off the microprocessor, the sensors, radio, and the motors to protect the data in memory until the battery is replaced. The motor drive pack contains the high-current drivers and controls the pulse signals, which drive the motors.

The centre of the robot architecture is its microprocessor, the Motorola 68HC11 [Mot96]. Most of the robot functionality is implemented in software and executed by the microprocessor. The sensors and motor drives are connected to the microprocessor

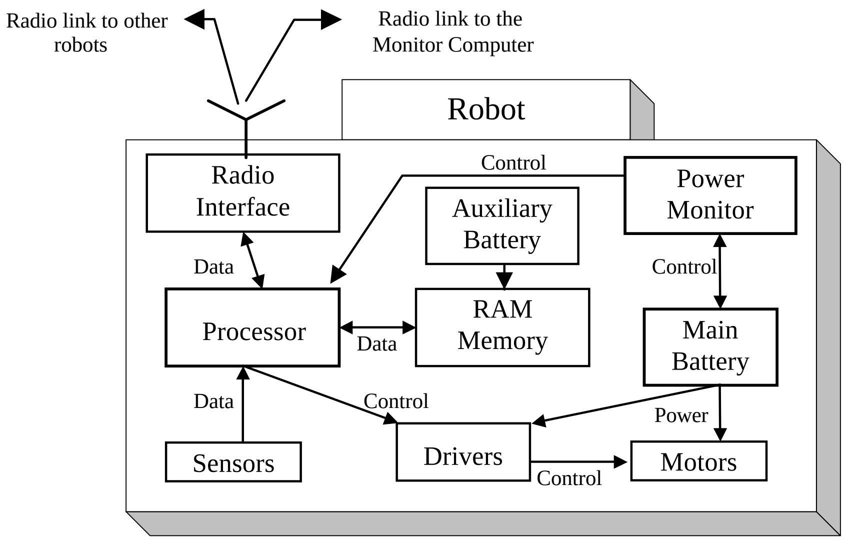


Figure 4.24 – Components of the robot architecture, showing how the microprocessor interfaces to the other units. The power monitor can shut down the processor if the main battery voltage goes low.

through I/O ports and the radio through a RS232 serial port. The RAM memory, which stores the program and data, has a small internal backup battery. A power monitor circuit supervises the condition of the main battery and shuts down the microprocessor if the voltage drops below a threshold, stopping all operations to protect the data stored in memory. The program counter and all other internal variables are preserved and the robot can resume its operation as soon as a new battery is replaced.

Figure 4.24 shows the components of the robot architecture. It displays the main data and control lines among the units. The microprocessor receives information from the other robots and the monitor computer, and specifies the internal configuration of the neural network, which will control the motors according to the sensor readings.

4.3.3 Computing System

The robot modular architecture permits different combinations of the modules with other hardware and software, so that new robot architectures can be generated. It was conceived in this way, because it allows future work to be carried out by

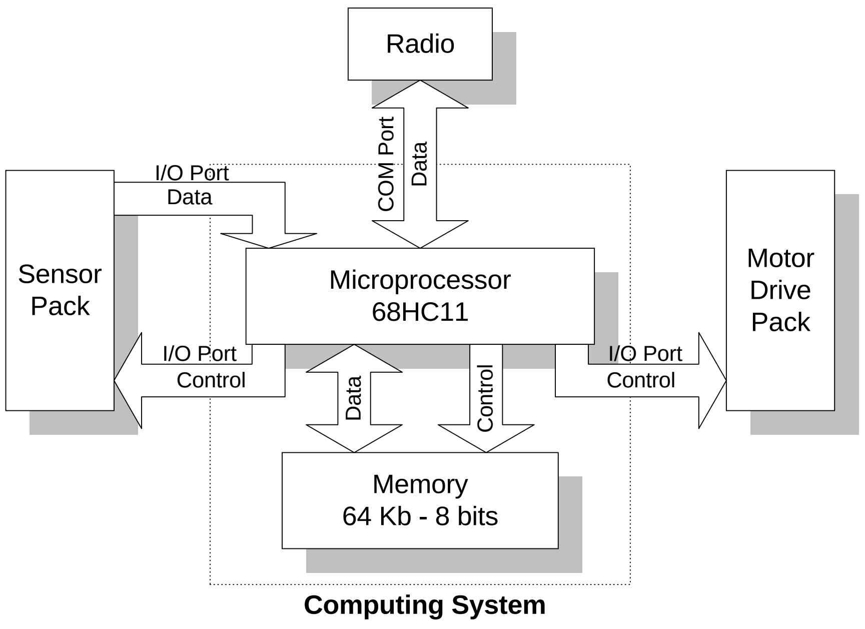


Figure 4.25 – Diagram of the computing system, showing how the processor interfaces the memory, the radio, and the sensor and motor packs.

modifying the existing robots to perform different, more complex tasks. Therefore, the computing system consists of a processor unit and memory, interfacing the other units throughout input/output ports (I/O ports) and a serial COM port. Figure 4.25 shows a diagram of the computing system and its ports.

The microprocessor and memory are used to implement the software algorithms of the central control, the communication, the sensor, and the motor drive modules. Therefore, the computing system takes part in the implementation of the software of every module. For the central control module, it implements the algorithms of the navigation control (the neural network functions), the supervisor algorithm (the evaluation of robot performance), and the evolutionary control (the coordination of the evolutionary process). For the sensor module, it controls the infrared pulses, calculates the sensor readings, and selects which sensors are active according to the chromosome. For the motor drive module, it interprets the command from the neural network and generates the pulse-width signals that will control the motor high-current drivers. For the communication module, it produces and identifies a series of commands and the standard RS232 signals necessary to establish the communication protocol with the other robots and the monitor computer.

- **Computing Module Requirements**

The main concerns when designing such a system are: *i)* will the microprocessor be fast enough to perform all subroutines of the modules and still manoeuvre the robot smoothly and with good clearance from obstacles? And *ii)* will the memory size be large enough to store all programs and variables? Basically, to guarantee a smooth control of the robot speed and direction, the navigation controller must provide at least 100 iterations a second. It means that the processor must be able to perform a sensor reading, choose the appropriate command, and control the motors accordingly in less than 10ms. It needs to execute not only these routines, but also the higher-level layers of fitness evaluation and radio monitoring.

The available memory needs to be enough to contain the programs of the modules and store the contents of the neurons and the chromosome. This is one of the reasons why a RAM neural network is so attractive. It is a binary architecture, what facilitates its implementation in a processor-based design. Considering the configuration presented in Figure 4.9, it is only necessary to have 1050 bits of memory to store the contents of all neurons. However, the current implementation needs 1050 bytes of memory, since the bits cannot be accessed individually and the chosen memory works with eight bits. It is still only a reasonably small amount of the memory required.

4.3.4 Communication Module

The communication module comprises a transmitter radio and a receiver radio, connected to an RS232 converter, which forms the serial communication link to the other robots and to the monitor computer (see Figure 4.26). The corresponding software is implemented in the microprocessor. It is a routine that monitors the radio for messages while the robot navigates. In case a message is detected, the communication module identifies which robot it was sent to, stopping the robot navigation to deal with it if the robot is the addressed one.

A radio board was also manufactured to be connected to the monitor computer via its serial port COM2. It contains the basic units shown in Figure 4.26, with a

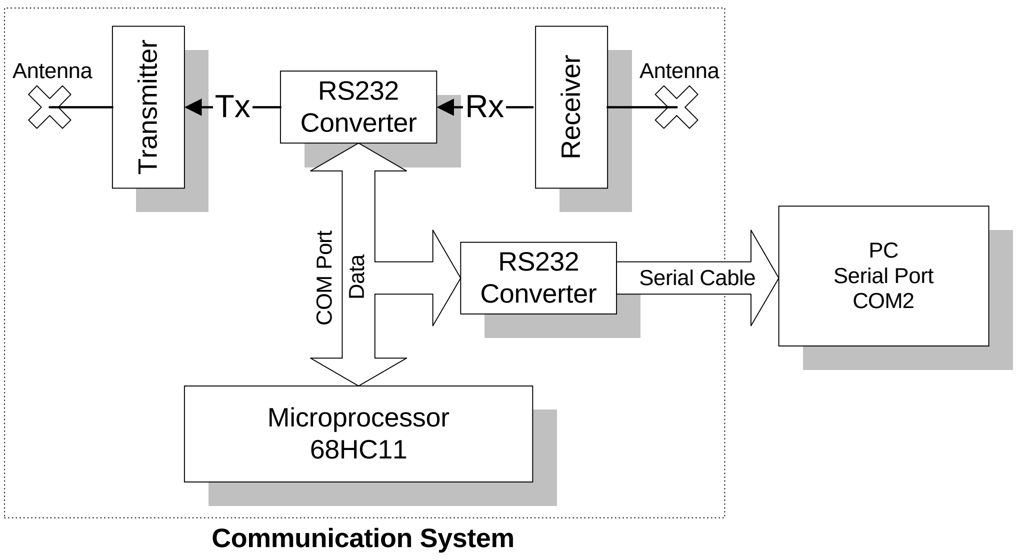


Figure 4.26 – Diagram of the Communication System, showing how two radios, a transmitter and a receiver, form the RS232 interface to the processor.

slightly different RS232 converter. Therefore, the PC can be seen by the other robots as another of their kind. It is also possible to communicate to the monitor computer via a cable, allowing a faster 9.6Kbps connection (see Figure 4.26). This cable is more often used when developing new software for the robots, because it allows the process of downloading, controlling the execution, and debugging software to be done much faster.

The robots communicate with each other via messages. The communication module also enables access to the monitor computer, which takes no part in the evolution, but monitors the communication among the robots. The messages are a sequence of bytes containing data and codes that identify which robot is calling, the receiver of the message, and coordinate the transmission. Table 4.4 presents a list of the codes of the robot language and their meaning.

The robots exchange many messages during the evolutionary process. They all follow one of the two structures shown in Figure 4.27. The long message in (a) is a data transmission formed by a heading (*Start*) followed by a code that in this case must be *Z* to indicate that data will be transmitted. Then, the codename of the robot to which the message goes is followed by the codename of the robot sending the message. A string of data bytes is sent next, followed by three hexadecimal bytes (*7F*, *FF*, *FF*). These bytes indicate the end of the data string and must be followed by the *End* code (*W*), marking the end of the message, and the checksum of the transmitted bytes. Figure 4.27 (b) shows a

Table 4.4 – The codes used in the robot messages and their translation.

Code	**Hex**	**Translation**
A	*41*	Robot ID: Message to/from Robot n°. 1
B	*42*	Robot ID: Message to/from Robot n°. 2
C	*43*	Robot ID: Message to/from Robot n°. 3
D	*44*	Robot ID: Message to/from Robot n°. 4
E	*45*	Robot ID: Message to/from Robot n°. 5
F	*46*	Robot ID: Message to/from Robot n°. 6
Y	*59*	Robot ID: Message to/from PC
T	*54*	Robot ID: Message to all Robots
U	*55*	Start: sent to clear noise in the radio receiver
X	*58*	Start: code to start message reception
Z	*5A*	Code: start data reception
N	*4E*	Code: data transmission error, retransmit
O	*4F*	Code: message received OK – checksums correspond
V	*56*	Code: abort listening for messages
S	*53*	Code: stop navigating
M	*4D*	Code: resume navigation
P	*50*	Code: Was last message received OK?
W	*57*	End: message terminator (last Byte before checksum);

short message or command, used to give orders (*N*, *O*, *V*, *S*, and *M*) or query other robots about the last transmitted message (*P*).

As an example of communication between two robots, consider the cyclic procedure described in Figure 4.11. When the robots finish their working phase, they try to find a partner to breed with by sending a mating call. Figure 4.28 exemplifies a communication between Robot 1 and Robot 2 (see also Table 4.4). The figure shows the contents of three messages using the ASCII codes and the corresponding hexadecimal bytes. Message 1 is a command from Robot 1 to order all robots to stop by the end of the working season. When this message is received, the robots stop navigating and start transmitting their fitness and chromosomes one by one, coordinated by Robot 1. Robot 2 is the first to send its data, and message 2 in the figure shows how it does that. Robot 1 then acknowledges having received the message, telling Robot 2 that it was transmitted without problems. If any problem occurred during the transmission (e.g., interference in the radio waves), Robot 1 would indicate it had problems receiving it, sending the message: "*UUUUUUUUUU-X-N-B-A*".

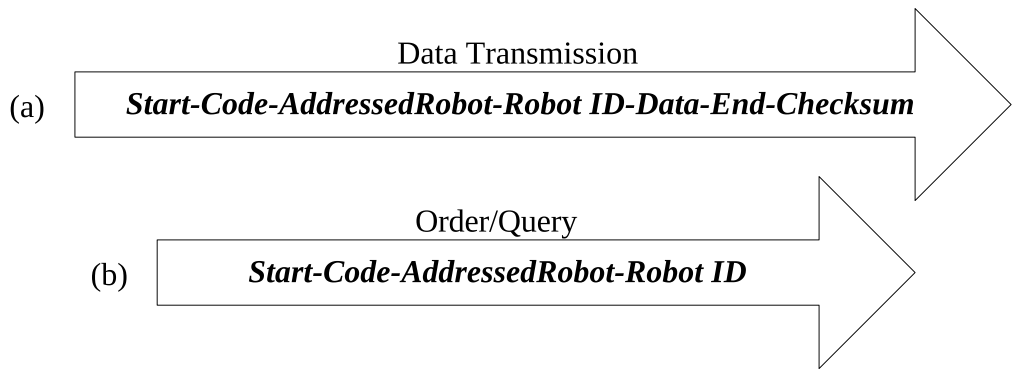

Data Transmission
(a)
Start-Code-AddressedRobot-Robot ID-Data-End-Checksum
Order/Query
(b)
Start-Code-AddressedRobot-Robot ID

Figure 4.27 – Structure of the messages sent by the robots via the radio.
Where: *Start* – Initialises the message. It contains a string of 10 *Us* to clear noise in the radio receiver, followed by *X* that initialises a transmission;
Code – A code that orders the receiving robot to start the message reception (*Z*) or a command that gives another order (see other "Codes" in Table 4.4);
AddressedRobot – Identifies to which robot the message goes (see "Robot ID" in Table 4.4);
Robot ID – Identifies which robot is sending the message (see "Robot ID" in Table 4.4);
Data – The body of the message. It is a string of data bytes followed by *7E*, *FF*, *FF* (bytes in Hexadecimal);
End –A code indicating the end of the message. It is the byte *W*;
Checksum – Is 1 byte containing the sum of all transmitted bytes.

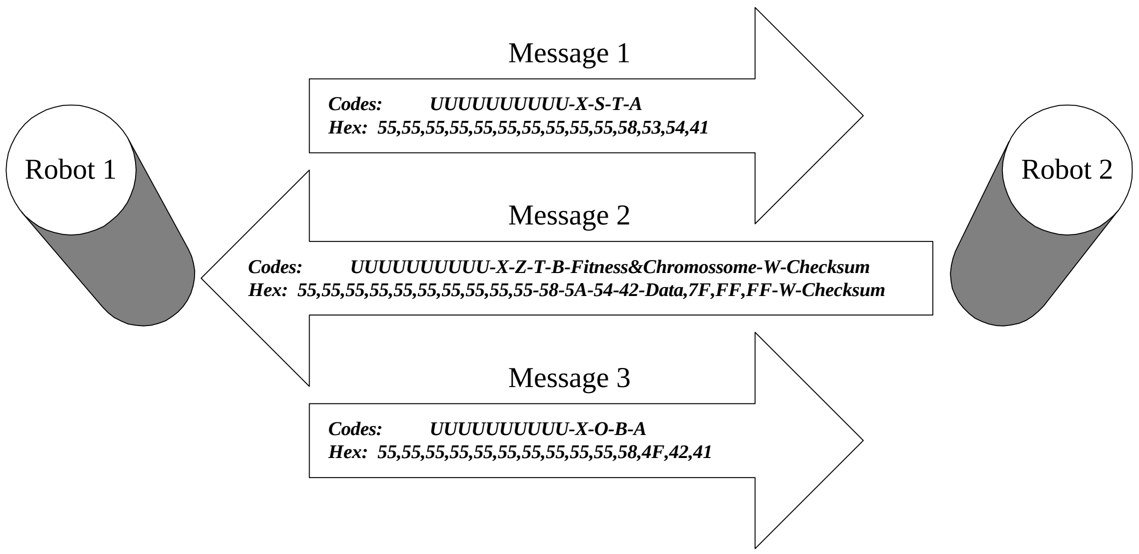

Message 1
Codes: UUUUUUUUUU-X-S-T-A
Hex: 55,55,55,55,55,55,55,55,55,55,58,53,54,41
Robot 1
Robot 2
Message 2
Codes: UUUUUUUUUU-X-Z-T-B-Fitness&Chromossome-W-Checksum
Hex: 55,55,55,55,55,55,55,55,55,55-58-5A-54-42-Data,7F,FF,FF-W-Checksum
Message 3
Codes: UUUUUUUUUU-X-O-B-A
Hex: 55,55,55,55,55,55,55,55,55,55,58,4F,42,41

Figure 4.28 – Example of three messages sent by Robot 1 and Robot 2. Message 1 orders all robots to stop. In message 2, Robot 2 sends its fitness and chromosome to all other robots. Robot1 sends message 3 to acknowledge that message 2 was received without problems.

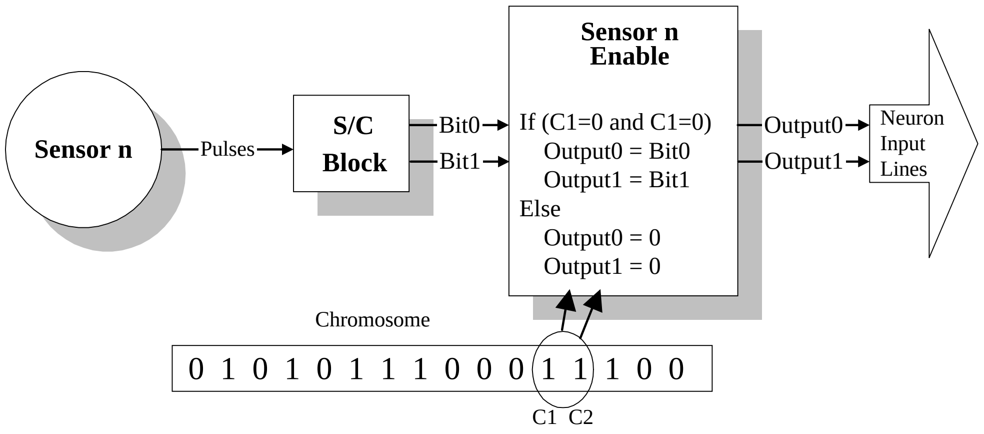


Figure 4.29 – Illustration of how each sensor reading is converted into a 2-bit signal by the analogue to digital block (S/C block) and selected to be connected to the neuron inputs.

- **Communication Module Requirements**

The speed of the radio link between the robots needs to be at least 1.2Kbps or the time wasted in communication will disturb the evolutionary process too much. Considering that six chromosomes containing about 1050 genes (considering the neural net configuration of Figure 4.9) need to be transmitted together with other codes, roughly 12 seconds will be wasted by communication alone during the mating season. Considering a generation time of one minute, it means that the robots will stop working for at least 1/6 of their lifetime. That is the reason why a good data transmission speed is so important. It also needs a safe protocol to ensure that data will not be lost in communication, unless a few wrong bits in the received chromosome can be considered as another form of mutation.

4.3.5 Sensor Module

In the sensor module, each analogue signal from the sensor is converted into two bits by the signal conversion block (S/C block) and selected to be connected to the navigation control in the central control module. Figure 4.29 shows how each sensor is selected according to two genes in the chromosome. If these two control bits are “0”, then the 2-bit signal from the S/C block is connected to the sensor output. The figure illustrates how “having a sensor enabled” is a recessive feature and illustrates the implementation of the “dominance” strategy described in Section 4.2.3.

Each sensor in the sensor module is connected to an S/C block and a sensor enable block. The S/C block converts the signals coming from the sensors into a 2-bit digital input to the sensor enable block. With two bits, this digital input can have four levels (“00”, “01”, “10”, and “11”) representing the distance to the obstacle: “00” means no obstacle; “01” means that the obstacle is *far*; “10” means the obstacle is at *medium* distance; and “11” means that the obstacle is *close* to the robot. The sensor enable block is implemented in software into the microprocessor. It has a simple algorithm that only outputs the sensor reading (*Bit0* and *Bit1*) if the control bits in the chromosome (*C1* and *C2*) are both equal to zero.

- **Sensor Module requirements**

To allow the neural network to drive the robot safely through the obstacles and the supervisor algorithm to evaluate when it is performing the right manoeuvre, the sensors need to discriminate the obstacles with at least four levels of range. Figure 4.30 illustrates how the proximity sensors are displayed around the robot and shows three consecutive curves representing the range of the sensors: *far* = 27cm; *medium* = 15cm; and *close* = 6.5cm. The curves were measured for the minimum distance that a sensor could detect the presence of a black pencil. That configuration was very efficient during preliminary tests of robot navigation. All sensors but one are positioned in the front of the robot. Considering that the robot will be moving when it is manoeuvring, there is no situation where a robot can approach an obstacle without seeing it. One sensor is placed in the back to allow the robot to detect when a faster robot approaches it from behind.

Figure 4.31 shows the position of four round bumpers and their collision sensors. Also shown in the figure is the position of four hypothetical collisions with the obstacle in different positions. The placement of the sensors on the bumpers permits the detection of a collision in 12 different positions. Each bumper is connected to the chassis by two independent sensors. Consider now bumper 1. If the collision occurs in the middle of the bumper (collision 1 in the figure), both sensors *Cs1* and *Cs8* will be activated. If the collision occurs to the right of the bumper (collision 2 in the figure), only *Cs8* will be

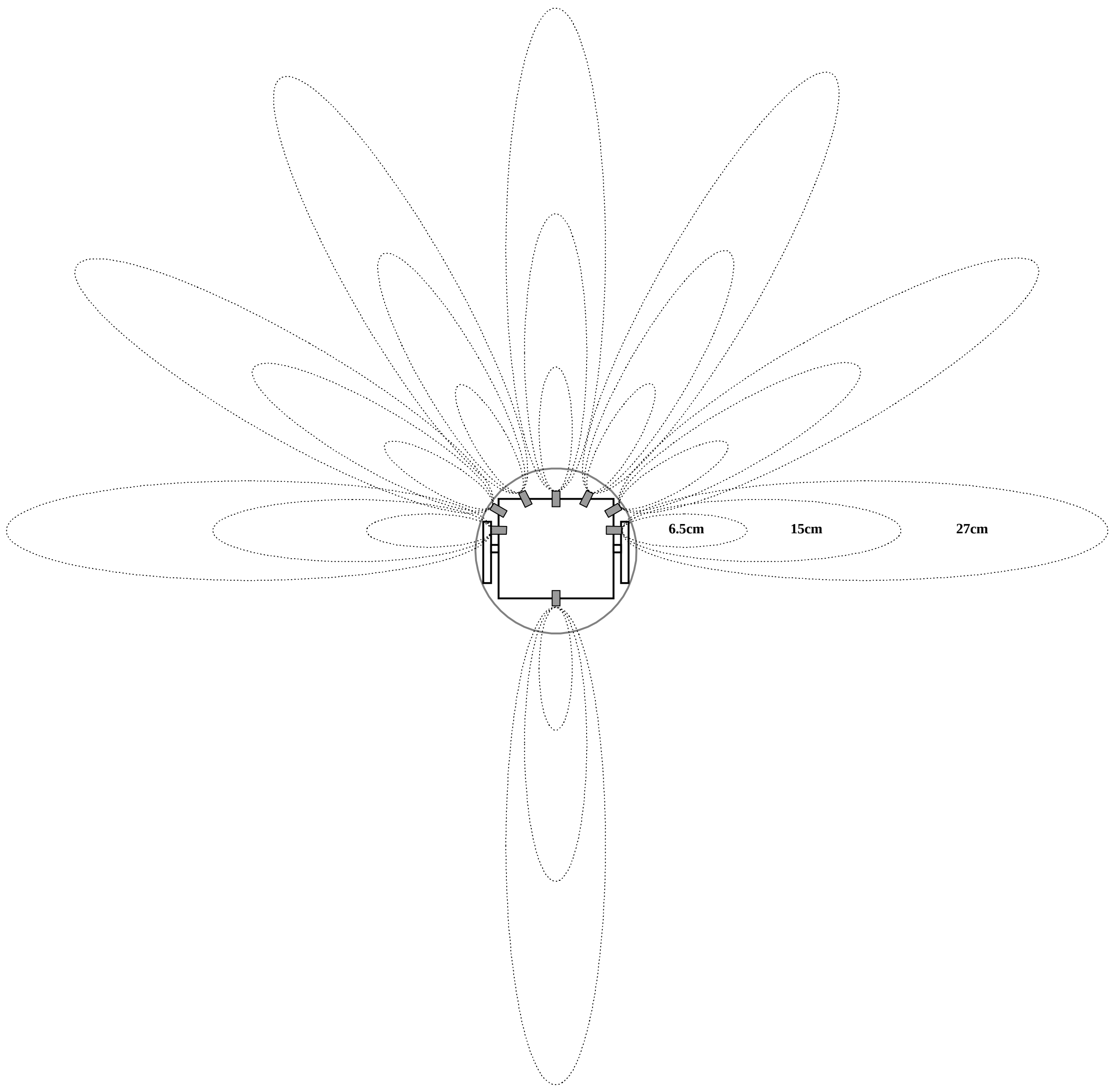


Figure 4.30 – Illustration of the position of the sensors in the robot periphery, showing approximately the range of each sensor relative to the minimum distance they can detect a white wall.

activated; if it occurs to the left, only *Cs1* will be activated. Therefore, collision 1 is detected by *Cs1* and *Cs8*; collision 2 is detected by *Cs8* only; collision 3 is detected by *Cs7* only; and collision 4 is detected by *Cs7* and *Cs6*. This is required to give the necessary accuracy to the supervisor algorithm in detecting the position of a collision around the robot. It can then calculate the appropriate manoeuvre and drive the robot away from the obstacle, returning the control to the navigation control circuit afterwards.

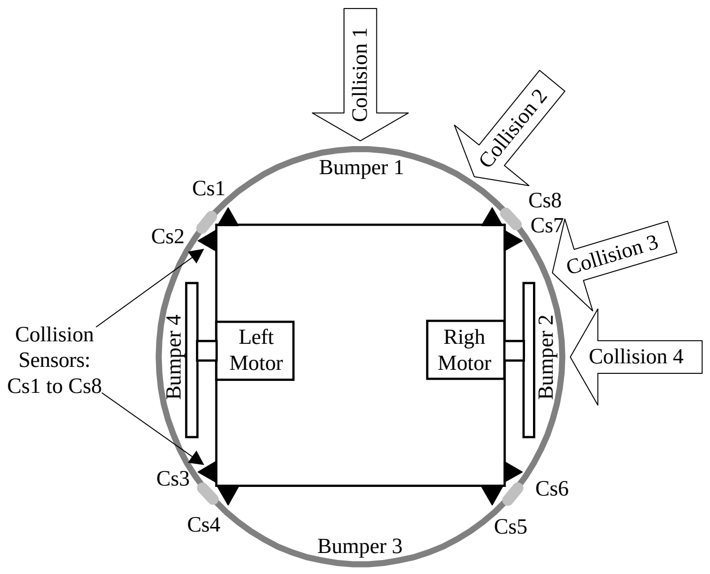


Figure 4.31 – The four round bumpers with collision sensors and the position of four possible collisions with obstacles. Collision 1 is detected by *Cs1* and *Cs8*; collision 2 by *Cs8* only; collision 3 by *Cs7* only; and collision 4 by *Cs7* and *Cs6*.

4.3.6 Motor Drive Module

Controlled by the chromosome, the motor drive module is implemented mostly in software by the microprocessor, including the pulse-width modulation signals that control the motors (see Figure 4.32). The speed levels *Fast*, *Medium*, and *Slow* are specified by ten bits in the chromosome, as described in Section 4.2.3.

The command interpretation block has a set of predefined routines to drive both motors that are selected by the command from the neural network. These routines are sequences of commands containing speed levels, directions, and delays for both motors. They are able to manoeuvre the robots according to predefined commands (e.g., *S*, *FS, FM, FF*, *TRS*, *TRL*, *TLS*, and *TLL* considering the example of Figure 4.9).

- **Motor Drive Module Requirements**

The pulse-width modulation is performed by three internal counters of the microprocessor, controlled by interrupt sub-routines. The selected speed can be generated

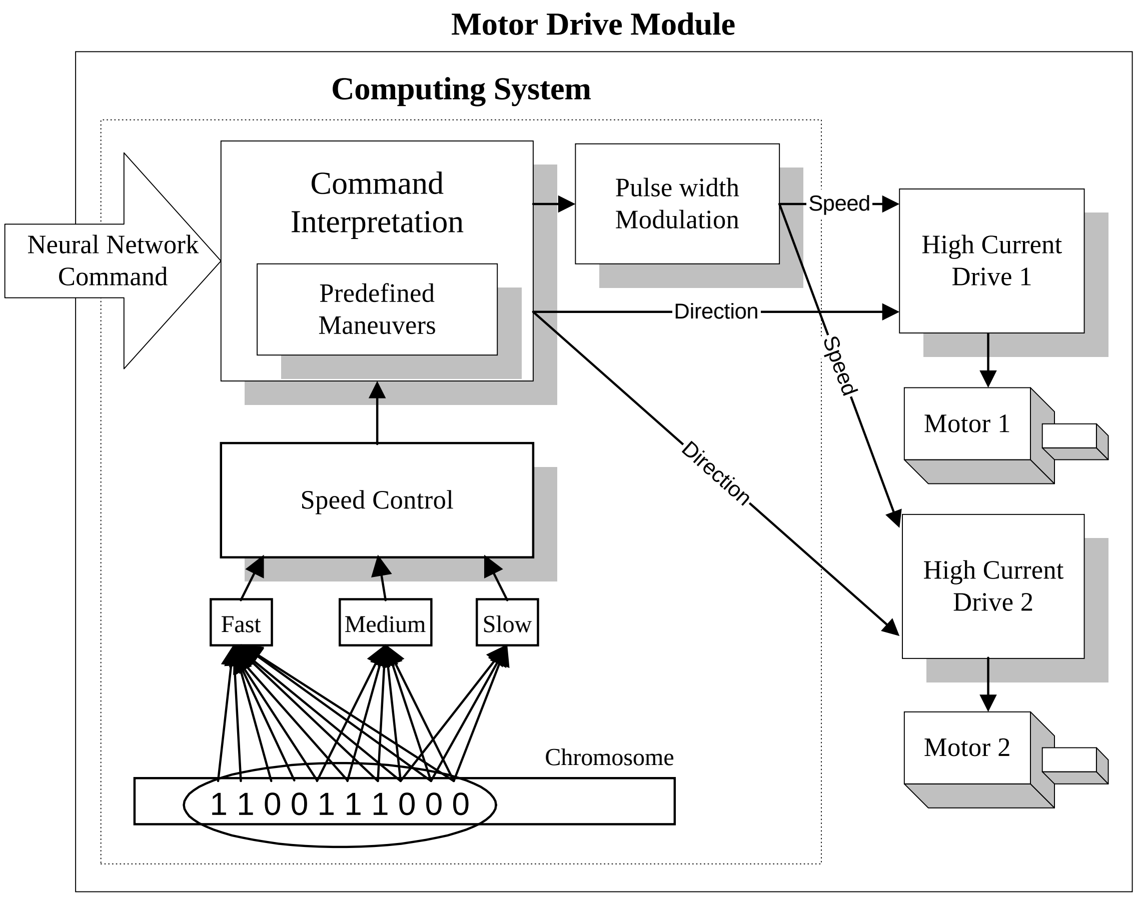


Figure 4.32 – The motor drive module. The speed levels *Fast, Medium,* and *Slow* are specified by the chromosome. The command interpretation block has a set of predefined routines to drive both motors. The microprocessor implements most of this module, including the pulse-width modulation signals that control the motors.

by controlling the width of a 12V pulse to the motor over a fixed period. It was observed that low frequencies cause the motor to vibrate in low speeds, disturbing the proximity sensor readings. Preliminary tests determined a minimum frequency of at least 1000 pulses per second. This frequency is necessary for a smooth control of the motors for any speed level.

The robots must be able to control their speed with at least 32 levels between zero and maximum speed. The faster the robots go, the more challenging situations they are going to face during the generation time, making it easier to spot poorly-adapted controllers. Considering the dimensions of the workspace and the relative size of the robots, they should be able to move at 0.26m/s at least.

4.3.7 Power Management

The power requirements for an open-ended embedded evolutionary system demands a constant delivery system. In most of the work with real robots, such as Brooks in [Bro92], Jakobi in [Jak98b] and [Jak98a], Harvey in [Har97b], or Floreano and Mondada’s experiments with Khepera robots [Flo98b], battery power does not usually last for more than a couple of hours. To keep the robots going for longer evolutionary runs many alternative power sources have been proposed. The most common is tethering the robots directly to a power source, as in [Mon96]. Another solution is to provide recharge stations and contacts in the robots that enable them to dock periodically. However, the amount of time the robots must spend docked (that is, not performing the task) is usually a considerable part of the process. If one robot stops to recharge its batteries, and therefore preventing the station being used by another one, it cannot take part in evolution and may force the others to stop and wait. A good solution used by Watson *et al*. [Wat99a] involves the construction of a powered floor to feed the robots throughout contact points on the underside of their bodies.

- **Power Management System Requirements**

All the above-mentioned solutions consider a fixed rechargeable battery that demands substantial technology. The developed solution is much simpler: it relies on an exchangeable battery, and a strategy to protect memory and the state of the robot hardware when it is replaced. The chosen batteries can sustain continuous operation of the robot for eight hours or more, which are enough to perform the evolutionary runs of the experiments. A monitoring circuit supervises the battery charge and will interrupt the operation of the robot if it drops below a threshold. The program counter and all other internal variables are preserved by a small fixed back-up battery, and the robot can resume its operation as soon as the main battery is replaced. Replacing the battery is considerably faster than providing power stations to recharge it, avoids tethers that may get tangled, and is considerably simpler and easier to implement than a powered floor.

5 ROBOT DESIGN

This chapter describes how the developed evolutionary system is embedded in the robots. It introduces the robot mechanics, the hardware and software developed to implement the sensors, the control circuit, the motor drive circuit, and the communication circuit of the robots. The software referred to in this chapter can be found in Appendix A, and the schematic diagrams of the complete robot circuit and radio board, together with a list of the used components, can be found in Appendix B.

The robot was designed in three phases: the first prototype; the second prototype; and the final robot. Figure 5.1 shows the first and the second prototypes. The first prototype was developed on a wire-up protoboard, so that the necessary modifications to the circuit could be easily rearranged. It can be seen in the figure that this design made use of two circuit boards, arranged in two layers. The first layer contained the processor, memory, and peripherals, and the second layer contained the sensors and motor drivers.

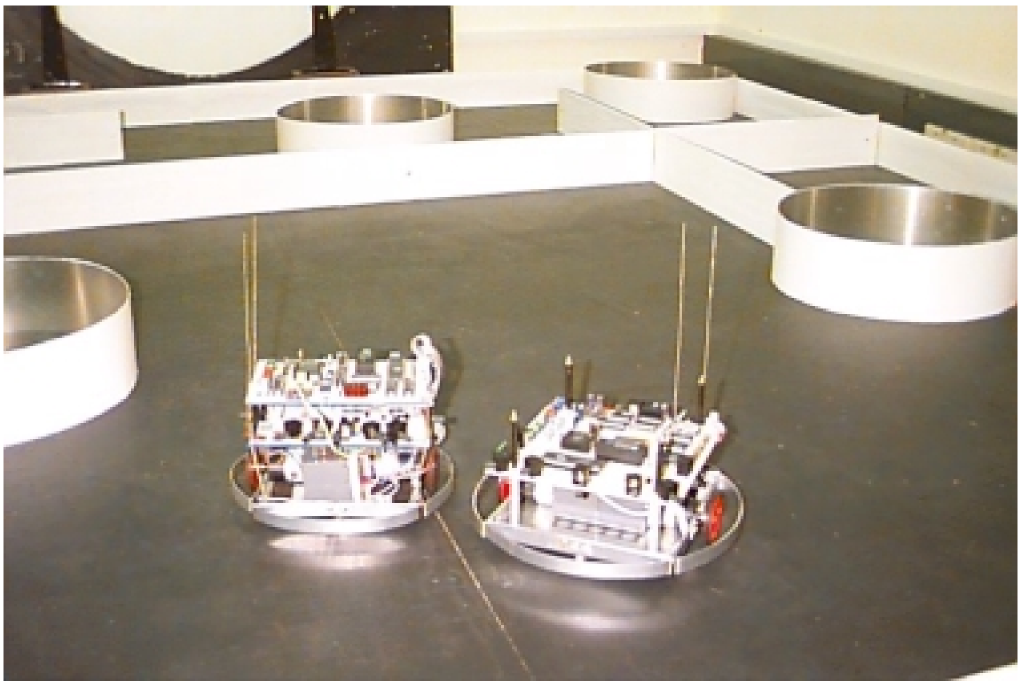


Figure 5.1 – The first (left) and the second (right) prototypes of the robot development.

Once a satisfactory performance was obtained, these two layers were combined into a single printed circuit board (PCB) layout containing the complete robot circuit. This was implemented in the second prototype, manufactured using a double-sided circuit board, which served to evaluate this compact design before it was reassembled into an even smaller final layout, which had the circuit board industrially manufactured.

5.1 The Chassis Design

The motors have to have small dimensions because of the small size of the robots. However, small motors do not have a very high-level of torque. Batteries of high capacity will be required to power two electric motors and the printed circuit board for a minimum of four hours without recharging. Therefore, because of the large weight of a suitable sized battery pack, extra torque needed to be produced by a gearbox so that the motors are able to move the robot at a reasonable speed. It is expected from the nature of the proposed experiments that actual contact between robots will occur. Therefore, the selected material must be able to support the robot weight and resist damage from contact with other robots, obstacles, and walls. As many robots are necessary for the experiments, simplicity of design and low cost are imperative. In that case, aluminium is the most suitable material to build the robot chassis, since it offers the lightness and strength needed, and is easily manipulated with the given resources of the mechanical workshop of the University.

5.1.1 Shape and Size

A rectangular robot provides a more useable shape in terms of mounting components, but the likelihood of the robot being stuck would be greatly increased, since it is not able to turn around within its perimeter, which can cause it to strike other robots and the surrounding walls. A circular chassis is more advantageous for the robots because it is less likely to be stuck in corners than a rectangular one. This design results in a robot that is highly manoeuvrable and able to turn around on its own centre of gravity and within its own perimeter. There is a reduced chance of the act of turning causing the robot to strike a wall or another robot. However, this results in a reduced usable space at the base. There is a need for a low centre of gravity to reduce the risk of the robot turning over because of its high mobility and fast directional changes. To provide a low centre of gravity, the heaviest parts (e.g., the battery pack and motors) need to be mounted as low as possible, but this leaves a lot of unusable space.

5.1.2 The Adopted Solution

Despite the advantages in space, leading to a smaller and lighter chassis, the rectangular base is more difficult to drive in tight, crowded spaces. As it can be inferred from the proposed experiments, initially, the robots will not have a proper trained controller, so that collisions are more likely to happen. When a collision is detected, the simple rescue algorithm will turn and drive the robot away from the obstacles before returning the control back to the main algorithm. This rescue program would be more complicated to design for the rectangular base, strongly suggesting a round chassis that can safely turn around its centre, leaving the robot facing back to the obstacles. However, designing and manufacturing circular printed circuit boards could also prove more difficult than the traditional rectangular ones. Therefore, as shown in Figure 5.2, the final solution was obtained by using a square base surrounded by round bumpers that produced a circular chassis.

This decision allowed the drive mechanism to be achieved by a differential steering system, consisting of two drive wheels and two non-pivoting casters. Each wheel can be driven independently and steering is achieved by changing the speed of each wheel. The castors are used to maintain balance. This solution is mechanically simple to implement and easy to control and steer. The motors are required to have enough torque to move the robot at 0.26m/s and to be small enough to fit into the chassis. The chosen motors are 12V DC with a built-in gearbox. The drive ratio with this gearbox is 1:42, which gives enough torque and drains a small current. The motors are connected to a set of plastic wheels available from electronic suppliers with a replaceable rubber tyre.

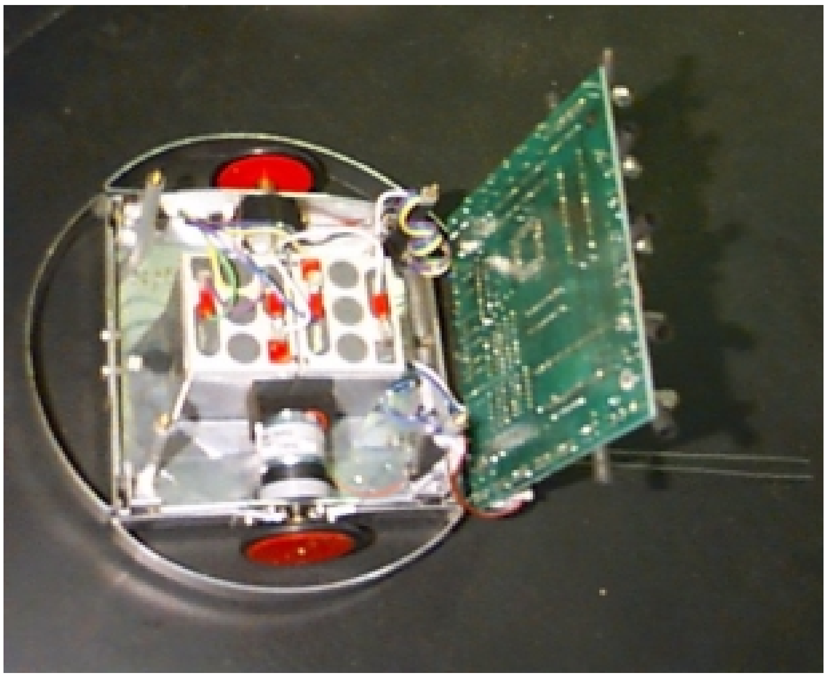


Figure 5.2 – Top view of the robot chassis showing the position of the motors and wheels, the batteries, and the round bumpers.

The rechargeable batteries are mounted between the motors to keep a low centre of gravity. If the battery charge runs low, these two 6V batteries can be exchanged for other two fully charged ones. This allows the robot to keep moving while the extra pair of batteries is charged. Figure 5.3 shows how the circular chassis was constructed using a square base to accommodate the batteries and motors.

5.2 Computing System

The computing system consists of the microprocessor, memory, and peripheral support circuitry. These devices should provide the necessary environment to allow a suitable autonomous motion for the robot and interfacing to other robots. The computing system should offer the necessary hardware and software for the processing of the infrared proximity sensors and should also provide the waveforms and direction signals required for controlling the motor drive circuit. To drive the motors, the processor should produce pulse width modulated (PWM) outputs. A means of sending and receiving

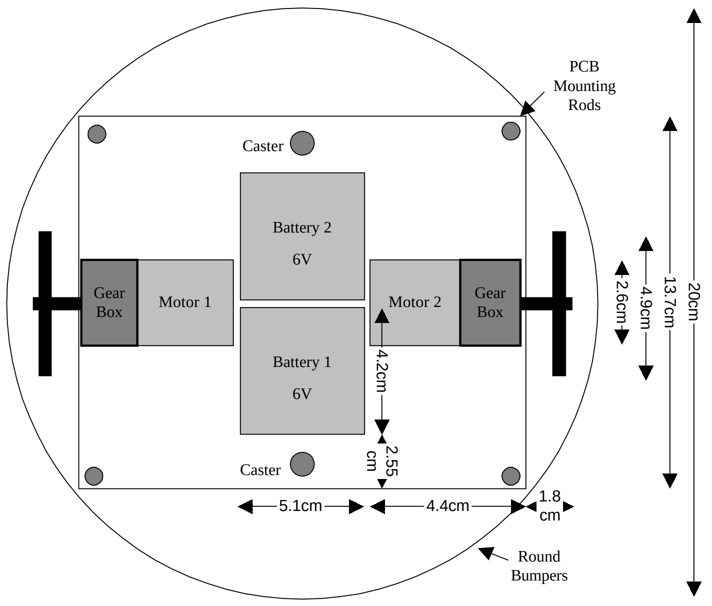


Figure 5.3 – Circular chassis construction and placement of the motors and batteries.

asynchronous serial data to communicate with other robots and the monitor PC is also required from the computing system. The schematic diagram of the complete robot circuit can be found in Appendix B, together with a list of the components used. Most of the information presented here originated from the Motorola 68HC11 reference manual [Mot96]. It provided vital information and technical data that allowed the design of the robot computing system.

The Electronics Laboratories of the University of Kent currently use the Motorola 68HC11 microprocessor as the basis of their embedded designs. This is an 8-bit micro-controller that is very versatile and well-suited for embedded computer systems. It has a limited instruction and register set; hence, the processing tasks able to be undertaken are of limited scope. The 68HC11 was chosen for it is a low-cost well-known microprocessor that could deliver the necessary functionality to support the operation of other peripheral circuits, as well as providing the means for carrying out the main system processing. The main functionality provided by the 68HC11 is: timing signals for the infrared proximity sensing and processing of the responses of the sensors; power and direction output signals to the motor drive circuits; and serial communications.

As well as providing the means for interfacing the external circuitry, the computing system contains the necessary circuits to support the operation of the processor itself: power regulation; power supervision; and system clock generation. These circuits are illustrated in the schematic diagram presented in Appendix B.

- **Regulated Power Supply**

The processor and all the other circuits in the robot printed circuit board require a 5V DC (150mA) power supply, plus a non-regulated 12V DC supply connected to the output buffers that drive the motors. The 12V DC (named *VCC2* in the schematic) is provided by connecting together two 6V batteries on board the robot. The first battery also supplies approximately 6V that is regulated via a 5V DC (500mA) low dropout voltage regulator. This battery produces a variable voltage dependent on the current load. Therefore, a low voltage dropout regulator such as the Maxim MAX603 can extend the usage of the battery down to 5.3V. Other regulators typically need the supply to be at least 1.5V higher than its output. This chip supplies 5V DC (*VCC1* in the schematic) to the robot board.

- **Power Supervision**

The 68HC11 needs 5V DC to operate properly and the system clock is generated from the same supply. If the board voltage supply drops from 5V, then the state of the processor and data in memory can become indeterminate. Therefore, the function of the power supervisory circuit is to ensure that the processor is in a known state below this voltage. The system external memory requires no management during the system reset and low voltage conditions, since it makes use of a memory chip that incorporates a static RAM to store the program and data, an internal 3V rechargeable lithium cell, and the necessary management circuit. This chip alone provides control and a standby voltage that allows the program to be retained during power off periods.

The supervisor circuit also provides the ability to deselect memory operations during low power conditions, so that the processor cannot write to the program memory during periods of indeterminate conditions. The Motorola 68HC11 processor has an active low reset line (*RST*) that can force the processor into reset state. The MAX691 device is used to provide the supervisory reset signal (*/RST*) necessary to control the 68HC11 reset and disable the external memory during reset and power-down events.

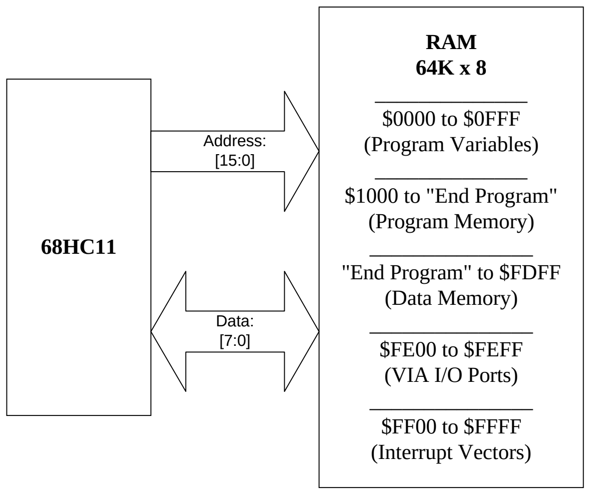


Figure 5.4 – Configuration of the system memory.

- **System Clock Circuit**

The system clock signal (*E*) is generated by the 68HC11 using an internal oscillator circuit. The necessary 2MHz reference signal is generated from an 8MHz crystal oscillator circuit connected to the processor *EXTAL* and *XTAL* pins.

- **System Memory**

The 68HC11 reduces the complexity of on-board memory support circuitry, since it provides internal chip select decoding to address the program and data memories. The memory configuration has an 8-bit wide 64Kbyte memory that is shared by the program and data. The program area starts at $1000 (“$” represents a number in hexadecimal) and data can use all available memory from the end of the program to $FDFF. The interrupt vectors occupy the area from $FF00 to $FFFF, and a small area for program variables is left from $0000 to $0FFF as shown in Figure 5.4. The addresses between $FE00 to $FEFF are reserved to be used by the port expansion chip, the VIA (Versatile Interface Adapter).

The battery backed-up RAM memory works as a pseudo ROM, providing a non-volatile memory for storing the program. Instead of using a PROM or EEPROM to provide the program memory, the battery backed-up RAM allows new programs to be installed much faster by just downloading them via the serial port from the computer where the software is developed and compiled. A problem arising from this approach is that there is no protection against unintentional changes or corruption of the program memory via stack overflow or other runtime failure. Therefore, extra care must be taken in programming to avoid this problem.

- **Serial Communication**

Serial communication is provided by the 68HC11 internal Serial Communication Interface (SCI). It can be configured to eight data bit, no-parity, 1200 baud, CMOS voltage level. A wired RS232 serial link was made available by the use of a cable containing a CMOS-RS232 conversion interface that can be attached to the IBM PC COM ports and connected to the *RXD* and *TXD* pins available in the microprocessor. This allows the downloading of the program software to the robots and permits runtime debugging of program variables and data exchange via a specially designed terminal program resident in the PC. This program was written in Borland C++ and provides a virtual screen where the 68HC11 can output data. The program is called VIRTUALSCREEN.CPP and is listed in Appendix A.

The serial port is also connected to a peripheral AM radio that provides a wireless link to the other robots and to the monitor computer. A DIP-switch defines if the serial communication will come from the cable or the radio link. The radio circuit is explained in Section 5.5.

- **Discrete Input and Output Ports**

Discrete I/O is available via two different sources: from the 68HC11 specialised I/O ports that can be pin-configured as input or output; or from the peripheral VIA that connects to the address and data busses. The 68HC24 is the device that works as a port expansion, providing two extra I/O ports that can be read from or written to as data in the memory. Each one of the pins has a control function and a corresponding I/O pin mapping that can be selected by setting register values through runtime routines in the program software.

5.3 Motor Drive Circuit

The most complex part of the motor drive system is the generation of the PWM signals, which is provided by the 68HC11 Timer Processor Unit (TPU) interface. The motor drive circuit constitutes a power driver that converts the low current CMOS signals provided from the processor to high current 12V pulses. This circuit is required to supply 12V to the motors and move the robot motors independently with proper speed and direction, according to the logical inputs from the processor.

The 68HC11 TPU interface can provide time-based functions by sequencing together match and compare operations. Many pre-programmed functions are provided in the TPU ROM mask, including PWM output, discrete I/O and input capture. As the drive motors are DC voltage controlled, the pulse-width modulation waveforms generated by the processor provide the necessary speed control by varying the average power of the signal. The 68HC11 generates these signals by combining the real-time clock system with output compare counters. The appropriate counters are set at the correct intervals during normal processing to generate the desired waveforms. This introduces a degree of complexity to the software design, since the main program is run in parallel with the waveform generation.

A push-pull driver is the main device of the motor drive circuit. The L239E was chosen because it has the ability to drive the two motors in two different directions separately. The L239E can output a maximum current of 1A that is more than enough to drive the chosen DC motors, which drain an average current of 65mA at maximum speed. The motor drive circuit operates by buffering two signals from the TPU interface in the 86HC11 chip for each motor. These signals control the speed and direction of the motors. The variable speed of the motors is generated by using a pulse width modulated signal with a period of 1.025ms to control the chip enable line of the L293E. The signal is switched on in the beginning of the 1.025ms period and stays on until the corresponding counter reaches the specified value t_1. Then, it is switched off and stays like that until the next period. The average power transferred to the motor is maximum when t_1 = 1.025ms and the signal is never switched off. With t_1 = 0.5125ms, the transferred power is 50%, as illustrated in Figure 5.5.

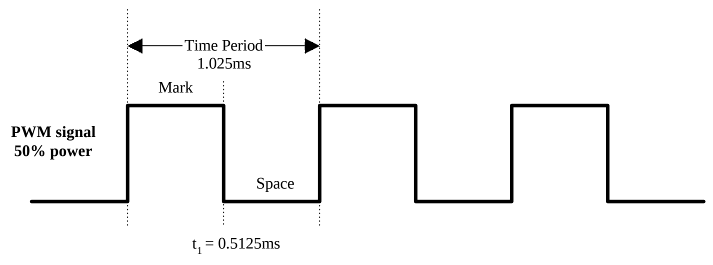


Figure 5.5 – PWM signal with 50% power transferred to the controlled motor.

5.4 Sensors Circuit

The sensors enable the robot to interpret the external environment in two ways: providing the distance between the robot and the obstacles and indicating where the collisions have occurred. The infrared proximity sensors allow the robot to determine object locations remotely and simple touch sensors in the bumpers can tell whether the robot is in contact with an object and where the collisions occurred. The proximity sensors can detect other robots, obstacles and walls without physical contact, so that they can be avoided.

Infrared sensors were chosen to implement the detection of obstacles because they are a simple low-cost solution that can provide a fairly accurate distance sensing. The standard infrared emitter-detector combinations were studied. They work by continuously emitting IR light and detecting the intensity of the received energy to indicate how far away the robot is from other objects. This method requires too much power, since to allow a good range the eight infrared LEDs (light-emitting diodes) would drain more current than the motors moving at full speed (180mA). Therefore, a more economic solution is to output a pulsed waveform of a known duration and then look at the detectors to see if anything was received. If the detector is activated, it means that the waveform had enough energy to reach an obstacle and bounce back to the robot. If nothing is received by the detector, the power of the pulse is gradually increased until some detection is achieved and the distance to the object will be proportional to the level of power in the emitter. Figure 5.6 shows how the current through the emitter is gradually increased to vary the energy of the output until an obstacle is detected.

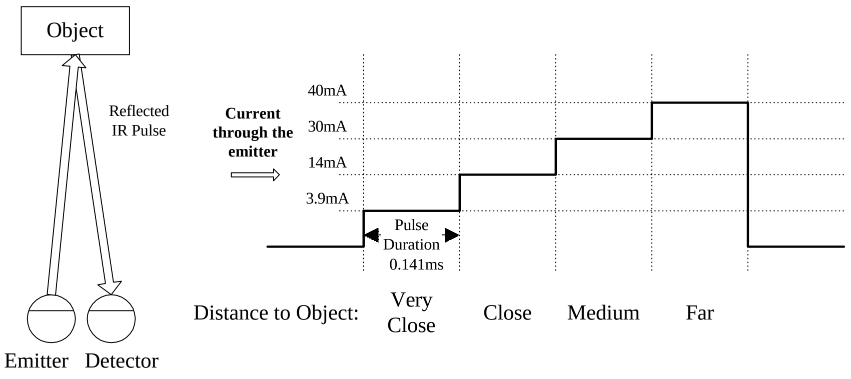


Figure 5.6 – The current through the emitter is progressively increased to allow the detection of obstacles with four ranges.

The necessary variation in the current that excites the IR emitters is obtained by using transistors to connect different resistors in parallel to the emitters. With more than two robots using infrared in the same area, there could be confusion about the source of the detected infrared. This approach was expected to present problems when the robots emit infrared at the same time, but this did not happen often and the robots could take again the correct manoeuvre after one iteration with correct sensor readings. This approach uses far less power than the previous solution, because the LEDs are only activated when the robot is sensing the environment. As the LEDs are working for a very short duration, the actual current that they can support is much higher than their nominal current and the average current used in this process is very low (0.5mA). The disadvantage of this solution is that it is more complex to implement in software, but this is a small price to pay when compared to the amount of power saved. The simple touch sensors in the bumpers that determine where the robot is in contact with objects are simple de-bounced microswitches mounted between the bumpers and the chassis. They work normally in the open position, but are closed when the bumpers are compressed.

5.5 Communication Circuit

The communication circuit provides a link among the robots and between the robots and the monitor PC. It enables the implementation of the embedded distributed evolutionary system and allows the monitor computer to produce a data record of the evolutionary experiment. The communication software developed specifically to deal with the interference of noise suffered by the radio link can distinguish between valid and corrupt data packets in real time. The baud rate of the radios of the robots and the PC was set to 1200bps to match the baud rate of the AM radios used.

The communication circuit consists of two AM radios, a transmitter and a receiver, on board of all robots and a radio board specially made to be connected to the serial port of the monitor PC. The chosen transmitter and receiver devices are the AMT21 and AMRW4, each connected to different antennas to avoid interference. These devices comprise modulation, filtering, and amplification of the signals and their logic levels are computed internally, where logic “0” is indicated by the presence of radio energy in the specified frequency and “1” is indicated by the absence of energy. The transmitter radio is connected to the *TXD* pin and the receiver to the *RXD* pin of the 68HC11 and a specific routine was implemented to handle communication and deal with noise immunity.

All radios on board the robots work at the same frequency: 418MHz. Therefore, a communication protocol is necessary to avoid having more than two robots emitting at the same time. Each experiment had different objectives and, therefore, different communication protocols were developed. Figure 5.7 shows an example of one of these protocols used by the robots to communicate and exchange data and messages. The robot can be interrupted while executing the main program if a “U” is received by the radio. This event transfers the control to the *GETDATA* subroutine. “get byte” is a command that reads the serial port to get a byte. From this subroutine, if the received byte is “X”, it goes to *SKIP1* and waits for an “Y” to go to *SKIP2* or a “V” to *ABORT*. In *SKIP2*, it can be ordered to abort (“V”), end communication and go back to the main program (“O”), transmit the stored data (“N”), or enter the *RECEIVE* routine (“Z”). Receiving noise in both *SKIP1* and *SKIP2* will bring the robot back to *GETDATA*. After transmitting data, the robot waits for the reply of the receiving robot and can be ordered to transmit again if there was any problem in the received data (“N”) or to go back to the main program if the transmission was OK (“O”). After receiving data, if the check sum comparison is OK, it sends “O” to indicate the sender that the data was received without problems. If noise is detected, it sends “N” and goes back to *GETDATA* to receive the next transmission.

A stand-alone radio board is connected to the monitor PC to allow it to communicate with all the robots. It has a transmitter and a receiver radio connected to a MAX233 chip that converts the CMOS signals (0V and +5V) to the RS232 voltage levels

(-15V and +15V). These are then connected to the *RX* and *TX* lines in the PC serial ports COM1 or COM2. Appendix B also includes the schematic diagram and a list of the components of this radio board.

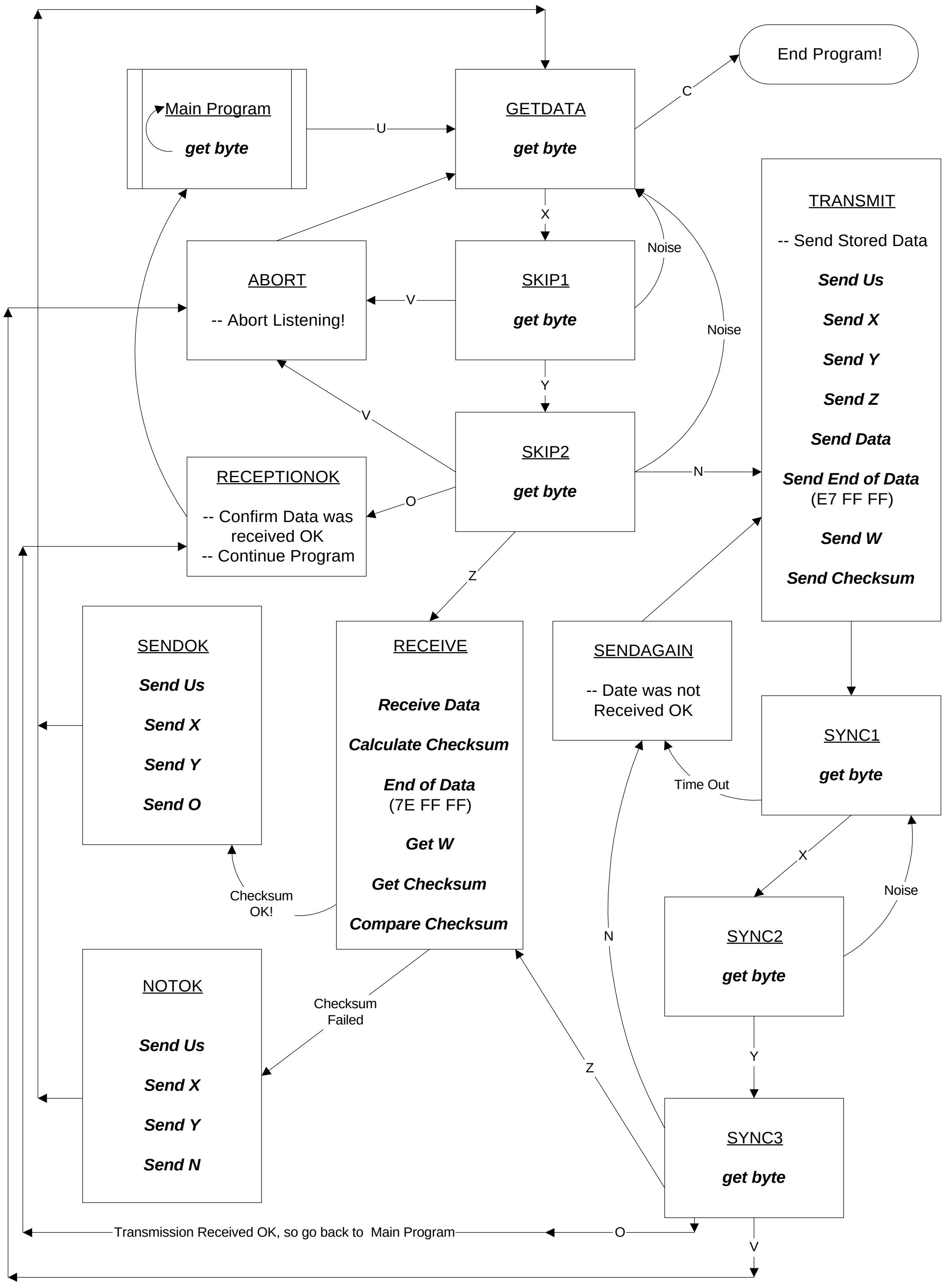


Figure 5.7 – The communication protocol used by the robots.

6 PRELIMINARY EXPERIMENTS

This chapter reports experiments that tested the effects of different lifetimes (the duration of each generation); determined the best sensor configurations; tested the capacity of the evolutionary system to evolve the configuration of the sensors with different mutation rates; and evaluated the evolution of an unstructured navigation control circuit with a simple and a biasing fitness functions. All the experiments but the first one were performed in environments of medium complexity, because a simple one does not provide the necessary selection pressure to distinguish between similar configurations in short lifetimes. In addition, in a complex environment, the robots tend to get too close to each other and unfit robots collide too much into the fit ones, causing their fitness to drop even though their configuration is well-adapted to the environment.

This chapter also introduces the workspace where the experiments were run. The workspace forms the environment where the robots operate and are evolved by the developed evolutionary system, which trains the controllers to behave according to the selected task: collision-free navigation in a constrained space with obstacles. It describes how the workspace was arranged to support the experiments and collect data from the evolving robots.

The experiments described here are just a small portion of a greater number of tests that were performed to provide the necessary insights that allowed the development of the evolutionary system. The results presented are the most important ones, the ones that evaluated new ideas that proved essential to the design. Many other experiments were run, providing more complementary information to the development of the system. Although most of these experiments are not described here for the simple reason of space, the information they generated is always present in the discussions, comments, and justifications described in this chapter. Therefore, this chapter shows only the most relevant experiments, chosen for representing the typical case and, in specific situations, the ones nearest to the average result. Appendix C contains the average results of many experiments

Table 6.1 – Initial position and orientation of the robots in the environment.

Robot ID	**Centre X** (cm)	**Centre Y** (cm)	**Orientation** (0 to 360^o)	**Speed M1** (parameter)	**Speed M2** (parameter)
Robot 1	190	208	85	10	21
Robot 2	112	145	70	223	118
Robot 3	141	47	352	56	3
Robot 4	166	179	87	61	144
Robot 5	13	19	318	187	47
Robot 6	194	122	60	112	193

that could not be included in the body of the thesis. For Chapter 6, it includes data related to Section 6.3.

6.1 Environment Organisation

To guarantee the random initialisation of all the variables involved in the experiments, it is essential to make sure that the robots start an experimental run at a different random position every time. This was made possible by using a positioning program, written to generate a random position, orientation, and initial speed for all robots. This program is called INITIALISE.CPP and can be found in Appendix A. Table 6.1 presents the resultant initialisation table produced by this program. *Centre X, Y* marks the position of the centre of the robot from the left lower corner of the workspace in centimetres. *Orientation* is the angle (between zero and 360^o) between the robot and the lower edge. *Speed M1* and *Speed M2* are the initial speeds for both motors of the robots (a value between zero and 255 that is later normalised to the 32 speed levels of the robot motors). If the position of the robot, which has 20cm diameter, overlaps the position of any other robot or the edges of the environment, then the coordinates are automatically discarded and calculated again.

After the initial position and orientation is obtained, the robots are placed by hand at their corresponding location. The dimensions of the environment were fixed (250cm $\times$ 250cm) throughout all experiments, but the position and form of the obstacles may vary. To make it possible, many round obstacles of various sizes were manufactured together with a set of 1m-long wall blocks, which can be combined to form many different configurations, as shown in Figure 4.19. Figure 6.1 shows some of the available forms.

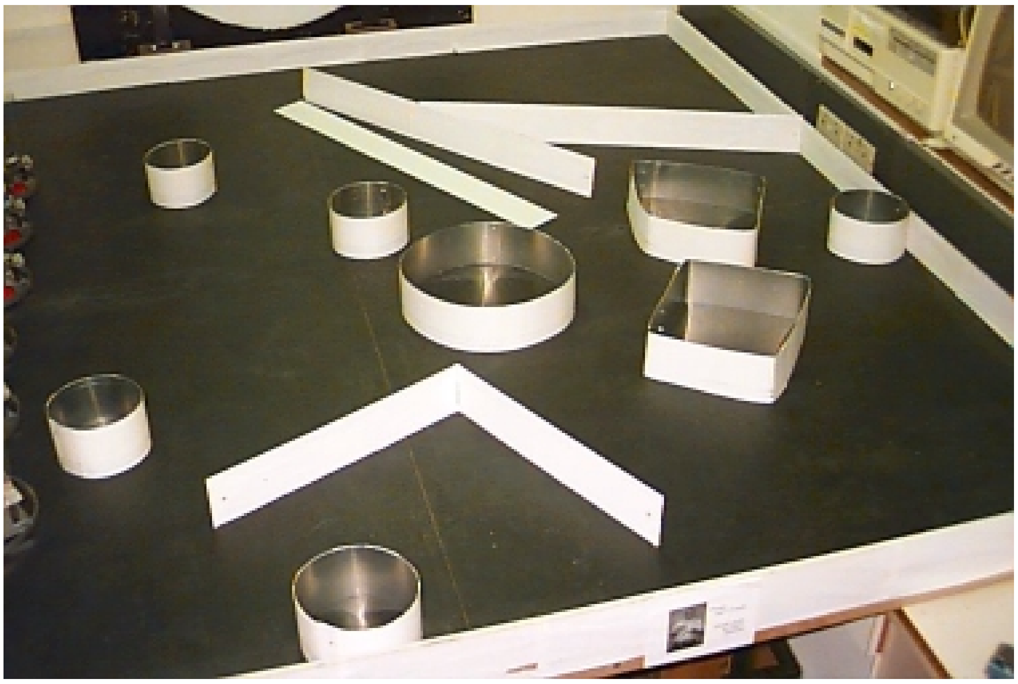
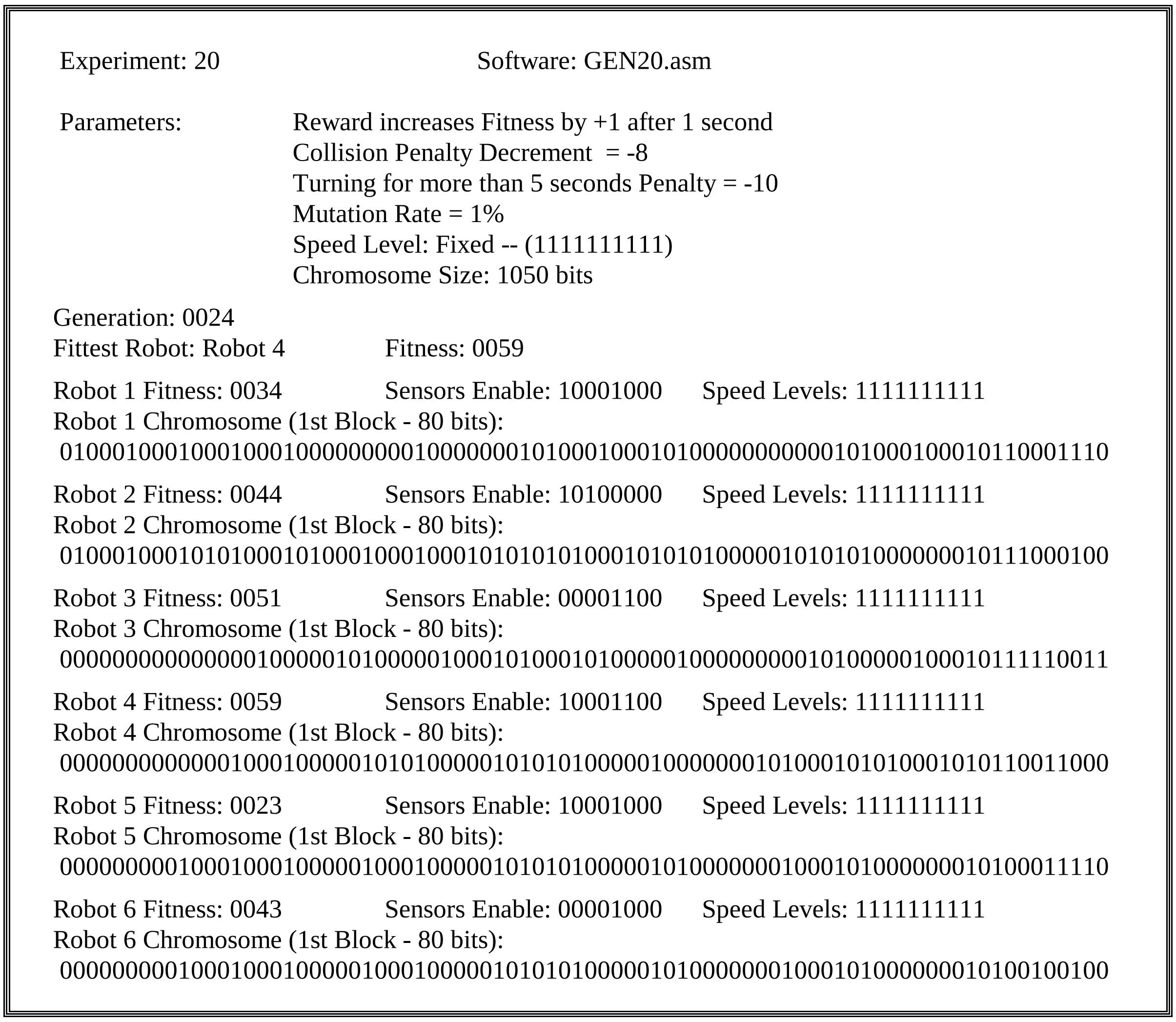


Figure 6.1 – The different shapes of the obstacles and walls.

Unlike the position of the robots, the configuration of the environment is not determined randomly by a program. On the contrary, it was specified by the user when the experiment was set up. Even though some of the user's prejudices would always interfere, it was set up as randomly as possible for each experiment. An automatic generator could not be used because it is very difficult to achieve a valid layout, where the robots would not be stuck in corners or small passages that would be seen as obstacles. The complexity of the walls and the obstacle configuration, however, play an important role in how evolution produces the controllers, so that selected experiments were performed just to evaluate this. Yet, many experiments were performed with the environment configuration unchanged. The reason for this was to concentrate effort on understanding the effects of other important characteristics, such as the mutation rate, crossover strategy, or the selection technique.

6.1.1 Data Monitoring

The monitor computer keeps a complete data record of the robot chromosome, fitness value, and other important parameters that change in every generation for all experiments. This computer does not influence the evolutionary process and can only be used to start or stop the robots, avoiding making the user run after the robots one by one to switch them off! As the data exchanged among the robots varied from experiment to experiment, so did the software running in the monitor computer, but it basically listened and identified the transmitted data and created an ordered data store. So there is practically a different program for each experiment. They are all named EXPxx.CPP and are listed in Appendix A. Figure 6.2 shows a typical screenshot of the monitor computer generated by the program EXP20.CPP.



```
 Experiment: 20                          Software: GEN20.asm

 Parameters:          Reward increases Fitness by +1 after 1 second
                      Collision Penalty Decrement  = -8
                      Turning for more than 5 seconds Penalty = -10
                      Mutation Rate = 1%
                      Speed Level: Fixed -- (1111111111)
                      Chromosome Size: 1050 bits

Generation: 0024
Fittest Robot: Robot 4          Fitness: 0059

Robot 1 Fitness: 0034           Sensors Enable: 10001000     Speed Levels: 1111111111
Robot 1 Chromosome (1st Block - 80 bits):
 01000100010001000100000000001000000010100010001010000000000010100010001011000111 0

Robot 2 Fitness: 0044           Sensors Enable: 10100000     Speed Levels: 1111111111
Robot 2 Chromosome (1st Block - 80 bits):
 01000100010101000101000100010001010101010001010101000001010101000000010111000100

Robot 3 Fitness: 0051           Sensors Enable: 00001100     Speed Levels: 1111111111
Robot 3 Chromosome (1st Block - 80 bits):
 00000000000000010000010100000100010100010100000100000000010100000100010111110011

Robot 4 Fitness: 0059           Sensors Enable: 10001100     Speed Levels: 1111111111
Robot 4 Chromosome (1st Block - 80 bits):
 00000000000001000100000101010000010101010000010000000101000101010001010110011000

Robot 5 Fitness: 0023           Sensors Enable: 10001000     Speed Levels: 1111111111
Robot 5 Chromosome (1st Block - 80 bits):
 00000000010001000100000100010000010101010000010100000001000101000000010100011110

Robot 6 Fitness: 0043           Sensors Enable: 00001000     Speed Levels: 1111111111
Robot 6 Chromosome (1st Block - 80 bits):
 00000000010001000100000100010000010101010000010100000001000101000000010100100100
```

Figure 6.2 – Data displayed on the screen of the monitor computer for one of the experiments at generation 24.

Data are obtained by the monitor computer and stored in a file called GENE.TXT. Every experiment in Appendix A has its own data file with this name. Figure 6.3 shows an example of how a data file looks. Data are a sequence of hexadecimal codes, beginning with “Gxx” that specify the current generation, followed by “a” that marks the beginning of the parameters of Robot 1 plus its chromosome, and so on. To save space, the bits in the robot chromosomes are grouped eight by eight and stored as bytes in hexadecimal.

```
G01a10700101010000010001010100000001000000010001010100000001111000100000001O
01010000010101000101010100000000000000101000000010001010100010101000100000010
0001010101000100000000010101000001000000100000100001010000010100010001400014AFA
8D27C014AFA8D27CD6BED87613DDB01F44A8EE8E0957500010100010004 2CFFFDFD

                                   . . .

0001000100000000100010100000100010000010100010000000001000100000101000100010 0
0001000000001010110001010000000000B3C17FB32DB789E72CF4B7F3153FABC0986719
1597156463D6B8581F39720B55488038421100010100010001435D8BA2C13DB690235DA
```

Figure 6.3 – Data obtained by the monitor computer stored in GENE.TXT.

Another program is then necessary to convert that information into a file that can be opened by Microsoft Excel. This software is called GRAFxx.CPP and is also listed in Appendix A for every experiment. It produces a file called GRAF.TXT. Figure 6.4 shows an example of the output generated by the program GRAF20.CPP. Data is displayed in rows containing the generation (*Gen*), followed by the current fitness value of all robots (*R1* to *R6*), plus the average fitness (*Av*) and the fitness of the fittest robot (*F*). Then, the information in this file can be displayed graphically.

```
Gen: 1 R1= 3928 R2= 3449 R3= 3922 R4= 4377 R5= 3390 R6= 4005 Av= 3845 F= 4377
Gen: 2 R1= 3671 R2= 4023 R3= 3525 R4= 3954 R5= 2806 R6= 3598 Av= 3596 F= 4023
Gen: 3 R1= 3899 R2= 4047 R3= 3893 R4= 4028 R5= 2966 R6= 2778 Av= 3601 F= 4047
Gen: 4 R1= 3870 R2= 3716 R3= 2861 R4= 4381 R5= 3824 R6= 2778 Av= 3571 F= 4381
Gen: 5 R1= 3411 R2= 3716 R3= 2777 R4= 3791 R5= 3852 R6= 2777 Av= 3387 F= 3852

                                   . . .

Gen: 147 R1= 4329 R2= 4364 R3= 4359 R4= 4354 R5= 4333 R6= 4360 Av= 4349 F= 4364
Gen: 148 R1= 4327 R2= 4327 R3= 4351 R4= 4097 R5= 4301 R6= 4327 Av= 4288 F= 4351
```

Figure 6.4 – Data converted into a file called GRAF.TXT, containing the fitness of the robots for every generation.

6.2 Experiment 1: The Influence of the Lifetime

In this first experiment, the influence of the robot lifetime, or the duration of a generation, is analysed. To differentiate between individuals with the same degree of adaptation that produce similar fitness values, the duration of a generation needs to be carefully specified to submit these individuals to the right selection pressure [Sim99] [Wat99a]. A possible alternative is to increase the duration of the generations at the end of the evolutionary experiment, to introduce more selection pressure when the robots start to behave similarly. This experiment intends to perform a qualitative analysis of the effects of the generation time on the evaluation of the robots.

- **Aim of the Experiment**

This experiment aimed to find the correct duration for a generation and to set reference standards to future experiments. To achieve this, the controller circuit was fixed and robots with different sensor configurations were tested continuously for a long period. Then, their fitness curves could be compared to indicate the optimal duration of a generation that is able to inflict the correct selection pressure and discriminate between two similar robots. During the evolutionary process, the robots were not allowed to reproduce and did not suffer any kind of mutation. The initial control circuit and morphology were preserved during the complete test.

- **Experimental Setting**

To produce a fixed controller that could navigate the robot during the experiment, the neural network that was implemented as the navigation control circuit of the robots was trained to behave according to a hand-designed controller. This controller operates according to the following algorithm:

```
-  If (All Sensors=0) then Command = FF;
P  If (Sensor2=1) then Command = TRS1;
r  If (Sensor3=1) then Command = TRS1;
i  If (Sensor4=1) then Command = TRS1;
o  If (Sensor6=1) then Command = TLS1;
r  If (Sensor7=1) then Command = TLS1;
i  If (Sensor8=1) then Command = TLS1;
t  If (Sensor1=1) then Command = TRS2;
y  If (Sensor1=1 and Sensor4=1) then Command = TRS1;
+  If (Sensor1=1 and Sensor6=1) then Command = TLS1;
```

In the experiment all sensors were set to operate at *medium* range, detecting obstacles closer than 15cm (see Figure 4.30). Sensor *S5* in the back of the robot has not been connected to the neural network, so it does not matter if it is selected or not by the sensor module. The sensor positions around the robot are presented in Figure 6.5.

The neural network used four commands to control the motor drive module: *Front Fast* (*FF*); *Turn Left Short1* (*TLS1*); *Turn Right Short1* (*TRS1*); and *Turn Right Short2* (*TRS2*). *Front Fast* means "move forward with maximum speed". To turn left/right short, the robot moves with reverse direction in one of its motors (with both motors at maximum speed), causing a spin around its own axis. The difference between *TRS1* and *TRS2* is that in the later, the robot keeps turning for 200ms, while the duration of the other

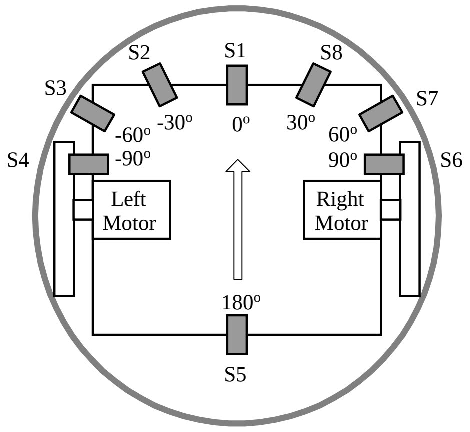


Figure 6.5 – Position of the sensors on the robot and their angle in relation to the central line of the robot.

three commands is just one iteration (10ms more or less). Using only four output commands means that the neural network needs only two bits to encode them and four classes of discriminators to work. The neural network architecture shown in Sections 3.2.5 and 4.1.1 is preserved and its configuration has four groups of seven neurons (*m*=4 and *n*=7) with two inputs each (the neuron size is four bits). Figure 6.6 shows how this configuration is connected to all seven sensors. Differently from other experiments, the interconnections between sensors and neurons are fixed, not random. Each neuron has four bits of memory to store its contents. The neural network has 28 neurons with a total memory size of 112 bits. This would also be the size of the corresponding bit string in the chromosome, if the neural network was to be evolved. The winner-takes-all block chooses the command that has more active neurons and encodes it with two bits, before sending it to the motor drive module. Other configurations are also possible, but this was the smallest network amongst the tested ones that could learn, without saturating, the right command for all possible 128 input combinations that the controller can face.

The neural network was trained for all input possibilities that it can face: for seven 1-bit sensors, it has 128 possibilities. For each input vector (0000000 to 1111111), the current output of the hand-designed controller was calculated and "1" was written to all neurons of the same class (the command output from the hand-designed controller) in the position addressed by the input vector. When the last input vector was trained, the network was able to respond exactly as the hand-designed controller. The contents of the neurons were not modified again in the experiment.

The speed of both motors was fixed at the maximum setting (Vmax = 32), enabling the robots the move at 0.26m/s. As the controller configuration and speed levels were fixed, the only feature that varied was the selection of the sensors. The sensor module can enable or disable each sensor according to two bits stored in the chromosome. For each robot tested, these bits were manually initialised so that different sensor configurations could be tested in the same experiment.

The experiments were run in two different environments: a simple one, with just the outer walls; and a complex one, containing many round obstacles of different sizes. Figure 6.7 shows the configuration of the simple environment and Figure 6.11 shows the complex one. All experiments were run for 30 minutes, so that the results could be compared.

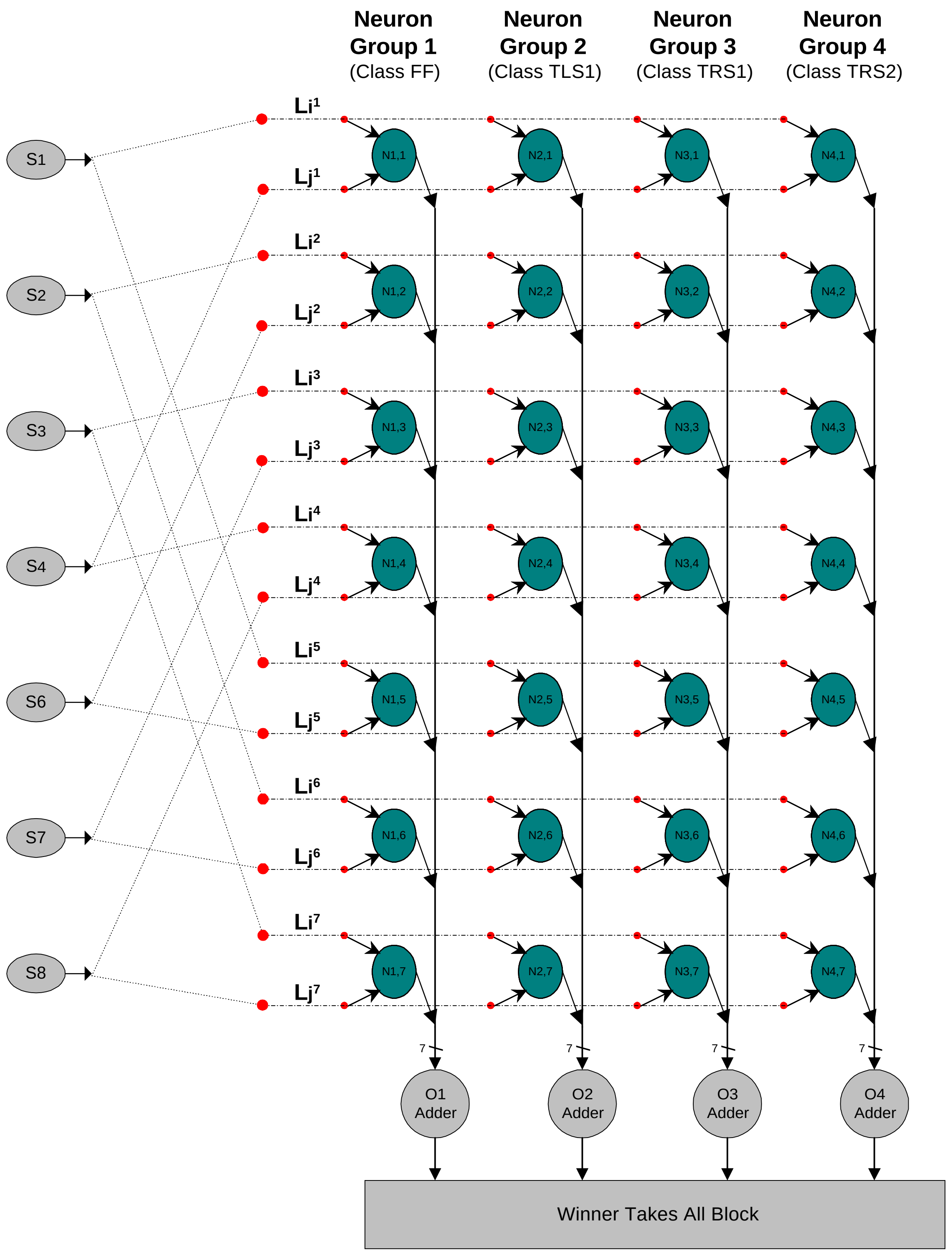


Figure 6.6 – Configuration of the neural net with four groups of seven neurons.

The data in these experiments were sampled by the program EXP01.CPP and converted to a Microsoft Excel file by the program GRAF01.CPP, both written specially for this experiment using the Borland C++ environment version 5.01. These programs are listed in Appendix A.

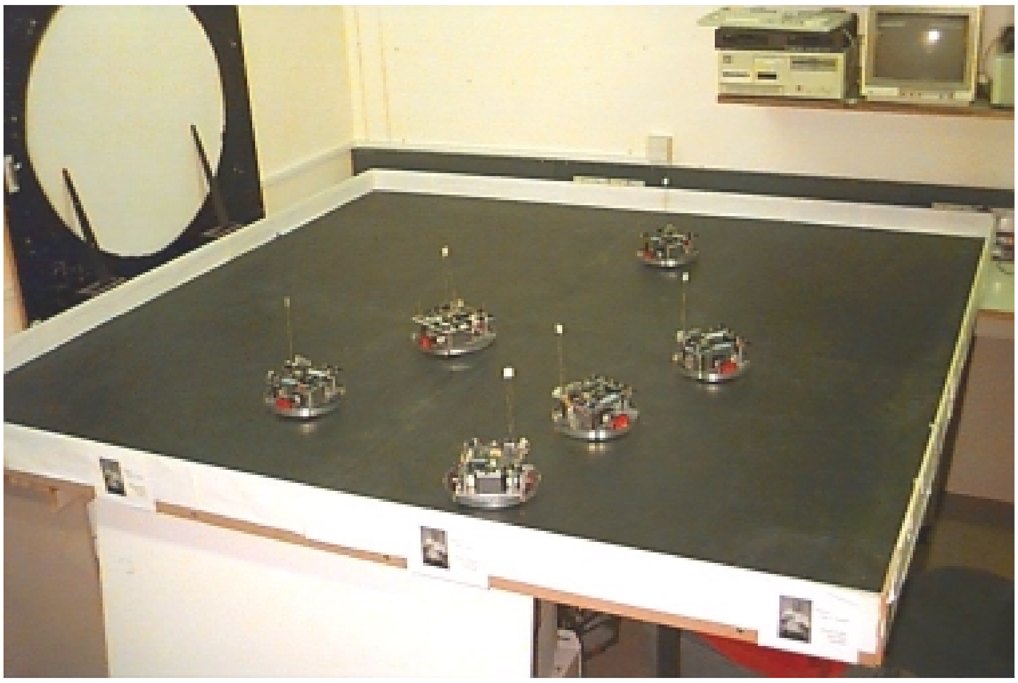


Figure 6.7 – Configuration of the simple environment for the experiments in this section.

The strategy for rewarding and punishing the robot fitness function for every test in this experiment was:

1- Start with 4096 points;

2- Reward: increase fitness by 5 points every 3 seconds of forward movement;

3- Punishment: decrease fitness by 10 points every collision.

Each robot is given 4096 points ($1000 in hexadecimal) at the beginning of the test. According to the reward rule, the maximum that a robot can score if it does not crash during the 30 minutes of the test is 7096 points. Therefore, it can be observed in the following charts that even the best-adapted robots still collide sometimes.

The robot sensors can be enabled by fixing their control bits in the chromosome. In this way, the corresponding pair of bits in the chromosome is set to "11" to disable the sensor, and "00" to enable it (see Section 4.3.5). The legend of the charts indicates the colour of the curve of the robot that has the corresponding sensor(s) enabled. For example, *S1*,*S2* indicate that the sensors *S1* and *S2* are enabled, and all the others disabled. The corresponding 16 bits in the chromosome are: 11-11-11-11-11-11-00-00 (observe that the order of the sensor control bits is: *S8*-*S7*-*S6*-*S5*-*S4*-*S3*-*S2*-*S1*). The position of the sensors in the robot is indicated in Figure 6.5.

6.2.1 Experiment 1.1: the Simple Environment

To facilitate comparison of the curves in the tests with the simple and complex environments, the robots with the same sensor configurations are represented by the same colours. Even in this simple environment, where there are no obstacles, the interaction among the robots was intense, causing many collisions [Set97]. This happened because a fit robot, with the most important sensors enabled, for example, can be crashed into by a "blind" robot, which has all sensors disabled. Therefore, the results would be different if the robots were tested alone in the environment, but this will not be the case in the developed evolutionary system, so the presence of other unfit robots is very important for evolution and must be considered. Figure 6.7 shows the configuration of the simple environment. Table 6.2 presents a summary of the settings for this experiment.

Table 6.2 – Summary of the Experimental Settings

Parameter	Definition
Fitness Function:	+5 points every 3s moving forward –10 points each collision
Initial Fitness Value:	4096 points
Maximum Fitness Value:	7096 points
Generation Time:	Population fixed at the first generation: 30 minutes
Mutation Rate:	Not present
Speed Levels:	Fixed at maximum speed (Vmax=32)
Sensors Enable:	All sensors but *S5* can be enabled independently *S5* is permanently disabled
Navigation Controller:	Fixed Neural Network (m=4, n=7, neuron size=4 bits)
Software:	EXP01.CPP and GRAF01.CPP (in Appendix A)

- **Results**

Two tests were performed with the simple environment. In the first one, Robot 1 had the sensors *S1* and *S4* enabled; Robot 2 had *S1* enabled; Robot 3 had *S1*, *S4* and *S6*; Robot 4 had *S3*; and Robot 5 had *S1*, *S3* and *S7* enabled. Robot 6 was not used in the test. Figure 6.8 shows the fitness results from the first test.

The second test was performed with all six robots. Robot 1 had sensors *S1*, *S2,* and *S8* enabled; Robot 2 had *S4* and *S6*; Robot 3 had *S3* and *S7*; Robot 4 had *S2* and *S6*; Robot 5 had *S2* and *S8*; and Robot 6 had the sensors *S2* and *S7* enabled. Figure 6.9 shows the curves of the second test.

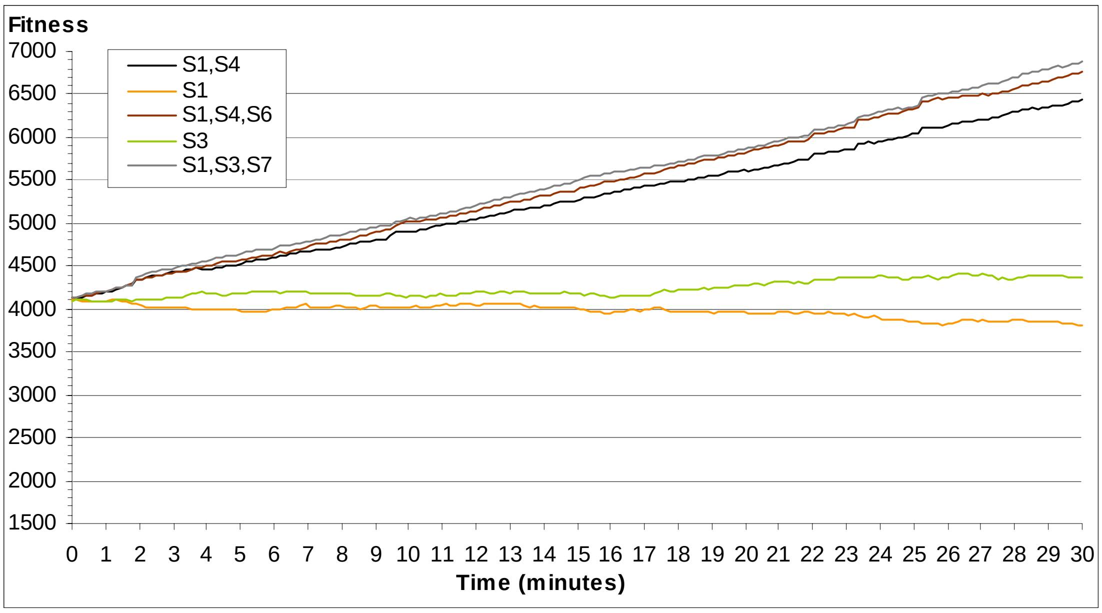


Figure 6.8 – ***Experiment 1.1a***: Comparison of five different sensor configurations evaluated in the simple environment during 30 minutes. Both controller and morphology are fixed during the experiment. The tested configurations are: *S1,S4*; *S1*; *S1,S4,S6*; *S3*; and *S1,S3,S7*.

- **Discussion**

As expected, both charts showed that robots with two sensors collided less than robots with just one of their sensors active. In the first test (Figure 6.8), the robots with only *S1* and *S3* performed less fit than *S1,S4,S6*, *S1,S4*, and *S1,S2,S7*. Contrary to expectation, *S3* behaved better than *S1* in this experiment. This has happened because *S3* can actually pick some IR reflection from the front as well as from the left side of the robot, and, in this way, protected the robot from colliding into a greater range of possible obstacles.

The robots with *S1,S4,S6*, and *S1,S3,S7* were able to avoid most of the obstacles with almost full cover of their surroundings. As expected, *S1,S4* was not better than *S1,S4,S6* or *S1,S3,S7*, and the later was the best configuration in this test. This can be explained by the fact that the sensors *S3* and *S7* with an angle of 60^o are better in detecting obstacles than *S4* and *S6*, with an angle of 90^o. Nevertheless, the robots with sensors *S1,S4,S6* and *S1,S3,S7* had roughly the same behaviour, showing that the combination of *S4* and *S6* is almost as effective as *S3* and *S7*.

Figure 6.8 also showed that having only two sensors is not a major problem in a simple environment, since the robots can spot the walls in most situations with only

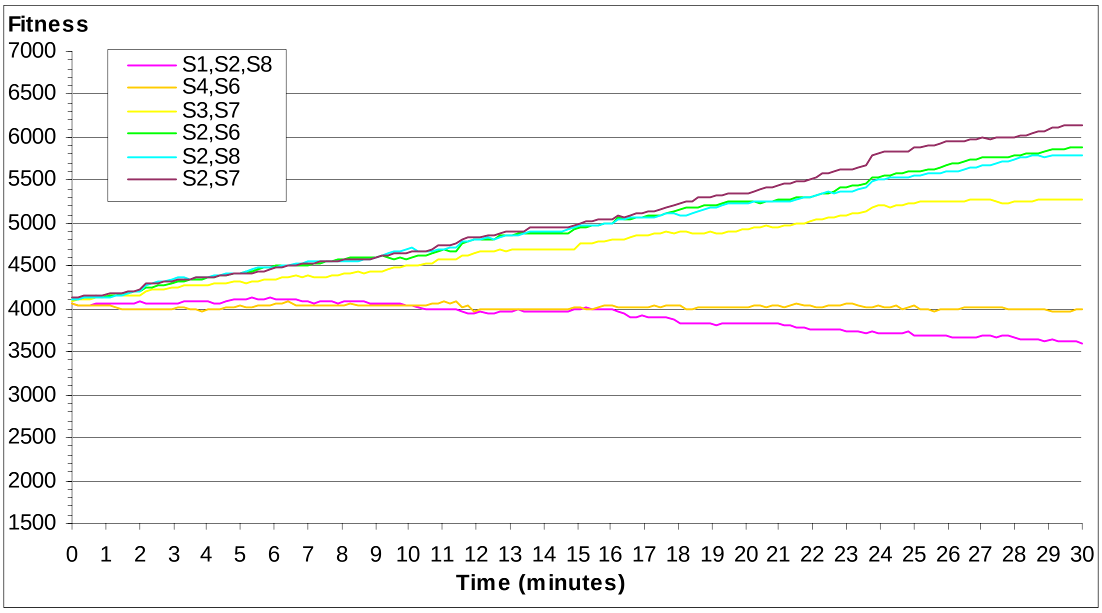


Figure 6.9 – ***Experiment 1.1b***: Comparison of six different sensor configurations evaluated in the simple environment during 30 minutes. Both controller and morphology are fixed during the experiment. The tested configurations are: *S1,S2,S8; S4,S6*; *S3,S7*; *S2,S6*; *S2,S8*; and *S2,S7*.

one or two sensors. The major cause of collisions in this case was the approach of another robot. This demonstrates the importance of using real robots rather than simulation, since this sort of interaction would be difficult to model [Set97] [Jak98b].

Figure 6.9 shows that robots that only have lateral sensors (*S4* or *S6*) could not sense obstacles in their front and hence gave bad performance. The figure suggests that sensors in the front are more important than lateral ones. *S3,S7* performed better than *S4,S6* in the figure, since the robot had the ability to sense obstacles in the front as well as lateral ones. *S2,S7*, *S2,S8*, and *S2,S6* showed excellent performance, behaving roughly in the same way until more or less 16 minutes and 30 seconds, when *S2,S7* started to improve on the others. This can be explained by the fact that robot *S2,S7* was running out of battery and moving slower than the other two. The other two robots had fully charged batteries and were running faster than *S2,S7*. The battery charge is an important factor. If the robot is moving faster, it will have less time to manoeuvre after spotting an obstacle and has a greater chance of colliding. This was observed here for the first time and since then, all robots in future experiments had their batteries equally charged at the start of the tests to prevent this effect in their evaluation.

Figure 6.9 shows that, against what was expected, the robot with sensors *S1*, *S2*, and *S8* active had the worst performance. This exemplifies how an evolutionary system

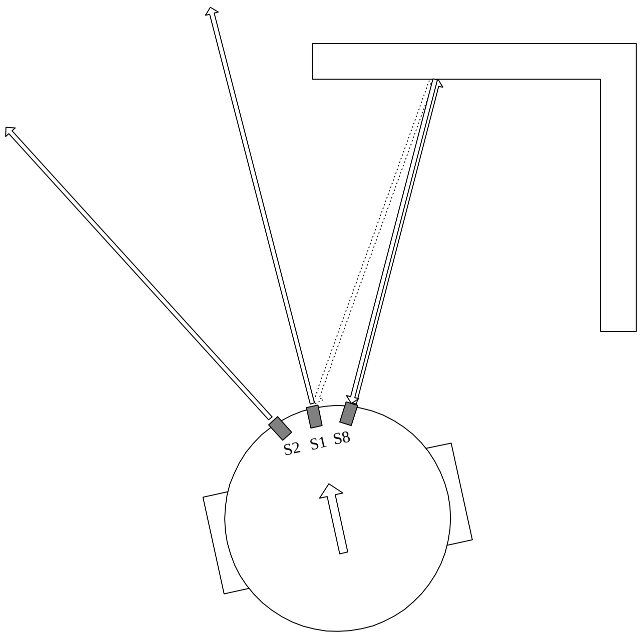


Figure 6.10 – The reflected infrared beam (regular arrows) from one sensor interferes (dotted arrow) with another.

can contribute to robot design, where the general idea would be fitting as many sensors as possible to the robots. In this case, however, the experiment showed that there is a problem with the *S1,S2,S8* configuration and a conflict between sensors *S1* and *S8* was observed. It only happens because of the way the controller was designed.

There is an instruction (*If (Sensor1=1) then Command = TR2*) in the algorithm of the hand-designed controller (see experimental settings) that orders the robot to turn right if sensor *S1* is activated. As *S1* and *S8* are close to each other, in the front of the robot, sometimes in the corners both sensors can sense an object at the same time and as sensor *S1* would tell the robot to turn right, sensor *S8*, in the other hand, would make it turn left with a conflicting command. Figure 6.10 shows when this interference can happen. The infrared beam from sensor *S8* can reflect back to the sensor (the regular arrows in the figure), but can also be detected by another sensor, such as sensor *S1* in the figure. The same interference can happen between sensor *S1* and sensor *S2*, but as both of them turn the robot to the right, it does not generate a problem. As the command line of sensor *S1* has greater priority in the algorithm than the command line of sensor *S8*, if sensor *S1* detects interference, it will order the robot to turn right. Then, sensor *S8* orders it to turn left, only to be turned right again by another interference, causing the robot to oscillate many times until it escapes that trap in the controller design. Every time an evolutionary system is designing a robot controller for an environment where this problem

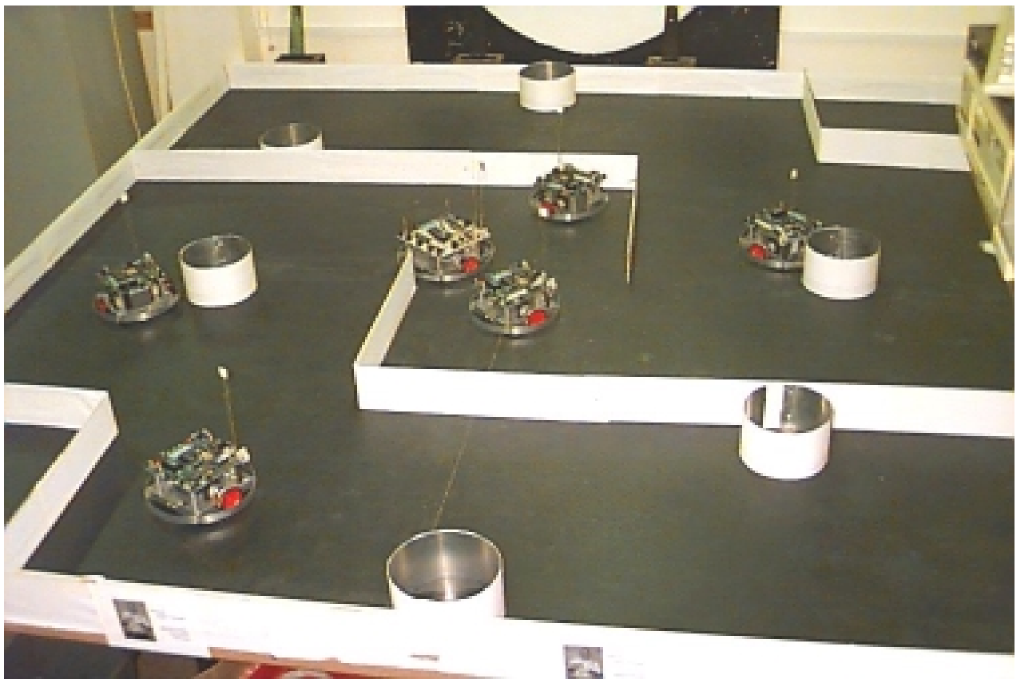


Figure 6.11 – Configuration of the complex environment for the experiments in this section.

can happen, that configuration of sensors will have a lower fitness value, and will be discarded as a good solution or a controller will be produced that can deal with this situation.

6.2.2 Experiment 1.2: the Complex Environment

The robots were also tested in a complex environment. Figure 6.11 shows how the complex environment was obtained by including many round obstacles of different sizes. The obstacles increase selection pressure because there are more collision possibilities and the population needs to manoeuvre much more. This increases the "temperature" of the system, since the population is much more agitated. Since the selection pressure was increased by the addition of obstacles, less time is expected to differentiate between fit and unfit sensor configurations. The same scale from the charts of the simple environment was preserved to facilitate comparison between the performances of the robot with the same sensor configuration. The same colours were also preserved from the curves of the previous charts, allowing the robots using the same sensors to be

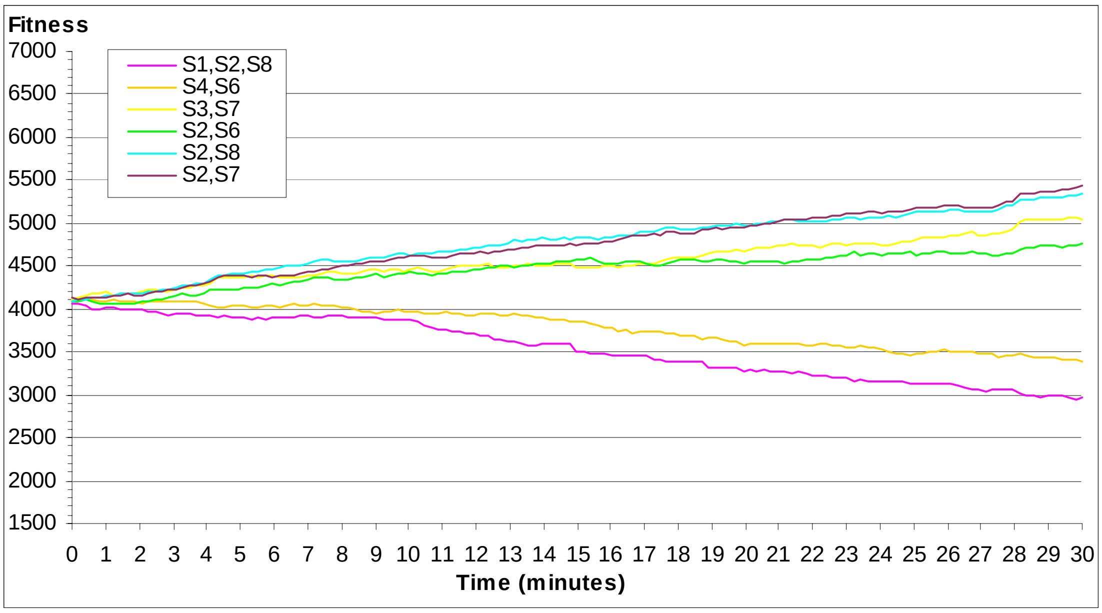


Figure 6.12 – *Experiment 1.2a*: Comparison of six different sensor configurations of Experiment 1.1b now tested in a complex environment during 30 minutes. Both controller and morphology are fixed during the experiment. The tested configurations are: *S1,S2,S8; S4,S6*; *S3,S7*; *S2,S6*; *S2,S8*; and *S2,S7*.

displayed with the same colours. All robots that were tested for the simple environment were also tested in the complex one. This experiment, though, tested some other configurations as well. As only the configuration of the environment changed, the same experimental settings presented in Table 6.2 can also be applied here.

- **Results**

This section presents four tests with the complex environment. The first, shown in Figure 6.12, presents the same sensor configurations of Figure 6.9. The second test was performed with six robots. Two of them, Robot 2 using sensor *S1* and Robot 5 using sensor *S3*, were tested before in the simple environment in Experiment 1.1a. Other configurations were tested by Robot 1, which had all sensors disabled; Robot 3 using only *S4*; Robot 4 using *S6*; and Robot 6, which had only sensor *S7* enabled. Figure 6.13 shows the curves of this second test.

The third test was performed with five robots. Two of them, Robot 1 using sensors *S1*, *S3* and *S7* and Robot 2 using sensors *S1* and *S4*, were tested before in the simple environment in Experiment 1.1a. Other configurations were tested by Robot 3 having sensors *S1* and *S3* enabled; Robot 4 using only *S1* and *S6*; and Robot 1, which had

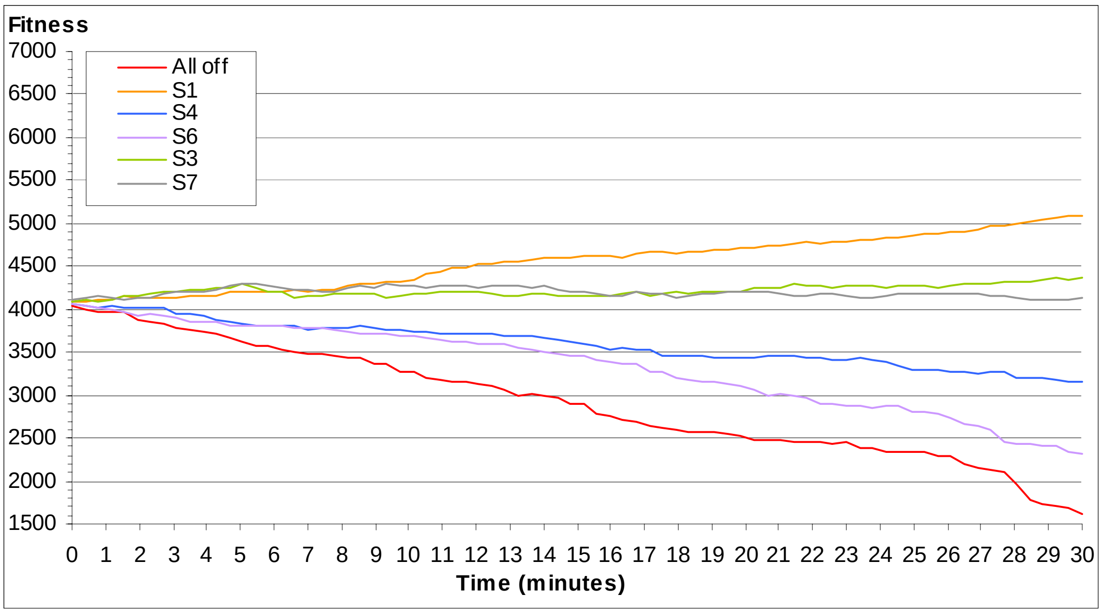


Figure 6.13 – ***Experiment 1.2b***: Comparison of six different sensor configurations evaluated in the <u>complex environment</u> during 30 minutes. Both controller and morphology are fixed during the experiment. The tested configurations are: *All off*; *S1*; *S4*; *S6*; *S3*; and *S7*. The configurations *S1* and *S3* were tested before in <u>Experiment 1.1a</u>.

the sensors *S1* and *S7* enabled. Figure 6.14 shows the curves of the third test. The fourth test in Figure 6.15 shows the performances of five different sensor configurations that are combinations of sensor *S1* with some other sensors.

- **Discussion**

Figure 6.12 shows a greater selection pressure than the one experienced by the robots in Experiment 1.1b. It indicates that *S1,S2,S8* and *S4,S6* were still the worst configurations, colliding even more in this more complex environment. *S2,S7* was still the best one, proving the power of the frontal sensors in spotting the obstacles in the way of the robot.

Generally, all performances were degraded by the inclusion of more obstacles in the way of the robots. *S2,S8* repeated a good performance, but *S2,S6* dropped to fourth place, behind *S3,S7*. This happens probably because of *S6* deficiency in detecting the obstacles that approach the robot, if compared to *S7*. This showed once more that a more forward placement of the sensors, such as *S3* and *S7*, or even as *S2* and *S8*, is better than a more lateral one, such as *S4* and *S6*.

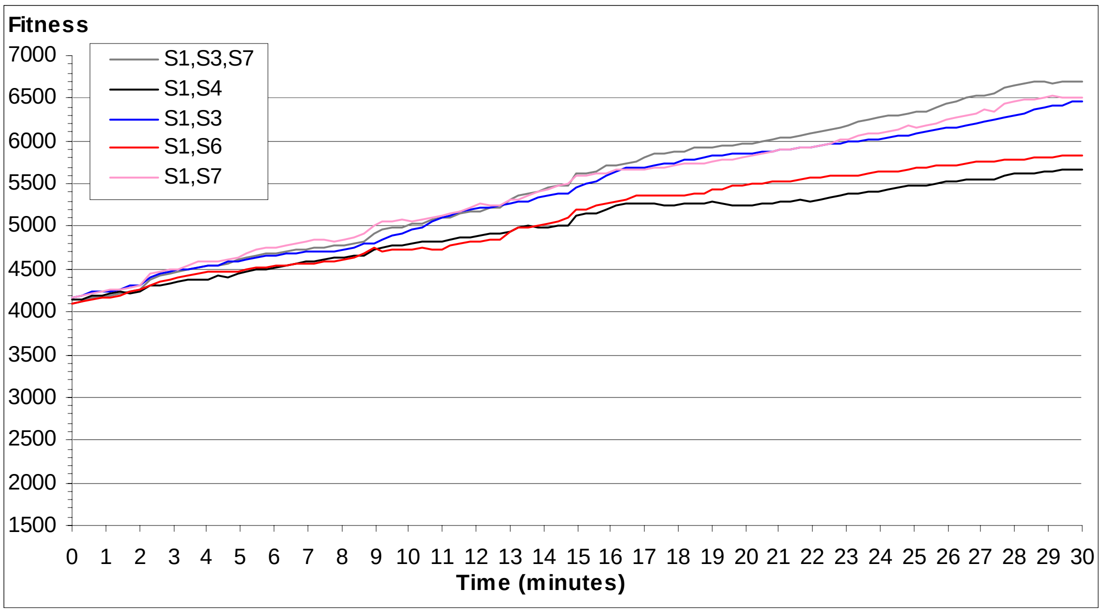


Figure 6.14 – ***Experiment 1.2c***: Comparison of five different sensor configurations evaluated in the complex environment during 30 minutes. Both controller and morphology are fixed during the experiment. The tested configurations are: *S1,S3,S7*; *S1,S4*; *S1,S3*; *S1,S6*; and *S1,S7*. The configurations *S1,S3,S7* and *S1,S4* were tested before in Experiment 1.1a.

Figure 6.13 compared single sensor solutions to a "blind" robot, which had all its sensors disabled. Robot 1, using *S1*, and Robot 2, using *S3*, were evaluated before in the simple environment of Experiment 1.1a (see Figure 6.8). *S3* showed roughly the same performance even with the addition of more obstacles, but *S1* performed better. Actually, *S1* performance in this experiment was more consistent with its sensing capability. It showed that a frontal sensor is more important than a lateral one when the environment is full of obstacles, and its bad performance in Experiment 1.1a was probably because of its fully charged battery, which made it travel faster.

Figure 6.14 shows three similar configurations differing only by the presence of sensors *S3* or *S7*: *S1,S7; S1,S3*; and *S1,S3,S7*. The robot that had three sensors activated, understandably, had better performance than *S1,S7* and *S1,S3*, which had roughly the same fitness value, indicating that a robot with a right sensor has the same probability of success as a robot with the corresponding left sensor. The same can be said of *S1,S4* and *S1,S6* that had similar performances, with *S1,S4* colliding much more than in the previous experiment in the simple environment.

Figure 6.15 shows a tight competition of three sensor configurations: *S7,S6,S4,S3,S2,S1*; *S7,S6,S3,S1*; and *S7,S3,S2,S1*. This once more illustrated the

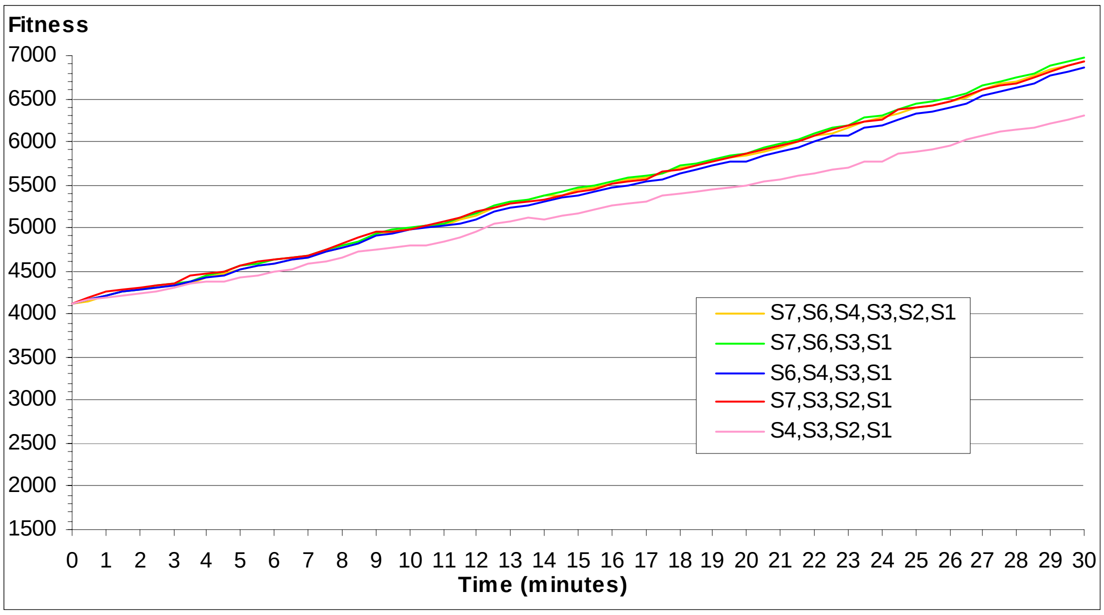


Figure 6.15 – *Experiment 1.2d*: Comparison of five different sensor configurations evaluated in the <u>complex environment</u> during 30 minutes. Both controller and morphology are fixed during the experiment. The tested configurations are: *S7,S6,S4,S3,S2,S1; S7,S6,S3,S1; S6,S4,S3,S1; S7,S3,S2,S1*; and *S4,S3,S2,S1*.

importance of *S7* and *S3* angled 60^o in both sides of the robot. *S7* and *S3* showed again superior to *S6* and *S4*, which form a 90^o angle with the robot central line. The *S6,S4,S3,S1* curve started to show a lower performance after three minutes, suggesting that a shorter generation time would not be enough to make any distinction between these two groups of sensors. The sensor configuration *S4,S3,S2,S1* was very powerful, but not as much as the others. Its curve started to differ from the others in the first minutes of the experiment, suggesting that a short lifetime would be enough to differentiate among these simple sensor combinations and richer ones, containing at least one sensor in the front, and one at each side of the robot.

6.2.3 Discussion of the Experimental Results

Evolution can be used to determine the best sensor configuration for the population of robots, since it was possible to differentiate between fit and unfit sensor configurations in the experiments. It was shown that the robots with just a few well-placed sensors performed better than the ones with conflicting, badly-placed sensors. As fit sensor configurations had very similar curves according to the selected fitness function, it was observed that this function could not discriminate between some of the best configurations, even after 30 minutes of evaluation. The implication of this is that using these settings, the optimal solution may never be found by the evolutionary technique with this fitness evaluation function. However, an optimal solution may not even exist for this problem, since even different battery charges can influence the performance of some solutions more than the configuration of their sensors. Small fitness differences (e.g., less than 100 points) after 30 minutes must be regarded as the same performance.

The aim of this experiment was to judge the amount of time that can safely discriminate between a good and a bad performance. It was observed that a generation that lasts at least one minute is enough to differentiate between fit and unfit sensor configurations. However, five or ten minutes may be necessary to choose the best among similar good solutions. In this case, an evaluation strategy that considers the gradual increase of the duration of the generations towards the end of the evolutionary experiment seems very attractive. This permits a better evaluation of the fitness of the solutions, where the robots must navigate for many minutes until the best performances can be selected.

This experiment showed so far that there might not be an optimal solution to the best sensor configuration. The scores of the best sensor configurations were very close and if a short generation time is applied, the noise of a real embedded evolutionary system will probably obscure these scores, making it very difficult, if not impossible, for evolution to find a definitive solution. What can be expected, however, is an evolutionary process that will converge to a group of best solutions, containing at least three sensors, one frontal, and two lateral ones. A short lifetime, and even a much longer one, will not apply the necessary selection pressure to differentiate among them. If mutation is active, the final result will probably be a population where all these best solutions coexist and keep reappearing in the population every now and then.

6.3 Experiment 2: Evolving the Sensor Configuration

In this experiment, an embedded evolutionary system was allowed for the first time to evolve the morphology of a robot population. To date, no other author has attempted this using a fully embedded evolutionary controller [Pol00]. Evolution could manipulate the specific genes in the robot chromosome that define the sensor configuration: the first eight pairs of bits. By evaluation, selection, and recombination, evolution was expected to be able to select the best sensor configuration to navigate the robots according to a small variation of the hand-designed controller described in Experiment 1. This hand-designed controller operated according to the following algorithm:

```
     Left = Right = 0;
  -  If (Sensor4=1) then Left = Left + 1;
  P  If (Sensor3=1) then Left = Left + 1;
  r  If (Sensor2=1) then Left = Left + 1;
  i  If (Sensor6=1) then Right = Right + 1;
  o  If (Sensor7=1) then Right = Right + 1;
  r  If (Sensor8=1) then Right = Right + 1;
  i  If (Sensor1=1) then Command = TRS2;
  t  If (Left > Right) then Command = TRS1;
  y  If (Left = Right) then Command = FF;
     If (Left < Right) then Command = TLS1;
  +  If (Sensor1=1 and Sensor4=1) then Command = TRS1;
     If (Sensor1=1 and Sensor6=1) then Command = TLS1;
```

The *Left* and *Right* variables were created to prevent the robot from turning left when more left sensors are detecting obstacles than the right sensors, as it was happening in the previous hand-designed controller from Experiment 1. This did not happen in this controller, since the robot will turn to the side where fewer sensors are detecting obstacles.

In this experiment, all sensors could be enabled by their controlling pair of bits in the chromosome. It will be recalled that both controlling bits need to be “0” to enable the corresponding sensor. Any other combination (e.g., “01”, “10”, and “11”) disables it. Therefore, to select sensors *S1* and *S2* only, the corresponding 16 bits must be for example 01-11-10-10-11-01-00-00. Observe that the first pair of bits in the chromosome controls sensor *S8*, followed by *S7*, *S6*, *S5*, *S4*, *S3*, *S2*, and *S1*. The phenotype with only sensors *S1* and *S2* enabled has many corresponding genotypes. As three combinations of both control bits disable each one of the other six sensors (e.g., "01", "10", and "11"), this phenotype can be produced by 3^6 or 729 genotypes. This genetic system is presenting some n*eutrality* [Ler76] and the reason for this is that the sensors have a higher probability (75%) of being disabled than enabled. This helps co-evolving the sensor configuration and the controller, since once a sensor is enabled it gives more time for the controller to learn how to use it before another sensor appears, forcing the controller configuration to adapt again [Shi00].

- **Aim of the Experiment**

This experiment aimed to test if the embedded evolutionary controller described in Chapter 4 was able to evolve part of the robot morphology: the sensor configuration. To achieve this, the controller circuit was fixed by training the neural network, the same architecture used in the previous experiment, to behave according to the hand-designed controller described above. The neural network was shown in Figure 6.6 and was trained as described in Experiment 1. It used the same four commands to control the motor drive module: *Front Fast* (*FF*); *Turn Left Short1* (*TLS1*); *Turn Right Short1* (*TRS1*); and *Turn Right Short2* (*TRS2*).

- **Experimental Setting**

A population of robots with different sensor configurations was evaluated in 30-second generations. The robots were allowed to reproduce and mutate, but only with the first 16 bits that enable the sensors. The initial control circuit was preserved during the complete experiment.

The chosen selection strategy for this evolutionary experiment was:

- *Select the robot with the highest fitness value in the generation to breed with all other robots and survive to the next generation.*

This strategy means that the robot with the highest fitness is selected, and will send its chromosome to the other five robots. Each one of the remaining five robots then combines its own chromosome with the one received from the best robot to produce a resultant chromosome. Next, the bits of this resultant chromosome are randomly selected to be mutated (logically inverted) according to the selected mutation rate. Then, the remaining five robots reconfigure themselves with the mutated chromosomes and the robots continue in the next generation. This tries to make sure that in the next generation the best fitness will be at least similar to the present one, since, at least, the surviving best robot has the same chances of repeating its good performance.

In the crossover phase, where the two chromosomes are combined to form a resultant offspring, each bit is randomly chosen from the corresponding location in the chromosome of the parents. Then, in the mutation phase, a random number *r* is generated between zero and 100 for each bit in the chromosome, and the bit will be flipped (from “0” to “1”, or “1” to “0”) each time the *r* is smaller than the mutation rate.

Sensor *S5* in the back of the robots has not been connected to the neural network, so it does not matter if it is enabled or not by the sensor module. In this experiment, all sensors have only one bit of precision and they were set to work at *medium* range (see Figure 4.30). The velocity of the motors in this experiment was again set to the maximum speed.

The selected fitness function, for every test in this experiment was:

1- Start with 4096 points;

2- Reward: increase fitness by 5 points every 1 second of forward movement;

3- Punishment: decrease fitness by 10 points every collision.

The maximum that a robot can score if it does not crash during the 30 seconds of the generation is 4246 points. Figure 6.16 shows the environment where this experiment was run. It was built with medium complexity, containing a few round obstacles of different sizes.

This experiment selected the best possible combinations of the eight sensors *S1*, *S2*, *S3*, *S4*, *S5*, *S6*, *S7*, and *S8*. The number of possible combinations of the eight sensors is: $2^8 = 256$ combinations. It is a relatively small search space. Hence, fast convergence was expected, since many combinations present very similar good performances (as demonstrated in Experiment 1) with fitness values near to the maximum

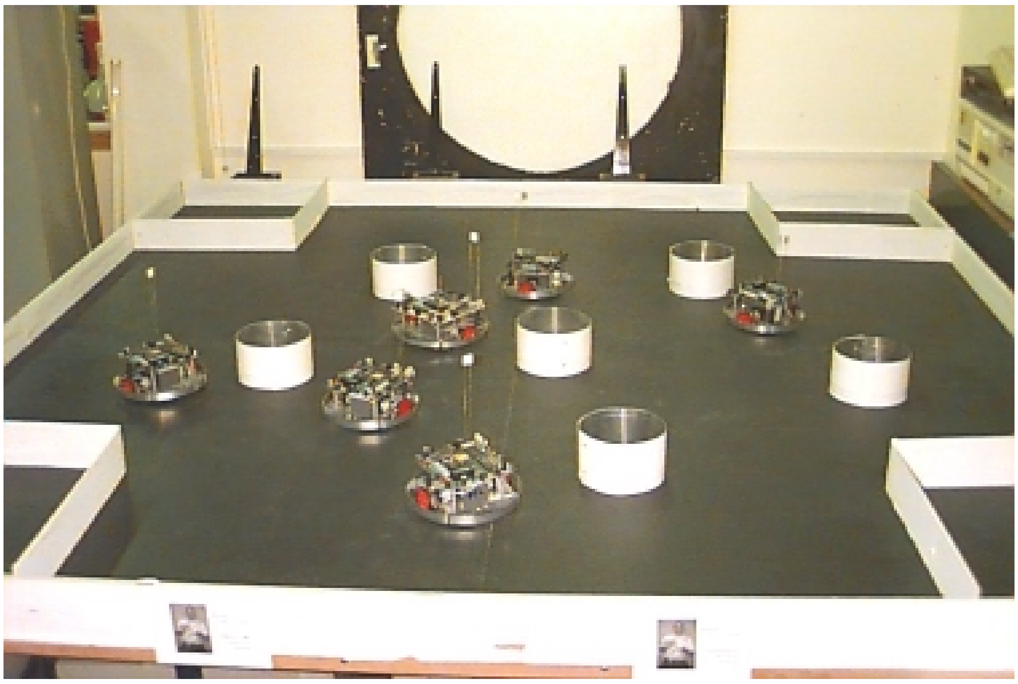


Figure 6.16 – Configuration of the environment used in this experiment.

score. For all four tests in this experiment, the population was initialised with all sensors disabled (i.e., with “10-01-10-01-10-01-10-01” written to the chromosomes of all robots). “10-01-10-01-10-01-10-01” was chosen instead of “11-11-11-11-11-11-11-11” to include more variety in the chromosomes (i.e., a chromosome containing different combinations of “0” and “1” has more variety of genes than one with all bits set to “1” or “0”, because it can produce more different combinations). Therefore, it is easier to produce a pair of zeros, enabling the sensor. The system would have to rely entirely on mutation to produce zeros, if the combination “11-11-11-11-11-11-11-11” had been chosen. Table 6.3 presents a summary of the settings for this experiment.

Table 6.3 – Summary of the Experimental Settings

Parameter	Definition
Fitness Function:	+5 points every 1s moving forward –10 points each collision
Initial Fitness Value:	4096 points
Maximum Fitness Value:	4246 points
Generation Time:	30 seconds
Mutation Rate:	0%, 5%, 10%, 20%, and 40%
Speed Levels:	Fixed at maximum speed (Vmax=32)
Sensors Enable:	All sensors can be enabled independently
Navigation Controller:	Fixed Neural Network (m=4, n=7, neur. size=4 bits)
Initial Sensor Configuration:	“10-01-10-01-10-01-10-01”
Initial Speed Configuration:	“1111111111”
Initial Controller Configuration:	Fixed from trained neurons.
Software:	EXP07.CPP and GRAF07.CPP (listed in Appendix A)

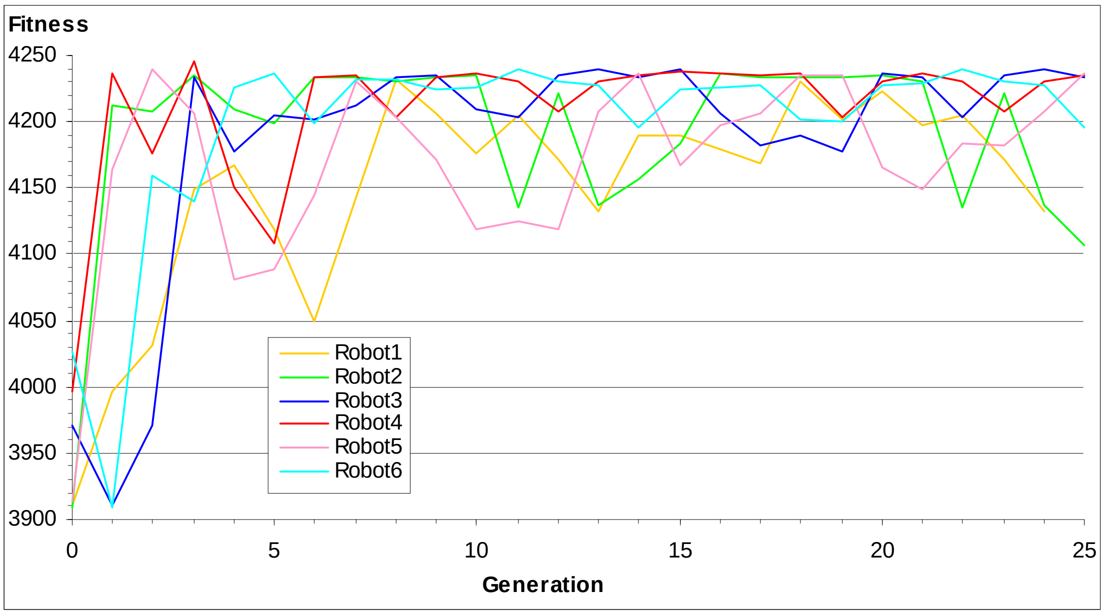


Figure 6.17 – ***Experiment 2.1***: Evolution of the sensor configuration in an environment of medium complexity, mutation rate of 5%, sexual reproduction, and generation time of 30s. The controller and speed levels are fixed during the experiment. Here, *Robotn* is the fitness of Robot *n* in the generation.

- **Results**

This experiment shows three charts representing three different evolutionary experiments, run in the same environment with six robots and different mutation rates: 0%, 5%, 10%, 20%, and 40%. To facilitate comparisons, all five evolutionary experiments run for 25 generations. Figure 6.17 shows the first evolutionary experiment with mutation rate of 5%. A good solution with a high fitness value was found by evolution much faster than expected. After the first two or three generations, a solution that scored at least 90% of the total possible points was found in all four evolutionary runs. The fitness values of the population dispersed more as the mutation rate increased.

Figure 6.17 shows that Robot 4 was the best robot in this experiment at the end of the second generation, and was selected to breed with all other robots. It was preserved to the third generation with the same sensor configuration, but could not repeat the same good performance. It is a good example of how much noise and interactions with other robots and different obstacles can influence the performance of the same robot, when it is tested again for the same period [Set97]. Robot 6, in the same figure, from generation

5 to 6, and 6 to 7, was selected as the best robot. This robot had a better sensor configuration, and was able to repeat the same good performance twice.

During the experiments, the most important factor that interfered with the performance of the system was the stochastic noise arising from the interactions of real physical systems. The encountered noise in the experiments was:

1- The infrared signals reflected by the walls in every direction (e.g., two point reflection from one emitter in two walls and back to another sensor receiver);
2- The dust on the floor can make the wheels slide differently, sometimes turning the robot differently than expected – this generates problems in performing programmed high-level actions, since a slide or two during the manoeuvre may take the robot away from the expected final position;
3- Although exhaustively refined, the sensors are still affected by environment conditions (such as intensity of illumination, colour of the walls, texture of the obstacles and floor, etc);
4- The inertia of motors and body, which vary with the robot speed causing difficulties to perform the expected turning routines.

In the next experiment, shown in Figure 6.18, the mutation rate was fixed at 0%, so that it was possible to analyse how much noise and interactions with other robots and obstacles can influence the performance of the robots [Set97]. All conditions from Experiment 2.1 were preserved with the exception of the new mutation rate and that instead of initialising the population with all sensors disabled, the robots now were randomly initialised. This was necessary because since mutation was set to 0%, the population relied only in crossover to produce new individuals. To do this, random bits were written to the 16 bits related to selecting the sensor configuration. As the population worked with a limited variety of genotype, the smaller phenotype range resulted in fast convergence.

Figure 6.19 shows a summary of the performance of the five mutation rates of 0%, 5%, 10%, 20%, and 40%, overlapping two curves for each mutation, one displaying the average fitness of all robots, and the other displaying the fitness of the best robot in each generation. This last curve was obtained by plotting the fitness of the winner robot, the one selected to reproduce with all the others, for every generation. Five colours were

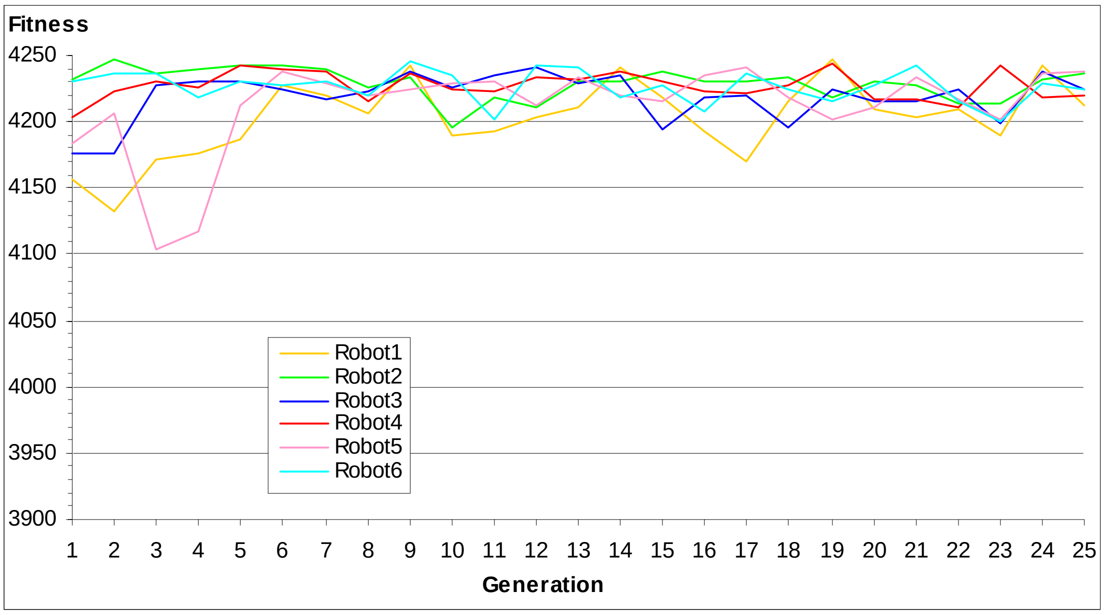


Figure 6.18 – ***Experiment 2.2***: Evolution of the sensor configuration in an environment of medium complexity, mutation rate of 0%, sexual reproduction, and generation time of 30s. The controller and speed levels are fixed during the experiment. Here, *Robotn* is the fitness of Robot *n* in the generation.

used, one for each mutation rate. Although the same colour was used to display the curves of the average fitness and the best robot, there is no possibility of mistaking them, since the best robot curve (*m*%*Best*, in the chart) is well above the average curve (*m*%*Average*, in the chart) for all five experiments. By the indicated mutation rate (*m*%), it is possible to identify to which chart the curves belong. Higher mutation rates introduce variation in the population and force the individual fitness to search the fitness landscape for other local optima. With these rates, the population was expected to present very different fitness values in each generation, even late in the evolutionary experiment.

- **Discussion**

Figure 6.18 shows that the population fitness still oscillates even without mutation. This represents all the noise in the system together with the interactions among the robots and among the robots and different obstacles for every generation. It generates a high degree of uncertainty in the process that would be difficult to simulate, highlighting once more the importance of employing a real physical system in evolutionary computation.

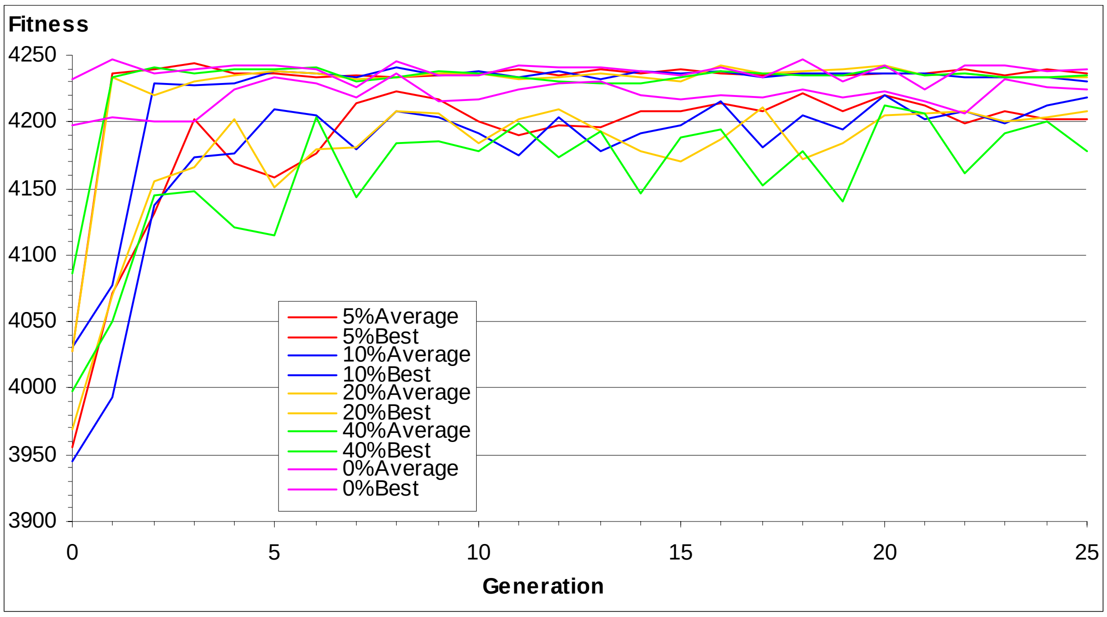


Figure 6.19 – *Experiment 2.3*: Comparison of five mutation rates (0%, 5%, 10%, 20%, and 40%) for the evolution of the sensor configuration in an environment of medium complexity, with sexual reproduction, and generation time of 30s. Here, *Average* is the average fitness of all robots in the generation and *Best* is the fitness of the best robot in the generation for the respective mutation rate.

For the experiment with 0% mutation, the random values that initialised the population were: *0,0,S6,0,0S3,S2,S1*; *S8,0,S6,S5,S4,S3,S2,0*; *S8,S7,0,0,S4,S3,0,S1*; *S8,S7,0,S5,S4,S3,0,0*; *S8,S7,S6,0,S4,0,S2,S1*; and *S8,S7,S6,S5,S4,0,S2,0*. Sensor *S5* is not connected to the neural network, so it did not matter if it was enabled or not. It is important to observe that initialised in this way, the population had the potential to generate any combination of the sensors. The system converged to *S8,0,S6,0,S4,S3,S2,0* in generation 16, so that all members of the population held this configuration to the end of the process. Therefore, the different performances of the same configuration from generation 17 on showed the influence of uncertainty on the robots. This experiment showed high fitness values right from the beginning of the process (generations 1 and 2). This happened because the randomly initialised configurations gave better performances than the robots with all sensors disabled from the experiments with non-zero mutations.

As predicted by Experiment 1, there was no optimal solution, since the best robot place was occupied by a series of different candidates with different sensor configurations, almost in every generation until the end of the experiment. Robots 6 and 4 were the best ones in the chart of Figure 6.17, repeating consistent performances all the way until the last generation. The experiment with 10% mutation, in Figure 6.19, showed a slightly more dispersed population according to the fitness value, but the system behaviour was stable again, producing a group of well-adapted robots right after the fifth generation. Again, the best robot place was occupied by different sensor configurations during the process.

The experiments with 20% and 40% mutation, show a very disperse population, but the fitness of the best robot is nevertheless as high as in the previous charts. This demonstrated that preserving the best robot, allowing it to survive to the next generation, was a good idea, which could keep the fitness of at least the best robot high during the whole process. A high mutation tends to degrade substantially the average performance, so the best robots that are selected to survive and do not suffer mutation have more chances of dominating the population, being selected as the best robot most of the time.

From the comparison of the average results shown in Figure 6.19, it is possible to observe that the lower mutation rates of 0% and 5% had the best averages (displayed in pink and red in the chart), the ones nearest to the best robot curve. As mutation increased, the average fitness reduced and the corresponding curves became more distant from the best robot fitness. Therefore, the distance between the average fitness curve and the best robot fitness curve can tell how dispersed the population was, or give an idea of the mutation rate used.

It can be observed in Figure 6.19 how the 0% mutation curves started with high fitness in the first generations, while the others climbed from much lower values. The rise of the average curves showed how the population got progressively better as the generations passed and evolution gradually tuned their sensor configurations.

6.3.1 Discussion of the Experimental Results

It can be concluded by the performed experiments that the implemented evolutionary system can produce a better population out of evaluation, selection, and recombination. It was able to evolve the sensor configuration of the robots in real time, in a physical environment with real robots. The aim of this experiment was therefore satisfied.

From the total search space of 2^8 sensor combinations, the evolutionary system was not able to find an optimal solution, but many efficient solutions containing at least three sensors, one in the front and two lateral ones in each side of the robot, were produced. The evolutionary system could not choose one amongst these solutions, which were alternately selected as the winner as the system tried to converge in the end of the process. This fact was predicted by Experiment 1, where it was shown that it was very difficult to differentiate the good solutions and establish a winner even in relatively long generation times of ten minutes or more.

This experiment established a relationship between mutation rate and the distance between the average fitness and the best fitness curves. It demonstrated that a higher mutation rate increases this distance, although some of this distance is due to noise and the interactions among the robots and obstacles, as it can be seen in Figure 6.18. It was shown that this uncertainty might never allow these two curves to meet in a real physical environment such as the one used for development.

6.4 Experiment 3: Evolving an Unstructured Controller

In this experiment, the embedded evolutionary system attempts to evolve a completely unstructured control circuit (i.e., a controller that has an unconstrained logic circuit, which can produce any binary function of its inputs). The eight sensors were permanently enabled and the speed of the motors was fixed at maximum speed. Therefore, the navigation control circuit was the only one under evolutionary control. This experiment did not use a structured, modular neural network (which can produce limited functionality, since only the contents of the neurons can be modified) to implement the navigation control circuit as shown in the previous experiments. Instead, it attempted to evolve an unstructured, undefined architecture, which was called the *black box* [Sim94] [Bot96] [Sim96].

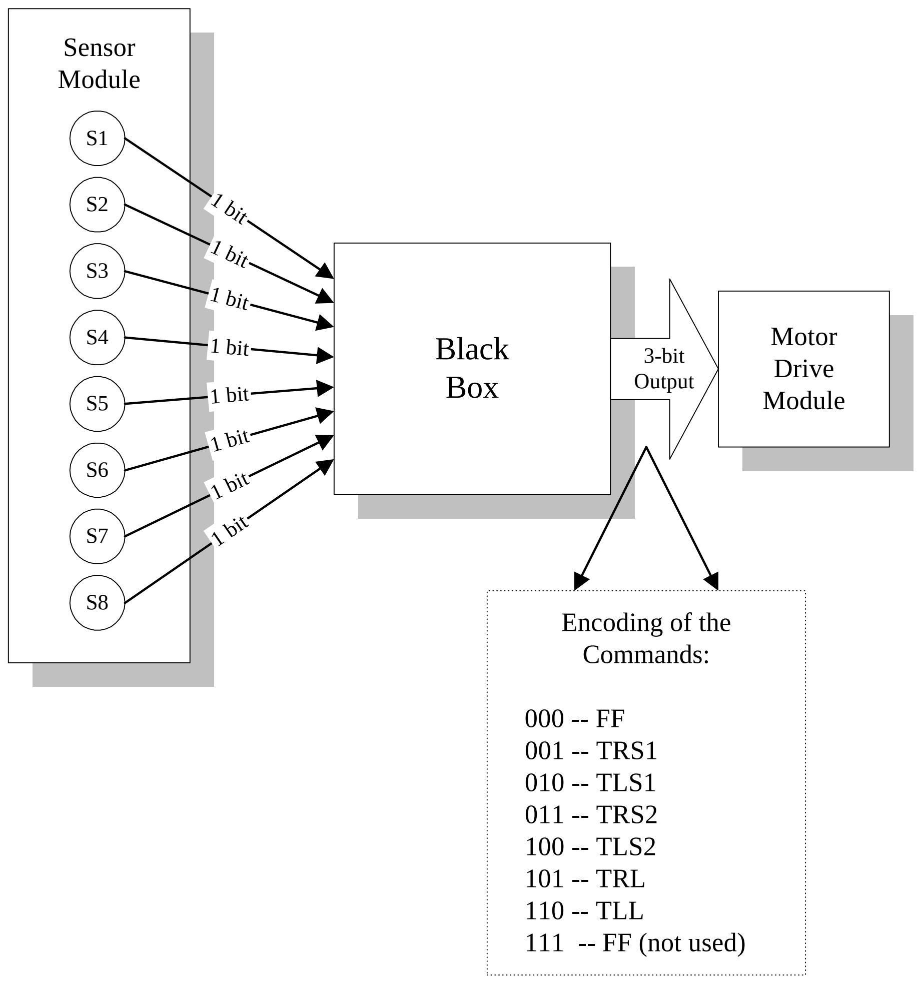


Figure 6.20 – The Black Box controller connected to eight sensors with 1-bit resolution. It produces a 3-bit signal to command the motor drive module according to the encoded commands. "111" is not used and is interpreted as *FF* by the motor drive module.

Evolving unstructured architectures for robot control has been attempted before by Thompson in [Tho94c] [Tho96a] [Tho96c], where he evolved a dynamic state machine to drive a small mobile robot and in [Tho96b], where he tried to evolve an FPGA connected directly to the motors to produce a pulse modulated signal. Both attempts employed simulation, and one real robot to evaluate the solutions. To date, no work has been reported where the evolution of an unstructured controller has been attempted with a real robot population in real time. Therefore, this experiment provides the first results obtained from the evolution of an unstructured control architecture by an embedded evolutionary system.

In this experiment, all sensors have only one bit of precision and they were set to work at *medium* range. Differently from the previous experiments, sensor *S5* in the back of the robots has been connected to the black box controller, and this time it can be used as any other sensor by the evolvable controller.

The controller used seven commands to control the motor drive module: *Front Fast* (*FF*); *Turn Left Short1* (*TLS1*); *Turn Right Short1* (*TRS1*); *Turn Right Short2* (*TRS2*); *Turn Left Short2* (*TLS2*); *Turns Right Long* (*TRL*); and *Turn Left Long* (*TLL*). These first four commands were explained in Experiment 1. In *TRL* and *TLL*, the robot turns right/left by breaking one wheel and turning around it with the other one in maximum speed in a wide arch of one wheel span of diameter. Figure 6.20 shows how the black box is connected to the sensors and the motor drive module.

The concept of the black box controller started when Simões *et al.* published a strategy of implementing a neural network in VHDL (VHSIC Hardware Description Language) (VHSIC – Very High Speed Integrated Circuit) by converting the neuron behaviours into lookup tables [Sim94] [Sim96]. Then, the mnemonics of each neuron were converted into VHDL language. To achieve this, after the neural network was trained, each one of its 8-input neurons was stimulated with all possible combinations of the input vector, from “00000000” to “11111111”. Then, each output for all input possibilities was written to a lookup table, representing the neuron combinational circuit. Next, these tables were used to convert the neuron behaviours into the VHDL symbols.

The same strategy used to implement the circuit of the neurons in [Sim96] as lookup tables is applied here to implement the navigation control circuit. The controller was represented by a black box that can generate any binary logic function of its eight binary inputs. Which logic circuit is actually used inside the black box to implement this logic function does not matter in this case, since only its behaviour is relevant to the problem. To adapt this strategy to be used as the navigation control circuit of the robot, it should be able to produce a 3-bit output.

Figure 6.21 shows how three lookup tables can be used to implement the black box circuit. Using three 256-bit memories to store the information of the three lookup tables is just one way to implement the black box circuit. The 8-bit signal from the sensors forms the address to the memories that output the corresponding content (1 bit). The outputs of the three memories form the 3-bit command that controls the motors.

Another way to implement the black box circuit is to use a 256-byte memory, which has enough space to store eight lookup tables of eight inputs. Figure 6.22 shows how this solution works. The 8-bit signal from the sensors is directly connected to

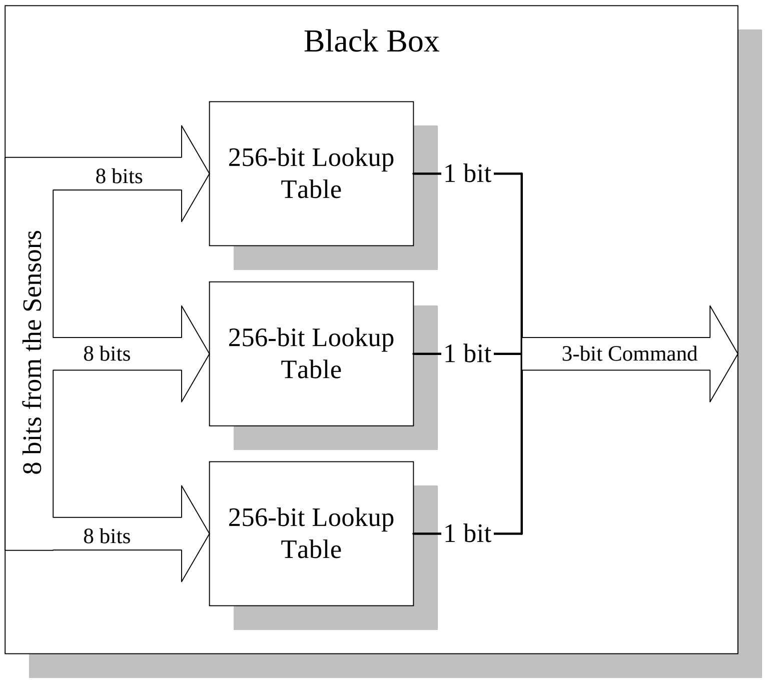


Figure 6.21 – The black box containing three 256-bit lookup tables that form the 3-bit command to the motor drive module.

the address lines of the memory. The output of the black box controller is a byte that has eight bits that can store the contents of eight lookup tables. In the figure, b_0, b_1, and b_2 are used to form the 3-bit command that controls the motors. The other bits are ignored, but they can be used in the future to give other parameters to the robots, such as speed levels for both motors. The five spare bits (b_3 to b_7) can be used to define the 32 levels of the robot speed ($2^5 = 32$).

Considering that the robot computing system already has a 64Kbyte memory, the natural way to implement the black box controller is by using the robot RAM memory to store the contents of the lookup tables. Since the robot microprocessor can access only one byte at a time of the memory, only the first three bits of this byte are used to form the command to the motors; the remaining bits are ignored. This strategy resulted in a powerful and fast controller, since only a single step or command line of the C++ language is necessary to implement the whole controller: *Command = Memory[Sensors]*. Considering that *Sensors* is a 1-byte long variable containing the sensor readings (8 bits) converted to an integer that address the memory array. *Memory[256]* is an array of 256 bytes that stores the contents of the lookup tables, and *Command* is the variable that holds the memory content for that address. In the robots, the black box was implemented using

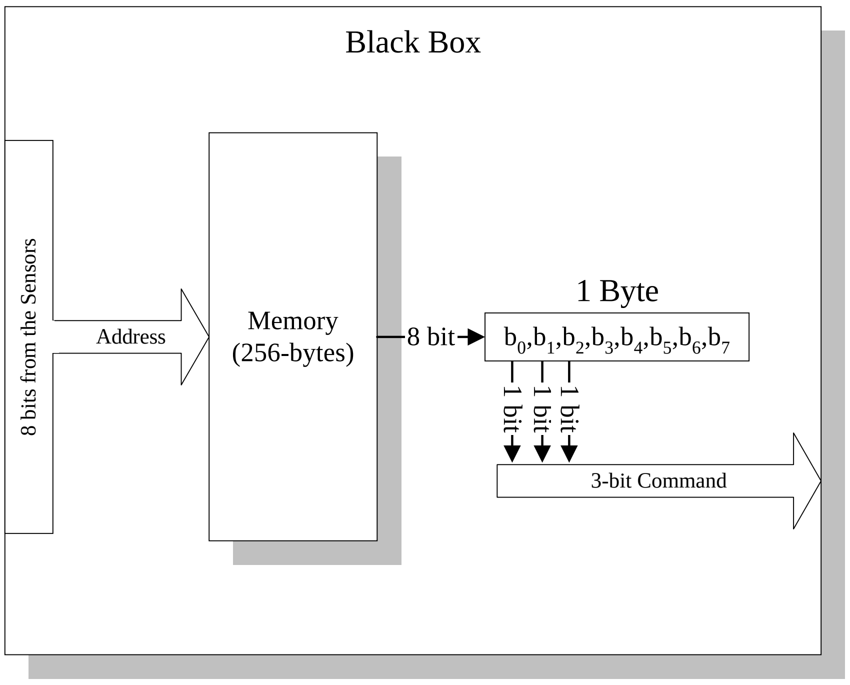


Figure 6.22 – The Black Box controller implemented using a 256-byte memory. Three bits from the output form the 3-bit command to the motor drive module.

the microprocessor (68HC11) assembler code. The corresponding part of this algorithm is shown below:

```
; Addressing Black Box in Memory according to the Sensor Readings
        ldab    sensors         ; b receives content of variable sensors
        ldx     #$8000          ; x receives Base Address
        abx                     ; Add b to x
        ldaa    0,X             ; Retrieve the Data from memory
        anda    #%00000111      ; Filter the first 3 bits containing the Command
        staa    command         ; Store the result to variable command
```

In this program, the variable *sensors* is a single byte that holds the sensor readings. Each bit of the byte corresponds to one sensor and is set to “0” if the sensor does not detect any obstacle, or “1”, if an obstacle is being detected. The base address *$8000* in hexadecimal is the address in memory of the first byte of the 256 locations of the black box memory. The variable *sensors* is directly added to it to point to any one of the 256 bytes stored in memory. The variable *command* is one byte that stores the command that corresponds to the sensor readings. These simple six lines of assembler code illustrate how fast the part of the robot software corresponding to the navigation control circuit works. The complete assembler program is called EVOLUTION.ASM and is listed in Appendix A. When compared to the 87 lines of assembler code that implement the hand-designed controller described in Experiment 2, the simpler black box strategy is at least 100 times faster. The assembler program that contains the hand-designed controller is called HANDCONTROL.ASM and is also listed in Appendix A.

In this experiment, instead of using the black box memory to implement a previously trained neural network or any other circuit as the previous work that inspired it, evolution was allowed to manipulate the memory contents directly, until a navigation control circuit emerged. Therefore, every byte of the memory was ordered in the chromosome to form a 2048-bit string. Crossover and mutation can affect each one of these bits as a normal binary chromosome. After these operations are completed, the 2048 bits in the chromosome are grouped again into 256 bytes and stored in the black box memory. Only the first three bits in the byte (b_0, b_1, and b_2) are relevant for this experiment, and the other ones (b_3 to b_7) are ignored by the robot. Therefore, from the 2048 bits corresponding to the eight lookup tables, only 768, which correspond to the first three tables (b_0, b_1, and b_2) are relevant to evolution. Hence, this is the size of the genotype of the robots, and considering that any one of them can produce a different phenotype (i.e., there is no neutrality in this case), the current search space is considerably large: $2^{768} = 1.55 \times 10^{231}$.

- **Aim of the Experiment**

This experiment aimed to test if the embedded evolutionary controller developed in Chapter 4 was able to evolve the navigation control circuit until the collision-free navigation behaviour emerges, while having the robot morphology (the sensor configuration and motor speeds) fixed. To achieve this, the sensor configuration was fixed with all sensors enabled and the velocity of both motors was fixed at maximum speed. The navigation control circuit is composed of only 256 bytes in the robot RAM memory and evolution is allowed to manipulate any bit of these bytes. The robots were allowed to reproduce and suffer mutation, but only with the 2048 bits in the chromosome that correspond to the navigation control circuit. The initial morphology remained fixed during the complete experiment.

- **Experimental Setting**

A hand-designed controller was developed as a reference to compare the results of the evolutionary system. It is a variation of the previously described ones, which includes the new commands for the motor drive module. In this experiment, the hand-designed controller operated according to the following algorithm:

```
     Left = Right = 0;
     If (Sensor4=1) then Left = Left + 1;
  -  If (Sensor3=1) then Left = Left + 1;
     If (Sensor2=1) then Left = Left + 1;
  P  If (Sensor6=1) then Right = Right + 1;
  r  If (Sensor7=1) then Right = Right + 1;
  i  If (Sensor8=1) then Right = Right + 1;
  o  If (Sensor1=1) then Command = TRS2;
  r  If (Left > Right) then Command = TRS1;
  i  If (Left = Right) then Command = FF;
  t  If (Left < Right) then Command = TLS1;
  y  If (Sensor1=1 and Sensor4=1) then Command = TRS2;
     If (Sensor1=1 and Sensor6=1) then Command = TLS2;
  +  If (Sensor4=1 and All other Sensors=0) then Command = TRL;
     If (Sensor6=1 and All other Sensors=0) then Command = TLL;
```

The black box controller can be trained to behave as the hand-designed controller by stimulating it with all possible combinations of the input vector (from “00000000” to “11111111”). Then, for each possible input, the output of the hand-designed controller is written in the corresponding addressed position in the black box memory. To do so, the 3-bit output of the hand-designed controller was converted to a byte by adding five zeros: *byte* equals to “00000” plus the 3-bit output from the hand-designed controller. Therefore, the black box is able to behave exactly as any controller that has eight binary inputs and produces up to eight binary outputs.

The selection, crossover and mutation strategies used here were explained in Experiment 2. This experiment was run in the same environment with medium complexity used by Experiment 2 (shown in Figure 6.16). In some tests in this experiment, the configuration of the controller of the robots was randomly initialised, but others started with a population trained according to the hand-designed controller described above. As explained before, simulations were done for a range of values and examples were selected out for a matter of presentation. Therefore, the mutation rate of 3% was chosen for this experiment because it produced the best results with the proposed settings (Appendix C contains the average results of many experiments that could not be included in the body of the thesis). Table 6.4 presents a summary of the settings for this experiment.

Table 6.4 – Summary of the Experimental Settings

Parameter	Definition
Fitness Function:	To be defined in the experiments
Initial Fitness Value:	4096 points
Maximum Fitness Value:	4696 points
Generation Time:	60 seconds
Mutation Rate:	3%
Speed Levels:	Fixed at maximum speed (Vmax=32)
Sensors Enable:	Fixed with all enabled
Navigation Controller:	Evolving Black box with 256 bytes
Initial Sensor Configuration:	“00-00-00-00-00-00-00-00”
Initial Speed Configuration:	“1111111111”
Initial Controller Configuration:	Random and as the hand-designed controller
Software:	To be defined in the experiments

6.4.1 Experiment 3.1: Evolving with a Simple Fitness Function

This experiment used a very simple fitness function in an attempt to prevent biasing evolution to any preconceived idea of an ideal controller. The selected fitness function for this test was:

1- Start with 4096 points;

2- Reward: increase fitness by 10 points every 1 second without collision;

3- Punishment: decrease fitness by 30 points for every time command is not FF for more than 15 seconds;

4- Punishment: decrease fitness by 10 points for every collision if command = FF, TLL, or TRL.

Rule 2 changed from Experiment 2 and constantly rewards the robot with five points for every second of the generation time. This attempted to solve a problem

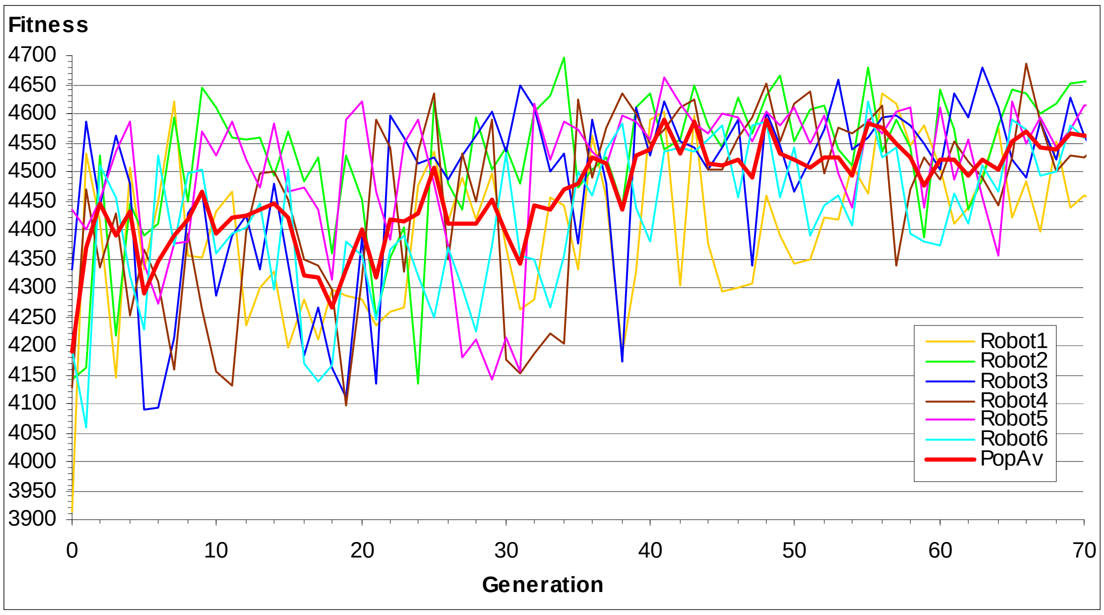


Figure 6.23 – *Experiment 3.1a*: Evolution of the black box controller using a simple fitness function in an environment of medium complexity, mutation rate of 3%, sexual reproduction, and generation time of 60s. The population was randomly initialised. Here, *Robotn* is the fitness of Robot *n* and *PopAv* is the average fitness of all robots in the generation.

where a robot that turned frequently was making fewer points, since the fitness function rewarded only the *FF* command. Rule 3 was introduced to prevent evolution from producing a solution that keeps turning around itself and never collided. This rule punishes the robots that keep turning for more than 15 seconds, encouraging them to move forward. Rule 4 was modified to prevent punishing the robots that are turning around themselves and are crashed by another robot. If a robot is executing any command but *FF*, *TLL*, or *TRL* its centre is not moving, so the collision cannot be its fault. The results in this experiment were acquired by the program EXP08.CPP and were converted by the program GRAF08.CPP, both listed in Appendix A.

- **Results**

Figure 6.23 shows results from an evolutionary experiment run in a medium complexity environment with six robots and a mutation rate of 3%. This experiment used a simple fitness function that did not bias evolution towards a preconceived solution. The chart displays the fitness of the six robots and the average fitness of the population (*PopAv*).

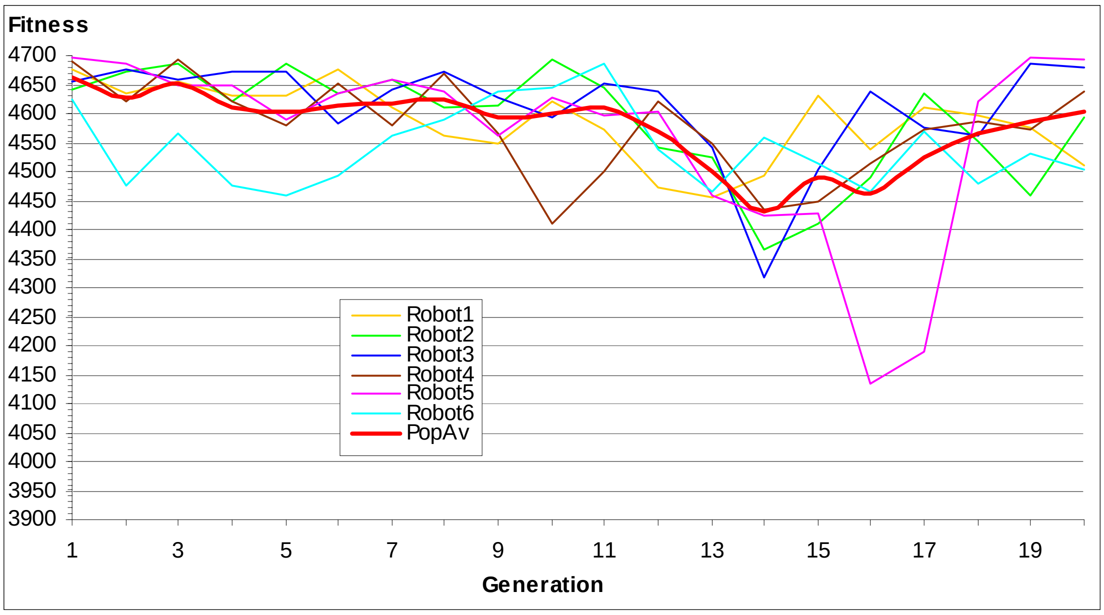


Figure 6.24 – ***Experiment 3.1b***: Evolution of the *black box controller* using a simple fitness function in an environment of medium complexity, mutation rate of 3%, sexual reproduction, and generation time of 60s. The population was initialised as the hand-designed controller. Here, *Robotn* is the fitness of Robot *n* and *PopAv* is the average fitness of all robots in the generation.

The experiment in Figure 6.24 was performed to investigate if the robot population, once initialised with a well-trained controller, could hold this configuration or would degenerate. The robots were initialised with a previously trained black box, which was taught to behave exactly as the above described hand-designed controller. The population was then allowed to mate and mutate with the same parameters of the previous evolutionary experiment to determine if it would ever get to such a good solution if the experiment was allowed to continue for more generations. The results presented in this figure can be compared to the ones obtained from the evolution of a randomly initialised population in Figure 6.23.

- **Discussion**

Figure 6.23 showed that the evolutionary system succeeded in evolving the population towards the expected behaviour, producing solutions that practically did not collide after 35 generations. It was observed in this experiment that one could not rely only on the fitness of a robot to know how well-adapted it is. A simple solution can be lucky enough to start its lifetime in a safe part of the environment, away from obstacles and other robots, which would produce a very high performance. This was observed in some robots during the evolutionary experiment (e.g., Robot 2 that scored 4644 in generation 10, or Robot 1 that scored 4621 in generation 8). Therefore, the human judgement was the best way to evaluate the abilities of the robots and the performance of the evolutionary system.

In Experiment 3.1a in Figure 6.23, all sensors were enabled from the beginning of the evolutionary experiment. The controller had information from all the sensors and needed to learn what to do with it. Based on observation, some robots learned how to use the frontal sensor, *S1*, which gave them an advantage early on in the process. This was the case for Robot 2 that could avoid obstacles detected by *S1* from generation 8. Robot 2 quickly became the best robot and throughout the crossover operation transferred to Robot 5 the ability of using *S1*. Robots 2 and 5 dominated the population until generation 23, where they were overtaken by Robot 3 and Robot 4. Robot 3 learned how to use *S1*, *S4*, and *S7* from generation 29, and was able to avoid most of the obstacles very well thereafter. From generation 47 on, all robots acquired the necessary skills to employ at least three sensors and the average performance was much better. From generation 50, Robot 2 was well-adapted to the environment, and could use *S1*, *S2*, *S4*, *S6*, and *S7* to avoid collisions with most of the obstacles and other robots.

The major observed problem was the instability of the system, where there was no guarantee that a good solution, such as Robot 2 in generation 50, would be selected as the best robot until it is outperformed by better robots. The population performance dropped when a robot with a poorly-adapted controller (Robot 1) was lucky enough to be selected as the best robot and spread its bad genes through the population. Robot 1 had a poorly-adapted controller that made it move forward each time *S6* detected an obstacle, independently of having something in its way. The average performance dropped, but after a few generations, the fitness of the population recovered.

The average of the population fitness is a good parameter to determine if the system is converging to the desired behaviour. The population performance oscillated, but kept improving through the evolutionary experiment. With such a large search space (1.55×10^{231}), a perfect robot that can deal with all sensors should take a very long time to obtain with this evolutionary approach. Nevertheless, the system succeeded in producing an even population of robots that can successfully avoid the obstacles in most of the situations faced and produces a fitness near to the maximum performance (4696).

Figure 6.24 shows how the system behaves after the population has been initialised with a black box trained according to the hand-designed controller. The aim of this test was to show that if such a good solution happen to be produced by evolution, it would be preserved in the process. If the evolutionary experiment cannot keep such a solution, it is unlikely that it will ever be produced by evolution under these circumstances. Unfortunately, the figure showed that the evolutionary system, as set in this experiment, could not keep a good solution for more than ten generations and the performance of the system degenerated. Although it was possible to recover and continue to improve performance after the loss of the hand-designed controller genes, this test indicated that it is unlikely that such a good solution will be produced by the process as it was configured for this experiment. Maybe a more biasing fitness function can produce a better result.

6.4.2 Experiment 3.2: Evolving a Biasing Fitness Function

This experiment intended to use a more biasing fitness function than the one presented in Experiment 3.1. It investigated whether a biased evolution could produce a better solution (such as the hand-designed controller) for the collision-free navigation task.

A new fitness function was designed taking in consideration the information of the infrared sensors to reward or punish the robots. This function rewards the robot when the command is *FF* and there are no obstacles around the robot. It also rewards it if the robot detects an obstacle on one side and turns to the other side. The robot is punished if it turns against the obstacle. To make it easier to use the sensor information, the variables *Left* and *Right* were used again as in the hand-designed controller. As it was described before, *Left* is increased by one each time one of the sensors *S2*, *S3*, and *S4* detect an obstacle and *Right* is increased by one each time *S8*, *S7*, and *S6* detect an obstacle. The selected fitness function, for this test was:

1- Start with 4096 points;

2- Reward: increase fitness by 10 points every 1 second if All Sensors=0 and command=FF;

3- Reward: increase fitness by 10 points if Left > Right and command=TRS1, TRS2, TRL;

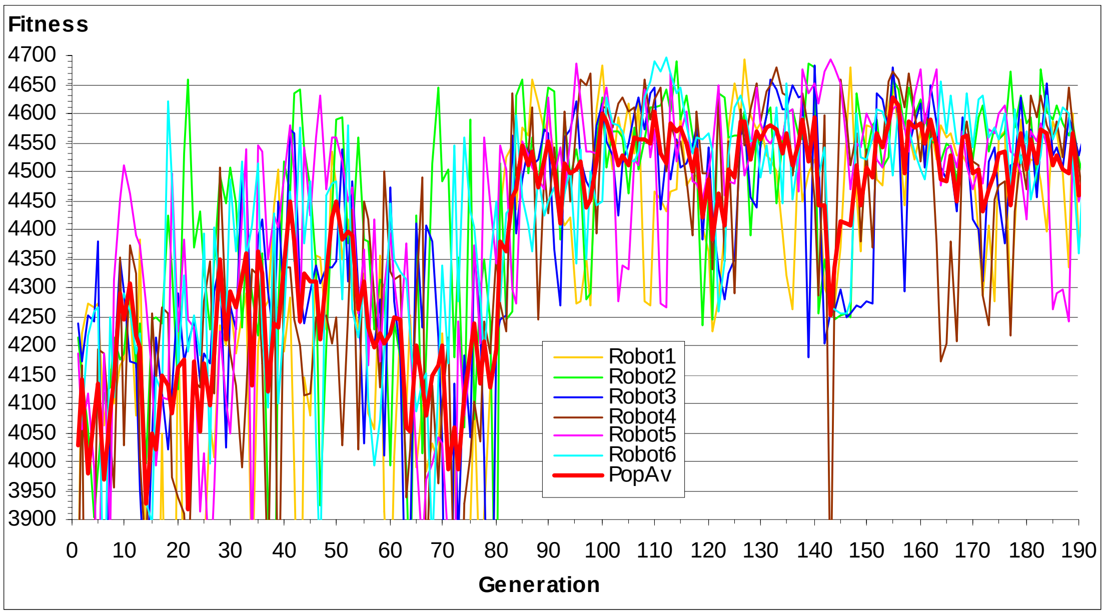


Figure 6.25 – *Experiment 3.2a*: Evolution of the *black box* controller using a biasing fitness function in an environment of medium complexity, mutation rate of 3%, sexual reproduction, and generation time of 60s. The population was randomly initialised. Here, *Robotn* is the fitness of Robot *n* and *PopAv* is the average fitness of all robots in the generation.

4- Reward: increase fitness by 10 points if Left < Right and command=TLS1, TLS2, TLL;

5- Punishment: decrease fitness by 10 points if Left < Right and command=TRS1, TRS2, TRL;

6- Punishment: decrease fitness by 10 points if Left > Right and command=TLS1, TLS2, TLL;

7- Punishment: decrease fitness by 20 points for each collision if command = FF, TLL, or TRL.

This fitness function is biasing because it takes consideration of not only what the evolving controller is doing, but also how it does it. It practically tells the controller how it should control the robot, limiting the possible solutions arising from the evolutionary process. The results in this experiment were acquired by the program EXP10.CPP and converted by the program GRAF10.CPP, both listed in Appendix A.

- **Results**

The system was allowed to evolve for a long period (more than three hours), to show if solutions as good as the hand-designed controller could be produced. Figure 6.25 shows a chart representing the evolutionary experiments in the medium complexity

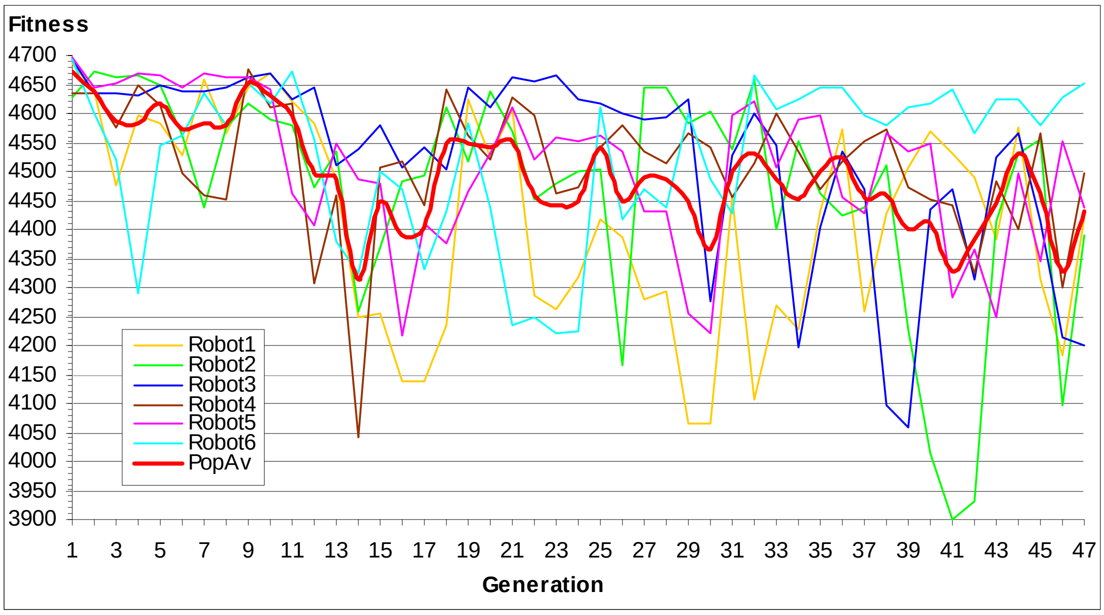


Figure 6.26 – *Experiment 3.2b*: Evolution of the <u>black box controller</u> using a <u>biasing fitness function</u> in an environment of <u>medium complexity</u>, <u>mutation rate of 3%</u>, <u>sexual reproduction</u>, and <u>generation time of 60s</u>. The population was initialised as the hand-designed controller. Here, *Robotn* is the fitness of Robot *n* and *PopAv* is the average fitness of all robots in the generation.

environment. It ran for 190 generations with a biasing fitness function and 3% mutation. The same test shown in Figure 6.24 was repeated here. It is shown in Figure 6.26 and examined whether the robot population, once initialised with a well-trained controller, such as the hand-designed controller, could hold this configuration or would degenerate.

- **Discussion**

Once more, the system succeeded in producing efficient solutions to the collision-free navigation task. This time, however, the biasing fitness function could itself almost drive the robot. One may ask why the designer should bother in applying evolution if almost the complete controller has to be specified in the fitness function. It was still valid as an experiment to see how close to a more refined solution, such as the hand-designed controller, the evolutionary system could get by means of a more biasing fitness function. Nevertheless, rule 7 in the fitness function included a new dimension to biasing the solution towards a preconceived controller. This rule punishes the robots for each collision, so that it is still being evaluated. Therefore, if evolution reaches the targeted controller and it still does not guarantee collision-free behaviour, it will keep modifying it until a better solution is found. The targeted controller is actually just a guideline to help evolution to reach a better solution faster.

Despite the fact that good solutions were in fact produced early on in the evolutionary experiments, the system still lacked stability as in Experiment 3.1. Robot 2 was a good example of an efficient solution that learned how to use the front sensor as soon as generation 20, but could keep this configuration for only five generations. This was due to it finding itself in a crowded area of the environment where it repeatedly collided with three other robots. It can be seen in Figure 6.25 that until these instability problems are solved, the system will still oscillate, while good solutions continue to be replaced by worse, but luckier ones. A final optimal solution could not be found in 190 generations and more than three hours of evolution. The process resulted in a group of competing efficient configurations that used at least three sensors, one in the front and two lateral ones on both sides of the robot. From generation 100 on, these solutions were alternately selected as the best robot.

The evolutionary experiment shown in Figure 6.26 could not keep a good solution for long when it had all members of the population replaced by a copy of the hand-designed controller. After 11 generations, Robot 6 mutated in an unfit configuration but was lucky enough to be selected as the father of the next generation and the average performance of the population degraded. It was a good sign that the population could recover after ten generations, where Robots 2, 3, and 4 combined their genetic information to produce once more a good solution that dominated the population until generation 27. Then, an unfit Robot 2 managed to score high enough to be selected to mate with all the other robots twice, erasing some important genetic information from Robot 3, and the performance of the population degraded once more between generations 30 and 40.

6.4.3 Experiment 3.3: Evolving a Strongly Biasing Function

This experiment examined the effects of punishment by collision on the stability of the evolutionary system. Rule 7 was erased from the fitness function in an attempt to take chance out of the system. In fact, without punishing the robots each time

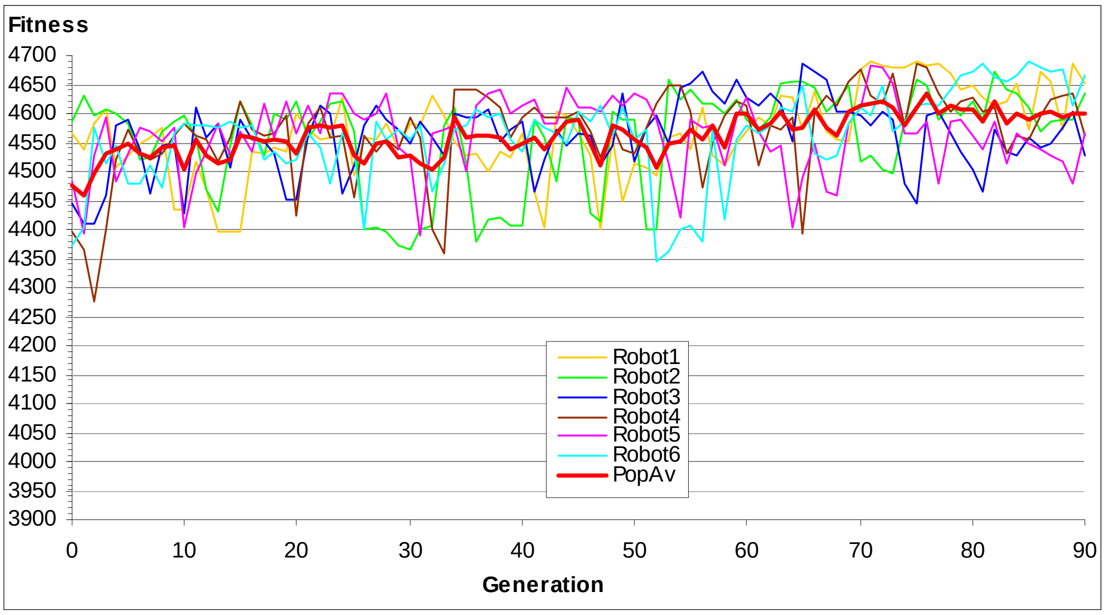


Figure 6.27 – *Experiment 3.3a*: Evolution of the black box controller using a strongly biasing fitness function in an environment of medium complexity, mutation rate of 3%, sexual reproduction, and generation time of 60s. The population was randomly initialised. Here, *Robotn* is the fitness of Robot *n* and *PopAv* is the average fitness of all robots in the generation.

they collide, a fully trained robot (e.g., one that behaves similarly to the black box) is expected to have always the same score, in spite of where it is placed in the environment. This is since its collisions, or more importantly the collisions it suffers from other robots, will not decrease its fitness. However, a chance factor still exists since an unfit robot (e.g., one that has only part of its controller well-developed) will be favoured if it faces obstacles from the side with which its controller can deal. This may seem a useless application of evolution, since its only benefit is training the black box controller to behave according to a preconceived solution. Nevertheless, these experiments are important because they help to evaluate the evolutionary system performance. The new fitness function was obtained by deleting rule 7 from the fitness function shown in Experiment 3.2. The software used for this experiment were EXP09.CPP and GRAF09.CPP. Both programs are listed in Appendix A.

- **Results**

Figure 6.27 represents the evolutionary experiments in the medium complexity environment, running for 89 generations with six robots, a strongly biasing fitness function, and a mutation rate of 3%. The experiment shown in Figure 6.28 examined whether the robot population, once initialised with a well-trained hand-designed controller, could hold this configuration or would degenerate.

- **Discussion**

The experiment shown in Figure 6.27 presented fitness curves that differ from the ones in Experiments 3.1 and 3.2. The fitness values of the robots varied much less than in the previous experiments, since the collisions were not taken into account. The population performance was more stable as well. However, the average fitness of the population still oscillates. Therefore, some other mechanisms must be contributing to this instability. This will be further investigated in Chapter 7.

Figure 6.28 showed that even without counting collisions, the robot population could not keep a well-adapted solution. It can be observed that all six robots had similar performances in generation 1. This indicates, as expected, that if a robot is well-adapted, it will present similar performances if tested in the environment for the same time. The figure also showed that the resulting average of the population fitness, around 4600 points, was close to the one obtained with 89 generations in Figure 6.27, demonstrating that the system succeeded in producing an efficient solution very close to the hand-designed controller.

6.4.4 Discussion of the Experimental Results

The results so far demonstrated that the developed evolutionary system works and can evolve the navigation control circuit of the robots. The search space for a solution in this experiment was considerably large: $2^{768} = 1.55 \times 10^{231}$. Even so, the evolutionary system was able to produce an acceptable solution that could drive the robots with very few collisions. It still could not produce, however, a solution as good as the hand-designed controller, even when biased.

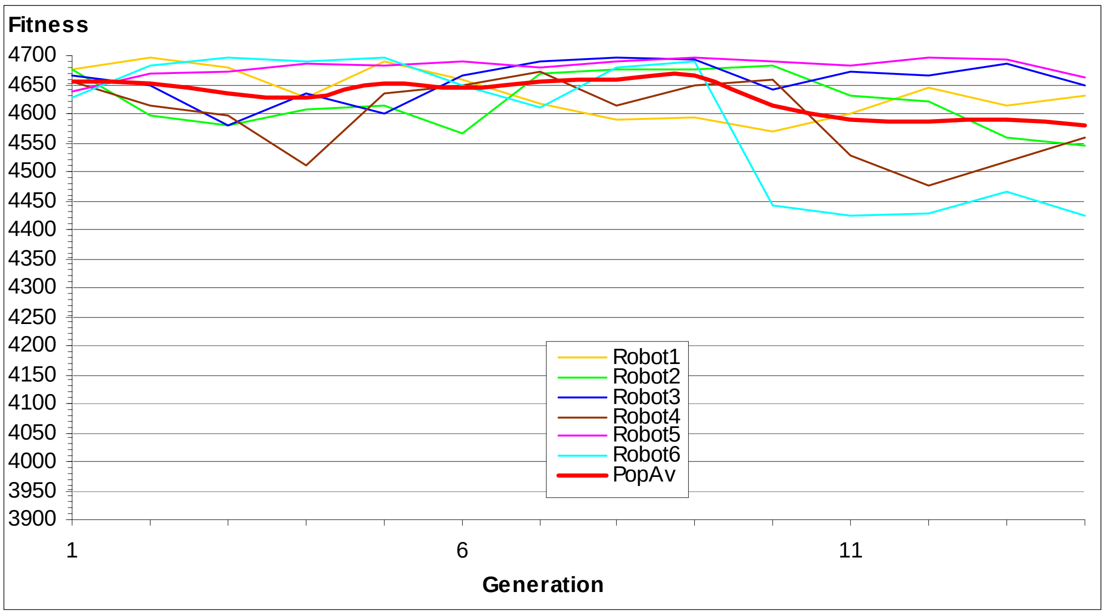


Figure 6.28 – *Experiment 3.3b*: Evolution of the *black box* controller using a strongly biasing fitness function in an environment of medium complexity, mutation rate of 3%, sexual reproduction, and generation time of 60s. The population was initialised as the hand-designed controller. Here, *Robotn* is the fitness of Robot *n* and *PopAv* is the average fitness of all robots in the generation.

By biasing evolution with a complex fitness function, some of the noise and interactions among the robots have less influence in the fitness evaluation, helping the system to converge faster [Set97]. However, as it could not achieve a solution as good as the hand-designed controller, it is clear that there was some other factor making the system oscillate and degrading performance. Even though it was shown that the population could recover after such degradations, it prevented evolution from producing a better solution.

The presented experiments showed that even a biased evolution could not prevent a lucky unfit robot from being selected as the best robot, spreading its bad genes throughout the rest of the population and causing the average performance to drop. There must be a way to prevent this from happening so often in the evolutionary experiment. The next experiment will try to explore other strategies for selecting the individuals that provide more stability to the system.

6.5 Experiment 4: Inheritance

This experiment developed a different selection strategy for evolutionary systems that was defined as *Inheritance Strategy*. From now on, the robots will be selected to breed based on not only their performance in the current generation, but also on their fitness in the previous generations. Therefore, the robots *inherit* part of the points scored by their predecessors. By considering the average fitness scored by the robot in the previous generations, this strategy attempts to reduce the effect of chance, that can produce bad performances if the robot is “unlucky”. This experiment tested this new approach, and evaluated if it could solve the instability problem pointed out by Experiment 3.

- **Aim of the Experiment**

The aim of this experiment is to develop and test a novel selection strategy involving inherited scores. It evaluated whether this new strategy could solve the problems with instability presented by the previous selection strategy, where a lucky unfit robot can be selected as the best robot instead of better ones that by chance started in a more crowded area of the environment.

- **Experimental Setting**

Apart from a different selection strategy, this experiment was set exactly as Experiment 3. It was run in the same environment used by the previous experiment, which is illustrated in Figure 6.16. It used the same biasing fitness function presented there with tests containing rule 7 and without it (this rule punishes the robot fitness each time it collides). The chosen selection strategy for this evolutionary experiment was:

  - *The score used to select the robot was the average of the robot fitness in the last three generations (i.e., inheriting the scores of its previous two generations). The robot with the best average survives, and breeds with all other robots.*

This approach favours the robots with the highest average, which are the ones that have performed well for the last three generations. If in the current generation one lucky unfit robot happens to achieve the highest score, it is still unlikely that it will have the best average score and will not be selected to mate. This was an attempt to improve the stability of the system and reduce the effect of noise and interactions among robots and obstacles. In some tests in this experiment, the configuration of the controller of the robots was randomly initialised, but others start with a population trained according to the hand-designed controller described in Experiment 3. Table 6.5 presents a summary of the settings for this experiment.

Table 6.5 – Summary of the Experimental Settings

Parameter	Definition
Fitness Function:	To be defined in the experiments
Initial Fitness Value:	4096 points
Maximum Fitness Value:	4696 points
Generation Time:	60 seconds
Mutation Rate:	3%
Speed Levels:	Fixed at maximum speed (Vmax=32)
Sensors Enable:	Fixed with all enabled
Navigation Controller:	Evolving Black box with 256 bytes
Initial Sensor Configuration:	“00-00-00-00-00-00-00-00”
Initial Speed Configuration:	“1111111111”
Initial Controller Configuration:	Random and as a hand-designed controller
Software:	EXP13.CPP and GRAF13.CPP (listed in Appendix A)

6.5.1 Experiment 4.1: Inheritance and a Biasing Fitness Function

This experiment applied the inheritance strategy to an evolutionary system using a biasing fitness function, the same one described in Experiment 3.2. The charts display the fitness of the six robots in the current generation as well as the average fitness value (*AvRn*) scored by each robot in the current and the previous two generations. For each robot:

$$AvRn = (FitnessRn_{G0} + FitnessRn_{G-1} + FitnessRn_{G-2})/3$$

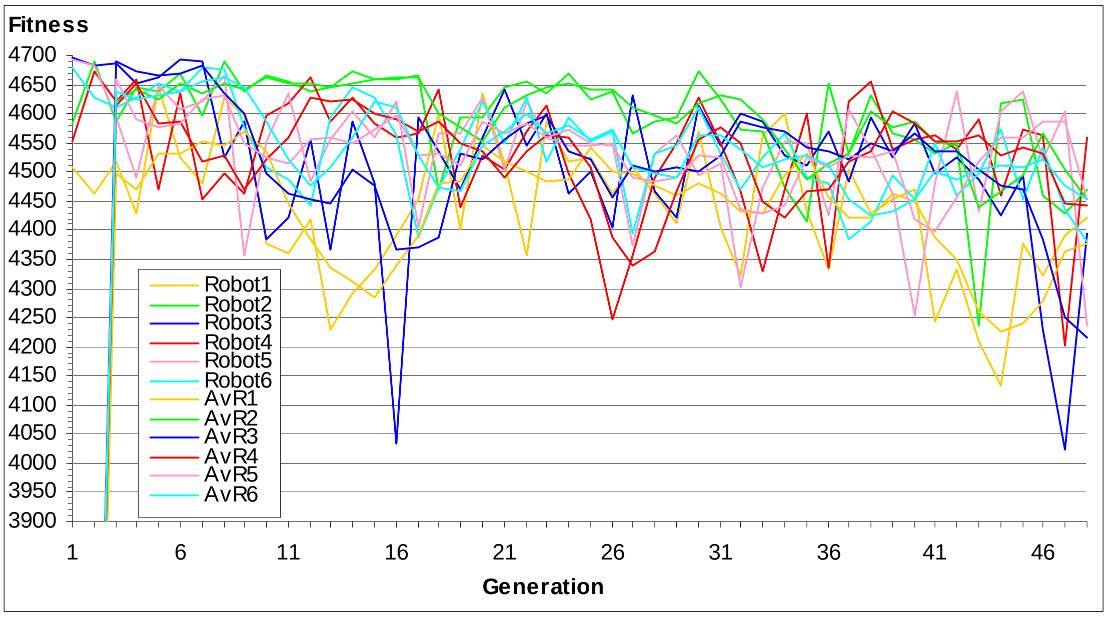


Figure 6.29 – *Experiment 4.1a*: First test of the evolution of the *black box* controller using inheritance selection and a biasing fitness function in an environment of medium complexity, mutation rate of 3%, sexual reproduction, and generation time of 60s. The population was initialised as the hand-designed controller. Here, *Robotn* is the fitness of Robot *n* in the generation and *AvRn* is the average fitness scored by robot *n* in the current and the previous two generations.

In the equation above, *n* is the number of the robot; $FitnessRn_{G0}$ is the fitness of the Robot *n* in the current generation; $FitnessRn_{G-1}$ is the fitness scored by Robot *n* in the previous generation; and $FitnessRn_{G-2}$ is the fitness scored by Robot *n* two generations before.

- **Results**

Figure 6.29 shows the first test of the new selection strategy where the population was initialised as the hand-designed controller. The experiment did not succeed. It could not keep the good performance of the initial population. By analysing the information in the figure, it was concluded that it did not work well because it did not protect a fit robot in the current generation that scored more than the best robot (the one with the highest average). The current selection strategy would mistake it for a lucky unfit robot and would not protect it from breeding with the best robot and mutating. This can destroy the precious good genes of this robot and evolution will lose this better performance. This fact can be observed in the figure, where, for example, Robot 4 was a potentially better solution that performed better than Robot 2 in generation 12, but was not protected and mated with Robot 2, which was potentially worse. After mating and mutating, Robot 4 could not perform as well and this possible better individual was discarded by the process. Therefore, this strategy needs to be modified to protect the robots that score more than the average of the best robot. A modified selection strategy is described below:

- *The score used to select the robot is the average of the robot fitness in the last three generations (i.e., inheriting the scores of its previous two generations). The robot with the best average survives, but only breeds with the robots with the fitness in the present generation lower than its own fitness.*

This approach protects new robots that are actually better than the one with the highest average, but need to be evaluated for more generations to be selected by their average. If one of these protected robots has a better configuration, it will probably repeat its good performance, and within one or two generations its average fitness may be the highest and the robot will be chosen to reproduce. If the protected robot is in fact a lucky unfit one, it may not repeat its good performance and, if so, will be allowed to breed in the next generation. Figure 6.30 shows another test with the updated selection strategy.

- **Discussion**

The new selection strategy presented in Figure 6.30 succeeded in improving system stability. The fitness of the population dispersed mostly because of mutation, but the performance of the best robot did not degrade as much as in the previous attempts. It now protects the potentially better configurations, as happened with Robot 2 in generation 53. It outperformed Robot 6 and was given the chance to repeat its better performance, and did just that in generation 54, when its average became finally higher than Robot 6 and it was selected as the best robot of the generation, breeding with all the other robots. In the next experiment, presented in Figure 6.31, this new selection strategy is applied to evolve a randomly initialised population.

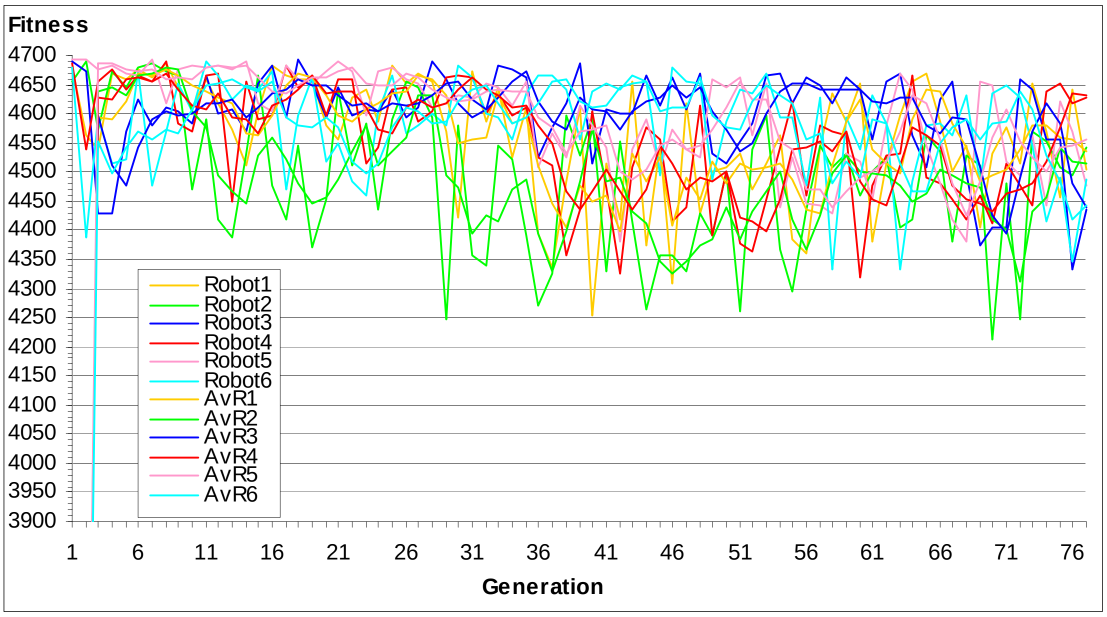


Figure 6.30 – *Experiment 4.1b*: Evolution of the <u>*black box* controller</u> using a corrected <u>inheritance selection</u> and a <u>biasing fitness function</u> in an environment of <u>medium complexity</u>, <u>mutation rate of 3%</u>, <u>sexual reproduction</u>, and <u>generation time of 60s</u>. The population was initialised as the hand-designed controller. Here, *Robotn* is the fitness of Robot *n* in the generation and *AvRn* is the average fitness scored by robot *n* in the current and the previous two generations.

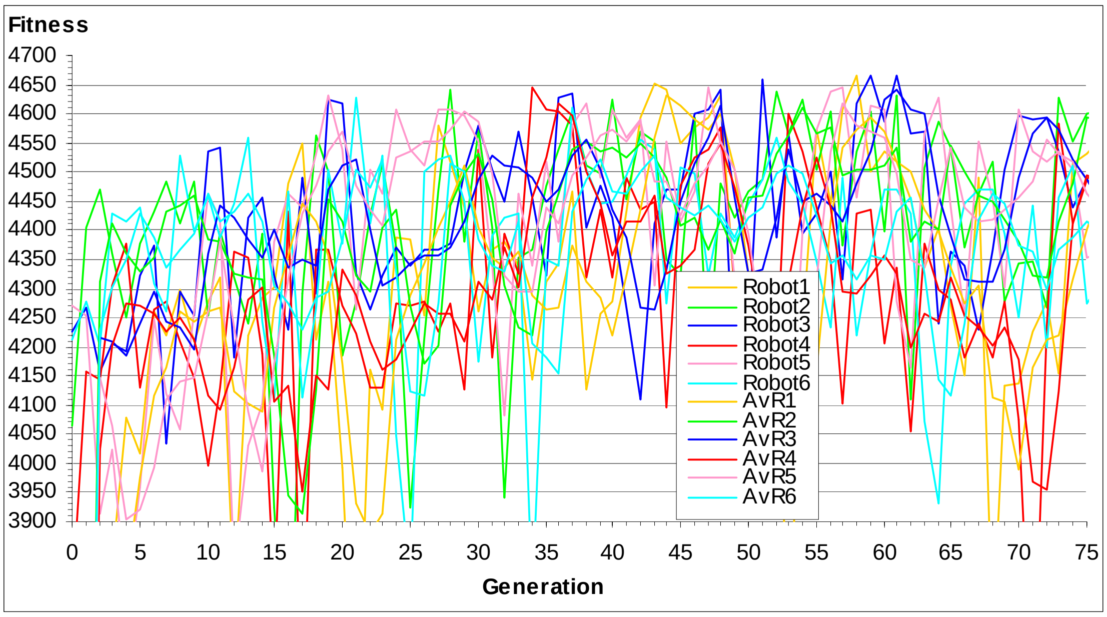


Figure 6.31 – *Experiment 4.1c*: Evolution of the <u>*black box* controller</u> using <u>inheritance selection</u> and a <u>biasing fitness function</u> in an environment of <u>medium complexity</u>, <u>mutation rate of 3%</u>, <u>sexual reproduction</u>, and <u>generation time of 60s</u>. The population was randomly initialised. Here, *Robotn* is the fitness of Robot *n* in the generation and *AvRn* is the average fitness scored by robot *n* in the current and the previous two generations.

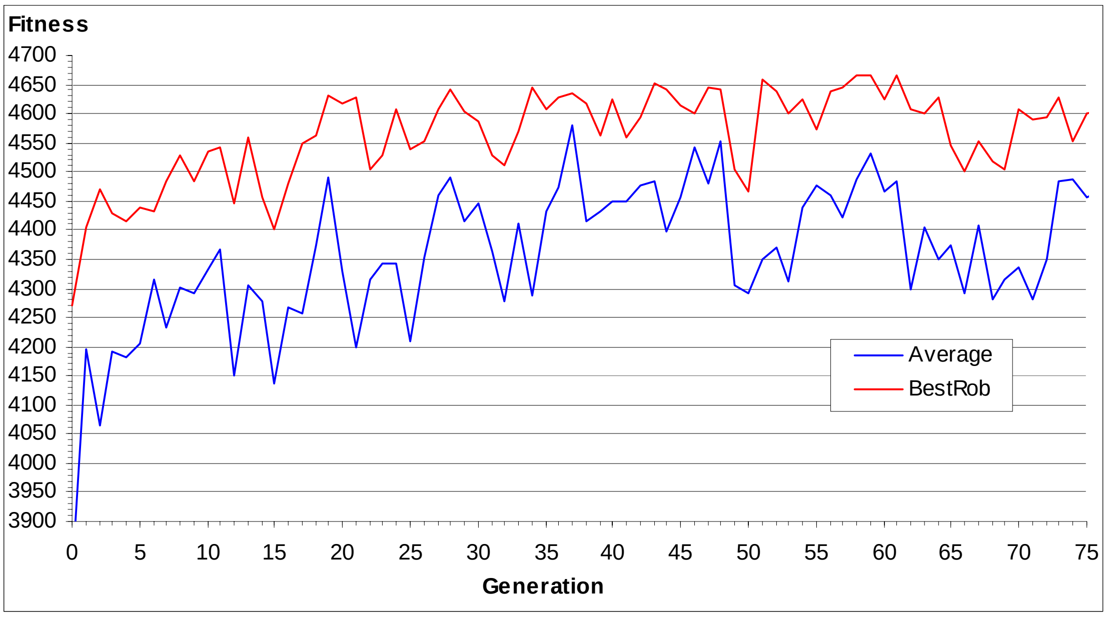


Figure 6.32 – *Summary of Experiment 4.1c*: Summary showing the average fitness (*Average*) of all robots and the fitness of the best robot (*BestRob*) in the generation.

Figure 6.32 shows that the system still suffered from instability. Evolution was able to produce efficient solutions and converged faster than in Experiment 3, but the population performance still oscillated despite the general upward trend. The interactions among the robots still made them collide [Set97].

6.5.2 Experiment 4.2: Inheritance and a Strongly Biasing Function

The next test intended to analyse if a strongly biasing fitness function together with the new inheritance selection can produce a more stable evolution. This is obtained by cutting rule 7 from the fitness function, as explained in Experiment 3.3. This experiment uses the same settings presented in Table 6.5.

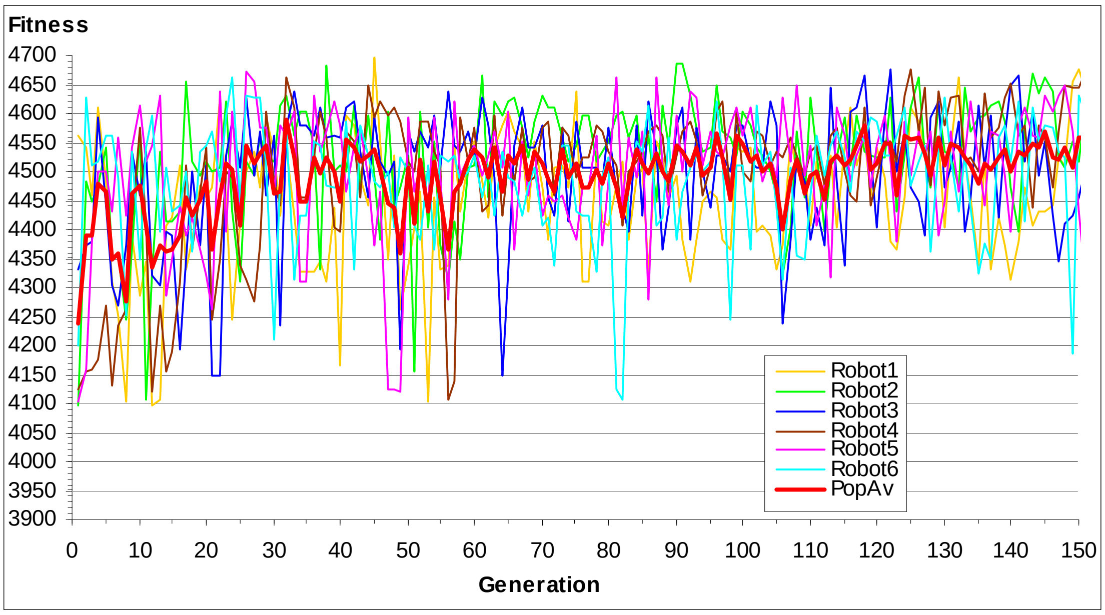


Figure 6.33 – ***Experiment 4.2a***: Evolution of the *black box* controller using inheritance selection and a strongly biasing fitness function in an environment of medium complexity, mutation rate of 3%, sexual reproduction, and generation time of 60s. The population was randomly initialised. Here, *Robotn* is the fitness of Robot *n* and *PopAv* is the average fitness of all robots in the generation.

- **Results**

The results presented by Figure 6.33 and Figure 6.34 show the behaviour of the robot population when initialised with a random and a hand-designed controller, using a strongly biasing fitness function.

- **Discussion**

It can be seen in Figure 6.33 that without punishing the robots for their collisions, the system is much more stable. The population performance did not degrade and the system did not oscillate much during the whole process. Evolution kept improving the population performance with a good rate and seemed to be converging towards an optimal solution. The question now is how long it would take to get there. The next test investigated what would happen when the population was initialised with a hand-designed controller. The results of this new test are shown in Figure 6.34.

Figure 6.34 shows that even now the evolutionary system could not prevent the population performance from degrading. This figure showed that in 50 generations, the

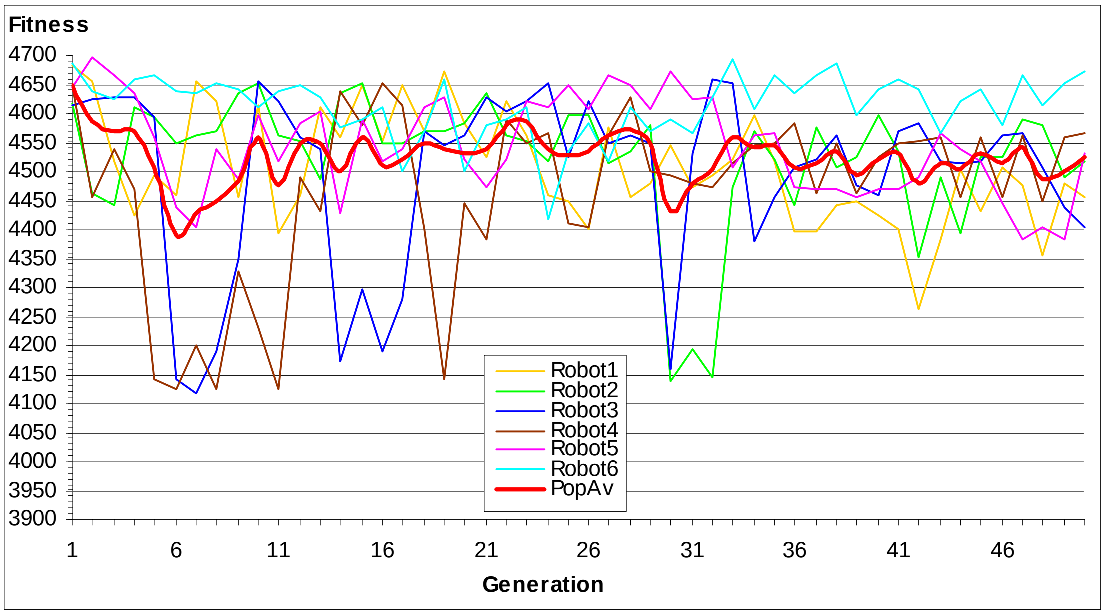


Figure 6.34 – *Experiment 4.2b*: Evolution of the *black box controller* using inheritance selection and a strongly biasing fitness function in an environment of medium complexity, mutation rate of 3%, sexual reproduction, and generation time of 60s. The population was initialised with a hand-designed controller. Here, *Robotn* is the fitness of Robot *n* and *PopAv* is the average fitness of all robots in the generation.

population performance degraded to a level similar to the one where the experiment of Figure 6.33 stopped, showing that it is unlikely that it will improve the population much more than this level.

6.5.3 Discussion of the Experimental Results

This experiment showed that the new inheritance selection helped to improve system performance, but did not completely solve the instability problem. The system does not seem to be able to produce an optimal solution to the collision-free navigation task. The experiments demonstrated that even using a strongly biasing fitness function, evolution still can take a considerable amount of time to achieve a solution as good as the hand-designed controller and may not even be able to get there.

The suggested inheritance selection is a powerful strategy to prevent much of the chance factor from affecting the system. It succeeded in preventing most of the unfit individuals being mistaken as the best robot and protected the better robots from being erased by reproduction, giving them a chance to survive and repeat their better performance, until their average performance overcame the one of the current best robot.

The evolutionary system needs to be evaluated more extensively to show the optimal parameters that can tune it to work at its maximum capacity. A simulator can provide fast long-term evolution that can exhaustively test the capacity of the system. Simulation can be an important tool to help finding the best mutation rates, selection strategies, and crossover parameters. Simulation can also provide an ideal environment, without noise and where the interactions among the robots can be carefully specified. It can take complexity away from the system, allowing the correct tuning of some important parameters. The next chapter will analyse the developed evolutionary system in simulation.

7 SIMULATED EXPERIMENTS

To provide a more comprehensive analysis of the evolutionary system in an ideal environment, where the interactions among the robots can be treated individually, a simulator was designed to take away the complexity of the physical robotic system. It was not an attempt to reproduce a detailed experimental environment. It was actually an attempt to discard all these details and concentrate on how the evaluation, selection, and crossover operators work. This is different to the normal use of simulation in evolution, where the real evaluation process is abstracted and reproduced in a computational environment. This simulator generated a completely different evaluation process. Instead of trying to reproduce how the performances of the robots are evaluated in the real environment, this simulator only compares the response of the evolving controller to a certain input to the response of a targeted hand-designed controller to the same input. Every time the two responses coincide, the evolving controller scores one point. This is clearly the strongest way to bias the fitness function. It tries to direct evolution towards a specific solution, rewarding more the solutions as they get closer to it. All the other phases of the evolutionary system were carefully reproduced in the simulator. The selection, crossover, and mutation operators are still the same algorithms used in the robots

This chapter presents the experiments that made use of this simulator to produce data much faster than using real robots. A simulator was applicable for two reasons: it can test the performance of the evolutionary system in the long-term; and it provides an ideal environment, without noise and where the interactions among the robots can be carefully specified. Simulated evolution can also take much of the complexity of the system away and provide important insights on the specification of mutation rates, selection strategies, crossover parameters, and population size.

It is acknowledged that there is no substitute for experimenting with real robots and simulation is not included as a permanent phase of the evolutionary system. Therefore, this chapter does not attempt to use simulation to train the robots in a computer program before transferring them to the real environment. Simulation is only applied to abstract complexity from the robots and the environment, to allow different characteristics of the system to be analysed individually and provide understanding of the complex evolutionary system. This study, it is hoped, may indicate better configurations and genetic operators to be applied later to the evolution of real robots with an improved system. Appendix C contains the average results of many experiments that could not be included in the body of the thesis. For Chapter 7, it includes data related to Sections 7.2, 7.3, 7.4, 7.5, 7.6, and 7.7.

7.1 The Simulator

The simulator was written in Borland C++ and different versions of the main routine were used for each experiment. The corresponding programs are all listed in Appendix A and will be referred to when each experiment is introduced. Figure 7.1 overviews how the simulator works.

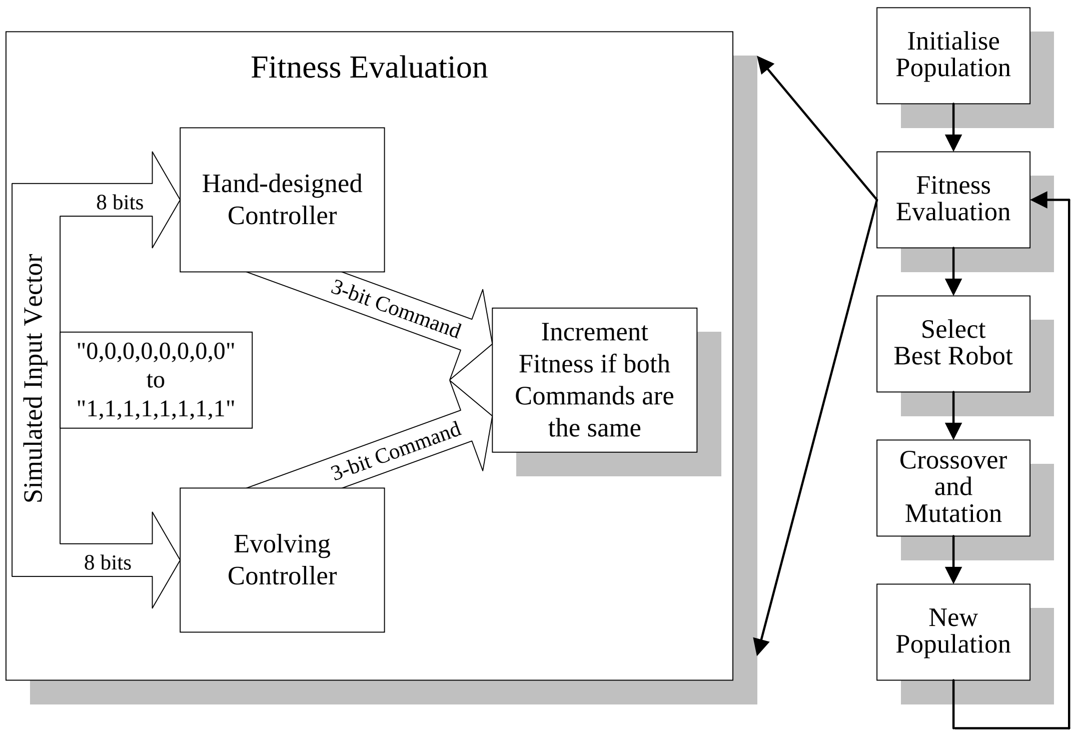


Figure 7.1 – General view of the evaluation phase of the simulator.

The sensor configuration for all the experiments in this chapter fixed all the 16 control bits of the eight sensors to “00-00-00-00-00-00-00-00”, enabling every sensor permanently. All sensors have only one bit of precision. In the evaluation phase, the simulator works by generating all the possible input combinations, which are the simulated sensor readings from “0,0,0,0,0,0,0,0” to “1,1,1,1,1,1,1,1”. Both the evolving controller and the hand-designed controller input signals are stimulated with the same 256 possible combinations of the sensor states (“0” means the sensor is not detecting anything and “1” means it is detecting an obstacle). Each time both controller outputs coincide, the fitness of the simulated robot is increased by one. This means that the maximum score that a robot can make if it behaves exactly as the hand-designed controller is 256 points.

After the evaluation phase, the robots are selected to breed according to the inheritance selection strategy described and evaluated in Section 6.5. The inheritance selection rule is reproduced below:

- *The score used to select the robot is the average of the robot fitness in the last three generations (i.e., inheriting the scores of its previous two generations). The robot with the best average survives, and breeds only with the robots with fitness in the present generation lower than its own fitness.*

All the experiments in this chapter use this selection strategy, which considers not only the performance of the robots in the current generation, but also the score of their predecessors. Using the above criteria, only one robot is selected to be the best robot, the one that supplies its chromosome to be combined with the other robots to form the next generation. Robots are protected if they have a score higher than the score of the best robot. All other robots will combine their chromosomes with the best robot and reconfigure afterwards with the resultant genes. The best robot and the protected ones do not change and survive to the next generations. The crossover phase combines the two chromosomes from the parent robots to form a resultant offspring; each bit is randomly chosen from the corresponding location in the chromosome of the parents. Then, in the mutation phase, a random number is generated between zero and 100 for each bit in the chromosome, and the bit will be flipped (from “0” to “1”, or “1” to “0”) each time the generated random number is smaller than the mutation rate. Different forms of mutation and crossover will also be tried in the experiments in this chapter and they will be explained each time.

7.2 Experiment S1: Simulated Evolution

This experiment is intended to provide an estimative of how long it would take for the real evolution of Section 6.5.1 to produce a solution similar to the hand-designed controller. It tried to provide understanding on why the experiment reported in that section was taking so long to evolve the robots. It was also carried out with different mutation rates, to provide a comparison of the effects of mutation in the system performance and indicate the most effective mutations.

The evolving controller is the black box look-up table described in Section 6.4 that consists of 256 bytes of memory (see also Figure 6.22). The commands that it can produce are encoded in the first three bits of the byte. These commands are: *Front Fast* (*FF*); *Turn Left Short1* (*TLS1*); *Turn Right Short1* (*TRS1*); *Turn Right Short2* (*TRS2*); *Turn Left Short2* (*TLS2*); *Turns Right Long* (*TRL*); and *Turn Left Long* (*TLL*). These commands were explained before in Section 6.4. Table 7.1 shows how these commands are encoded with three bits.

Table 7.1 – Encoding of the Black Box Commands.

Command	**Encoding**
FF	0,0,0
TLS1	0,0,1
TRS1	0,1,0
TRS2	0,1,1
TLS2	1,0,0
TRL	1,0,1
TLL	1,1,0
Not used, interpreted as *FF*	1,1,1

- **Aim of the Experiment**

The aim of this experiment is to run a simulated evolution for as long as 30,000 generations to see how close the hand-designed controller could become to the evolving controller. It also evaluated the simulator to see if it can be used to train a programmable architecture, such as the black box controller, to behave exactly as a targeted circuit, the hand-designed controller. Both the evolving controller and the targeted circuit can be other architectures. This simulator can be used to train a neural network, for example, to behave as a PID controller or a fuzzy logic system. Virtually anything that can be programmed can be trained using the same principle if both circuits can be described in Borland C++ language.

- **Experimental Setting**

In these experiments, the hand-designed controller operated according to the following algorithm:

```
     Left = Right = 0;
     If (Sensor4=1) then Left = Left + 1;
 -   If (Sensor3=1) then Left = Left + 1;
     If (Sensor2=1) then Left = Left + 1;
 P   If (Sensor6=1) then Right = Right + 1;
 r   If (Sensor7=1) then Right = Right + 1;
 i   If (Sensor8=1) then Right = Right + 1;
 o   If (Sensor1=1) then Command = TRS2;
 r   If (Left > Right) then Command = TRS1;
 i   If (Left = Right) then Command = FF;
 t   If (Left < Right) then Command = TLS1;
 y   If (Sensor1=1 and Sensor4=1) then Command = TRS2;
     If (Sensor1=1 and Sensor6=1) then Command = TLS2;
 +   If (Sensor4=1 and All other Sensors=0) then Command = TRL;
     If (Sensor6=1 and All other Sensors=0) then Command = TLL;
```

These experiments used a biasing fitness function that directs evolution to produce a solution that behaves like the hand-designed controller. This fitness function works according to the following rules:

1- Start with 4100 points;

2- Reward: increase fitness by 1 point every time the outputs from both controllers are the same;

3- Each robot is evaluated 256 times to test all possible input combinations;

The fitness functions used in simulation were directly imported from the robot software and preserved the original parameters. This means that they still start with 4100 points, even though the robot score cannot decrease in the simulator, but this facilitates comparison of the results to the ones obtained with real experiments, since they have similar scales. The maximum that a robot can score if it responds to all 256 input combinations exactly as the hand-designed controller is 4356 points. The 2048 bits of the chromosome are randomly initialised before the simulated evolution starts. Table 7.2 presents a summary of the settings for this experiment.

Table 7.2 – Summary of the Experimental Settings

Parameter	Definition
Fitness Function:	+1 point each time both outputs are the same
Initial Fitness Value:	4100 points
Maximum Fitness Value:	4356 points
Generation Time:	Simulated evolution
Mutation Rate:	0.0%, 0.1%, 0.5%, 1%, 3%, 10%, 20%, 50%, 80%
Selection Strategy:	Inheritance
Crossover Strategy:	Sexual
Speed Levels:	Simulated evolution
Sensors Enable:	Fixed with all enabled
Navigation Controller:	Evolving Black box with 256 bytes
Initial Sensor Configuration:	"00-00-00-00-00-00-00-00"
Initial Speed Configuration:	Simulated evolution
Initial Controller Configuration:	Random.
Software:	SIM01.CPP and GRAFSIM01.CPP (listed in Appendix A)

- **Results**

The chart shown in Figure 7.2 presents a comparison of three evolutionary experiments simulated for 30,000 generations with a population of 6, 50, and 100 robots. For these tests, the robots were initialised with a random black box controller and went on evolving with mutation rates of 0.1% and 0.5%. The simulator showed that the closer evolution gets to the optimal solution, the longer it takes to improve the solutions. This happens because evolving a black box look-up memory involves searching in a search space of size 2^{768}. This made the problem intractable by the evolutionary system, since 30,000 generations of one minute each would last almost 21 days of non-stop evolution and so far would not solve the problem.

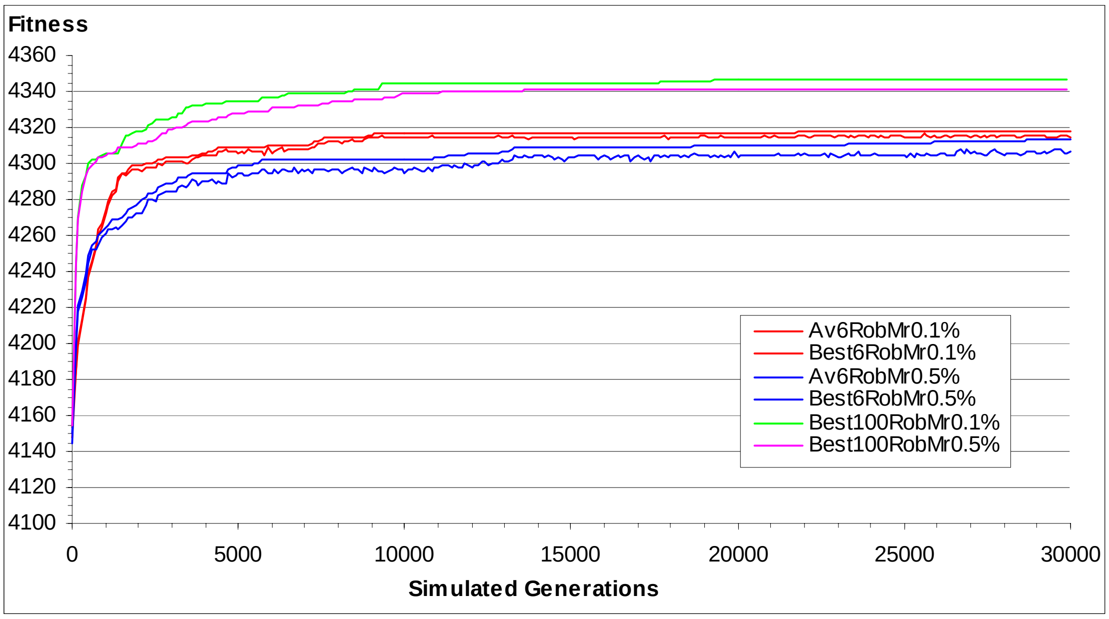


Figure 7.2 – ***Experiment S1.1***: Simulated evolution of the *black box* controller using inheritance selection, and sexual reproduction for two different mutation rates: 0.1% and 0.5%. Here, *AvnRobMrm%* and *BestnRobMrm%* are the average fitness of all the robots and the fitness of the best robot in the generation for the tests with *n* robots and mutation rate of *m*%.

The experiments shown in Figure 7.2 were simulated for 30,000 generations and gave much insight into the way the evolutionary system works. They demonstrated why the real evolution presented in Section 6.5 was taking so long in producing a solution like the hand-designed controller: this is not so easy to find. It can be seen in the figure that even after 30,000 generations in an idealised virtual environment, where the robot evaluation does not suffer the effects of noise and interactions among robots, evolution still did not get close to the optimal solution. For a population of six robots, the maximum fitness after 10,000 generations was 4317 points for 0.1% mutation and 4302 points for 0.5%, still far from the fitness of the optimal solution, which is 4356 points. After running the program for a further 20,000 generations, 0.1% mutation only increased one point, resulting in 4318 points. The 0.5% mutation curve increased only 11 points. Even when the number of robots in the population was increased to 100, the system still could not reach to maximum score in 30000 generations. This shows that the long time necessary for the system to evolve is probably a consequence of the large search space.

Evolution gets slower when the population converges because when mutation modifies genes in a chromosome that has more than 50% of good genes, the chance of changing a good gene to a bad one is greater than changing a bad gene to a good

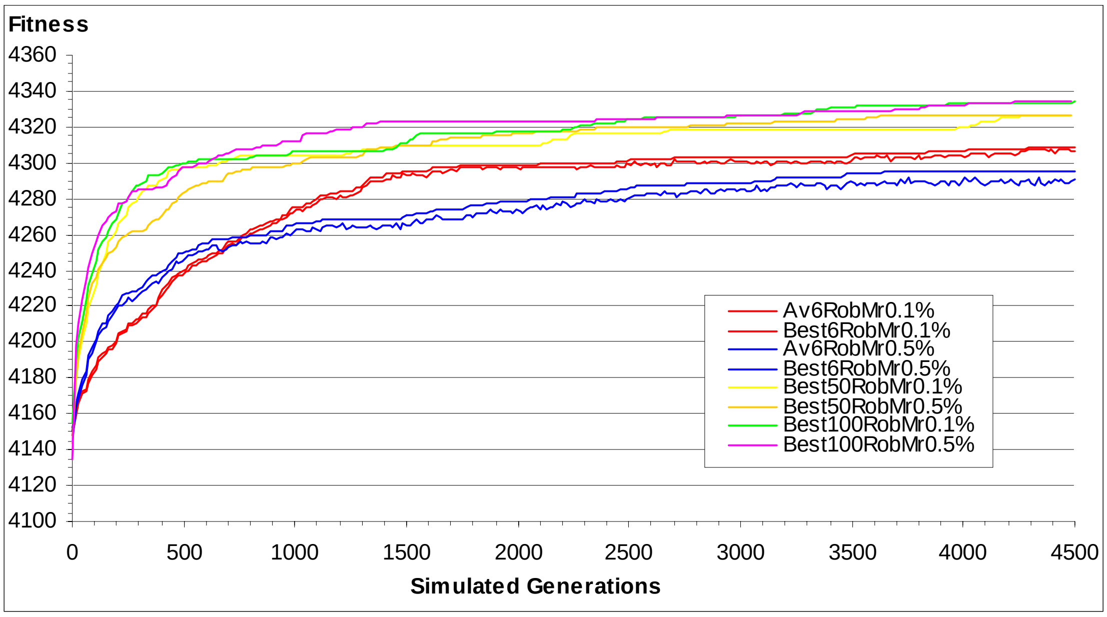


Figure 7.3 – ***Experiment S1.2***: First 4500 generations of Experiment S1.1, showing more details of the simulated evolution of the *black box* controller using inheritance selection, and sexual reproduction for two different mutation rates: 0.1% and 0.5%. Here, *AvnRobMrm%* and *BestnRobMrm%* are the average fitness of all the robots and the fitness of the best robot in the generation for the tests with *n* robots and mutation rate of *m*%.

one. When the chromosome has 99% of good genes, the chance of having a positive mutation is only 1%. Figure 7.3 shows a comparison of the previous curves in the first 4500 generations. For a small population of six robots, in the long term, a 0.1% mutation is better than 0.5%, which helped to evolve faster in the beginning of the process, until generation 810, when 160 genes in the chromosome, from a total of 256, were already correct. Then, a higher mutation had more chances to change good genes to bad ones and 0.1% mutation began to show a better performance. For larger populations, these effects are less prominent. Both curves for 100 and 50 robots evolve quickly up to 4280 points, when 100 robots start to make a greater difference.

Although the current configuration of the system deals with a large search space and was very slow in finding an optimal solution, it may still be useful to test the effects of different mutation rates. The simulator is good for such a task because it is less stochastic than the real world and the effects of mutation can be analysed individually. It is actually only mutation and the randomly initialised population that cause its non-determinism. Figure 7.4 shows a comparison of four small mutation rates: 0.1%; 0.5%; 1%; and 3%. Figure 7.5 shows a comparison of four higher mutation rates: 10%; 20%; 50%; and 80%.

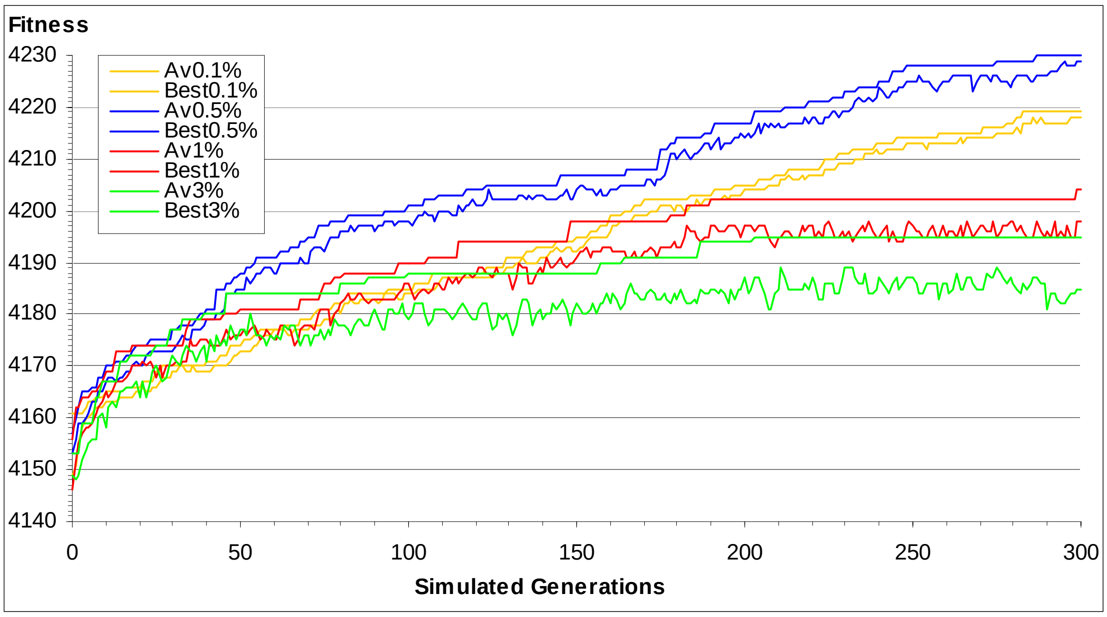


Figure 7.4 – ***Experiment S1.3***: Simulated evolution of the *black box* controller using inheritance selection, and sexual reproduction for four different mutation rates: 0.1%, 0.5%, 1%, and 3%. Here, *Av* is the average fitness of all the robots and *Best* is the fitness of the best robot in the generation.

The last two figures show that with such a small population, after the first 20 generations or so the population relies only on mutation to increase its performance. Higher mutations help evolving in the beginning of the process, when there are more bad than good genes in the chromosome. It can be seen in Figure 7.4 that 3%, 1%, and 0.5% mutation evolved faster than 0.1% in the beginning of the process, but as the genes in the robot chromosomes became better, 0.1% mutation outperformed 1% and 3%. The best mutation was 0.5%, which showed a good performance during the whole process. It can be observed how the curves of the average fitness and the fitness of the best robot got more distant from each other when mutation was increased, showing that higher mutations make the population more disperse according to the fitness of the robots.

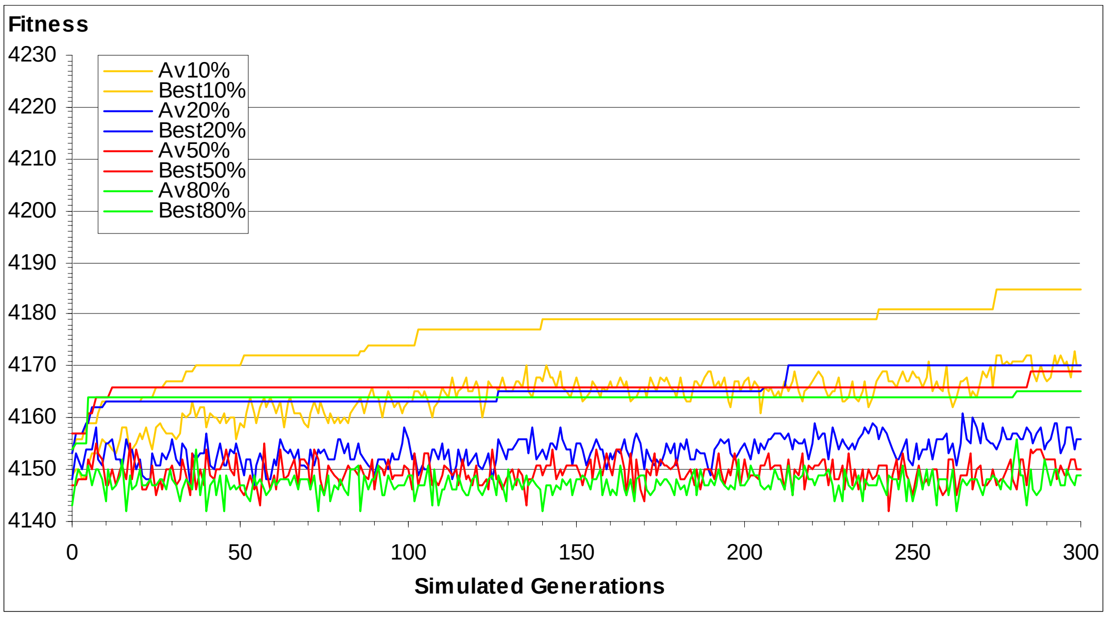


Figure 7.5 – ***Experiment S1.4***: Simulated evolution of the *black box* controller using inheritance selection, and sexual reproduction for four different mutation rates: 10%, 20%, 50%, and 80%. Here, *Av* is the average fitness of all the robots and *Best* is the fitness of the best robot in the generation.

The chart in Figure 7.5 shows that the same trend of small mutations is true for higher rates: the higher the mutation rate, the faster the system evolves in the first generations, but the slower it becomes in the end of the process. Higher mutations also increase the distance between the curves of the average fitness and the fitness of the best robot. Figure 7.6 shows that without mutation, the population relies only on its diversity to increase performance and after the initial convergence in the first ten or 20 generations, the population stopped evolving.

7.2.1 Discussion of the Experimental Results

This experiment suggested that unstructured evolution generates a very large search space that makes the problem intractable for the developed evolutionary system as it is set. The black box controller, containing 2^{768} possibilities is just too complex to be treated even by an ideal simulated environment, and seems practically impossible to be evolved towards an optimal solution in a real environment. Even when the population is

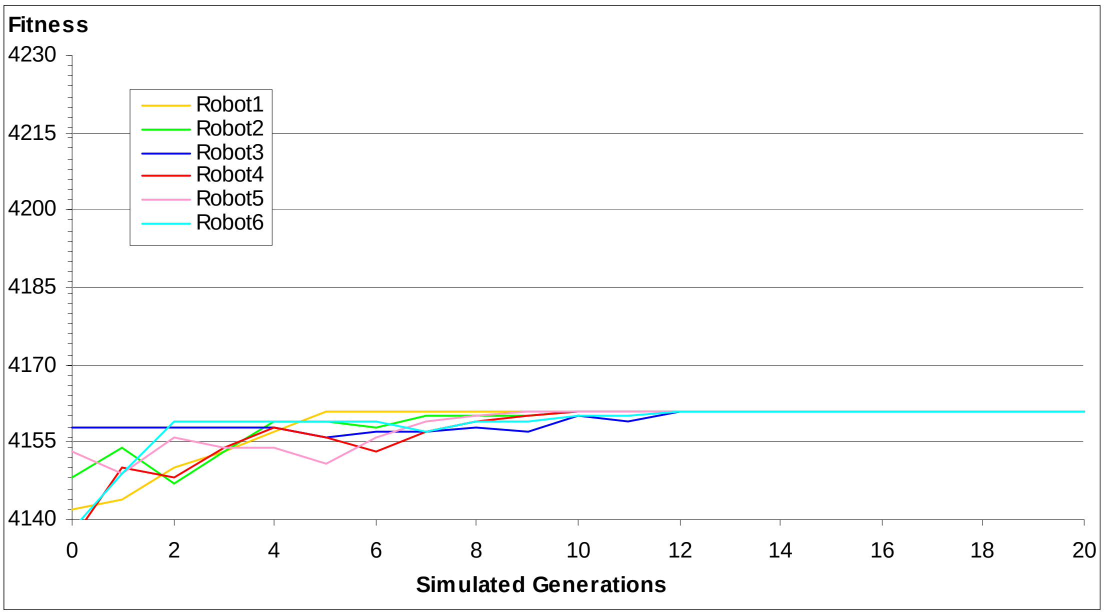


Figure 7.6 – *Experiment S1.5*: Simulated evolution of the *black box* controller of the six robots using inheritance selection, sexual reproduction, and mutation rate of 0.0%. Here, *Robotn* is the fitness of Robot *n* in the generation.

increased to 100 robots, the system still could not reach the maximum score. An unstructured evolvable controller avoids the designer's prejudices. It does not restrict the possibilities that evolution needs to investigate, slowing down the evolutionary process.

This experiment showed that for 300 generations with six robots, 0.5% mutation was the most effective one, improving system performance considerably in relation to other settings. Even though the system worked much better with the right mutation level, after 300 generations the fitness of the best robot was just 4230 points, only 50.78% of the maximum score (4230 – 4100 = 130, which is 50.78% of the maximum score: 4356 – 4100 = 256). The smaller the mutation, the higher the chance of changing only bad genes when the chromosome has more good than bad ones.

Concluding, this experiment showed two important things: choosing the right mutation rate drastically improves the speed of the system; and the unstructured black box controller makes evolution look for the optimal solution in a very large search space, which slows down the process too much, making it almost intractable for the evolutionary system in a physical environment. Another possibility of obtaining a better operation without increasing the population and preserving the large search space of an unstructured controller is to change how the system works, with new selection, mutation, or crossover strategies. The following experiments explore these possibilities.

7.3 Experiment S2: Asexual Reproduction

This experiment used exactly the same settings as the previous experiment, with the exception of the crossover strategy. This experiment makes use of a simple strategy that uses inheritance to select the fittest robot, allows it to survive, and reconfigures all the others with a small variation of the fittest robot chromosome. This is a form of *asexual reproduction*, where the robots do not cross over their chromosomes. The selection operator has not changed from the previous experiment. The only operation that was modified was the crossover of the genes. The combined inheritance-asexual reproduction operator is defined below:

- *The score used to select the robot is the average of the robot fitness in the last three generations. The robot with the best average survives, and the robots with the fitness in the present generation lower than the fitness of the best robot overwrite their chromosome with a copy of the chromosome of the best robot, and then suffer mutation in a few genes.*

- **Aim of the Experiment**

The aim of this experiment is to determine if the combined inheritance-asexual reproduction operator is more efficient than the one used in the previous experiment. This experiment explored an attempt to make the evolution search faster for a solution amongst the large number of possibilities of an unstructured controller such as the black box.

- **Experimental Setting**

This experiment used the same settings of Experiment S1 with a different selection-crossover strategy. Table 7.3 presents a summary of the settings for this experiment.

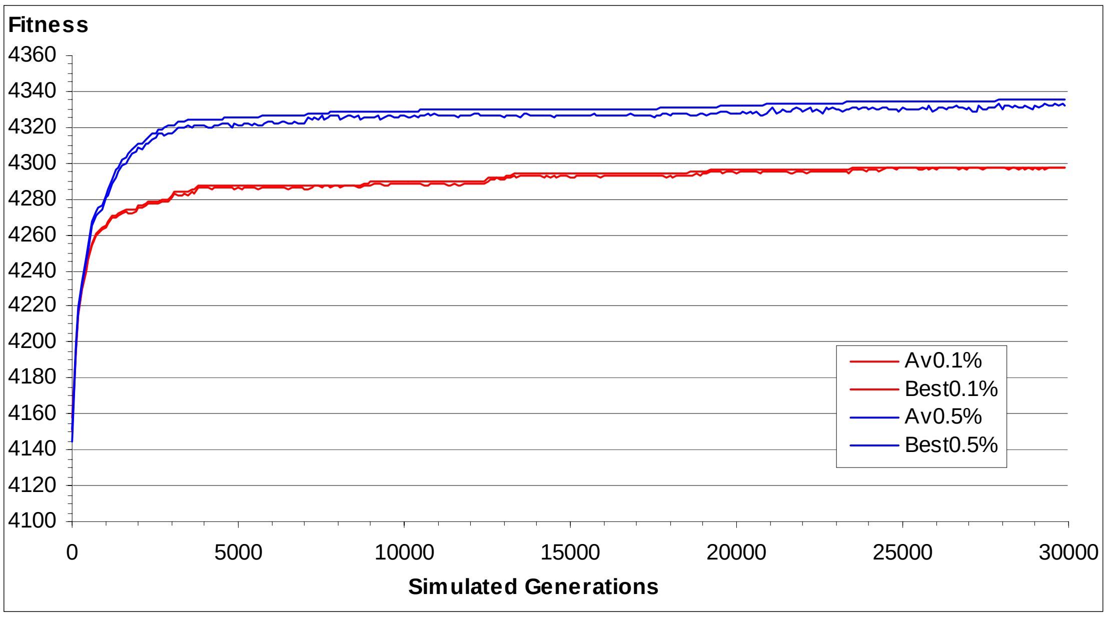


Figure 7.7 – ***Experiment S2.1***: Simulated evolution of the *black box* controller using inheritance selection, and asexual reproduction for two different mutation rates: 0.1%, and 0.5%. Here, *Av* is the average fitness of all the robots and *Best* is the fitness of the best robot in the generation.

Table 7.3 – Summary of the Experimental Settings

Parameter	**Definition**
Fitness Function:	+1 point each time both outputs are the same
Initial Fitness Value:	4100 points
Maximum Fitness Value:	4356 points
Generation Time:	Simulated evolution
Mutation Rate:	0.1%, 0.5%, 1%, 10%
Selection Strategy:	Inheritance
Crossover Strategy:	Asexual
Speed Levels:	Simulated evolution
Sensors Enable:	Fixed with all enabled
Navigation Controller:	Evolving Black box with 256 bytes
Initial Sensor Configuration:	"00-00-00-00-00-00-00-00"
Initial Speed Configuration:	Simulated evolution
Initial Controller Configuration:	Random.
Software:	SIM02.CPP and GRAFSIM02.CPP (listed in Appendix A)

- **Results**

The experiment presented in Figure 7.7 shows the long-term evolution of the robot population for 30,000 generations. In the figure, the new selection-crossover strategy behaved better than the one used by the previous experiment, but was still very slow and would take a long time to converge towards the optimal solution in a real experiment. It still did not solve the problem. The figure shows a comparison between two mutation rates:

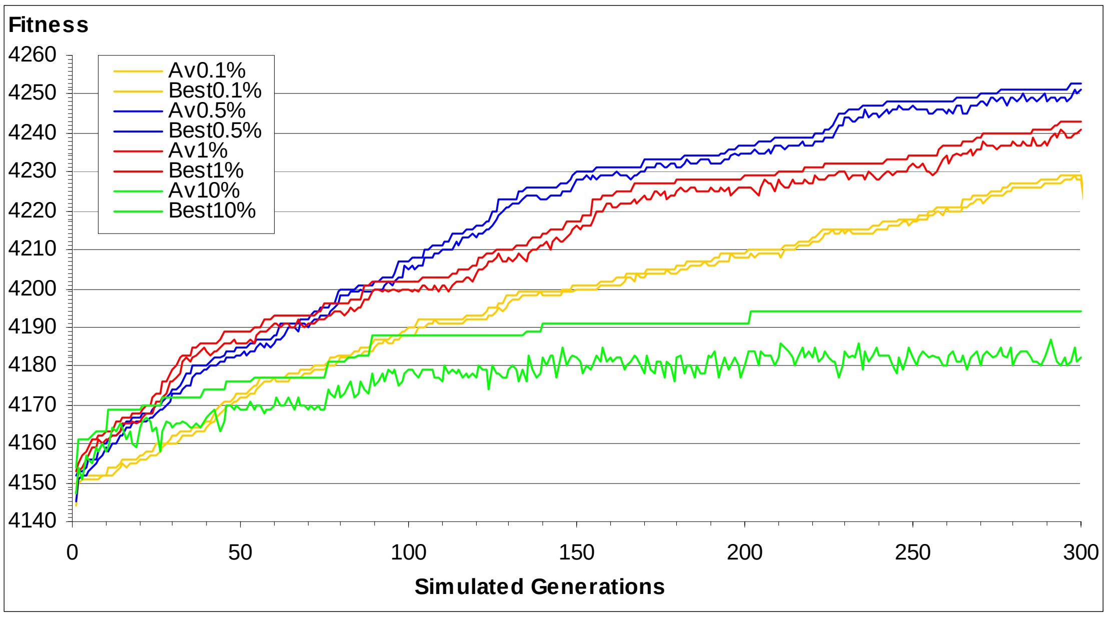


Figure 7.8 – ***Experiment S2.2***: Simulated evolution of the *black box* controller using inheritance selection, and asexual reproduction for four different mutation rates: 0.1%, 0.5%, 1%, and 10%. Here, *Av* is the average fitness of all the robots and *Best* is the fitness of the best robot in the generation.

0.1% and 0.5%. The most relevant point when comparing the information in this figure to Figure 7.2 is that here, 0.1% mutation is not better than 0.5% in the long term, which is contrary to the results of Experiment S1. An explanation for this is that in sexual reproduction, evolution would employ mutation and crossover to manipulate the bits in the chromosome. With asexual reproduction, only mutation can change the bits in the chromosome. For each possible input, one byte of the black box look-up memory is chosen. The first three bits in this byte form the command to the motors. The evolving controller is evaluated by comparing this command with the one produced by the hand-designed controller. If the black box is outputting the wrong command for a specific input, *FF* for example, which is encoded by "0,0,0", instead of *TRL*, encoded by "1,0,1", evolution cannot generate "1,0,1" from "0,0,0" by changing only one bit each generation. With sexual reproduction, the robot can, for example, inherit "0,0,1" from the chromosome of one of the parents and mutate the first "0" to "1", producing the "1,0,1" combination and the *TRL* command.

Figure 7.8 shows a comparison among four different mutation rates (01%, 05%, 1%, and 10%) with the new inheritance selection and asexual reproduction operators. This solution lacks the opportunity of exploring combinations of the population diversity, but worked faster than the strategy chosen by Experiment S1. As it was observed in Experiment S1, the curves with higher mutations evolved faster in the beginning, where mutation operated on chromosomes predominated by bad genes. However, when the population improved, lower rates showed the best performances, with 0.5% presenting the best result for 300 generations.

The experiment in Figure 7.9 shows a comparison between two selection-crossover strategies: the inheritance selection plus sexual reproduction from Experiment S1 and inheritance selection with asexual reproduction from this experiment. It compares the two best mutation rates from each strategy: 0.1% and 0.5%. Although sexual reproduction worked faster in the beginning of the process, in the long term it was outperformed by asexual reproduction. This can be explained by the ability of sexual reproduction to combine the best genes of the robot population in the beginning of the process. Once the initial diversity of the population was reduced by convergence and the population relied only in mutation to evolve, asexual reproduction was the best strategy.

Sexual reproduction applies mutation on a population of robots resultant from the combination of the chromosome from the best robot and the chromosome from an average robot. Therefore, after the population converges in the initial generations, mutation is applied on a resultant chromosome that is potentially worse than the best robot. After the original randomly initialised population has converged to similar configurations, asexual reproduction behaved better because it applies mutation directly on copies of the best chromosome.

7.3.1 Discussion of the Experimental Results

The combined inheritance selection and asexual reproduction, in this experiment, was a better strategy for the developed evolutionary system to evolve in the long term. In this experiment, Sexual reproduction was better only in the beginning of the process, when evolution can still work with the original diversity of the population.

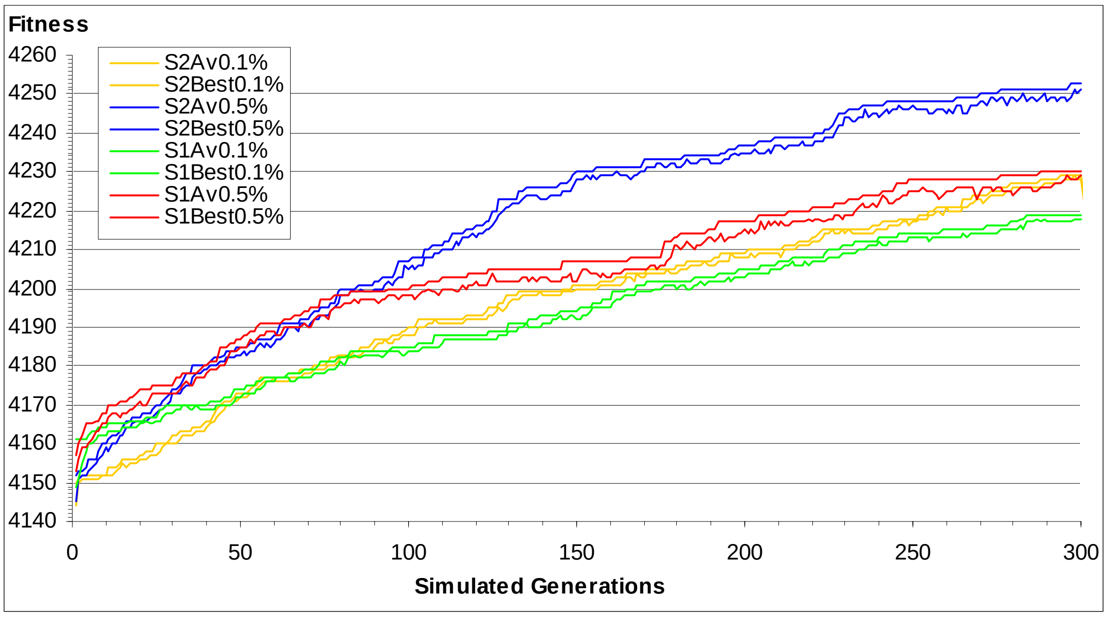


Figure 7.9 – ***Summary***: Comparison between the results of Experiments S1.1 and S2.1 for the mutation rates of 0.1%, and 0.5%. The experiments tested the simulated evolution of the *black box* controller using inheritance selection for sexual and asexual reproduction. Here, *Av* is the average fitness of all the robots and *Best* is the fitness of the best robot in the generation.

In the long term, the best mutation rates are still the smaller ones in these experiments. The best rate is the one that has the highest probability of changing only a few bits of the chromosome each generation. In these experiments, the robot chromosome contained 2048 bits, but only 768 were relevant to evolution, since only the first three bits of the byte in the controller output contained the encoded command. The right mutation for this case was the one able to modify only one or two of these 768 bits, because it can produce all other six commands from the current one. For sexual reproduction, a mutation rate of 0.5% has the probability of changing 3.84 bits each time, and is better early on in the process. The mutation rate of 0.1% has the probability of changing 0.768 bits. Hence, 0.1% is the closest probability of changing only one bit and showed the best results, since another bit or two can be changed by the crossover operation. For asexual reproduction, only mutation manipulates the bits, so 0.4% or 0.5% produce better results.

This solution, however, is still too slow to be applied to an unstructured controller such as the developed black box controller in a real environment containing a small population. To allow the developed evolutionary system to reach an optimal solution with these setting, another possible way is to change the crossover operator, as the next experiment will demonstrate.

7.4 Experiment S3: A Different Way to Evolve

This experiment attempted to evolve the bits in the chromosome that specify the controller behaviour in a different way that had not been attempted by any of the previous experiments. Here, the bits in the chromosome are grouped in small portions. From the analogy to nature, as every amino acid that forms the proteins is encoded by three nucleotides [Ler76] [Cam99], the first three bits (b_0, b_1, and b_2) in the byte that represent the commands (this experiment is using the same settings of Experiment S1) are grouped and treated as an entity. It can be seen from Table 7.1 that this encoding still presents some neutrality, since the command *FF* can be coded by “0,0,0” and “1,1,1”. In this experiment, when the robots mate, they do not exchange or mutate single bits in the chromosome, they exchange the 3-bit entity that represents the commands. This developed technique was called *command exchanging*.

In this new strategy, the robots do not exchange bits in the crossover phase, they exchange commands. In the mutation phase, the bits are not inverted one by one, instead, the 3-bit entities that form the commands mutate together. In the crossover phase, the resulting command in the chromosome (the 3-bit entity) is randomly chosen from one of the parent chromosomes to occupy its corresponding position in the offspring chromosome. Mutation generates a random number *r* between zero and 100 for each one of the 256 commands (the 3-bit entity) and mutates it if *r* is smaller than the mutation rate. If the command is to be mutated, it is substituted by another command chosen randomly to occupy its position.

- **Aim of the Experiment**

The aim of this experiment is to evaluate a new strategy to deal with the bits in the robot chromosome. It tested how evolution performed in the long term, to indicate if the optimal performance could be obtained using this new approach.

- **Experimental Setting**

This experiment used the same settings of Experiment S1 and examined the new strategy of manipulating the bits with the selection-crossover strategy from

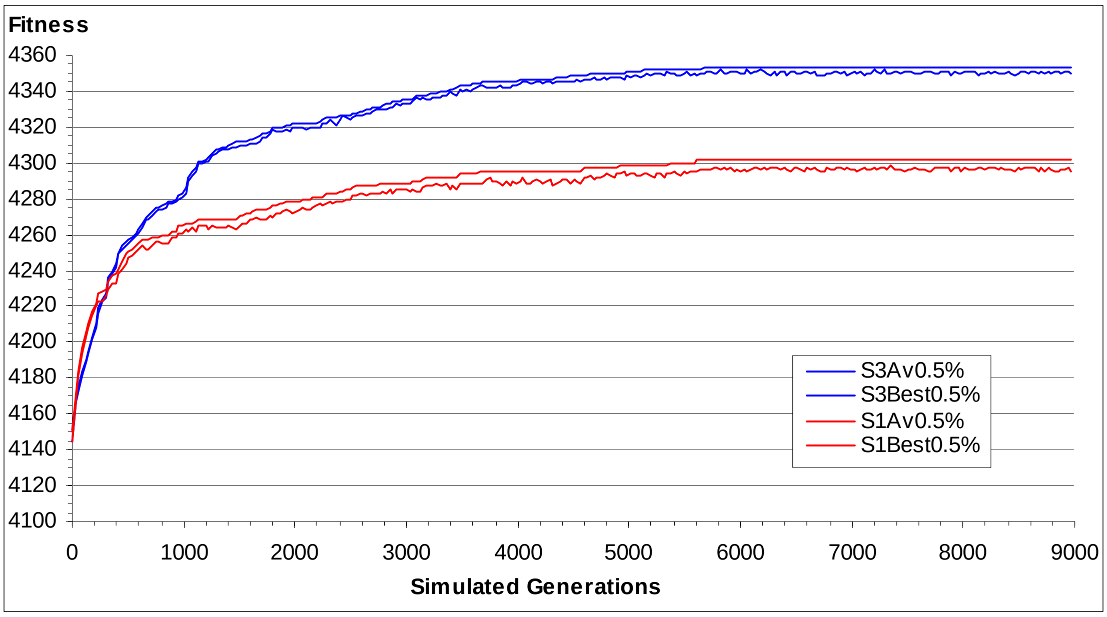


Figure 7.10 – ***Experiment S3***: The results of Experiment S3 compared to Experiment S1.1 for the mutation rate of 0.5%. The experiments tested the simulated evolution of the *black box* controller using inheritance selection and sexual reproduction with and without command manipulation. Here, *Av* is the average fitness of all the robots and *Best* is the fitness of the best robot in the generation.

Experiment S1: inheritance with sexual reproduction. Table 7.4 presents a summary of the settings for this experiment.

Table 7.4 – Summary of the Experimental Settings

Parameter	**Definition**
Fitness Function:	+1 point each time both outputs are the same
Initial Fitness Value:	4100 points
Maximum Fitness Value:	4356 points
Generation Time:	Simulated evolution
Mutation Rate:	0.5%
Selection Strategy:	Inheritance
Crossover Strategy:	Sexual
Speed Levels:	Simulated evolution
Sensors Enable:	Fixed with all enabled
Navigation Controller:	Evolving Black box with 256 bytes
Initial Sensor Configuration:	“00-00-00-00-00-00-00-00”
Initial Speed Configuration:	Simulated evolution
Initial Controller Configuration:	Random.
Software:	SIM04.CPP and GRAFSIM04.CPP (listed in Appendix A)

- **Results**

Figure 7.10 compares two experiments with the same mutation rate (0.5%) and sexual reproduction, but with the original strategy of bit-by-bit manipulation in the chromosome assumed in Experiment S1, against the new approach, that manipulates the commands directly. They started with the same performance until generation 330 from where the new approach showed a much superior performance evolving the population all the way to the optimal solution in 8500 generations.

7.4.1 Discussion of the Experimental Results

This strategy of manipulating the whole command each time was much faster and reached the optimal solution, being able to evolve the robot population all the way from random to the hand-designed controller. The most interesting fact is that the search space did not change. It is still the same large number of possibilities that seemed impossible to overcome in the previous experiments. Before, evolution needed to find the right position for 768 bits in the chromosome, and this imposed a search space of 2^{768} that equals to 1.55×10^{231} possibilities. Here, evolution needed to find the right command from a set of eight for 256 positions in the chromosome corresponding to all possible input combinations, and the imposed search space was 8^{256}, which is also equal to 1.55×10^{231} possibilities. Therefore, the size of the search space did not change and is still very large.

This strategy is the most effective until now. However, it is still very slow if it is to be applied to the evolution of a real system, since it took 2500 generations to reach 90% of the maximum performance. This would correspond to more than 40 hours of continuous evolution with a generation time of one minute. Therefore, there is still room for improvement in this approach.

7.5 Experiment S4: Disabling Back Mutation

In the previous experiments, mutation was randomly choosing which bits in the chromosome to change. This meant that any bit in the chromosome could mutate to a different value and later mutate back to its original value. In the scope of this work, these events were called *back mutations* [Ler76] [Cam99]. This fact was causing evolution to waste time trying to find a good solution by testing some configurations that have been tested before. Consider a chromosome that has 99% of its bits set to the correct ones. If any bit can be changed by mutation, the chance of changing the bad ones is only 1%. This means that 99% of the time, this strategy produces configurations that are potentially worse than the current chromosome, and many generations are wasted in evaluating them.

To prevent evolution from wasting time in evaluating configurations that have been tested before, a new strategy that prevents back mutations is developed in this experiment. It consists of marking each bit in the chromosome that suffered mutation and only allowing the bits that are not marked to mutate. To achieve this, a binary array was created in memory with the same size of the chromosome. It is initialised with zeros and when one bit in the chromosome is mutated, "1" is written to the corresponding position in the array. Only the bits in the chromosome that have zeros in the corresponding position in the array are allowed to mutate. Once all bits have mutated, the array will be full of "1s". Then, the evolutionary system resets the array to "0s" and every bit in the chromosome will be allowed to mutate again. This strategy was called *back-mutation prevention*.

By preventing back mutations and mutating a single bit every generation, this strategy forces evolution to evaluate the effect produced by each bit in the chromosome. A problem that emerges from applying this strategy to an evolutionary system that uses asexual reproduction is that by changing only one bit per generation, mutation may not be able to produce all possible combinations of the bits in the chromosome. As explained in Section 7.4, if two bits needed to be changed to produce the correct command, changing one bit at a time will not increase performance and the change will be discarded.

This new strategy can be successfully applied to an evolutionary system that uses asexual reproduction if it can manipulate directly the 3-bit entities that define the commands. Therefore, the same strategy for crossing over and mutating directly the commands presented in Experiment S3 was applied here.

- **Aim of the Experiment**

The aim of this experiment is to apply a new strategy that prevents back mutation of the bits in the chromosome. It tested whether this strategy could be applied in sexual and asexual reproduction and provided data to analyse which is the best solution for evolving the robots in the long term.

- **Experimental Setting**

This experiment used the same settings of Experiment S3 where the evolutionary system is able to manipulate the commands in the chromosome directly. It examined the new strategy of back-mutation prevention with both selection-crossover strategies from Experiments S1 and S2: inheritance with sexual and asexual reproduction. Table 7.5 presents a summary of the settings for this experiment.

Table 7.5 – Summary of the Experimental Settings

Parameter	Definition
Fitness Function:	+1 point each time both outputs are the same
Initial Fitness Value:	4100 points
Maximum Fitness Value:	4356 points
Generation Time:	Simulated evolution
Mutation Rate:	Only 1 command (3 bits) is modified in the chromosome each time
Back Mutation:	Disabled
Selection Strategy:	Inheritance
Crossover Strategy:	Sexual and Asexual
Speed Levels:	Simulated evolution
Sensors Enable:	Fixed with all enabled
Navigation Controller:	Evolving Black box with 256 bytes
Initial Sensor Configuration:	“00-00-00-00-00-00-00-00”
Initial Speed Configuration:	Simulated evolution
Initial Controller Configuration:	Random.
Software:	SIM05.CPP and GRAFSIM05.CPP (listed in Appendix A)

- **Results**

Figure 7.11 compared the new back-mutation prevention strategy with sexual and asexual reproduction when mutation was allowed to modify only one command in the chromosome for each generation. The figure shows a considerable increase in performance for both sexual and asexual reproduction. As demonstrated by the previous

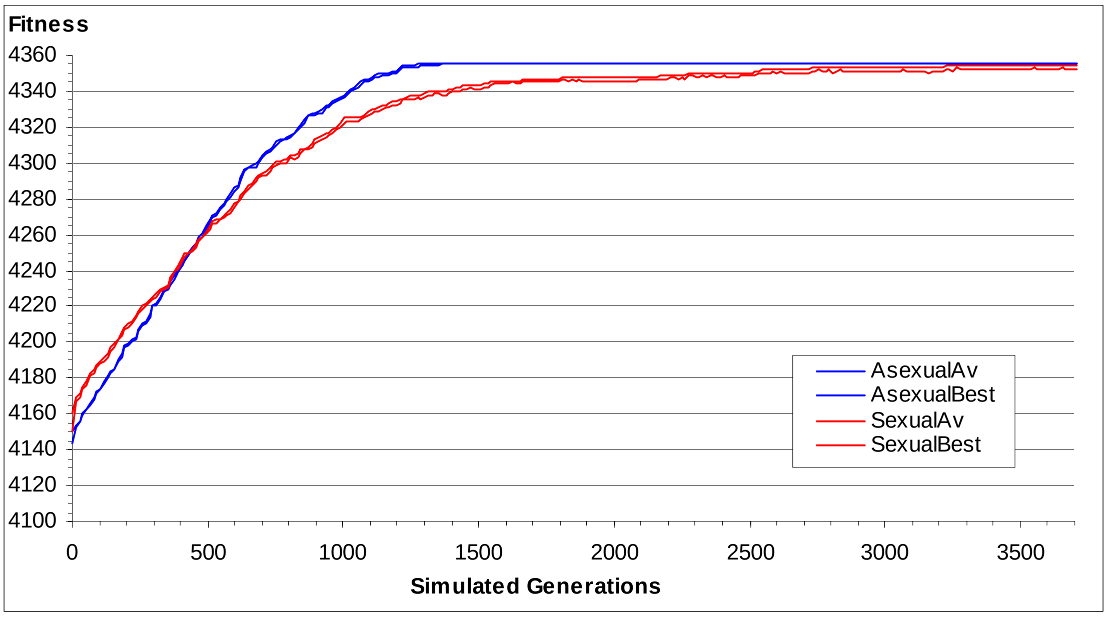


Figure 7.11 – ***Experiment S4.1***: Simulated evolution of the new strategy of back-mutation prevention for sexual and asexual reproduction. The experiment tested the simulated evolution of the *black box* controller using inheritance selection with command manipulation. Only one command (3 bits) is allowed to mutate each time. Here, *Av* is the average fitness of all the robots and *Best* is the fitness of the best robot in the generation.

experiments, sexual reproduction worked better in the beginning of the process, but after 600 generations, asexual reproduction dominated, reaching the optimal solution in less than 2000 generations. Evolution with sexual reproduction was slower, but still efficient, reaching the optimal performance in 3700 generations.

The techniques developed so far succeeded in allowing the employment of the evolutionary system to control such a small population of robots. Figure 7.12 shows the effect in the performance of the system when the number of robots in the population is increased up to 100 individuals. The experiment indicates that a good gain in performance can be obtained by increasing the number of robots from 6 to 15. It can also be inferred from the experiment that increasing the population to more than 35 robots does not bring great improvements to the performance of the system until it reaches 90% of the maximum score. After this, the gain in performance is so small that it is difficult to justify the cost of doubling or tripling the number of robots.

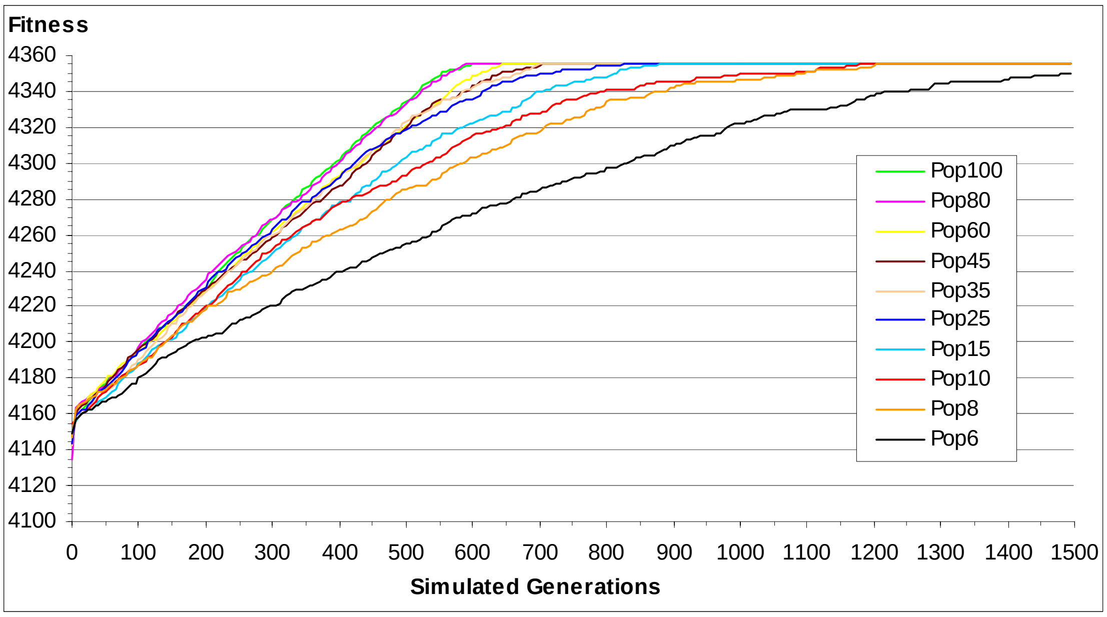


Figure 7.12 – ***Experiment S4.2***: Simulated evolution of the strategy of back-mutation prevention with asexual reproduction for different population sizes. The experiment tested the simulated evolution of the *black box* controller using inheritance selection with command manipulation. Only one command (3 bits) is allowed to mutate each time. Here, *Popn* is the fitness of the best robot in the generation for the test with a population of *n* robots.

7.5.1 Discussion of the Experimental Results

Preventing back mutation was very efficient in speeding up the process of finding a solution. Evolution was able to reach 90% of maximum performance in less than 950 generations for asexual reproduction and 1300 generations for sexual reproduction. These results allow the evolution of a real system in 15 hours. The combined strategies of preventing back mutations and manipulating the commands made viable the use of an unstructured controller such as the developed black box.

If the number of robots in the population is increased up to 35, the system can present a very good performance, reaching 90% of maximum performance in about 500 generations. This is almost half of the time the six robots take to evolve to the same point. Figure 7.12 demonstrates that building six or ten more robots would improve considerably the performance of the system. More robots though will not bring an important gain and would probably just generate more interference between the robots, making the whole system slower. This experiment was intended to be a reference for future work with larger populations.

The strategy of back-mutation prevention tested in this experiment was very efficient in looking for a solution in a considerably large search space of 1.55×10^{231} possibilities. It demonstrated the power of the developed evolutionary system for many other applications of genetic algorithms, not only in robotics. The performance of the system can be improved even more if the search space is reduced by using a neural network to implement a structured evolving controller.

7.6 Experiment S5: The Neural Network Controller

In this experiment, a neural network is used to implement the evolving controller in order to impose some organization to evolution. It tried to propose a control circuit that can be configured by fewer bits than the black box, but one that is still able to drive the robot with a similar performance to the black box controller.

In this experiment, a neural network was designed according to the same architecture presented in Chapter 4. It has four groups of neurons (discriminators) with seven 2-input neurons each. This neural network is able to send four commands to the motor drive module: *Front Fast* (*FF*); *Turn Left Short1* (*TLS1*); *Turn Right Short1* (*TRS1*); and *Turn Right Short2* (*TRS2*). These commands were explained before in Chapter 6, Section 6.2. Only two bits are necessary for encoding these four commands. Table 7.6 shows how these commands were encoded with two bits.

Table 7.6 – Encoding of the Neural Network Commands.

Command	**Encoding**
FF	0,0
TLS1	0,1
TRS1	1,0
TRS2	1,1

The evolving neural network controller is evaluated in the same way as the black box controller, according to the strategy presented in Figure 7.1 and explained in Section 7.1. The sensors work with 1-bit precision in the *medium* range. Sensor *S5* is not connected to the neural network, but still all possible 256 combinations of the eight sensors are used to evaluate the evolving neural network. Therefore, only 128 configurations are effectively relevant to evaluate the neural network. In the other 128 it will just repeat the same score obtained for the first 128 ones. Hence the neural network will score twice for the “x,x,x,0,x,x,x,x” and the “x,x,x,1,x,x,x,x” combinations. It was left this way because it facilitates the comparison of the results, since it still gives the same maximum score to the optimal solution: 4356 points. The evaluation proceeds by giving one point each time the commands coincide.

The optimal solution in this experiment is represented by a simplified hand-designed controller similar to the one described in Section 6.3, which operates according to the following algorithm:

```
-   Left = Right = 0;
    If (Sensor4=1) then Left = Left + 1;
P   If (Sensor3=1) then Left = Left + 1;
r   If (Sensor2=1) then Left = Left + 1;
i   If (Sensor6=1) then Right = Right + 1;
o   If (Sensor7=1) then Right = Right + 1;
r   If (Sensor8=1) then Right = Right + 1;
i   If (Sensor1=1) then Command = TRS2;
t   If (Left > Right) then Command = TRS1;
y   If (Left = Right) then Command = FF;
    If (Left < Right) then Command = TLS1;
+   If (Sensor1=1 and Sensor4=1) then Command = TRS1;
    If (Sensor1=1 and Sensor6=1) then Command = TLS1;
```

In the evaluation phase of the simulated evolution, the neural network controller is compared to the above hand-designed controller for all 256 possible input combinations and each time the two outputs are the same, the robot scores one point. The process of evaluation works similarly to the one shown in Figure 7.1, but with only two bits in the output of both controllers encoding the four possible commands.

- **Aim of the Experiment**

The aim of this experiment is to evaluate the performance of the neural network controller, determining if it can be evolved faster than the black box controller. It tested whether the structured neural network controller can accelerate the evolution of the system with four different mutation rates.

- **Experimental Setting**

This experiment used the same evaluation strategy of Experiment S1 with the same selection-crossover strategy: inheritance with sexual reproduction. Differently from the last experiment, the neural network controller here was evolved without the strategy that prevented back mutation from occurring and by manipulating the bits in the chromosome, not the commands. This was necessary because the neural network does not allow evolution to manipulate directly the 3-bit entities that define commands. Table 7.7 presents a summary of the settings for this experiment.

Table 7.7 – Summary of the Experimental Settings

Parameter	**Definition**
Fitness Function:	+1 point each time both outputs are the same
Initial Fitness Value:	4100 points
Maximum Fitness Value:	4356 points
Generation Time:	Simulated evolution
Mutation Rate:	0.5%, 1%, 3%, 15%
Selection Strategy:	Inheritance
Crossover Strategy:	Sexual
Speed Levels:	Simulated evolution
Sensors Enable:	Fixed with all but *S5* enabled
Navigation Controller:	Evolving Neural Network (m=4, n=7, neuron size=4)
Initial Sensor Configuration:	“00-00-00-11-00-00-00-00”
Initial Speed Configuration:	Simulated evolution
Initial Controller Configuration:	Random.
Software:	SIMNEU01.CPP and GRAFNEU01.CPP (listed in Appendix A)

- **Results**

The results shown in Figure 7.13 compared four different mutation rates: 0.5%, 1%, 3%, and 15%. The neural network controller evolved much faster than the black box controller did, even without manipulating the commands directly or preventing back mutations from occurring. It reached 90% of the best performance in the first 310 generations, dropping the estimated experiment time from 15 hours (the best performance

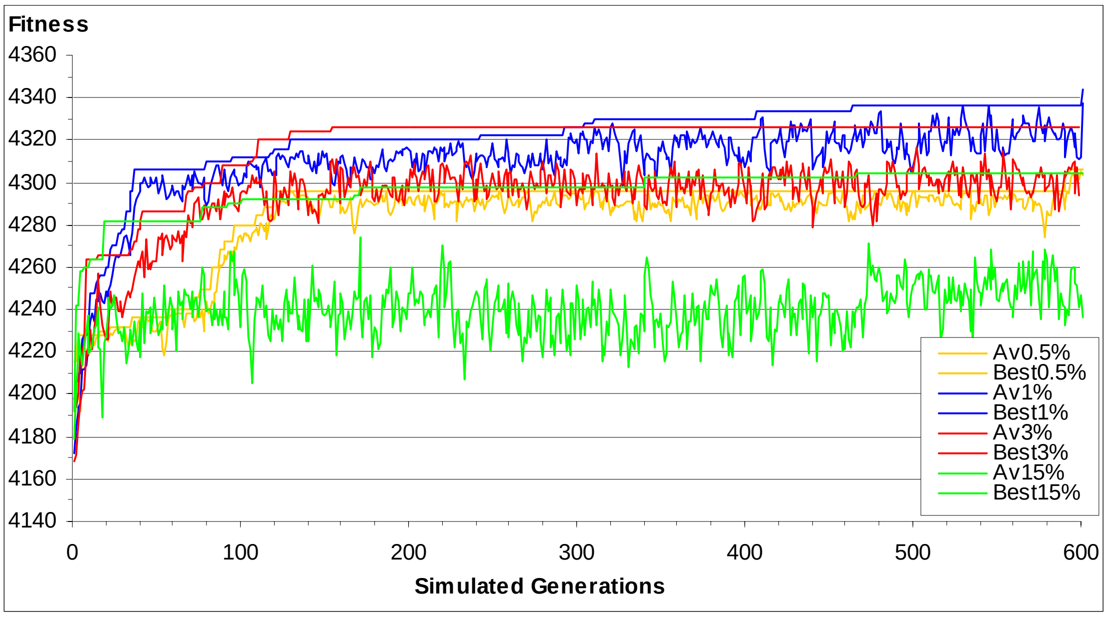


Figure 7.13 – *Experiment S5*: Simulated evolution of the neural controller using inheritance selection and sexual reproduction for four different mutation rates: 0.5%, 1%, 3%, and 15%. Here, *Av* is the average fitness of all the robots and *Best* is the fitness of the best robot in the generation.

obtained in Experiment S4) to 5.2 hours if the real robots are evolved with 1-minute generations.

7.6.1 Discussion of the Experimental Results

According to Figure 7.13, the higher mutation rates of 3% and 15% produced better results in the beginning of the process, where there were more bad genes than good ones in the chromosomes. Therefore, mutating a greater number of genes in these chromosomes actually helped to evolve faster. Even then, the average fitness of the population was already lower than that produced by a lower mutation such as 1%. The mutation rate of 3% could reach 4320 points, which is 85.93% of the maximum result, in less than 120 generations, outperforming 1% mutation. The mutation rate of 15% reached 4280 points, which is 70.31% of the maximum result, in less than 22 generations, outperforming all other rates. Therefore, the correct mutation rate depends on what is the desired behaviour of the system. If what is important is to converge quickly to a reasonable solution, a higher mutation is the best option. If a more solid performance is expected, a lower mutation rate should be chosen.

The neural network has fewer configuration bits than the black box controller does; four groups of seven neurons with four bits each, in a total of 112 bits. The search space is considerably smaller than the black box one: $2^{112} = 5.19\times10^{33}$ possibilities. With 112 bits to adjust, 1% mutation performed better than 0.5%. As the relationship between inputs and commands is intrinsic to the neural network generalisation, there is no way to manipulate directly the commands, since it is not possible to identify which bits encode each command inside the neural network. Therefore, the same strategy used in the last experiment cannot be applied to increase the performance of the neural network.

7.7 Experiment S6: Predation

As the direct manipulation of the commands cannot be applied to the neural network controller, a different approach was developed to increase its performance. It was called *predation*. In analogy to nature, the robot population can suffer regular attacks of a “predator” that selects the worst (“weakest”) robot in the specified generation and destroys (“kills”) it [Cam99], opening space in the population for the migration of new individuals, what brings more genetic diversity to the group [Ler76]. To achieve this, every 20 generations, the robot with the smallest average fitness is selected and substituted by a robot with a random configuration.

It is important to give enough time to the population so that the attacked robot can recover from the “attack” and its new genetic material can be incorporated in the population. If the attacks occur in less than ten generations, the attacked robot will not have time to recover and will be selected as the worst one in the next attack. The necessary time for the attacked robot to recover depends on the complexity of the system.

- **Aim of the Experiment**

The aim of this experiment is to test the effects of the predation strategy on the evolving robot population. It tested whether this new strategy could make the neural network evolve faster. It also tested different mutation rates to indicate the one that provided the best performance.

- **Experimental Setting**

This experiment used the same evaluation strategy of Experiment S1 with the same selection-crossover strategy: inheritance with sexual reproduction. The robots controlled by the neural network suffered predation every 20 generations. Table 7.8 presents a summary of the settings for this experiment.

Table 7.8 – Summary of the Experimental Settings

Parameter	Definition
Fitness Function:	+1 point each time both outputs are the same
Initial Fitness Value:	4100 points
Maximum Fitness Value:	4356 points
Generation Time:	Simulated evolution
Mutation Rate:	0.5%, 1%, 3%, 15%
Selection Strategy:	Inheritance with Predation
Crossover Strategy:	Sexual
Frequency of the Attacks:	Every 20 generations
Speed Levels:	Simulated evolution
Sensors Enable:	Fixed with all but *S5* enabled
Navigation Controller:	Evolving Neural Network (m=4, n=7, neuron size=4)
Initial Sensor Configuration:	“00-00-00-11-00-00-00-00”
Initial Speed Configuration:	Simulated evolution
Initial Controller Configuration:	Random.
Software:	SIMNEU02.CPP and GRAFNEU02.CPP (listed in Appendix A)

- **Results**

The results presented in Figure 7.14 show the effects of the attacks in the robot population that evolved with 0.0% mutation. After the initial improvement resultant from combining the population diversity, the system would stop evolving as shown by the curve from Experiment S5. However, the new genetic material produced by the attacks kept the population evolving up to 4300 points in 300 generations. It can be seen in the figure that the attacks in generations 40, 60, and 120 created a random robot that, combined with the best robot, produced a superior configuration that improved the population

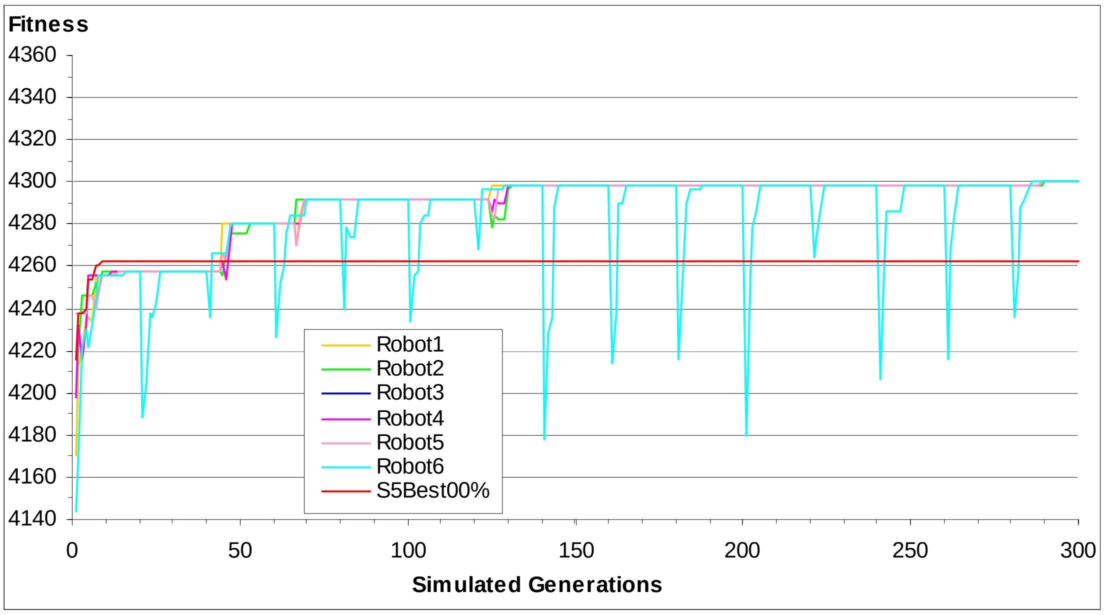


Figure 7.14 – ***Experiment S6.1***: Simulated evolution of the predation strategy compared to evolution without predation. The experiments tested the simulated evolution of the *neural* controller using inheritance selection, sexual reproduction, and mutation rate of 0.0%. Here, *Robotn* is the fitness of Robot *n* in the generation in the experiment with predation and *S5Best00%* is the fitness of the best robot in the generation in the experiment without predation (Experiment S5).

performance. If all robots converge to the same configuration, which would only happen if mutation is 0.0%, the worst robot is by default Robot 6.

It can be seen in Figure 7.15 that this strategy improved the system performance if compared to Experiment S5. With the mutation rate of 1%, it reached 90% of the maximum performance in 150 generations. A real system would then reach a good performance in less than 2.5 hours. Once again, 0.5% mutation performed worse than 1%, since it is not enough to modify at least one of the 112 genes every generation. The mutation rate of 1% has a higher chance of modifying only one or two of the genes, and behaved better in adjusting a chromosome that has most of its genes correct.

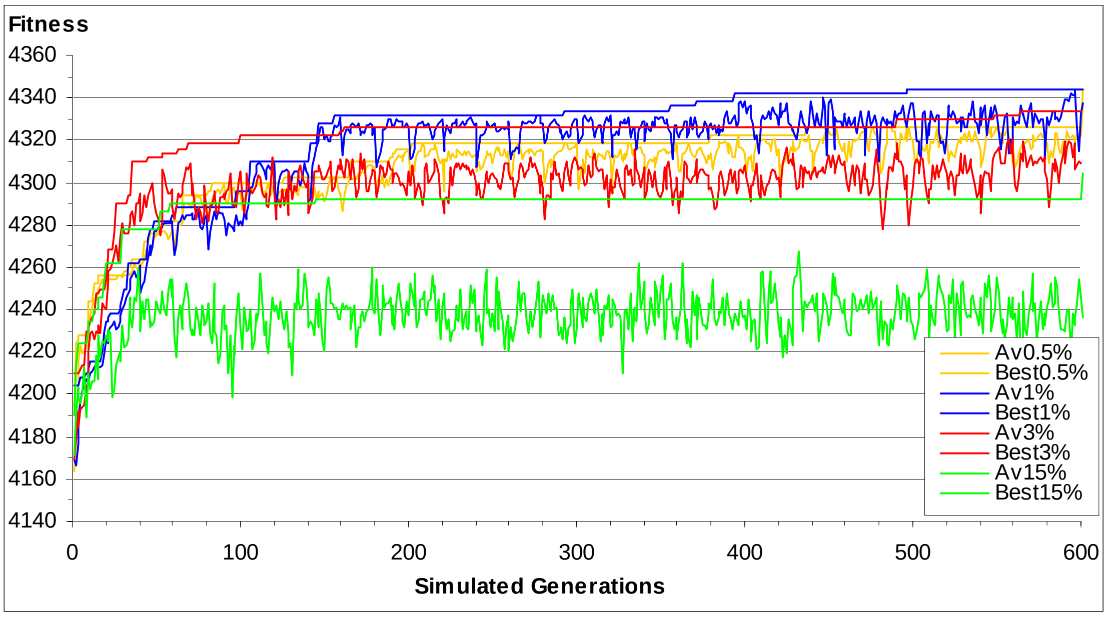


Figure 7.15 – ***Experiment S6.2***: Simulated evolution of the neural controller using inheritance selection, sexual reproduction, and predation for four different mutation rates: 0.5%, 1%, 3%, and 15%. Here, *Av* is the average fitness of all the robots and *Best* is the fitness of the best robot in the generation.

7.7.1 Discussion of the Experimental Results

The predation strategy presented in this experiment was able to improve the performance of the evolutionary system. The purpose of predation in this experiment was not to eliminate unfit (weak) robots, since the fitness of the random configurations used to substitute the destroyed robot was often worse than the original one. Even though the resulting random configuration was, after many attacks, worse than the destroyed robot, the new genetic material that it contained, combined with the chromosome of the best robot, produced in many occasions a better performance. Therefore, the purpose of predation in this experiment was to bring more diversity to the population.

The instant of the attacks can be identified in the figures by the drop of the fitness of the attacked robot, occurring every 20 generations. This happened because the resulting fitness of the attacked robot, which is reconfigured by a random chromosome, is usually worse than the one that the original robot would produce. Therefore, the average performance of the population usually drops after the attacks. Nevertheless, after a few generations, the new genetic material is filtered by the selection-crossover operators and incorporated in the population, often increasing the average performance. By protecting the best robot and allowing it to survive to the next generation, the maximum fitness of the population did not change, and the chromosome resulting from the evolutionary process until the current generation was always protected. In the ideal environment of simulation, this chromosome is only overwritten when one of its descendants produces a better performance. In the real world of physical robots, however, protecting this chromosome is much more difficult, since the noise and interactions among the robots can make a lucky unfit robot outperform the best one. It is therefore necessary to use all means to protect the best configuration, and a selection strategy that considers even more than three generations may help it.

Predation is a powerful strategy to prevent the population from being stuck in local optima, since it introduces new genetic material that may help the population to crawl down the slope and explore new possibilities in the fitness landscape [Har93a]. Getting stuck in local optima is an intrinsic problem of most evolutionary systems [Har97b], since it is very difficult, and sometimes impossible, to know, from the point of view of the evolving individuals, if the population can be improved even more, or if the optimal solution was actually achieved. With the developed predation strategy, the population can depend on a steady supply of new genetic material to bring in more diversity, even after it completely converged to a local optima [Hus95].

This was the last experiment in simulation. The results obtained with the help of the simulator were essential in providing useful data on the performance of the system. The analysis of the obtained information provided vital insights on developing new strategies that improved considerably the performance of the system. The simulator also made possible the evaluation of different parameters such as different mutation rates, reproduction and selection strategies, and evolvable controllers. The next chapter makes use of the developed techniques, combining the best results to produce an evolutionary system able to evolve not only the navigation controller, but also the sensor configuration and robot speed, which form the morphology of the robot.

8 EMBEDDED EVOLUTION

This chapter incorporates the strategies developed and tested in the preliminary experiments and in simulation (Chapters 6 and 7) to design a fully embedded evolutionary system that is able to evolve not only the robot navigation control circuit, but also the morphology of the robot: the configuration of the sensors and the speed levels of the motors. It describes the implementation of two controllers: one is based on an unstructured black box (see also Section 6.4); and the other on a neural network (see also Section 7.6). Appendix C contains the average results of many experiments that could not be included in the body of the thesis. For Chapter 8, it includes data related to Sections 8.1 and 8.2.

All the experiments shown in this chapter were evolved in the environment shown in Figure 8.1. It is an environment of medium complexity where the six robots could interact with each other and the obstacles. The robot sensors were set to have only one bit of precision and work in the *medium* range (15cm) (see Figure 4.30 and Figure 6.5). Sensor

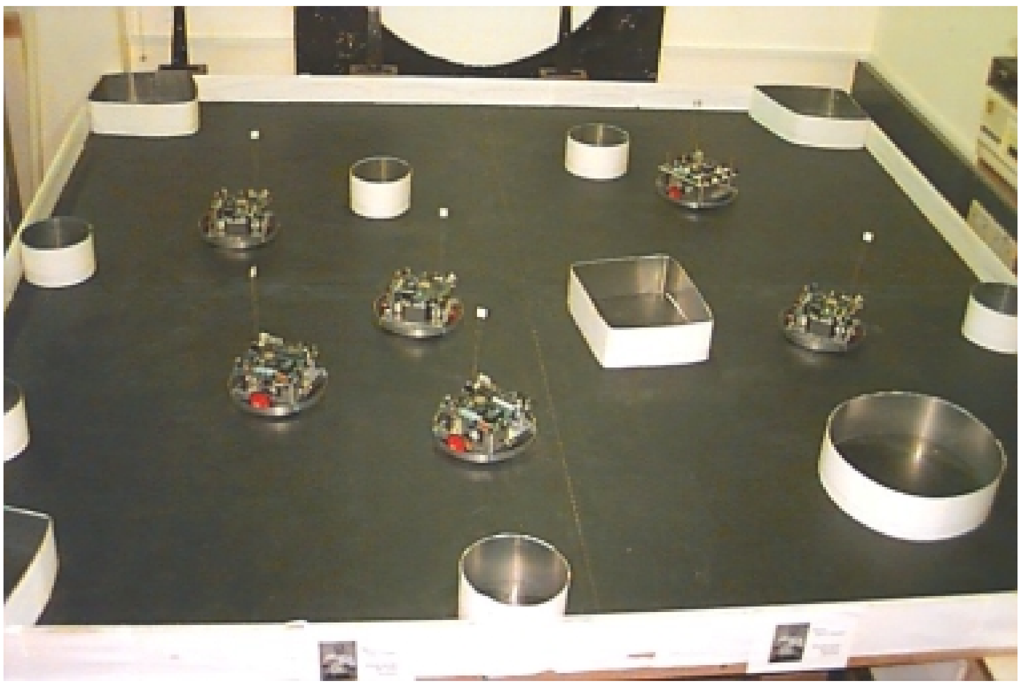


Figure 8.1 – General view of the six robots in the environment of medium complexity used in the experiments in this chapter.

S5 in the back of the robots can be used by the black box controller but has not been connected to the neural network controller. The position of the sensors around the robots is under evolutionary control. In the experiments shown in this chapter, all sensors can be selected by their controlling pair of bits.

8.1 Evolving with a Black Box Controller

This experiment applies every strategy developed in the previous chapters that can improve the performance of an evolving black box controller. This is an unstructured, undefined architecture, which was described in Section 6.4 and was presented in Figure 6.22.

The controller used eight commands to control the motor drive module: *Front Fast* (*FF*); *Turn Left Short1* (*TLS1*); *Turn Right Short1* (*TRS1*); *Turn Right Short2* (*TRS2*); *Turn Left Short2* (*TLS2*); *Turns Right Long* (*TRL*); *Turn Left Long* (*TLL*); and *Front Medium* (*FM*). The first seven commands were explained before in Chapter 7 and in the last one, the *Front Medium* command encoded by “111”, the robot moves forward with *Medium* speed. The speed levels of the robot *Fast* and *Medium*, activated by the commands *FF* and *FM*, are defined by ten bits stored in the robot chromosome as explained in Section 4.2.3 and illustrated in Table 4.4. The speed of the robots is not fixed at the maximum speed as in the previous experiments and is now under evolutionary control. In this experiment, to keep the robots moving all the time with a reasonable speed, the first five bits that define the speed levels in the chromosome are fixed at “1”. These guarantee that both speeds *Fast* and *Medium* will have at least five bits on, so the lower speed that the robots can travel is 17 (from a total of 32 possible speeds) according to Table 4.4. The maximum that *Medium* can be is 20 and *Fast* is 32.

The eight commands are encoded by the first three bits (b_0, b_1, and b_2) of the byte stored in the black box memory, as explained in Section 6.4. The black box lookup table is implemented using 256 bytes stored in the robot RAM memory. Every byte of the memory is arranged in the chromosome to form a 2048-bit string and crossover and mutation can affect each one of these bits. From the 2048 bits corresponding to the eight lookup tables, only 768, which correspond to the first three tables (b_0, b_1, and b_2) are relevant to evolution. This is the size of the genotype of the robots that correspond to the navigation control circuit, and considering that any one of them can produce a different phenotype (i.e., there is no neutrality in this case), the imposed search space is considerably large: $2^{768} = 1.55\times10^{231}$.

- **Aim of the Experiment**

The aim of this experiment is to provide the first experimental proof of an embedded evolutionary system that evolves an unstructured controller and the morphology of the robot. The 16 bits that control the sensor configuration, plus five of the ten bits that control the robot speed levels, and the 2048 bits of the black box are under evolutionary control. The total number of bits controlled by evolution is 2069 plus 5 fixed ones.

- **Experimental Setting**

A new hand-designed controller was developed as a reference to compare the results obtained by the evolutionary system. It is a variation of the previously-described ones, which includes the new command *FM* for the motor drive module. In this experiment, the hand-designed controller operated according to the following algorithm:

```
     Left = Right = 0;
-    If (Sensor4=1) then Left = Left + 1;
     If (Sensor3=1) then Left = Left + 1;
     If (Sensor2=1) then Left = Left + 1;
P    If (Sensor6=1) then Right = Right + 1;
r    If (Sensor7=1) then Right = Right + 1;
i    If (Sensor8=1) then Right = Right + 1;
o    If (Sensor1=1) then Command = TRS2;
r    If (Left > Right) then Command = TRS1;
i    If (Left = Right) then Command = FM;
t    If (Left < Right) then Command = TLS1;
y    If (Sensor1=1 and Sensor4=1) then Command = TRS2;
     If (Sensor1=1 and Sensor6=1) then Command = TLS2;
     If (Sensor4=1 and All other Sensors=0) then Command = TRL;
+    If (Sensor6=1 and All other Sensors=0) then Command = TLL;
     If (All Sensors=0) then Command = FF;
```

This hand-designed algorithm behaves as the ones explained in Chapter 6, but will only allow full speed (*Fast*) if all sensors are zero. If the robot detects obstacles in both sides, but the variable *Left* is equal to *Right*, the robot proceeds with caution: *Front Medium*. In this experiment, the first three bits (b_0, b_1, and b_2) of the byte that represent the commands are grouped and treated as an entity, as explained in Section 7.4. Table 8.1 shows how these commands are encoded with three bits in this experiment.

Table 8.1 – Encoding of the Black Box Commands.

Command	**Encoding**
FF	0,0,0
TLS1	0,0,1
TRS1	0,1,0
TRS2	0,1,1
TLS2	1,0,0
TRL	1,0,1
TLL	1,1,0
FM	1,1,1

This experiment uses the strategy of *back-mutation prevention* developed in Section 7.5. A new *inheritance selection* that calculates the average fitness of the robots in the last six generations was adopted in this experiment. This strategy also protected up to two robots that score more than the fitness of the best robot, as explained in Section 6.5. This experiment makes use of *asexual reproduction*. As it was explained in Section 7.3, the robots do not cross over their chromosomes. The modified selection strategy is described below:

- *The score used to select the robot is the average of the robot fitness in the last six generations (i.e., inheriting the scores of its previous five generations). The robot with the best average survives, but does not breed with up to two robots with the fitness in the present generation higher than its own fitness. The other robots overwrite their chromosome with a copy of the chromosome of the best robot, and then suffer mutation in the three bits that represent a command.*

Two charts in this chapter display the average fitness value (*AvRn*) scored by each robot in the current and the previous five generations. For each robot:

$$AvRn = (FitnessRn_{G0} + FitnessRn_{G-1} + FitnessRn_{G-2} + FitnessRn_{G-3} + $$
$$+ FitnessRn_{G-4} + FitnessRn_{G-5})/6$$

In the equation above, n is the number of the robot; $FitnessRn_{G0}$ is the fitness of the Robot n in the current generation; $FitnessRn_{G-1}$ is the fitness scored by Robot n in the previous generation; and so on until $FitnessRn_{G-5}$, which is the fitness scored by Robot n five generations before.

This experiment used a fitness function designed to take into consideration the information of the infrared sensors to reward or punish the robots. This function rewards the robot when the command is *FF* or *FM* and there are no obstacles around the robot. It also rewards it if the robot detects an obstacle in one side and turns to the other side. The robot is punished if it turns towards the obstacle. This algorithm does not punish the robot for colliding, only for making a bad manoeuvre. Table 8.2 presents a summary of the settings for this experiment. The selected fitness function for this test was:

1- Start with 4096 points;

2- Reward: increase fitness by 10 points every 1 second if (All Sensors=0 or Left=Right) and command=FF, FM;

3- Reward: increase fitness by 10 points if Left > Right and command=TRS1, TRS2, TRL;

4- Reward: increase fitness by 10 points if Left < Right and command=TLS1, TLS2, TLL;

5- Punishment: decrease fitness by 10 points if Left < Right and command=TRS1, TRS2, TRL;

6- Punishment: decrease fitness by 10 points if Left > Right and command=TLS1, TLS2, TLL;

- **Results**

This experiment is described in three charts: Figure 8.2 shows the fitness values of the six robots; Figure 8.3 shows the fitness of the best robot in the generation and the average fitness of the population; and Figure 8.4 shows the average fitness of each robot in the previous six generations that is used to select the best robot according to the inheritance selection strategy.

Table 8.2 – Summary of the Experimental Settings

Parameter	Definition
Fitness Function:	+10 points every 1s if all Sensors=0 or *Left*=*Right* and command=*FF* or *FM*; +10 points if *Left* > *Right* and command=*TRS1*, *TRS2*, *TRL* or *Left* < *Right* and command=*TLS1*, *TLS2*, *TLL* –10 points if *Left* < *Right* and command=*TRS1*, *TRS2*, *TRL* or *Left* > *Right* and command=*TLS1*, *TLS2*, *TLL*
Initial Fitness Value:	4096 points
Maximum Fitness Value:	4696 points
Generation Time:	60 seconds
Mutation Rate:	Only 1 command (3 bits) is modified in the chromosome each time
Back Mutation:	Disabled
Selection Strategy:	Inheritance (6 generations)
Crossover Strategy:	Asexual
Frequency of the Attacks:	Predation Disabled
Speed Levels:	Fast and Medium with first 5 bits fixed and the others evolving
Sensors Enable:	All sensors under evolutionary control
Navigation Controller:	Evolving Black box with 256 bytes
Initial Sensor Configuration:	Random
Initial Speed Configuration:	Random
Initial Controller Configuration:	Random.
Software:	EXP15.CPP and GRAF15.CPP (listed in Appendix A)

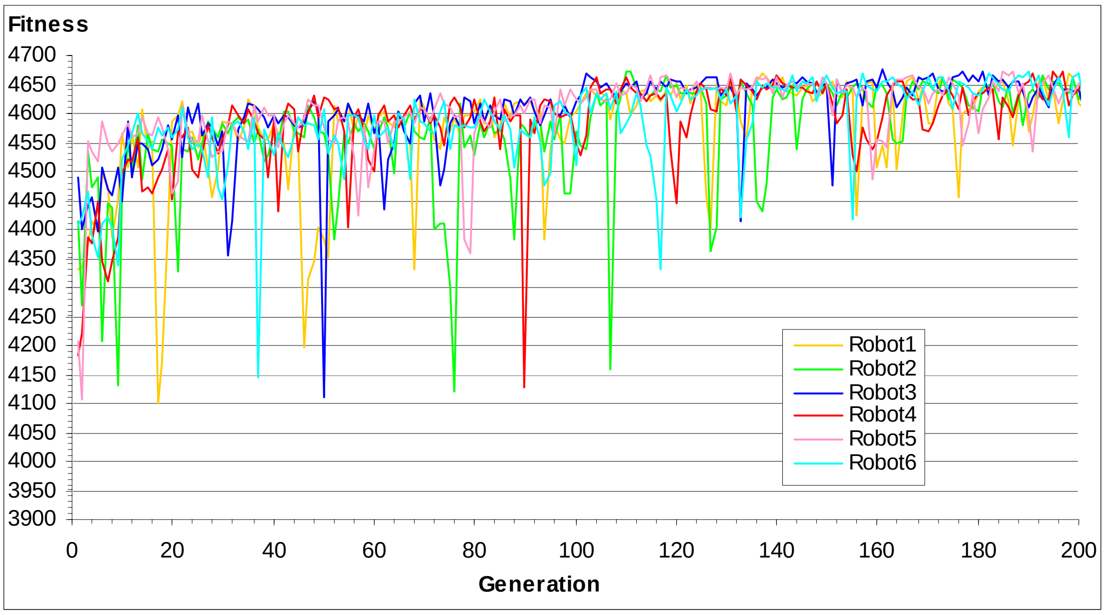


Figure 8.2 – ***Experiment E1***: Evolution of the <u>*black box* controller and morphology</u> using <u>inheritance selection</u> and a <u>biasing fitness function</u> in an environment of <u>medium complexity</u>, with <u>sexual reproduction</u>, <u>command manipulation</u>, <u>back-mutation prevention</u>, and <u>generation time of 60s</u>. Only one command (3 bits) is allowed to mutate each time. Here, *Robotn* is the fitness of Robot *n* in the generation.

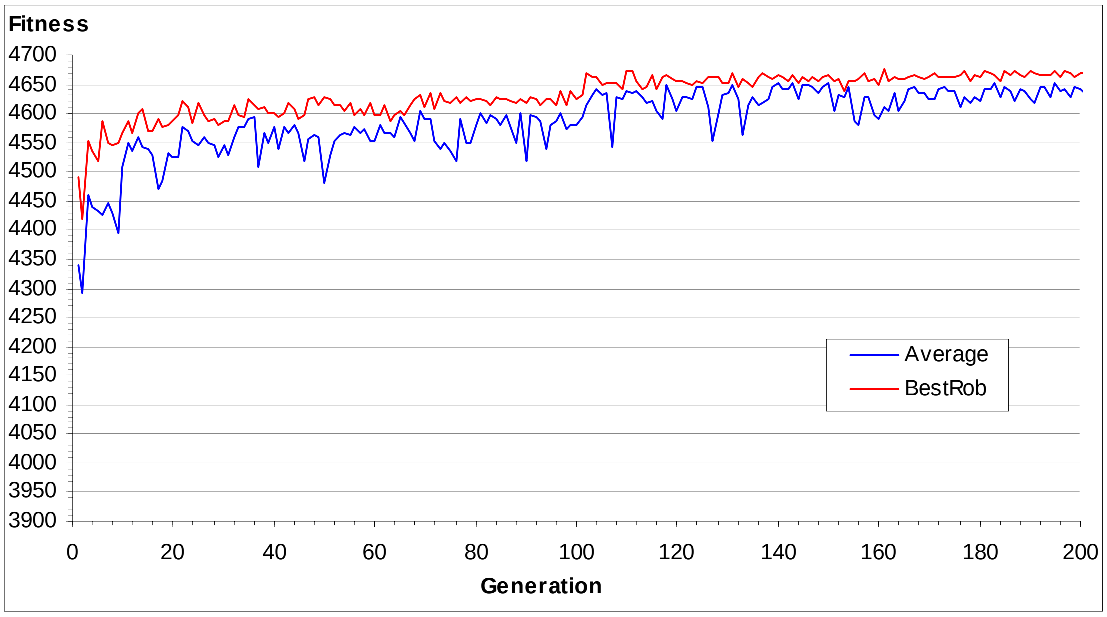


Figure 8.3 – ***Summary of Experiment E1***: Summary showing the average fitness (*Average*) of all robots and the fitness of the best robot (*BestRob*) in the generation.

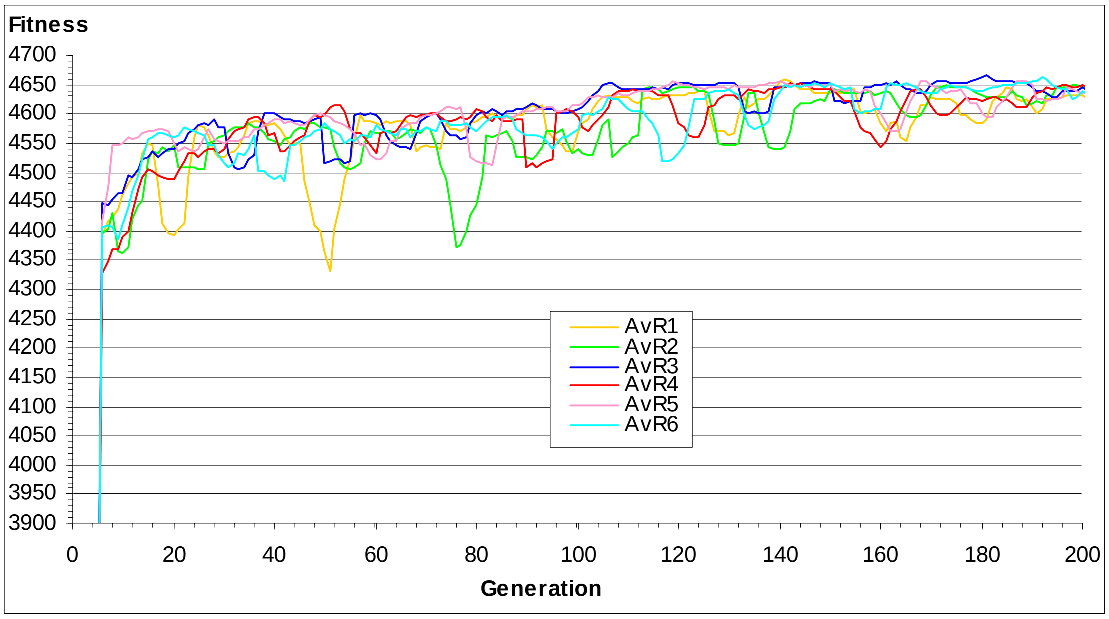


Figure 8.4 – ***Details of Experiment E1***: More details of the experiment showing the average fitness of each robot in the last six generations. Here, *AvRn* is the average fitness scored by robot *n* in the current and the previous five generations.

8.1.1 Discussion of the Experimental Results

The first chart shown in Figure 8.2 presents the behaviour of the six evolving robots in 200 generations of 60 seconds. They evolved from a random controller with random speed levels and sensor configuration to a group of competing efficient solutions containing four or five sensors (*S1,S3,S4,S6,S8*; *S3,S4,S6,S8*; *S2,S3,S4,S7*), and a controller that learned how to deal with them. Robot 5 started with four sensors enabled (*S3,S4,S6,S8*), and was an efficient combination that was soon transferred to the other robots in the first 20 generations. Next, their controllers had to adapt to work with small variants from this configuration. This took more than 80 generations.

The speed levels started at random values ("1111110011"), but were controlled by the evolutionary system and soon both levels (*Medium* and *Fast*) dropped to 17. This was an emergent strategy used by evolution that made the robots travel slower in the beginning of the process, when their controllers were not well-developed to deal with all situations. Level *Medium* stayed at 17, varying very little during the process, but *Fast* was progressively increased after generation 100, when the robots were better-adapted to deal with a faster speed. Level *Fast* oscillated between 20 and 26 in the last 35 generations, averaging 23. The resultant configuration for the bits that define the robot speed was "1111101100", which kept level *Medium* in the lower speed level (*Medium* = 17), but increased level *Fast* to 23. The first five bits were fixed at "1" to prevent the robots from going slower than 17, so evolution could control only the last five bits: "11111xxxxx". The *Medium* level is calculated by counting the number of ones in the first six of these configuration bits and *Fast* is calculated by counting the ones in all of the bits and then converting the count according to Table 4.4.

The chart in Figure 8.3 shows that the population gradually converged to a small set of efficient configurations combining morphology and control that could drive the robots practically without colliding. The average fitness of the population oscillated in the beginning of the process, but in the end it got very close to the curve of the fitness of the best robot, since with asexual reproduction the population is a copy of the best robot that suffered only one mutation. Therefore, all the robots in the end of the evolutionary experiment had very similar behaviours, or at least the ability to produce similar performances. The small differences that still persisted were a consequence of the influence of noise and the interactions between the robots. As the generations passed, there were fewer poorly-adapted robots crashing into the good ones, so the average performance increased.

The chart shown in Figure 8.4 represents the average fitness of each robot in the previous six generations and illustrates how the robots were selected to breed. The increase in performance that happened around generation 105 was initiated by Robot 5, which learned how to manoeuvre properly with four sensors enabled. It soon taught Robot 3, which began to dominate the population for 30 generations, since it had the best combination of controller and morphology.

The bits corresponding to the commands of the black box controller mutate in units of three bits, corresponding to each command. However, the bits corresponding to the sensor configurations and the speed of the robot mutate bit-by-bit, with only one bit allowed to be changed each time. An interesting event was observed in this experiment: after the population started to converge in the last 30 generations of the process, the sensor configurations still varied into a small set of solutions with four or five sensors enabled, but the controller configuration did not change much. In fact, it was able to perform the same manoeuvres regardless of being attached to four or five sensors. This helped to preserve a good performance while the sensor and speed configurations kept changing, since the controller did not need to adapt again after the robot morphology changed.

This experiment succeeded in showing that the embedded evolutionary system was able to evolve a population of real robots controlled by an unstructured black box and having their morphology manipulated by evolution. Together with the data presented in Appendix C, it provided the first experimental proof of the embedded evolution concept where both an unstructured controller and morphology were evolved. The system was able to find efficient solutions in a considerably large search space of 1.987×10^{234} possibilities (eight possible commands to occupy the 256 possible outputs of the black box controller make: $8^{256} \times 2^{8}$ sensor configurations $\times$ 5 possible speed levels).

In this experiment, the final version of the evolving controller, obtained after 200 generations, did not behave exactly as the preconceived hand-designed controller when connected to five sensors (*S1*, *S3*, *S4*, *S6*, and *S8*). It was, nevertheless, able to manoeuvre the robots without colliding and produced a high performance. The system did not converge to an optimal solution, because it was not necessary, since many configurations obtained after 200 generations were able to produce the higher performance.

8.2 Evolving with a Neural Network Controller

This experiment shows for the first time an embedded evolutionary system evolving a structured neural network controller together with the morphology of the robot. Both the sensor configuration and the speed of the motors are under evolutionary control. This experiment makes use of the most efficient mutation rate and the selection and reproduction strategies that were developed in simulation. It also incorporates the developed strategy of predation to increase the performance of the system as it was demonstrated in Section 7.7.

In this experiment, the robots were evolved in the environment of medium complexity presented in Figure 8.1. The sensor configuration and motor speed are controlled by evolution in the same way as explained before in Section 8.1. The main difference between this experiment and the previous one is the use of a structured neural network to implement the navigation control circuit.

- **Aim of the Experiment**

The aim of this experiment is to provide the first experimental proof of an embedded evolutionary system that evolves a structured neural network controller and the morphology of the robot. The evolutionary system is able to manipulate the 16 bits that control the sensor configuration, plus five of the ten bits that control the robot speed levels, and the 112 bits that define the contents of the neurons of the neural network. The total number of bits controlled by evolution is 133 plus 5 fixed ones. The neural network can produce four different commands and is connected to seven sensors (*S5* is controlled by evolution, but is not connected to the neural network, so it is irrelevant if it is enabled or not). The number of possible genotypes for the controller alone is $2^{112} = 5.19\times10^{33}$. However, because of the generalisation ability of the neural network, many of these genotypes produce the same phenotype.

- **Experimental Setting**

The neural network architecture was explained before in Section 4.1.1 and its configuration is the same shown in Figure 6.6, which was described in Section 6.2. It has four groups of neurons (discriminators) with seven 2-input neurons, each neuron containing four bits. The only difference is that instead of the command *Turn Right Short2* (*TRS2*), the neuron group 4 is now representing the class of the command *Front Medium* (*FM*). Therefore, the neural network controller provides four commands to drive the motors: *Front Fast* (*FF*); *Turn Left Short1* (*TLS1*); *Turn Right Short1* (*TRS1*); and *Front Medium* (*FM*). The commands are encoded by only two bits: *FF* – "00"; *TLS1* – "01"; *TLS2* – "10"; and *FM* – "11". These commands were explained before in Sections 6.2 and 8.1. Five of the ten bits in the chromosome that define the speed levels of the robot *Fast* and *Medium*, activated by the commands *FF* and *FM*, are again under evolutionary control, as explained before in Section 8.1. The first five bits that define the speed levels in the chromosome are fixed to "1" to prevent the robots from moving too slowly.

The neural network controller was evolved without the strategy that prevented back mutation from occurring and by manipulating the bits in the chromosome, not the commands. This was necessary because the neural network does not allow evolution to manipulate directly the commands. An *inheritance selection* that calculates the average fitness of the robots in the last six generations was adopted in this experiment. This strategy also protected up to two robots that score more than the fitness of the best robot, as explained in Sections 6.5 and 8.1. This experiment uses *sexual reproduction* as the crossover operator, where the robot with the best average is selected to breed by combining its chromosome with the ones of the robots that are not protected in the current generation. The modified selection strategy is described below:

- *The score used to select the robot is the average of the robot fitness in the last six generations (i.e., inheriting the scores of its previous five generations). The robot with the best average survives, but does not breed with up to two robots with the fitness in the present generation higher than its own fitness. The other robots combine randomly their chromosomes with the one of the best robot, and then suffer mutation.*

The mutation strategy generates a random number *r* between zero and 100 for each one of the 133 bits in the chromosome and flips it if *r* is smaller than the mutation rate. This experiment uses a very simple fitness function in order to prevent biasing evolution towards a pre-conceived solution. Rule 3 punishes the robots that keep turning for more than 15 seconds, encouraging them to move forward. The selected fitness function for this experiment is:

1- Start with 4096 points;

2- Reward: increase fitness by 10 points every 1 second;

3- Punishment: decrease fitness by 30 points for every time command is not FF or FM for more than 15 seconds;

4- Punishment: decrease fitness by 10 points for every collision if command = FF or FM.

In this experiment, the robot population suffered regular attacks (every ten generations) of a "predator" that selected the robot with the smallest average fitness in the specified generation and substituted it by a random one. Table 8.3 presents a summary of the settings for this experiment.

Table 8.3 – Summary of the Experimental Settings

Parameter	**Definition**
Fitness Function:	+10 points every 1s –30 points for turning for more than 15s –10 points for colliding when command=*FF*, *FM*
Initial Fitness Value:	4096 points
Maximum Fitness Value:	4696 points
Generation Time:	60 seconds
Mutation Rate:	1%
Back Mutation:	Enabled
Selection Strategy:	Inheritance (6 generations)
Crossover Strategy:	Sexual
Frequency of the Attacks:	Every 10 generations
Speed Levels:	Fast and Medium with first 5 bits fixed and the others evolving
Sensors Enable:	All sensors under evolutionary control (*S5* is not connected)
Navigation Controller:	Evolving Neural Network (m=4, n=7, neuron size=4)
Initial Sensor Configuration:	Random
Initial Speed Configuration:	Random
Initial Controller Configuration:	Random.
Software:	EXP16.CPP and GRAF16.CPP (listed in Appendix A)

- **Results**

As in the previous experiment shown in Section 8.1, the evolutionary experiment here is also described in three charts: Figure 8.5 shows the fitness values of the six evolving robots; Figure 8.6 shows the fitness of the best robot in the generation and the average fitness of the population; and Figure 8.7 shows the average fitness of each robot in the previous six generations.

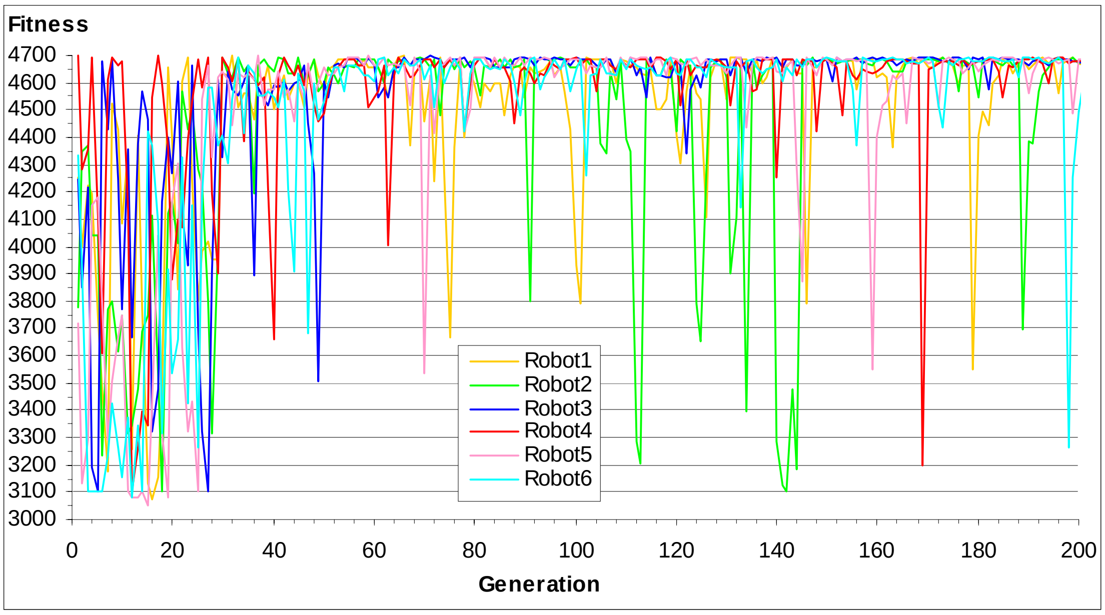


Figure 8.5 – ***Experiment E2***: Evolution of the neural controller and morphology using inheritance selection and a simple fitness function in an environment of medium complexity, with sexual reproduction, predation, mutation rate of 1%, and generation time of 60s. Here, *Robotn* is the fitness of Robot *n* in the generation.

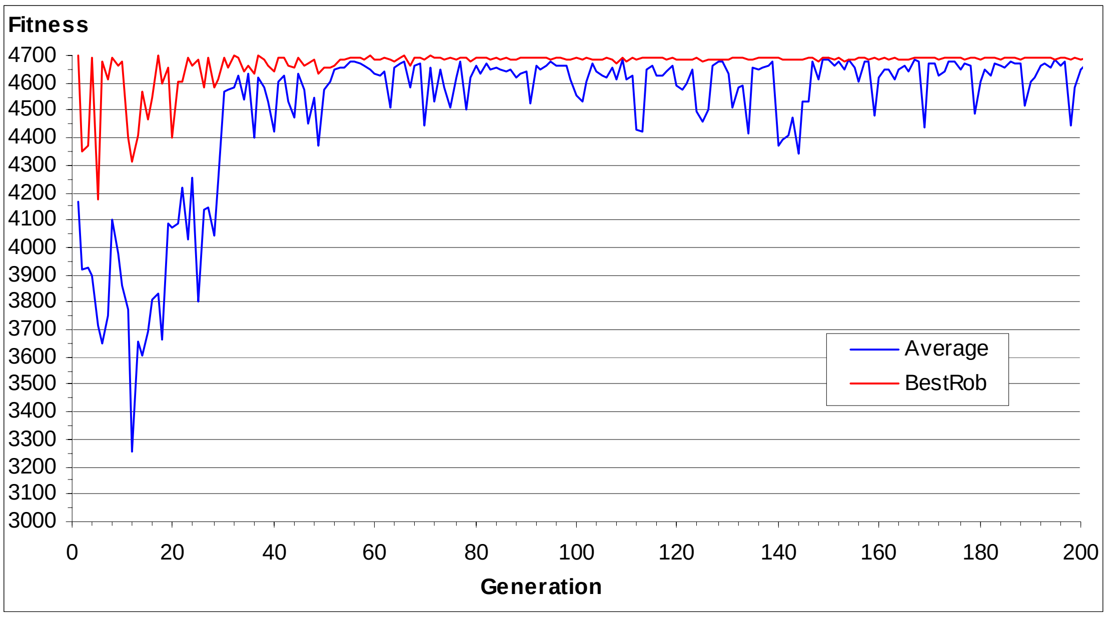


Figure 8.6 – ***Summary of Experiment E2***: Summary of the experiment showing the average fitness (*Average*) of all robots and the fitness of the best robot (*BestRob*) in the generation.

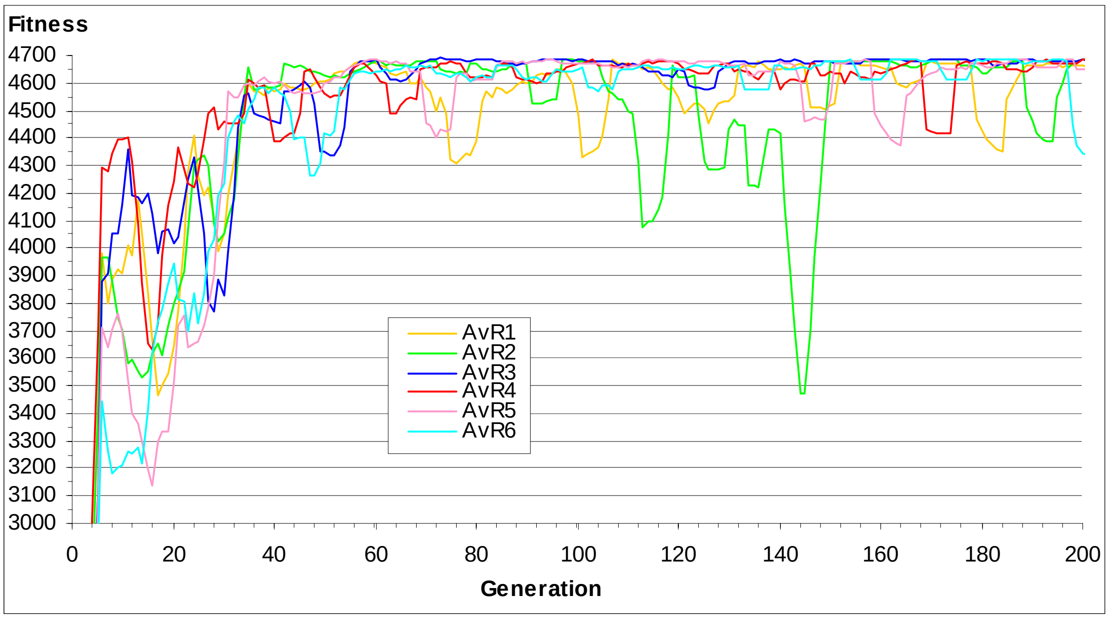


Figure 8.7 – *Details of Experiment E2*: More details of the experiment showing the average fitness of each robot in the last six generations. Here, *AvRn* is the average fitness scored by robot *n* in the current and the previous five generations.

8.2.1 Discussion of the Experimental Results

Figure 8.5 shows a very distinct behaviour, since this experiment used a non-biasing fitness function, many different solutions were produced and evaluated by evolution. An interesting point is that maximum fitness was obtained from the first generation. These happened because some robots produced in the first generations developed a kind of wall following behaviour, where they tried to stay close to the walls or obstacles, but keeping a safe distance, so they did not collide with them. This could produce maximum fitness in some generations, but as it can be observed in the chart, one robot could have very distinct performances with the same controller from one generation to the other. They could avoid colliding into the walls, but suffered collision from other robots, so that in some generations their performance dropped substantially. This configuration nevertheless spread quickly through the population and a particular event started to happen when two robots with this configuration started to walk around each other, each considering its fellow robot a wall to be followed. They kept spinning around, crashing upon everything in their way. This was responsible for the drop in performance near generation 13, which can be better observed in Figure 8.6. The population ended up converging to this unstable solution, a local optima, and could reach a better configuration only because of a predator attack that brought in new genetic material.

The third attack of the predator, in generation 30, resulted in a robot that could for the first time use one sensor in the front (*S2*) and produced a more steady performance. This can be observed in Figure 8.7, where the average of the fitness of each robot in the last six generations started to improve. This happened because, although this configuration did not produce such high scores as the previous wall-following behaviour, its average result was more consistent, and the average fitness of the population increased considerably between generations 30 and 40.

By generation 42, Robot 2, which was able to use properly *S2* and *S7*, had been selected as the best robot for six generations and started to improve the others. From successive combinations with Robot 2, Robot 5 learned one more sensor, mastering *S2*, *S4*, and *S7*, a very powerful combination. From generation 60 on, all robots learned to manoeuvre properly according to at least three sensors, one in the front, and two lateral ones. Robot 3, from generation 70, dominated a population where a small set of efficient solutions competed to be selected as the best robot.

In this experiment, both speed levels Fast and Medium varied from 17 to 20 through the whole experiment, indicating that a less biasing fitness function tends to be more careful, preserving lower, but safer, speed levels. The system oscillated considerably in the first 30 generations and convergence was only possible because of the inheritance selection that calculated the average fitness of the previous six generations for each robot. It can be seen in Figure 8.7 that in spite of all oscillations in the performances of the robots in Figure 8.5, the average fitness of the robots in six generations shown in Figure 8.7 provided a better way of indicating the real capacity of the robots.

This experiment illustrated the power of the developed predation strategy in providing more diversity when the population was trapped in a local optima. The new genetic material it supplied in the first thirty generations was essential to allow the population to explore more widely the fitness landscape. The disadvantage of this strategy is that it never allowed the average of the population fitness to reach the maximum score, since a random robot was introduced every ten generations, causing a drop in the performance of the population that can be easily identified in Figure 8.6.

The developed evolutionary system succeeded in evolving the real robots, initialised with random controllers and morphologies, reaching efficient solutions after 200 generations of 60 seconds. This experiment, together with the data presented in Appendix C (Section C.3.1) provided the first experimental proof of an embedded evolutionary system that was able to evolve a structured neural network controller together with the morphology of the robots in real time in the real world. It produced a satisfactory collision-free behaviour after 200 generations.

This experiment closes the last experimental chapter of this thesis. The experiments reported in this work analysed of the developed evolutionary strategies and their configuration parameters. These experiments demonstrated that the developed evolutionary system is a very powerful approach to developing a self-adapting distributed robot control system that enabled collision-free navigation behaviour in a population of six autonomous mobile robots.

9 CONCLUSION

This final chapter shows how the most relevant aspects of the developed work presented in this thesis can contribute to the state-of-the-art of evolutionary robotics and artificial intelligence with important ideas, novel strategies, and inspiring discussions. This chapter also reviews whether the obtained results achieved the proposed objectives. It highlights the most relevant conclusions arising from the experiments carried out in the specially built test environment and provides guidelines for the future exploration of the developed techniques.

This research work has been concerned with the development of a real-time high-performance distributed evolutionary system embedded in a population of six autonomous mobile robots, to be applied in producing collision-free navigation. To achieve this goal, one objective has been to develop suitable genetic operators and methods of embedding them into the robots. Furthermore, a robotic hardware system has been designed that allows the developed strategies to be tested in real time using the physical world as its best model [Bro91b].

Chapters 2 and 3 presented an overview of recent work in the area of evolutionary robotics together with a discussion of their limitations and drawbacks, which led to breaches in the research field that were explored by this work. These chapters showed that most of the work with evolutionary robotics has been done in simulation and inspired the pursuit of a challenging goal: providing the means of implementing a physical evolutionary system, where the population physically exists and artificially breeds, combining their configuration material to produce the new generations.

To achieve this goal, because of the dimensionality of the problem, the project specification had to set limits to the applicability of the research and select a non-complex task-behaviour, reducing the system complexity and consequently the application of the obtained results. Nevertheless, a new concept emerged and was experimentally proven for the first time: *embedded evolutionary systems*. It was also the first time that an embedded evolutionary system was put to test in co-evolving the robot controller and morphology at the same time. Therefore, the major contribution of this work is in developing basic research on embedded evolution, which is a new concept, proposing novel techniques that enabled an evolutionary system to be applied to a group of real autonomous mobile robots. This forms the basis of a technique that has very important industrial applications. Evolutionary robotics is a research area in its infancy [Har97b] [Bro00] [Pol00] [Tho00], which tests whether all newborn AI philosophies can grow up into the real world, and scale up with increasing complexity.

9.1 Conclusions Arising From the Experimental Results

It was noticed when comparing the results obtained with simulation (Chapter 7) to the ones with real evolution (Chapter 8) that the fitness of the robots improved faster in the experiments run in a real environment than the ones run in simulation, in terms of the number of generations, despite all the noise and interaction amongst the robots. The reason for this is that in a real environment the robots are evaluated for only a few minutes and are subjected to fewer situations that test their ability to manoeuvre properly. In the developed simulator, however, the response of the navigation control circuit is always exhaustively tested against all possible situations that the robot can face. This produced lower fitness values than the evaluations in real environments. Sometimes, for example, as demonstrated in Section 8.2, a randomly initialised population can produce the maximum fitness in the first generation of the experiment, if one of the robots is “lucky” enough to face only the situations it can deal with during the generation.

The importance of a long lifetime, the duration of a generation, was observed: improper solutions may take, by chance, the place of more robust individuals if they have been evaluated only for a short period. That happens because an unfit robot can be “lucky” enough to start its life in an easy place in the workspace or in a position where it can deal with the obstacles even when only part of its controller and sensors are well adjusted. In a similar way, a fit robot may start in a very difficult and crowded position, where its chances of collisions with other robots are greater. It may also happen, by chance, that other unfit robots end up colliding many times with a fit one. A longer lifetime will give more opportunities to differentiate between the fit and the unfit robots. The disadvantage of a longer lifetime is that it will take longer to evolve the population and achieve an efficient solution. However, the benefits generated by a better evaluation can speed up the evolutionary process and make the overall result faster than one using a shorter lifetime.

The previous work on the evolution of collision-free navigation was reported mostly by Floreano and Mondada [Flo96a]. Their work employed physical Khepera robots tethered to external computation provided by a workstation. They could only achieve the desired behaviour after three days of non-stop evolution. During the development of this work, Watson *et al.* also proposed an embedded evolutionary system that used eight real robots [Wat99a]. Their robots were simpler than the ones used in this work and they evolved only their navigation control circuits to drive the robots towards a light source, a task of similar complexity to the chosen collision-free behaviour. They achieved an efficient solution (compared to a hand-designed controller) in less than two hours of evolution. After applying the understanding and insights produced in the real experiments in Chapter 6 and in simulation in Chapter 7, the developed evolutionary system could evolve the controller and morphology of the robots and achieve collision-free behaviour in less than two hours. This good result could only be achieved after refining the configuration of the system, which was taking days to evolve with the wrong parameters.

Monitoring fitness from inside the robots is necessary in a distributed embedded controller, but it generates problems on how to evaluate behaviours. The monitor computer is just recording the evolutionary data and cannot be used to distinguish between fit and unfit behaviours by evaluating distances from goals, contact with other robots and obstacles, or the position of the robots. The fitness function needed to be implemented from the point of view of the robot, using its own sensors (contact and infrared proximity sensors). The fitness function has to be kept as simple as possible, to avoid biasing evolution to produce a pre-conceived solution.

The on-board fitness evaluation function required the use of a microprocessor, since it is the simplest way of implementing such functionality. The presence of a microprocessor also permitted high-level modularity of the controller and facilitated its implementation using a neural network. Although biasing evolution, the high-level modularity of the neural network permitted post-evolution analysis of the obtained result, a better user interface, and particular fitness evaluation of the modules. Moreover, and more importantly, it permitted the implementation of a considerably smaller chromosome to contain the configuration of the controller. An unstructured controller can provide much more variety and has more chances of producing an unforeseen result, but it is less organised and needs a longer configuration bit string. This increased drastically the search space and slowed down the evolutionary process, reducing system performance.

Real time evolution of an embedded hardware means that the evolutionary controller must cope with the robot limited processing speed and small memory size. Consequently, Boolean neural networks offered small, high-speed solutions. To minimise the execution time of the controller, making it smaller and faster, the evolving neural network, for every generation, can be converted to a look-up table and be implemented into the robot RAM memory. This is achieved by taking the resulting chromosome, after the mating process, and using it to configure the neural network. Then, the neural network is simulated with every possible input and a corresponding output table is written. This output table is then stored into the robot RAM memory and becomes the current controller that drives the robot in the current generation. For each iteration, the sensor readings are converted into an address to the memory and the returned data is converted into a command to drive the robot. In this way, the operation of the complete neural network that would take hundreds of lines of code to be implemented can be executed with one single instruction that reads the command byte in the memory. It can make the controller work hundreds of times faster. This strategy was used in some experiments and was very efficient.

Mutation in embedded evolution has a more important effect than in simulated evolution, since the population in real systems is considerably smaller than in simulated evolution. Therefore, crossover is limited by the small variety of genetic material that it can combine to produce better individuals and, without mutation, the population usually converge in a few generations and stop evolving. However, a high mutation rate does not help to evolve faster. It did not prove to be a good strategy. Although it may help in the beginning of the process to bring more variety into such a small population (e.g., six robots), it slows down the process after the initial genetic material is combined and the population converges to individuals that have more than half of their chromosome with the right genes. Therefore, in the long term, it produces a very distinct population, where the fittest individual is considerably distant from the average fitness of the population. Generally, only the best robot that survives to the next generation and does not mutate is well adapted to the environment. A relatively high mutation can still be used to accelerate the initial search for a solution to a problem that, once an acceptable fitness is achieved, will be replicated into other robots that will work independently. A low mutation rate makes the whole evolutionary experiment much faster. It produces a population with small variation where the average fitness is almost as high as the fitness of the fittest individual. A low mutation rate is very important if the robots are working as a team (e.g., combining their strength to push heavy objects or even playing soccer) and a bad performance can severely affect the overall operation.

Evolution produces a solution that is good enough to solve the problem, but not necessarily the expected one. It all depends on how the fitness evaluation function is specified. It is important to use the simplest function possible to prevent biasing evolution, limiting the possible solutions. Working from bottom-up, evolution generates what the fitness function rewards. It can get very close to the solution achieved by working out the problem from top-down, but probably never there. The main problem is how to specify the reward and punishment functions, so that evolution can deal with the complexity of the task. The best way encountered was by "trial and error": rewarding particular responses and watching to see what it can produce. Then, by gradually adding small sub-functions and observing the overall result, the designer can keep the ones that improved the result. Usually, the longer the process, the better the solution. With a poor fitness function, fitness evaluation is not enough to determine a good solution (e.g., fitness can be high, but the solution can be far from the expected one). The final result is probably more complex and can take longer to achieve than the top-down design of a controller by hand. Nevertheless, what can be achieved by applying an open-ended evolution is a self-adaptable controller, which can continuously modify its internal configuration to coupe with a variable environment. This is very difficult to be obtained with a traditional design, since all the variability of the environment has to be foreseen.

The experiments in Chapters 6, 7, and 8 demonstrated that evolution can work in two ways: biased or non-biased. Biased evolution appears when the fitness function rewards not only *what* the robot should do, but also *how* it is doing it. It happens when sub-functions are rewarding the robots that behave in a similar way to an expected solution foreseen in the problem (e.g., when they avoid the obstacles by turning in a specific way). Non-biased evolution is the one that only rewards when the robot does *what* it should do (or punishes when it does what it should not do). Non-biased evolution is an interesting alternative to conventional top-down design, because it can produce unique, unexpected solutions where the robots can actually tell the designer how they should be designed (e.g., showing the best positions to place sensors, their best range and angle, the proper speed the robot should navigate, etc). It was observed that biased evolution is not yet a better alternative to manual design, because if the designer foresees a good solution for the problem, why cannot it be implemented right away? This approach can, though, be used to program more efficient controllers (such as look up tables or neural networks) to behave similarly to a complex solution developed in a modular top-down design. Moreover, with the inclusion of extra sub-functions, a biased evolution can produce a continuously adaptable system able to deal with variations of the environment.

As the number and position of the sensors and the speed levels of the motors are under evolutionary control, not only the control circuit is produced, but also the physical characteristics of the robots can change into different configurations according to the complexity of the environment. In addition, the designer can fix the number of sensors, for example, and let evolution decide where they should be placed. The most successful configurations according to the sensor positions that were observed in the experiments are:

a) Configuration 1 – One sensor in the front;
b) Configuration 2 – Two sensors, one in the front and a lateral one;
c) Configuration 3 – Three sensors, one in the front and one in each side of the robot.

It was observed that, in a simple environment containing few obstacles, all three configurations coexist "peacefully", because it does not present enough selection pressure and the fitness of all three configurations are roughly the same after a small lifetime (evaluation time). The longer the lifetime though, the greater the number of opportunities to distinguish between a more efficient configuration and an ordinary one and the robots with more sensors positioned in the right places are more likely to succeed. When more obstacles are added and the environment becomes more complex, the competition is tougher and the configurations with more resources gradually lead the less-adapted ones to "extinction" from the population.

Late in the evolutionary experiment, when the robots are well-adapted to the environment and the task, refining the solutions takes progressively more time. The better a solution is, the slower it takes to refine it. Many crossover strategies and mutation rates were investigated to speed up the final refining process. The general strategy was to choose the best robot and breed it with all the others. The best robot then survives to the next generation and all the other robots are replaced by their offspring. Another solution developed for the mating phase is a strategy that does not cross over the genes at all. This *asexual reproduction* technique consists in choosing the best robot, allowing it to survive to the next generation, and replacing all the other robots with a copy of its chromosome plus a small degree of mutation. This technique, although slower in the beginning of the process, was the fastest overall evolution.

Another technique was developed in order to provide new genetic material for evolution to play with: every ten generations, the worst robot is selected and destroyed ("killed" by a virtual "predator"), and has its chromosome replaced by a randomly generated configuration. It models the natural world, where from time to time a predator calls taking out weak members of the pack, preventing them spreading its genes and opening space for new individuals to migrate, bringing more diversity to the group. This was called "the *predation* strategy" and was shown to improve the overall performance of the evolutionary process. After an "attack", enough time must be given to allow the population to recover and stabilise again. Otherwise, the "killed" robot that was replaced by a random configuration would probably be the worst one again in the next attack, and would be repeatedly destroyed. Therefore, the number of generations between the attacks must be carefully chosen according to the complexity of the evolving controller. The predation strategy brings in vital diversity that can prevent the population from being stuck in local optima.

Determining when the desired behaviour has been achieved on physical robots is difficult and relies on a quantitative analysis based on human judgement and experience. However, human observers can apply reasonable phenomenological descriptions of the performance of the robots [Mat97c]. Typically, statistical analysis is not significant as insufficient data are available. Because of the uncertainty and variability of physical experiments, an average performance is difficult to establish as trials vary significantly. Therefore, the conclusions provided in this work should be taken as guidelines to future experimentation and applications of the developed techniques, instead of an exhaustive study of all possible behaviours that such an evolutionary system can produce. Although it was proven that the developed evolutionary system can produce efficient solutions and unforeseen results, it cannot be affirmed that it will always behave as described in the experiments. The analysis provided is only descriptive [Men92].

9.2 Principal Achievements

The main achievements of this work is the proposal of the new concept of embedded evolution and the implementation of a novel and unique evolutionary system that is decentralised, distributed, and fully embedded in a population of real autonomous mobile robots, and is able to produce collision-free navigation by evolving not only the robot controller, but also its morphology. It provided a genetic system where the population exists in a real environment, where they exchange genetic material and reconfigure themselves as new individuals to form the next generations. The evolutionary algorithm is distributed amongst and embedded within the robot population. Most of the work to date on evolutionary robotics has been done by serially evaluating candidate controllers through simulation or on a single robot, so the importance of this research is in providing the means of running genetic evolutions in parallel, in a real physical platform. As new robots cannot be spontaneously created, the offspring are implemented by reconfiguring other robots of the same population. The adaptive mechanism is distributed in the population and is carried out autonomously.

This work provided the first experimental proofs of an embedded evolutionary system that can manipulate the robot control circuit and morphology. This system was inspired by biology and can be applied to address questions in the natural world and in artificial life, since it offers opportunities for conducting experiments that are extremely complicated in traditional biology or not feasible at all, such as the characterisation of the “evolvability” of ecological systems, modelling isolated populations, or ecological monitoring tools. Hence, most of the strategies developed to allow the implementation of such a system into real robots are also unique and relevant to the scientific community in biology and electronics:

- A real physical platform was developed where the population physically exists, which allows genetic techniques to be tested in a real environment, and distinct hardware and software were developed to allow the integration and distribution of reproduction and other genetic operators into the autonomous mobile robots;

- Guidelines were provided to allow the implementation of the developed evolutionary system in different robot platforms, so that the experiments can be repeated by other researchers using their own robots;
- Existing genetic algorithm techniques that had been used only in simulated systems had been adapted and converted to be implemented within real physical robots;
- New genetic operations such as evaluation, selection, and mutation strategies were developed, implemented within the embedded hardware and software of the robots, and evaluated in real-time experiments;
- An original solution was developed to accommodate both the robot navigation controller and the distributed evolutionary system on board the robots; and a unique communication protocol was also developed to allow the establishment of a radio link that permitted the synchronisation and exchange of genetic material during the mating phase of the evolutionary system;
- Different ways of evolving real robots were investigated and relevant data analysis was provided to guide future experiments and the application of the developed techniques;
- The developed *Predation strategy* proved a powerful concept that not only improved the performance of the system but also prevented the population from being stuck in local optima, by bringing in more diversity;
- It was demonstrated that evolution can actually help in the traditional design of robotic platforms, since it can suggest the best features a robot should incorporate to develop specific tasks. Evolution was shown to be efficient in testing and selecting the best sensor configurations, or speed levels for the motors. It can give important clues to the designer of how the morphology of the designed robot should be to achieve maximum performance and economy of energy;
- Finally, this work provided understanding on the implementation of real evolutionary systems and inspiring insights that have great potential of application in the area of automation.

The principal aim of this thesis has two major aspects. First, it has led to an understanding of the applicability of an evolutionary system to evolve a population of physical robots in real time. Second, the work has produced new evolutionary techniques and genetic operators that can have immediate potential value to researchers working in this area. Taken together, these two strands have contributed to an integrated study which, it is hoped, will be of continuing value in the future development of techniques in automation and cybernetics, since the embedded system can be applied to situations where the agents must evolve while deployed "in the field" – an issue not usually approached by evolutionary robotics.

A strategy was developed to allow the direct application of the evolutionary system in typical industrial control problems. For applications in optimisation and solution searching in robotics, it is possible that there will be many experiments with poorly-developed configurations. These can be discarded and their poor performance has no effect. However, in industrial applications, if evolution is set free to "play" with the resources, poorly developed controllers may damage the system or cause dangerous situations. Therefore, this work developed a strategy to apply safely the evolutionary system to industry by pre-training it to behave properly, using a traditional approach as a model. It performs in simulation the initial evolutionary trial-and-error phase, and transfers the result to be refined by evolution in the field. The simulator developed in Section 7.1 can be used to train the black box controller developed in Section 6.4 or any other controller that can be described algorithmically in C language, such as a neural network or fuzzy logic system, to behave according to a traditional approach, such as a PID controller (Proportional-Integral-Derivative controller).

The illustration in Figure 9.1 describes how the developed system can be adapted to be used in industrial control applications. To avoid damaging the controlled device, the evolutionary controller can be pre-trained by applying a biased simulated evolution to train it to behave according to the traditional approach to such control, as it was described in Sections 6.2 and 7.1. After the training phase, the evolving controller can safely actuate on the device and can be evolved by a supervisory evolutionary system to achieve the appropriate settings specified by the user. The fitness can be calculated by comparing the device settings to the current behaviour and giving a proportional score. If something goes wrong, the supervisory system can reconnect the original controller to prevent the device from being damaged until the evolutionary controller reaches a safe level again.

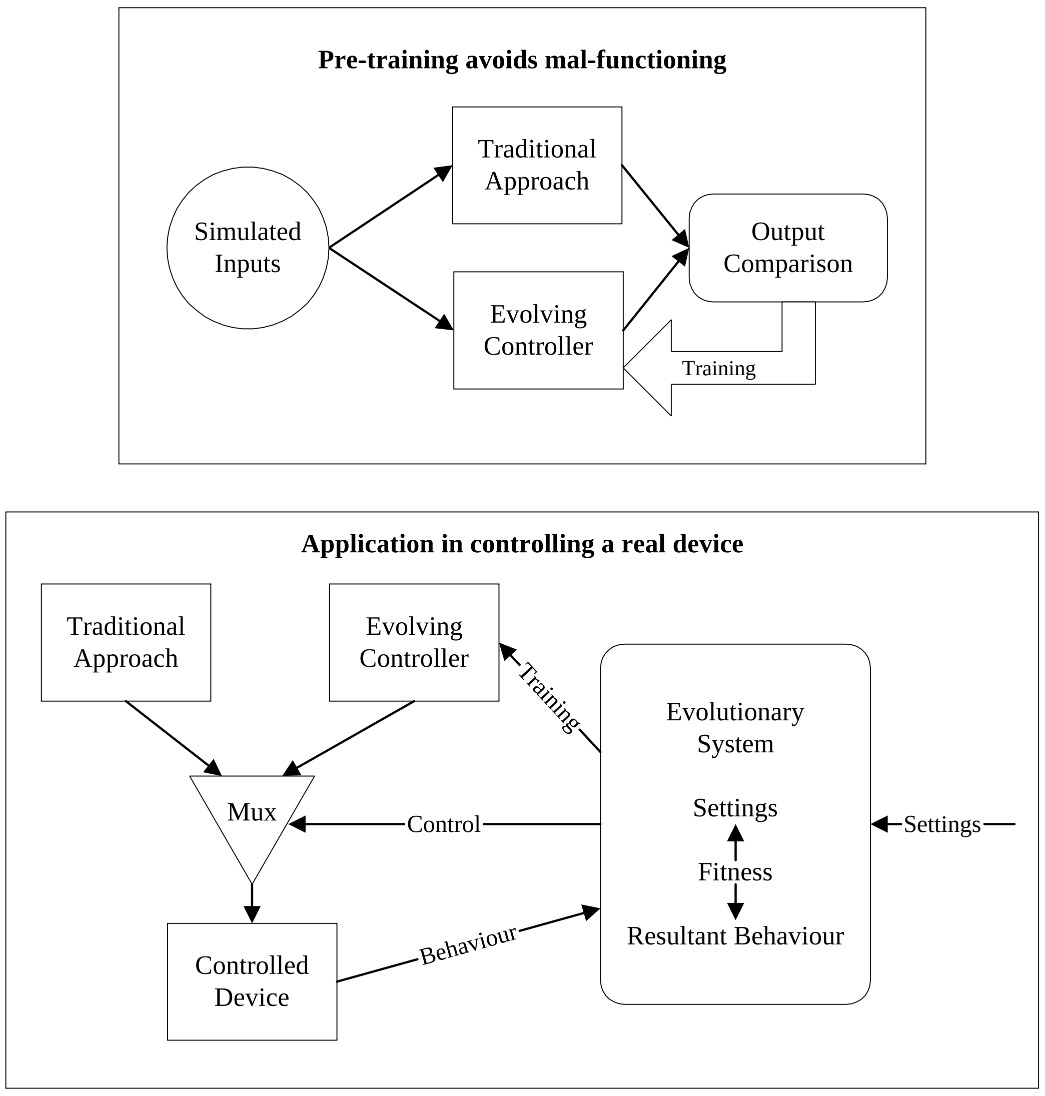


Figure 9.1 – A pre-trained evolutionary controller connected to a real device.

9.3 Suggestions for Future Research

To provide a practical focus to the work described in the thesis the developed evolutionary system has been confined to the task-behaviour of collision-free navigation. However, the methodology described here may be generalized to many other applications, such as object fetching and collection, environment mapping, cleaning and foraging; and some collective tasks as well, such as pushing heavy objects, task sharing, team cooperation and competition.

An investigation of other genetic strategies such as species adaptation genetic algorithms can improve the system performance in terms of accuracy and speed. The implemented techniques have shown good performances, but if new approaches can reduce the necessary time to achieve an efficient behaviour, longer lifetimes can be employed, which can better evaluate the robot ability to perform the task.

The gradual increase of the complexity of the evolved task-behaviours will be a primary subject in future work. A limited robot population is constraining the possible applications of the system. Therefore, more robots will be built, providing more diversity that can be combined into collective tasks where cooperation and competitive behaviours can emerge.

The self-adapting distributed controller developed in this work can be adapted to applications in situations where the designer cannot easily interfere and change the configuration of the robots and also where the conditions of the environment where the robots are going to work are difficult to foresee. Such applications are space exploration, underwater search and rescue, and rescue robots for large-scale disasters such as fire, flood, and earthquakes. The team (population) of robots can be pre-programmed and trained according to a preconceived solution (as demonstrated in the experiments) and can continuously adapt when deployed in the field to variations in the environment conditions, without depending on the designer to modify their configuration. Different teams of robots working in different environments that are distant from each other can learn or adapt to different behaviours, according to their experiences. These isolated populations can still communicate (via the internet, for example) and exchange the learned skills so that a team of robots that has never faced a particular situation or disaster can be taught the proper behaviour to deal with it from the experiences of another team. Such future work will probably have to deal with new arising concepts such as a robotic common knowledge, or culture, or even a robotic civilisation.

References

[Ale66] Aleksander, I., *Self-Adaptive Universal Logic Circuits*. In IEE Electronic Letters, v. 2, pp. 321-322, 1966.

[Ale84] Aleksander, I., Thomas, W. V., and Bowden, P. A., *WISARD: a Radical Step Forward in Image Recognition*. In Sensor Review, v. 4, n. 3, pp. 120-124, 1984.

[Ale90] Aleksander, I. and Morton, H., *An Introduction to Neural Computing*. Chapman and Hall (Ed.), London, UK, ISBN: 0412377802, 240p., 1990.

[Ang94a] Angeline, P. J., Saunders, G., and Pollack, J. B., *An Evolutionary Algorithm That Constructs Recurrent Networks*. In IEEE Transactions on Neural Networks, v. 5, n. 1, ISSN: 1045-9227, pp. 54-65, 1994.

[Ang94b] Angeline, P. J., *Genetic Programming and Emergent Intelligence*. In Advances in Genetic Programming, Chapter 4, Kinnear, K. E. Jr. (Ed.), Publisher: MIT Press, pp. 75-98, 1994.

[Aus88] Austin, J., *Grey Scale N Tuple Processing*. Proceedings of the 4th International Conference on Pattern Recognition (Lecture Notes in Computer Science, V. 301), Cambridge, UK, Kittler, J. (Ed.), Publisher: Springer-Verlag, Berlin, Germany, pp. 110-120, 1988.

[Aus94] Austin, J., *A Review of RAM-Based Neural Networks*. Proceedings of the Fourth International Conference on Microelectronics for Neural Networks and Fuzzy Systems - MICRONEURO94, Publisher: IEEE Computer Society Press, pp. 58-66, 1994.

[Aus98] Austin, J., *RAM-Based Neural Networks, a Short History*. In RAM-Based Neural Networks, Austin, J. (Ed.), York, UK, pp. 3-17, 1998.

[Bac91] Back, T., Hoffmeister, F., and Schwefel, H., *A Survey of Evolution Strategies*. Proceedings of the Fourth International Conference on Genetic Algorithms, Belew, R. and Booker, L. (Eds.), Publisher: Morgan Kaufmann, San Mateo, CA, USA, pp. 2-9, 1991.

[Ban93] Banzhaf, W., *Genetic Programming for Pedestrians*. Proceedings of the 5th International Conference on Genetic Algorithms - ICGA-93, University of Illinois at Urbana Champaign, Forrest, S. (Ed.), Publisher: Morgan Kaufmann, pp. 17-21, 1993.

[Ban94] Banzhaf, W., *Genotype-Phenotype Mapping and Neutral Variation - A Case Study in Genetic Programming*. In Parallel Problem Solving From Nature - PPSN III, Davidor, Y., Schwefel, H. P., and Manner, R. (Eds.), Publisher: Springer Verlag, Berlin, Germany, pp. 322-332, 1994.

[Bar98] Barnett, L., *Ruggedness and Neutrality: The NKP Family of Fitness Landscapes*. Proceedings of the Sixth International Conference on Artificial Life, Adami, C., Belew, R., Kitano, H. et. al. (Eds.), Publisher: MIT Press, pp. 17-27, 1998.

[Bed97] Bedau, M. A., Snyder, E., Brown, C. T., and Packard, N., *A Comparison of Evolutionary Activity in Artificial Evolving Systems and in the Biosphere*. Proceedings of the Fourth European Conference on Artificial Life, Husbands, P. and Harvey, I. (Eds.), Publisher: MIT Press/Bradford Books, pp. 125-134, 1997.

[Bee91] Beer, R., *Toward the Evolution of Dynamical Neural Networks for Minimally Cognitive Behaviour*. Proceedings of the Fourth International Conference on Simulation of Adaptive Behaviour, Maes, P., Mataric, M., Meyer, J. A. et. al. (Eds.), Publisher: MIT Press, Cambridge, MA, USA, pp. 421-429, 1991.

[Bla98] Blair, A., Sklar, E., and Funes, P., *Co-Evolution, Determinism and Robustness*. In Simulated Evolution and Learning - Lecture Notes in Artificial Intelligence 1585, McKay, B. et. al. (Eds.), Publisher: Springer-Verlag, Berlin, Germany, ISSN: 0302-9743, pp. 389-396, 1998.

[Ble59] Bledsoe, W. and Browning, I., *Pattern Recognition and Reading by Machine*. Proceedings of the Eastern Joint Computer Conference, Birmingham, UK, pp. 225-232, 1959.

[Bli96] Blickle, T., *Evolving Compact Solutions in Genetic Programming: A Case Study*. Proceedings of the International Conference on Evolutionary Computation: Parallel Problem Solving From Nature IV, v. 1141 of LNCS, Voigt, H., Ebeling, W., Rechenberg, I. et. al. (Eds.), Publisher: Springer Verlag, Berlin, Germany, pp. 564-573, 1996.

[Bot96] Botelho, S. C., Simoes, E. D. V., Uebel, L. F., and Barone, D. A. C., *High Speed Neural Control for Robot Navigation*. Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Beijing, China, pp. 421-429, 1996.

[Bro00] Brooks, R. A., *From Robot Dreams to Reality*. In Nature, v. 406, n. 6799, ISSN: 0028-0836, pp. 945-947, 2000.

[Bro91a] Brooks, R. A., *Elephants Don't Play Chess*. In Designing Autonomous Agents: Theory and Practice From Biology to Engineering and Back, Publisher: MIT Press, Cambridge, MA, USA, pp. 3-15, 1991.

[Bro91b] Brooks, R. A., *Intelligence Without Reason*. Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, Publisher: Morgan Kauffman, San Mateo, CA, USA, pp. 569-95, 1991.

[Bro92] Brooks, R. A., *Artificial Life and Real Robots*. Proceedings of the First European Conference on Artificial Life: Toward a Practice of Autonomous Systems, Varela, F. J. and Bourgine, P. (Eds.), Publisher: MIT Press / Bradford Books, Cambridge, MA, USA, pp. 3-10, 1992.

[Bul95] Bullock, S. G., *Co-Evolutionary Design: Implications for Evolutionary Robotics*. Report ID: Cognitive Science Research Paper Serial No. CSRP 384, The University of Sussex School of Cognitive and Computing Sciences, Falmer Brighton, BN1 9QH, England, UK, 10p., 1995.

[Cal95] Calabretta, R., Nolfi, S., and Parisi, D., *An Artificial Life Model for Predicting the Tertiary Structure of Unknown Proteins That Emulates the Folding Process*. In Advances in Artificial Life - Lecture Notes in Artificial Intelligence, v. 929, Publisher: Springer-Verlag, Berlin, Germany, pp. 862-875, 1995.

[Cam99] Campbell, N. A., Reece, J. B., and Mitchell, L. A., *Biology*. 1st ed., Publisher: Addison Wesley Longman, California, USA, 1999.

[Cha98] Chavas, J., Corne, C., Horvai, P., Kodjabachian, J., and Meyer, J. A., *Incremental Evolution of Neural Controllers for Robust Obstacle-Avoidance in Khepera*. Proceedings of the First European Workshop on Evolutionary Robotics - EvoRobot'98, Apr. 1998, Paris, France, Husbands, P. and Meyer, J. A. (Eds.), Publisher: Springer Verlag, pp. 227-247, 1998.

[Chi93] Chiaberge, M., Del Corso, D., Gregoretti, F., and Reyneri, L. M., *A Neural Network Chip Using CPWM Modulation*. In New Trends in Neural Computation (From Proceedings of the International Workshop on Artificial Neural Networks - IWANN93, June 1993, Sitges, Spain), Publisher: Springer-Verlag, pp. 420-425, 1993.

[Chi95] Chiaberge, M., Sologuren, M. E., and Reyneri, L. M., *A Pulse Stream System for Low-Power Neuro-Fuzzy Computation*. In IEEE Transactions on Circuits and Systems - Fundamental Theory and Applications, v. 42, n. 11, Publisher: IEEE Press, New York, USA, ISSN: 1057-7122 , pp. 946-954, 1995.

[Chi96] Chiaberge, M., Sologuren, E. M., and Reyneri, L. M., *A Low-Power Neuro-Fuzzy Pulse Stream System*. Proceedings of the Fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems - MICRONEURO96, Feb., 1996, Lausanne, Switzerland, Publisher: IEEE Computer Society Press, pp. 191-199, 1996.

[Cli92] Cliff, D., Harvey, I., and Husbands, P., *Incremental Evolution of Neural Network Architectures for Adaptive Behaviour*. Report ID: Cognitive Science Research Paper Serial No. CSRP 256, The University of Sussex School of Cognitive and Computing Sciences, Falmer Brighton, BN1 9QH, England, UK, 15p., 1992.

[Cli94] Cliff, D., Harvey, I., and Husbands, P., *General Visual Robot Controller Networks Via Artificial Evolution*. Report ID: Cognitive Science Research Paper Serial No. CSRP 318, The University of Sussex School of Cognitive and Computing Sciences, Falmer Brighton, BN1 9QH, England, UK, 11p., 1994.

[Cli96] Cliff, D. and Miller, G. F., *Co-Evolution of Pursuit and Evasion II: Simulation Methods and Results*. Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior - SAB96: From Animals to Animats 4, Maes, P., Mataric, M., Meyer, J. A. et. al. (Eds.), Publisher: MIT Press, pp. 506-515, 1996.

[Cli97] Cliff, D., Harvey, I., and Husbands, P., *Artificial Evolution of Visual Control Systems for Robots*. In From Living Eyes to Seeing Machines, Srinivisan, M. and Venkatesh, S. (Eds.), Publisher: Oxford University Press, pp. 126-157, 1997.

[Dau98] Dautenhahn, K. and Nehaniv, C., *Artificial Life and Natural Stories*. Proceedings of the Third International Symposium on Artificial Life and Robotics - AROB III'98, v. 2, January 19-21, 1998, Beppu, Japan, pp. 435-439, 1998.

[Fic00] Ficici, S. G. and Pollack, J. B., *Effects of Finite Populations on Evolutionary Stable Strategies*. Proceedings of the 2000 Genetic and Evolutionary Computation Conference, Las Vegas, Nevada, USA, Whitley, L. D., Goldberg, D., Cantu-Paz, E. et. al. (Eds.), Publisher: Morgan-Kaufmann, ISBN: 1-55860-708-0, pp. 927-934, 2000.

[Fic99] Ficici, S. G., Watson, R. A., and Pollack, J. B., *Embodied Evolution: A Response to Challenges in Evolutionary Robotics*. Proceedings of the Eighth European Workshop on Learning Robots, Wyatt, J. L. and Demiris, J. (Eds.), pp. 14-22, 1999.

[Fil92] Filho, E. C. D. B., Fairhurst, M. C., and Bisset, D. L., *Analysis of Saturation Problem in RAM-Based Neural Network*. In Electronics Letters, v. 28, n. 4, pp. 345-346, 1992.

[Flo01] Floreano, D., Mondada, F., and Nolfi, S., *Co-Evolution and Ontogenetic Change in Competing Robots*. In Advances in the Evolutionary Synthesis of Intelligent Agents, Publisher: MIT Press, Cambridge, MA, USA, ISBN: 0-262-16201-6, 30p., 2001.

[Flo94] Floreano, D. and Mondada, F., *Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural-Network Driven Robot*. Proceedings of the 3rd International Conference on Simulation of Adaptive Behavior - SAB'94, From Animals to Animats 3, Cliff, D., Husbands, P., Meyer, J. A. et. al. (Eds.), Publisher: MIT Press/Bradford Books, Cambridge, MA, USA, pp. 421-430, 1994.

[Flo96a] Floreano, D. and Mondada, F., *Evolution of Homing Navigation in a Real Mobile Robot*. In IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics, v. 26, n. 3, pp. 396-407, 1996.

[Flo96b] Floreano, D. and Mondada, F., *Evolution of Plastic Neurocontrollers for Situated Agents*. Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior, From Animals to Animats 4, Maes, P., Mataric, M., Meyer, J.-A. et. al. (Eds.), Publisher: MIT Press/Bradford Book, Cambridge, MA, USA, pp. 402-410, 1996.

[Flo96c] Floreano, D., *Evolutionary Re-Adaptation of Neurocontrollers in Changing Environments*. Proceedings of the Conference on Intelligent Technologies, v. 2, Kosice, Slovakia, Sincak, P. (Ed.), Publisher: Efa Press, 10p., 1996.

[Flo97a] Floreano, D. and Nolfi, S., *Adaptive Behavior in Competing Co-Evolving Species*. Proceedings of the Fourth European Conference on Artificial Life, Husbands, P. and Harvey, I. (Eds.), Publisher: MIT Press, Cambridge, MA, USA, pp. 378-387, 1997.

[Flo97b] Floreano, D., *Reducing Human Design and Increasing Adaptability in Evolutionary Robotics*. In Evolutionary Robotics I, Gomi, T. (Ed.), Publisher: AAI Books, Ontario, Canada, pp. 187-220, 1997.

[Flo98a] Floreano, D., Nolfi, S., and Mondada, F., *Competitive Co-Evolutionary Robotics: From Theory to Practice*. Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior, Pfeifer, R. (Ed.), Publisher: MIT Press-Bradford Books, Cambridge, MA, USA, pp. 515-524, 1998.

[Flo98b] Floreano, D. and Mondada, F., *Hardware Solutions for Evolutionary Robotics*. Proceedings of the First European Workshop on Evolutionary Robotics, Husbands, P. and Meyer, J.-A. (Eds.), Publisher: Springer Verlag, Berlin, Germany, pp. 137-151, 1998.

[Fun00a] Funes, P., Lapat, L., and Pollack, J. B., *EvoCAD: Evolution-Assisted Design (Poster Abstract)*. Proceedings of the Artificial Intelligence in Design, Key Centre of Design Computing and Cognition, University of Sidney, Australia, pp. 21-24, 2000.

[Fun00b] Funes, P. and Pollack, J. B., *Measuring Progress in Coevolutionary Competition*. Proceedings of the Sixth International Conference on the Simulation of Adaptive Behavior: From Animals to Animats 6, Publisher: MIT Press, Meyer, J. et. al. (EDs.), pp. 450-459, 2000.

[Fun98] Funes, P., Sklar, E., Juillé, H., and Pollack, J. B., *Animal-Animat Coevolution: Using the Animal Population As Fitness Function*. Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior, From Animals to Animats 5, Pfeifer, R. (Ed.), Publisher: MIT Press, pp. 525-533, 1998.

[Fun99] Funes, P. and Pollack, J. B., *Computer Evolution of Buildable Objects*. In Evolutionary Design by Computers, Bentley, P. (Ed.), Publisher: Morgan Kaufmann, San Francisco, USA, pp. 387-403, 1999.

[Gar94a] Garis, H. D., *CAM-BRAIN: The Genetic Programming of an Artificial Brain Which Grows/Evolves at Electronic Speeds in a Cellular Automata Machine*. Proceedings of the 1st IEEE International Conference on Evolutionary Computation - ICEC'94, v. 11, pp. 337-339, 1994.

[Gar94b] Garis, H. D., *Evolution of an Artificial Brain Via Cellular Automata Based GenNets*. Proceedings of the International Symposium on Robotics and Manufacturing, Maui, Hawaii, USA, 9p., 1994.

[Gar97] Garis, H. D., *One Chip Evolvable Hardware : 1C-EHW*. Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms, Norwich, UK, 7p., 1997.

[Gra96] Grand, S., Cliff, D., and Malhotra, A., *Creatures: Artificial Life Autonomous Software Agents for Home Entertainment*. Report ID: Cognitive Science Research Paper Serial No. CSRP 434, The University of Sussex School of Cognitive and Computing Sciences, Falmer Brighton, BN1 9QH, England, UK, 8p., 1996.

[Gru94] Gruau, F., *Automatic Definitions of Modular Neural Networks*. In Adaptive Behavior, v. 3, n. 2, pp. 151-183, 1994.

[Har92] Harvey, I., *Species Adaptation Genetics Algorithms: A Basis for a Continuing SAGA*. Proceedings of the First European Conference on Artificial Life: Toward a Practice of Autonomous Systems, Varela, F. J. and Bourgine, P. (Eds.), Publisher: MIT Press/Bradford Books, Cambridge, MA, USA, pp. 346-354, 1992.

[Har93a] Harvey, I., *Evolutionary Robotics and SAGA: the Case for Hill Crawling and Tournament Selection*. In Artificial Life III, Langton, C. (Ed.), Publisher: Addison-Wesley, pp. 299-326, 1993.

[Har93b] Harvey, I., Husbands, P., and Cliff, D., *Genetic Convergence in a Species of Evolved Robot Control Architectures*. Report ID: Cognitive Science Research Paper Serial No. CSRP 267, The University of Sussex School of Cognitive and Computing Sciences, Falmer Brighton, BN1 9QH, England, UK, 14p., 1993.

[Har93c] Harvey, I., Husbands, P., and Cliff, D., *Issues in Evolutionary Robotics*. In From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior, Roitblat, Meyer, and Wilson (Eds.), Publisher: The MIT Press/Bradford Book, Cambridge, MA, USA, pp. 364-373, 1993.

[Har94] Harvey, I., Husbands, P., and Cliff, D., *Seeing the Light: Artificial Evolution, Real Vision*. Report ID: Cognitive Science Research Paper Serial No. CSRP 317, The University of Sussex School of Cognitive and Computing Sciences, Falmer Brighton, BN1 9QH, England, UK, 11p., 1994.

[Har96] Harvey, I. and Thompson, A., *Through the Labyrinth Evolution Finds a Way: a Silicon Ridge*. Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware - ICES96, Oct. 7-8, 1996, Tsukuba, Japan, Higuchi, T. and Iwata, M. (Eds.), Publisher: Springer-Verlag LNCS, ISBN: 3540631739, pp. 406-422, 1996.

[Har97a] Harvey, I., *Artificial Evolution for Real Problems*. In Evolutionary Robotics, Gomi, T. (Ed.), Publisher: AAI Books, Ontario, Canada, pp. 187-220, 1997.

[Har97b] Harvey, I., Husbands, P., Cliff, D., Thompson, A., and Jakobi, N., *Evolutionary Robotics: the Sussex Approach*. In Robotics and Autonomous Systems, v. 20, pp. 205-224, 1997.

[Har97c] Harvey, I., *Is There Another New Factor in Evolution?* In Evolutionary Computation, v. 4, n. 3, pp. 311-327, 1997.

[Har97d] Harvey, I. and Bossomaier, T., *Time Out of Joint: Attractors in Asynchronous Random Boolean Networks*. Proceedings of the Fourth European Conference on Artificial Life, Husbands, P. and Harvey, I. (Eds.), Publisher: MIT Press, pp. 67-75, 1997.

[Hay95] Haynes, T., Sen, S., Schoenefeld, D., and Wainwright, R., *Evolving Multiagent Coordination Strategies With Genetic Programming*. Report ID: Technical Report UTULSA-MCS-95-04, The University of Tulsa, USA, 17p., 1995.

[Hem97] Hemelrijk, C. K., *Cooperation Without Genes, Games Or Cognition*. Proceedings of the Fourth European Conference on Artificial Life - ECAL97, Publisher: MIT Press, Cambridge, MA, USA, ISBN: 0-262-58157-4, pp. 511-520, 1997.

[Hig94] Higuchi, T., Iba, H., and Manderick, B., *Evolvable Hardware*. In Massively Parallel Artificial Intelligence, Publisher: MIT Press, pp. 398-421, 1994.

[Hig96a] Higuchi, T., Iwata, M., Kajitani, I., Yamada, B., Manderick, B., Hirao, Y., Murakawa, M., Yoshizawa, S., and Furuya, T., *Evolvable Hardware With Genetic Learning*. Proceedings of the IEEE International Symposium on Circuits and Systems - ISCAS96, v. 4, pp. 29-32, 1996.

[Hig96b] Higuchi, T., Iwata, M., Kajitani, I., Murakawa, M., Yoshizawa, S., and Furuya, T., *Hardware Evolution at Gate and Function Levels*. Proceedings of the Biologically Inspired Autonomous Systems: Computation, Cognition and Action, Mar., 1996, Durham, North Carolina, USA, 6p., 1996.

[Hol62] Holland, J. H., *Outline for a Logical Theory of Adaptive Systems*. In Journal of the Association for Computing Machinery, v. 3, pp. 297-314, 1962.

[Hus93] Husbands, P., Harvey, I., and Cliff, D., *Analysing Recurrent Dynamical Networks Evolved for Robot Control*. Report ID: Cognitive Science Research Paper Serial No.CSRP 265, The University of Sussex School of Cognitive and Computing Sciences, Falmer Brighton, BN1 9QH, England, UK, 13p., 1993.

[Hus95] Husbands, P., Harvey, I., and Cliff, D., *Circle in the Round: State Space Attractors for Evolved Sighted Robots*. In Robotics and Autonomous Systems, v. 15, n. 1-2, pp. 83-106, 1995.

[Hus98] Husbands, P., *Evolving Robot Behaviours With Diffusing Gas Networks*. Proceedings of the First European Workshop on Evolutionary Robotics - EvoRobot98, Husbands, P. and Meyer, J. (Eds.), Publisher: Springer Verlag, pp. 123-136, 1998.

[Iba97] Iba, H., Iwata, M., and Higuchi, T., *Gate-Level Evolvable Hardware: Empirical Study and Application*. In Evolutionary Algorithms in Engineering Applications, Publisher: Springer-Verlag, pp. 259-276, 1997.

[Jak95] Jakobi, N., Husbands, P., and Harvey, I., *Noise and the Reality Gap: The Use of Simulation in Evolutionary Robotics*. In Advances in Artificial Life: Proceedings of the Third European Conference on Artificial Life - ECAL95, LNAI 929, Moran et. al. (Eds.), Publisher: Springer Verlag, pp. 704-720, 1995.

[Jak96] Jakobi, N., *Facing The Facts: Necessary Requirements For The Artificial Evolution of Complex Behaviour*. Report ID: Cognitive Science Research Paper Serial No. CSRP 422, The University of Sussex School of Cognitive and Computing Sciences, Falmer Brighton, BN1 9QH, England, UK, 9p., 1996.

[Jak97a] Jakobi, N., *Evolutionary Robotics and the Radical Envelope of Noise Hypotesis*. In Adaptive Behavior, v. 6, n. 1, pp. 131-174, 1997.

[Jak97b] Jakobi, N., *Half-Backed, Ad Hoc, and Noisy: Minimal Simulations for Evolutionary Robotics*. Proceedings of the Fourth European Conference on Artificial Life, Husbands, P. a. H. I. (Ed.), Publisher: MIT Press, pp. 348-357, 1997.

[Jak98a] Jakobi, N., Husbands, P., and Smith, T., *Robot Space Exploration by Trial and Error*. Proceedings of the Third Annual Conference on Genetic Programming, University of Wisconsin, Madison, Wisconsin, Koza, J., Banzhaf, W., Chellapilla, K. et. al. (Eds.), Publisher: Morgan Kaufmann, pp. 807-815, 1998.

[Jak98b] Jakobi, N. and Quinn, M., *Some Problems (and a Few Solutions) for Open-Ended Evolutionary Robotics*. Proceedings of the First European Workshop on Evolutionary Robotics - EvoRobot98, Husbands, P. and Meyer, J.-A. (Eds.), Publisher: Springer Verlag, 15p., 1998.

[Kan97] Kanada, Y., *Web Pages That Reproduce Themselves By Javascript*. In ACM SIGPLAN Notices, v. 32, n. 11, ISSN: 0362-1340, pp. 49-56, 1997.

[Kep94] Kephart, J. O., *A Biologically Inspired Immune System for Computers*. In Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems, Brooks, R. A. and Maes, P. (Eds.), Publisher: MIT Press, Cambridge, MA, USA, pp. 130-139, 1994.

[Key97a] Keymeulen, D., Durantez, M., Konaka, K., Kuniyoshi, Y., and Higuchi, T., *An Evolutionary Robot Navigation System Using a Gate-Level Evolvable Hardware*. In Evolvable Systems: From Biology to Hardware, Lecture Notes in Computer Science 1259, Higuchi, Iwata, and Liu (Eds.), Publisher: Springer-Verlag, pp. 195-209, 1997.

[Key97b] Keymeulen, D., Konaka, K., Iwata, M., Kuniyoshi, Y., and Higuchi, T., *Evolvable Reactive Execution System Using Reconfigurable Hardware: a Robot Navigation System Case Study* . In Working Notes of the American Association for Artificial Intelligence (AAAI) Fall Symposium on Model-Directed Autonomous Systems, Publisher: AAAI Press, 8p., 1997.

[Key97c] Keymeulen, D., Konaka, K., Iwata, M., Kuniyoshi, Y., and Higuchi, T., *Off-Line Evolution for a Robot Navigation System Based on a Gate-Level Evolvable Hardware*. Proceedings of the 4th European Conference on Artificial Life - ECAL97, Jul. 28-31, Brighton, UK, Publisher: MIT Press/Bradford Books, Cambridge, MA, USA, 11p., 1997.

[Key97d] Keymeulen, D., Konaka, K., Iwata, M., Kuniyoshi, Y., and Higuchi, T., *Robot Learning Using Gate-Level Evolvable Hardware*. Proceedings of the Sixth European Workshop on Learning Robots - EWLR-6, Brighton, UK, Birk, A. and Demiris, J. (Eds.), Publisher: Springer Verlag, pp. 173-187, 1997.

[Key98] Keymeulen, D., Iwata, M., Konaka, K., Suzuki, R., Kuniyoshi, Y., and Higuchi, T., *Off-Line Model-Free and on-Line Model-Based Evolution for Tracking Navigation Using Evolvable Hardware*. Proceedings of the First European Workshop on Evolutionary Robotics, Publisher: Springer Verlag, pp. 208-223, 1998.

[Kni48] Knight, R. L., *Dictionary of Genetics*. 1st ed., Waltham (Ed.), Publisher: Mass.: Chronica Botanica Company, 1948.

[Kod98] Kodjabachian, J. and Meyer, J. A., *Evolution and Development of Modular Control Architectures for 1-D Locomotion in Six-Legged Insects*. In Connection Science: Special Issue on BioRobotics, v. 10, n. 3-4, pp. 211-237, 1998.

[Koz92] Koza, J. R. and Rice, J., *Automatic Programming of Robots Using Genetic Programming*. Proceedings of the National Conference on Artificial Intelligence of the American Association for Artificial Intelligence - AAAI-92, Publisher: MIT Press, Cambridge, MA, USA, pp. 194-201, 1992.

[Koz97] Koza, J. R., Bennett III, F. H., Hutchings, J. L., Bade, S. L., Keane, M. A., and Andre, D., *Rapidly Reconfigurable Field-Programmable Gate Arrays for Accelerating Fitness Evaluation in Genetic Programming*. Proceedings of the Genetic Programming Conference: Late Breaking Papers, Stanford University, Stanford, CA, USA, Koza, J. R. (Ed.), Publisher: Stanford University Bookstore, Stanford, CA, USA, pp. 121-131, 1997.

[Koz98] Koza, J. R., *Genetic Programming*. In Encyclopedia of Computer Science and Technology, Williams, J. G. and Kent, A. (Eds.), Publisher: Marcel-Dekker, pp. 29-43, 1998.

[Lan96] Langdon, W. B., *Data Structures and Genetic Programming*. In Advances in Genetic Programming 2, Chapter 20, Angeline, P. and Kinnear, K. E. Jr. (Eds.), Publisher: MIT Press, Cambridge, MA, USA, ISBN: 0-262-01158-1, pp. 395-414, 1996.

[Lan97] Langdon, W. B. and Poli, R., *Genetic Programming in Europe*. Report ID: Report of the EvoGP Working Group on Genetic Programming of the European Network of Excellence in Evolutionary Computing, Centrum voor Wiskunde en Informatica, Kruislaan, 413, NL-1098 SJ, Amsterdam, Holland, 18p., 1997.

[Lan89] Langton, C. G., *Artificial Life*. In Artificial Life, Santa Fe Institute Studies in the Sciences of Complexity, Langton, C. G. (Ed.), Publisher: Addison-Wesley, Redwood City, USA, pp. 1-44, 1989.

[Lay99a] Layzell, P., *Inherent Qualities of Circuits Designed by Artificial Evolution: A Preliminary Study of Populational Fault Tolerance*. Proceedings of the First NASA/DoD Workshop on Evolvable Hardware - EH99, Jul. 19 - 21, 1999, Jet Propulsion Laboratory, California Institute of Technology, USA, Stoica, A., Keymeulen, D., and Lohn, J. (Eds.), Publisher: IEEE Computer Society Press, pp. 85-86, 1999.

[Lay99b] Layzell, P., *Reducing Hardware Evolution's Dependency on FPGAs*. Proceedings of the 7th International Conference on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems - MicroNeuro '99, Publisher: IEEE Computer Society Press, pp. 171-178, 199.

[Ler76] Lerner, I. M. and Libby, W. J., *Heredity, Evolution and Society*. 2nd ed., Publisher: W.H.Freeman and Company, San Francisco, CA, USA, pp. 119-174, 1976.

[Lip00] Lipson, H. and Pollack, J. B., *Automatic Design and Manufacture of Robotic Lifeforms*. In Nature, v. 406, pp. 974-978, 2000.

[Lud99] Ludermir, T., Carvalho, A., Brage, A., and Souto, M., *Weightless Neural Models: a Review of Current and Past Works*. In Neural Computing Surveys, v. 2, pp. 41-61, 1999.

[Lun97] Lund, H. H., Hallam, J., and Lee, W. P., *Evolving Robot Morphology*. Proceedings of the IEEE Fourth International Conference on Evolutionary Computation, Publisher: IEEE Press, 6p., 1997.

[Mat93] Mataric, M. J., *Kin Recognition, Similarity, and Group Behavior*. Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society, Boulder, CO, USA, pp. 705-710, 1993.

[Mat94] Mataric, M. J., *Learning to Behave Socially*. In From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior - SAB-94, Meyer, J. A. et. al. (Eds.), pp. 453-462, 1994.

[Mat95] Mataric, M. J., *Issues and Approaches in the Design of Collective Autonomous Agents*. In Robotics and Autonomous Systems, v. 16, n. 2-4, pp. 321-331, 1995.

[Mat96] Mataric, M. J., *Challenges In Evolving Controllers for Physical Robots*. In Evolutional Robotics: Special Issue of Robotics and Autonomous Systems, v. 19, n. 1, pp. 67-83, 1996.

[Mat97a] Mataric, M. J., *Behaviour-Based Control: Examples From Navigation, Learning, and Group Behaviour*. In Journal of Experimental and Theoretical Artificial Intelligence: Special Issue on Software Architectures for Physical Agents, v. 9, n. 2-3, Hexmoor, Horswill, and Kortenkamp (Eds.), pp. 323-336, 1997.

[Mat97b] Mataric, M. J., *Reinforcement Learning in the Multi-Robot Domain*. In Autonomous Robots, v. 4, n. 1, pp. 73-83, 1997.

[Mat97c] Mataric, M. J., *Using Communication to Reduce Locality in Distributed Multi-Agent Learning*. Proceedings of the National Conference on Artificial Intelligence of the American Association for Artificial Intelligence - AAAI-97, Providence, Rhode Island, pp. 643-648, 1997.

[Mat98a] Mataric, M. J., *Behavior-Based Robotics As a Tool for Synthesis of Artificial Behavior and Analysis of Natural Behavior*. In Trends in Cognitive Science, v. 2, n. 3, pp. 82-87, 1998.

[Mat98b] Mataric, M. J., *Reducing Locality Through Communication in Distributed Multi-Agent Learning*. In Journal of Experimental and Theoretical Artificial Intelligence: Special Issue on Learning in Distributed Artificial Intelligence Systems, v. 10, n. 3, Weiss, G. (Ed.), pp. 357-369, 1998.

[May96] Mayley, G., *Landscapes, Learning Costs and Genetic Assimilation*. In Evolutionary Computation: Special Issue on Evolution, Learning, and Instinct: 100 Years of the Baldwin Effect, v. 4, n. 3, Turney, P., Whitley, D., and Anderson, R. (Eds.), pp. 213-234, 1996.

[Maz98] Mazoyer, J. and Rapaport, I., *Inducing an Order on Cellular Automata by a Grouping Operation*. In Lecture Notes in Computer Science: STACS '98, v. 1373 , pp. 116-127, 1998.

[Mee96] Meeden, L. A., *An Incremental Approach to Developing Intelligent Neural Network Controllers for Robots*. In IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics, v. 26, n. 3, pp. 474-485, 1996.

[Men92] Mendenhall, W. and Sincich, T., *Statistics for Engineering and the Sciences*. 3rd ed., Dellen and Macmillan (Eds.), 1992.

[Mey97] Meyer, J. A., *From Natural to Artificial Life: Biomimetic Mechanisms in Animat Design*. In Robotics and Autonomous Systems, v. 22, pp. 3-21, 1997.

[Mic97] Michaud, F. and Mataric, M. J., *Behavior Evaluation and Learning From an Internal Point of View*. Proceedings of the Florida AI Research Symosium - FLAIRS-97 - Special Session ROBOLEARN-97: Evaluating Robot Learning, May 11-14, 1997, Daytona Beach, FL, USA, 10p., 1997.

[Mic95] Michel, O., *An Artificial Life Approach for the Synthesis of Autonomous Agents*. Proceedings of the European Conference on Artificial Evolution, Alliot, J., Lutton, E., Ronald, E. et. al. (Eds.), Publisher: Springer-Verlag, pp. 220-231, 1995.

[Mig95] Miglino, O., Lund, H. H., and Nolfi, S., *Evolving Mobile Robots in Simulated and Real Environments*. In Artificial Life, v. 2, pp. 417-434, 1995.

[Mil94] Millery, F. and Cliff, D., *Co-Evolution of Pursuit and Evasion I: Biological and Game-Theoretic Foundations*. Report ID: Technical Report CSRP311, The University of Sussex School of Cognitive and Computing Sciences, Falmer Brighton, BN1 9QH, England, UK, 40p., 1994.

[Mit95] Mitchell, M., *Genetic Algorithms and Artificial Life*. In Artificial Life- An Overview, Langton, C. G. (Ed.), Publisher: MIT Press, pp. 267-289, 1995.

[Mon93] Mondada, F. and Franzi, E., *Biologically Inspired Mobile Robot Control Algorithms*. Proceedings of the NFP-PNR 23 - Symposium on Artificial Intelligence and Robotics of the Swiss National Research Program - NFP, Oct. 22, 1993, Zurich, Switzerland, pp. 47-60, 1993.

[Mon95] Mondada, F. and Floreano, D., *Evolution of Neural Control Structures: Some Experiments on Mobile Robots*. In Robotics and Autonomous Systems, v. 16, pp. 183-195, 1995.

[Mon96] Mondada, F. and Floreano, D., *Evolution and Mobile Autonomous Robotics*. In Towards Evolvable Hardware, Sanchez, E. and Tommasini, M. (Eds.), Publisher: Springer Verlag, Berlin, Germany, pp. 221-249, 1996.

[Mor96] Moriarty, D. E. and Miikkulainen, R., *Evolving Obstacle Avoidance Behavior in a Robot Arm*. Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior - SAB96: From Animals to Animats, Maes, P., Mataric, M., Meyer, J.-A. et. al. (Eds.), Publisher: MIT Press, Cambridge, MA, USA, pp. 468-475, 1996.

[Mot96] Motorola Products, *MC68HC11A8 HCMOS Single-Chip Microcontroller*. In MC68HC11A8 Technical Data, Publisher: MOTOROLA, INC., 150p., 1996.

[Neb96] Nebel, B., *Artificial Intelligence: A Computational Perspective*. In Principals of Knowledge Representation, Studies in Logic, Language and Information, Brewka.G. (Ed.), Publisher: CSLI Publications, pp. 237-266, 1996.

[Nol92] Nolfi, S. and Parisi, D., *Growing Neural Networks*. Proceedings of the Third International Conference on Artificial Life - Artificial Life III, Jun. 15-19, 1992, Santa Fe, New Mexico, USA, Langton, C. G. (Ed.), Publisher: Addison-Wesley, Reading, MA, 17p., 1992.

[Nol94] Nolfi, S., Floreano, D., Miglino, O., and Mondada, F., *How to Evolve Autonomous Robots: Different Approaches in Evolutionary Robotics*. Proceedings of the Fourth International Conference on Artificial Life - Artificial Life IV, Brooks, R. and Maes, P. (Eds.), Publisher: MIT Press/Bradford Books, pp. 190-197, 1994.

[Nol95] Nolfi, S. and Parisi, D., *Evolving Non-Trivial Behaviors on Real Robots: an Autonomous Robot That Picks Up Objects*. Proceedings of the Fourth Congress of the Italian Association for Artificial Intelligence: Topics in Artificial Intelligence, Gori and Goda (Eds.), Publisher: Springer Verlag, pp. 243-254, 1995.

[Nol98a] Nolfi, S. and Floreano, D., *Co-Evolving Predator and Prey Robots: Do 'Arm Races' Arise in Artificial Evolution?* In Artificial Life, v. 4, n. 4, pp. 311-335, 1998.

[Nol98b] Nolfi, S. and Floreano, D., *How Co-Evolution Can Enhance the Adaptive Power of Artificial Evolution: Implications for Evolutionary Robotics*. Proceedings of the First European Workshop on Evolutionary Robotics - EvoRobot98, Apr. 16-17, 1998, Paris, France, Husbands, P. and Meyer, J. A. (Eds.), Publisher: Springer Verlag, pp. 22-38, 1998.

[Nor96] Nordin, J. P. and Banzhaf, W., *An On-Line Method to Evolve Behavior and to Control a Miniature Robot in Real Time With Genetic Programming*. In Adaptive Behavior, v. 5, n. 2, pp. 107-140, 1996.

[Nor97] Nordin, J. P. and Banzhaf, W., *Real Time Control of a Khepera Robot Using Genetic Programming*. In Control and Cybernetics, v. 26, n. 3, pp. 533-561, 1997.

[Nor94] Northmore, D. P. M. and Elias, J. G., *Evolving Synaptic Connections for a Silicon Neuromorph*. Proceedings of the IEEE Conference on Evolutionary Computation, v. 2, Orlando, FL, USA, pp. 753-758, 1994.

[Och99a] Ochoa, G., Harvey, I., and Buxton, H., *On Recombination and Optimal Mutation Rates*. Proceedings of the Genetic and Evolutionary Computation Conference - GECCO'99, Orlando, FL, USA, Banzhaf, W., Daida, J., Eiben, A. E. et. al. (Eds.), Publisher: Morgan Kaufmann, ISBN: 1-55860-611-4, pp. 488-495, 1999.

[Och99b] Ochoa, G. and Jaffe, K., *On Sex, Parasites, and the Red Queen*. In Journal of Theoretical Biology, v. 199, pp. 1-9, 1999.

[Par96] Paredis, J., *Coevolutionary Computation*. In Artificial Life, v. 2, n. 4, Langton, C. (Ed.), Publisher: MIT Press/Bradford Books, pp. 355-375, 1996.

[Per96a] Perez-Uribe, A. and Sanchez, E., *The FAST Architecture: A Neural Network With Flexible Adaptable-Size Topology*. Proceedings of the International Conference on Microelectronics for Neural and Fuzzy Systems - MicroNeuro '96, Publisher: IEEE Press, pp. 337-340, 1996.

[Per96b] Perez-Uribe, A. and Sanchez, E., *FPGA Implementation of an Adaptable-Size Neural Network*. Proceedings of the International Conference on Artificial Neural Networks - ICANN '96, pp. 383-388, 1996.

[Pol00] Pollack, J. B., Lipson, H., Ficici, S. G., Funes, P., Hornby, G. S., and Watson, R. A., *Evolutionary Techniques in Physical Robotics*. Proceedings of the Third International Conference on Evolvable Systems - ICES 2000: From Biology to Hardware (Lecture Notes in Computer Science; V. 1801), Publisher: Springer , pp. 175-186, 2000.

[Rao96] Rao, R. and Fuentes, O., *Learning Navigational Behaviors Using a Predictive Sparse Distributed Memory*. In From Animals to Animats 4, Maes, P. et. al. (Eds.), Publisher: MIT Press, Cambridge MA, USA, pp. 382-390, 1996.

[Rey93] Reyneri, L. M., Chiaberge, M., and Del Corso, D., *Using Coherent Pulse Width and Edge Modulation in Artificial Neural Systems*. In International Journal on Neural Systems: Special Issue on Microneuro 93, v. 4, n. 4, pp. 407-417, 1993.

[Rey94] Reyneri, L. M., Wiyhagen, H. C. A. M., Hegt, J. A., and Chiaberge, M., *A Comparison Between Analog and Pulse Stream VLSI Hardware for Neural Networks and Fuzzy Systems*. Proceedings of the International Conference on Microelectronics for Neural and Fuzzy Systems - MICRONEURO '94, Sep., 1994, Turin, Italy, pp. 77-86, 1994.

[Rey95] Reyneri, L. M., Chiaberge, M., and Zocca, L., *CINTIA: A Neuro-Fuzzy Real Time Controller for Low Power Embedded Systems*. In IEEE Micro Magazine (M-MICRO), Special Issue on Hardware for Artificial Neural Networks, v. 15, n. 3, pp. 40-47, 1995.

[Sch96] Schneider-Fontan, M. and Mataric, M. J., *A Study of Territoriality: The Role of Critical Mass in Adaptive Task Division*. In From Animals to Animats IV, Maes, P. et. al. (Eds.), Publisher: MIT Press, pp. 553-561, 1996.

[Sch95] Schwefel, H. P. and Rudolph, G., *Contemporary Evolution Strategies*. Proceedings of the Third International Conference on Artificial Life: Advances in Artificial Life (Lecture Notes in Artificial Intelligence, V. 929), Publisher: Springer Verlag, Berlin, Germany, pp. 893-907, 1995.

[Set97] Seth, A. K., *Interaction, Uncertainty, and the Evolution of Complexity*. Proceedings of the Fourth European Conference on Artificial Life, Husbands, P. and Harvey, I. (Eds.), Publisher: MIT Press, pp. 521-530, 1997.

[Set98a] Seth, A. K., *Evolving Action Selection and Selective Attention Without Actions, Attention or Selection*. Proceedings of the Fifth International Conference of the Society for Adaptive Behaviour, Pfeifer, R., Blumberg, B., Meyer, J. et. al. (Eds.), Publisher: MIT Press, pp. 139-147, 1998.

[Set98b] Seth, A. K., *Noise and the Pursuit of Complexity, a Study in Evolutionary Robotics*. Proceedings of the First European Workshop on Evolutionary Robotics - EvoRobot98, Husbands, P. and Meyer, J. (Eds.), Publisher: Springer Verlag, pp. 123-136, 1998.

[Set99] Seth, A. K., *Evolving Behavioural Choice: an Investigation into Herrnstein's Matching Law*. Proceedings of the Fifth European Conference on Artificial Life - ECAL99, Lausanne, Switzerland, Floreano, D., Nicoud, J.-D., and Mondada, F. (Eds.), Publisher: Springer-Verlag, pp. 225-235, 1999.

[Shi00] Shipman, R., Shackleton, M., Ebner, M., and Watson, R. A., *Neutral Search Spaces for Artificial Evolution: A Lesson From Life*. Proceedings of the Seventh International Workshop on the Synthesis and Simulation of Living Systems - Artificial Life VII, Aug. 1-2, 2000, Reed College, Portland, Oregon, USA, Bedau, M., McCaskill, J., Packard, N. et. al. (Eds.), pp. 52-59, 2000.

[Sim94] Simoes, E. D. V., Uebel, L. F., and Barone, D. A. C., *Fast Prototyping of Artificial Neural Network: GSN Digital Implementation.* Proceedings of the IV International Conference on Microelectronics for Neural Networks and Fuzzy System, Sep., 1994,Turin, Italy, pp. 192-201, 1994.

[Sim96] Simoes, E. D. V., Uebel, L. F., and Barone, D. A. C., *Hardware Implementation of RAM Neural Networks*. In Pattern Recognition Letters, n. 17, pp. 421-429, 1996.

[Sim97] Simoes, E. D. V., Ueno, Y., and Barone, D. A. C., *The Adaptive Weight Using RAM Model*. Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics, Oct., 1997, Orlando, USA, pp. 234-239, 1997.

[Sim99] Simoes, E. D. V. and Dimond, K. R., *An Evolutionary Controller for Autonomous Multi-Robot Systems*. Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, v. 6, Oct., 1999, Tokyo, Japan, pp. 596-601, 1999.

[Sip95] Sipper, M., *An Introduction to Artificial Life*. In AI Expert: Special Issue on Explorations in Artificial Life, Publisher: Miller Freeman, San Francisco, CA, USA, pp. 4-8, 1995.

[Smi98] Smith, T., *Blurred Vision: Simulation-Reality Transfer of a Visually Guided Robot*. Proceedings of the First European Workshop on Evolutionary Robotics - EvoRobot98, Husbands, P. and Meyer, J. (Eds.), Publisher: Springer Verlag, pp. 123-136, 1998.

[Ste94] Steels, L., *Emergent Functionality in Robotic Agents Through on-Line Evolution*. Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems - Artificial Life IV, Brooks, R. A. and Maes, P. (Eds.), Publisher: MIT Press, Cambridge, MA, USA, pp. 8-16, 1994.

[Ste95] Steels, L., *The Homo Cyber Sapiens, the Robot Homonidus Intelligens, and the Artificial Life Approach to Artificial Intelligence*. Proceedings of the Burda Symposium on Brain-Computer Interfaces, Muenchen, 17p., 1995.

[Tem95] Tempesti, G., *A New Self-Reproducing Cellular Automaton Capable of Construction and Computation*. In Advances in Artificial Life (LNAI-929), Moran, F. (Ed.), Publisher: Springer-Verlag, pp. 555-563, 1995.

[Teu99] Teuscher, C., Sanchez, E., and Sipper, M., *Romero's Pilgrimage to Santa Fe: A Tale of Robot Evolution*. Proceedings of the Genetic and Evolutionary Computation Conference - GECCO'99 (Workshop), Orlando, FL, USA, Wu, A. S. (Ed.), pp. 409-410, 1999.

[Tho00] Thompson, A. and Layzell, P., *Evolution of Robustness in an Electronics Design*. Proceedings of the 3rd International Conference on Evolvable Systems - ICES 2000, Publisher: Springer Verlag, pp. 218-228, 2000.

[Tho94a] Thompson, A., *Evolving Fault-Tolerant Systems*. Report ID: Cognitive Science Research Paper Serial No. CSRP 385, The University of Sussex School of Cognitive and Computing Sciences, Falmer Brighton, BN1 9QH, England, UK, 6p., 1994.

[Tho94b] Thompson, A., *Real Time for Real Power: Methods of Evolving Hardware to Control Autonomous Mobile Robots*. Report ID: Cognitive Science Research Paper Serial No. CSRP 350, The University of Sussex School of Cognitive and Computing Sciences, Falmer Brighton, BN1 9QH, England, UK, 6p., 1994.

[Tho94c] Thompson, A., Harvey, I., and Husbands, P., *Unconstrained Evolution and Hard Consequences*. Report ID: Cognitive Science Research Paper Serial No. CSRP 397, The University of Sussex School of Cognitive and Computing Sciences, Falmer Brighton, BN1 9QH, England, UK, 31p., 1994.

[Tho95] Thompson, A., *Evolving Electronic Robot Controllers That Exploit Hardware Resources*. Proceedings of the 3rd European Conference on Artificial Life - ECAL'95 (LNAI 929), Moran (Ed.), Publisher: Springer-Verlag, pp. 640-656, 1995.

[Tho96a] Thompson, A., *Evolutionary Techniques for Fault Tolerance*. Proceedings of the UKACC International Conference on Control - CONTROL'96 (IEE Conference Publication No.427), pp. 693-698, 1996.

[Tho96b] Thompson, A., *An Evolved Circuit, Intrinsic in Silicon, Entwined With Physics*. Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware - ICES96, Oct. 7-8, 1996, Tsukuba, Japan, Higuchi, T. and Iwata, M. (Eds.), Publisher: Springer-Verlag LNCS, pp. 390-405, 1996.

[Tho96c] Thompson, A., *Silicon Evolution*. Proceedings of the First Annual Conference on Genetic Programming - GP96, Koza, J. R., Goldberg, D. E., Fogel, D. B. et. al. (Eds.), Publisher: MIT Press, Cambridge, MA, USA, pp. 444-452, 1996.

[Tho97a] Thompson, A., *Artificial Evolution in the Physical World*. In Evolutionary Robotics: From Intelligent Robots to Artificial Life - ER'97, Gomi, T. (Ed.), Publisher: AAI Books, pp. 101-125, 1997.

[Tho97b] Thompson, A., *Temperature in Natural and Artificial Systems*. Proceedings of the Fourth European Conference on Artificial Life, Husbands, P. and Harvey, I. (Eds.), Publisher: MIT Press, pp. 388-397, 1997.

[Tho98] Thompson, A., *On the Automatic Design of Robust Electronics Through Artificial Evolution*. Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware - ICES98, Sep. 23-26, 1998, Lausanne, Switzerland, Sipper, M., Mange, D., and Pres-Uribe, A. (Eds.), Publisher: Springer-Verlag, pp. 13-24, 1998.

[Tho99] Thompson, A. and Layzell, P., *Analysis of Unconventional Evolved Electronics*. In Communications of the Association of Computing Machinery, v. 42, n. 4, Yao, X. (Ed.), pp. 71-79, 1999.

[Tho96] Thornton, C., *Unsupervised Constructive Learning*. Report ID: Cognitive Science Research Paper Serial No. CSRP 449, The University of Sussex School of Cognitive and Computing Sciences, Falmer Brighton, BN1 9QH, England, UK, 9p., 1996.

[Tod97] Todd, P. M. and Miller, G. F., *Biodiversity Through Sexual Selection*. In Artificial Life V, Langton, C. G. and Shimohara, K. (Eds.), Publisher: MIT Press, Cambridge, MA. USA, pp. 289-299, 1997.

[Tom95] Tomassini, M., *A Survey of Genetic Algorithms*. In Annual Reviews of Computational Physics, World Scientific, pp. 87-118, 1995.

[Tom96] Tomassini, M., *Evolutionary Algorithms*. In Towards Evolvable Hardware, Publisher: Springer-Verlag, Berlin, Germany, pp. 19-47, 1996.

[Wag95] Wagner, G. P., *Adaptation and the Modular Design of Organisms*. Proceedings of the 3rd European Conference on Artificial Life - ECAL'95: Advances in Artificial Life (LNAI 929), Moran et al. (Ed.), Publisher: Springer-Verlag, pp. 317-328, 1995.

[Wat00a] Watson, R. A., Reil, T., and Pollack, J. B., *Mutualism, Parasitism, and Evolutionary Adaptation*. Proceedings of the Seventh International Workshop on the Synthesis and Simulation of Living Systems - Artificial Life VII, Bedau, M., McCaskill, J., Packard, N. et. al. (Eds.), pp. 170-178, 2000.

[Wat00b] Watson, R. A. and Pollack, J. B., *Recombination Without Respect: Schema Combination and Disruption in Genetic Algorithm Crossover*. Proceedings of the Genetic and Evolutionary Computation Conference, Las Vegas, Nevada, USA, Whitley, D. (Ed.), Publisher: Morgan Kaufmann, ISBN: 1-55860-708-0, pp. 112-119, 2000.

[Wat99a] Watson, R. A., Ficici, S. G., and Pollack, J. B., *Embodied Evolution: Embodying an Evolutionary Algorithm in a Population of Robots*. Proceedings of the Congress on Evolutionary Computation, Angeline, P., Michalewicz, Z., Schoenauer, M. et. al. (Eds.), Publisher: IEEE Press, pp. 335-342, 1999.

[Wat99b] Watson, R. A. and Pollack, J. B., *Incremental Commitment in Genetic Algorithms*. Proceedings of the Genetic and Evolutionary Computation Conference - GECCO99, Banzhaf, Daida, Eiben et. al. (Eds.), Publisher: Morgan Kauffmann, pp. 710-717, 1999.

[Wer96] Werger, B. B. and Mataric, M. J., *Robotic "Food" Chains: Externalization of State and Program for Minimal-Agent Foraging*. Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior - SAB-96: From Animals to Animats 4, Maes, P., Mataric, M., Meyer, J.-P. et. al. (Eds.), Publisher: MIT Press/Bradford Books, pp. 625-634, 1996.

[Wer99] Werger, B. B. and Mataric, M. J., *Exploiting Embodiment in Multi-Robot Teams*. Report ID: Technical Report IRIS-99-378, University of Southern California, Institute for Robotics and Intelligent Systems, 14p., 1999.

[Xil96] Xilinx, *XC6200 FPGA Advanced Product Specification*. Publisher: Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, USA, version 1.0 edition, Jun., 1996, URL: http://www.xilinx.com/partinfo/6200.pdf, 1996.

[Yao97] Yao, X. and Higuchi, T., *Promises and Challenges of Evolvable Hardware*. In Evolvable Systems: From Biology to Hardware, (Lecture Notes in Computer Science 1259), Publisher: Springer-Verlag, pp. 55-78, 1997.

APPENDIX A

Contents of the Attached CD-ROM

This Ph.D. thesis includes a CD-ROM containing the software implemented during the development of this work. The CD-ROM also contains a collection of selected photos and videos that help to illustrate how the proposed evolutionary system was implemented and operates.

The material included in the CD-ROM is organised in four folders: *Experimental Data*; *General Software*; *Videos*; and *Photos*. The last two folders contain video files and pictures. *General Software* contains some files with the main programs and data records produced and *Experimental Data* contains the programs developed during the experiments described in Chapters 6, 7, and 8, organised by chapters and by Experiments within the chapters. The structure of the CD-ROM is presented in Figure A.1 and the contents of each folder are listed below.

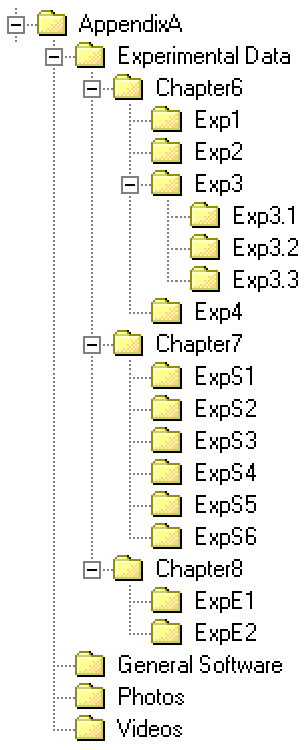


Figure A.1 – Structure of the folders in the attached CD-ROM.

LIST OF THE CD-ROM CONTENTS:

```
Volume in drive D is CD-ROM

Directory of D:\AppendixA

VIDEOS
GENERAL SOFTWARE
PHOTOS
EXPERIMENTAL DATA

Directory of D:\AppendixA\Experimental Data

CHAPTER6      <DIR>       23/11/00   1:56 Chapter6
CHAPTER7      <DIR>       23/11/00   1:56 Chapter7
CHAPTER8      <DIR>       23/11/00   1:56 Chapter8
        0 file(s)           0 bytes

Directory of D:\AppendixA\Experimental Data\Chapter6

EXP1          <DIR>       23/11/00   1:56 Exp1
EXP2          <DIR>       23/11/00   1:56 Exp2
EXP3          <DIR>       23/11/00   1:56 Exp3
EXP4          <DIR>       23/11/00   1:56 Exp4
        0 file(s)           0 bytes
```

```
Directory of D:\AppendixA\Experimental Data\Chapter6\Exp1

Graf01.cpp
graf1.exe
exp1.exe
Exp01.cpp
        4 file(s)        345,424 bytes

Directory of D:\AppendixA\Experimental Data\Chapter6\Exp2

exp7.exe
Graf07.cpp
graf7.exe
Exp07.cpp
        4 file(s)        345,424 bytes

Directory of D:\AppendixA\Experimental Data\Chapter6\Exp3

EXP3     1     <DIR>        23/11/00   2:00 Exp3.1
EXP3     2     <DIR>        23/11/00   2:00 Exp3.2
EXP3     3     <DIR>        23/11/00   2:00 Exp3.3
        0 file(s)             0 bytes

Directory of D:\AppendixA\Experimental Data\Chapter6\Exp3\Exp3.1

Exp08.cpp
Graf08.cpp
exp8.exe
graf8.exe
        4 file(s)        345,424 bytes

Directory of D:\AppendixA\Experimental Data\Chapter6\Exp3\Exp3.2

Exp10.cpp
Graf10.cpp
exp10.exe
graf10.exe
        4 file(s)        345,424 bytes

Directory of D:\AppendixA\Experimental Data\Chapter6\Exp3\Exp3.3

Exp09.cpp
Graf09.cpp
exp9.exe
graf9.exe
        4 file(s)        345,424 bytes

Directory of D:\AppendixA\Experimental Data\Chapter6\Exp4

graf13.exe
exp13.exe
Graf13.cpp
Exp13.cpp
        4 file(s)        345,424 bytes

Directory of D:\AppendixA\Experimental Data\Chapter7

EXPS1        <DIR>        23/11/00   2:16 ExpS1
EXPS2        <DIR>        23/11/00   2:16 ExpS2
EXPS3        <DIR>        23/11/00   2:16 ExpS3
EXPS4       <DIR>        23/11/00   2:17 ExpS4
```

```
EXPS5       <DIR>        23/11/00   2:17 ExpS5
EXPS6       <DIR>        23/11/00   2:17 ExpS6
        0 file(s)             0 bytes

Directory of D:\AppendixA\Experimental Data\Chapter7\ExpS1

Sim01.cpp
Grafsim01.cpp
Grafsim01.exe
Sim01.exe
        4 file(s)       334,522 bytes

Directory of D:\AppendixA\Experimental Data\Chapter7\ExpS2

Grafsim02.cpp
Grafsim02.exe
Sim02.cpp
Sim02.exe
        4 file(s)       337,387 bytes

Directory of D:\AppendixA\Experimental Data\Chapter7\ExpS3

Sim04.exe
Grafsim04.exe
Grafsim04.cpp
Sim04.cpp
        4 file(s)       370,108 bytes

Directory of D:\AppendixA\Experimental Data\Chapter7\ExpS4

Sim05.exe
Grafsim05.exe
Grafsim05.cpp
Sim05.cpp
        4 file(s)       370,301 bytes

Directory of D:\AppendixA\Experimental Data\Chapter7\ExpS5

Grafneu01.cpp
Grafneu01.exe
Simneu01.exe
Simneu01.cpp
        4 file(s)       348,143 bytes

Directory of D:\AppendixA\Experimental Data\Chapter7\ExpS6

Grafneu02.cpp
Simneu02.exe
Grafneu02.exe
Simneu02.cpp
        4 file(s)       349,489 bytes

Directory of D:\AppendixA\Experimental Data\Chapter8

EXPE1         <DIR>        23/11/00   2:34 ExpE1
EXPE2         <DIR>        23/11/00   2:34 ExpE2
        0 file(s)             0 bytes

Directory of D:\AppendixA\Experimental Data\Chapter8\ExpE1

Exp15.exe
```

```
Graf15.exe
Graf15.cpp
Exp15.cpp
        4 file(s)       357,630 bytes

Directory of D:\AppendixA\Experimental Data\Chapter8\ExpE2

Exp16.exe
Exp16.cpp
Graf16.cpp
Graf16.exe
        4 file(s)       357,953 bytes

Directory of D:\AppendixA\General Software

Evolution.asm
virtualscreen.cpp
Handcontrol.asm
Graf20.cpp
GRAFIC.TXT
Initialise.cpp
Gene.txt
Exp20.cpp
        8 file(s)       155,934 bytes

Directory of D:\AppendixA\Photos

Image46.jpg
IMAGE41.KDC
IMAGE39.KDC
IMAGE38.KDC
IMAGE33.KDC
IMAGE31.KDC
IMAGE19.KDC
robot_top.jpg
Robot_side.jpg
robot_front3.jpg
Robot_front1.jpg
Radio1.jpg
Z.jpg
r_n+obs.jpg
r_lab.jpg
r_cima.jpg
r_c+obs.jpg
lab_simple.jpg
lab_med.jpg
lab_complex.jpg
Image48.jpg
IMAGE47.jpg
pspbrwse.jbf
IMAGE44.jpg
Image43.jpg
IMAGE42.jpg
Image41.jpg
Image40.jpg
Image39.jpg
Image38.jpg
IMAGE37.jpg
Image36.jpg
Image33.jpg
IMAGE31.jpg
```

```
Image30.jpg
Image28.jpg
Image27.jpg
IMAGE26.jpg
IMAGE25.jpg
IMAGE24.jpg
IMAGE23.jpg
Image22.jpg
IMAGE21.jpg
IMAGE20.jpg
Image19.jpg
Image18.jpg
IMAGE17.jpg
IMAGE16.jpg
IMAGE15.jpg
IMAGE14.jpg
Image13.jpg
Image12.jpg
IMAGE11.jpg
IMAGE10.jpg
Image1.jpg
IMAGE09.jpg
IMAGE08.jpg
IMAGE07.jpg
IMAGE06.jpg
Image05.jpg
IMAGE04.jpg
IMAGE03.jpg
Image02.jpg
IMAGE01.jpg
Fig1_5.jpg
eu_robo.jpg
Robot1.bmp
board2.bmp
board1.bmp
        69 file(s)      7,219,290 bytes

Directory of D:\AppendixA\Videos

video1.jpg
video2.jpg
video3.jpg
video4.jpg
video5.jpg
         1 file(s)             1753,270 bytes
```

APPENDIX B

Schematic Diagrams

Appendix B contains two schematic diagrams. The first one, Diagram 1, contains the schematics of the circuit of the robot board, and a list of the used components. The second one, Diagram 2, contains the schematics of the circuit of the radio board, and a list of the used components.

Diagram 1:

Schematics of the circuit of the robot board

List of the used components:

Part Type	Designator
0.01uF	C7
0.1uF	CU4
0.1uF	CU3
0.1uF	CU2
0.1uF	CU7
0.1uF	C8
0.1uF	CU5
0.1uF	CU6
100M	R3
10K	R7
10K	R6
10K	R9
10K	R8
10K	R10
10K	R5
10M	R1
10uF	C5
10uF	C9
10uF	C6
1K	R2
1uF	C3
1uF	C4
1uF	CU1
24pF	C1
24pF	C2
2K2	R11
2K2	R12
330R	R15
330R	R16
4011	U4
4K7	R4
74LS04	U12
820R	R14
820R	R13
8MHz	X1
AMRECEIVER-AMRW	TX1
AMTRANSMITER-AMT21	TX1
ANTENNA	A1
ANTENNA	A2
BC182B	T1
BC182B	T6
BC182B	T5
BC182B	T4
BC182B	T2
BC182B	T3

HEADER 10	JP3
HEADER 10	JP6
HEADER 10	JP1
HEADER 10	JP2
HEADER 10	JP4
HEADER 10	JP5
HEADER 2	JP9
HEADER 2	JP8
HEADER 2	JP6
HEADER 2	JP7
HEADER 3	JP10
L293NE	U11
LED-Green	L3
LED-Green	L2
LED-Green	L5
LED-Green	L4
LED-Red	L7
LED-Red	L6
LED-Red	L1
LED-Red	L9
LED-Red	L8
MAX233	U7
MAX603	U6
MAX691	U8
MC68HC11A1FN(52)	U1
MC68HC24	U9
MC74HC373	U2
OPE5594	IR4
OPE5594	IR3
OPE5594	IR5
OPE5594	IR7
OPE5594	IR8
OPE5594	IR6
OPE5594	IR2
OPE5594	IR1
RN-10K	RN3
RN-10K	RN2
RN-10K	RN1
RN-4.7K	RN4
RN-4.7K	RN5
S4576DY	U5
SW DIP-8	DP1
SW DIP-8	DP2
SW SPDT	S1
SW SPDT	S2
SW-PB	PB4
SW-PB	PB2
SW-PB	PB3
SW-PB	PB1
SW-PBSMALL	PB5

Diagram 2:

Schematics of the circuit of the radio board

List of the used components:

Part Type	Designator
0.1uF	C2
10uF	C3
2K2	R2
2K2	R3
2K2	R1
AMRECEIVER-AMRW	TX1
AMTRANSMITER-AMT21	TX1
ANTENNA	A1
ANTENNA	A2
CON25	JP1
DIODE	D1
HEADER 2	JP2
LED-GREEN	L2
LED-RED	L1
MAX233	U2

APPENDIX C

Experimental Data

Appendix C contains complementary data that for a reason of space could not be presented in the body of the thesis. The data is organised by chapters and by sections within the chapters.

C.1 - Data Relative to Chapter 6:

C.1.1 - Section 6.3: Experiment 2

The data presented here is the result of experiments that were carried out to determine the best sensor configurations to drive the robots according to the hand-designed controller used in Experiment 2 (Section 6.3). Many sensor combinations were tested in different environments containing simple (11 tests), medium (20 tests), and complex (9 tests) configurations of obstacles. The best 42 combinations, presented in Table C.2, were selected and ordered by the average fitness they scored in all 40 experiments. A "blind" robot with all sensors disabled (*0,0,0,0,0,0,0*) was also included to provide a comparison to the worst case. In the sensor specifications shown below, a disabled sensor is represented by 0 and an enabled sensor is represented by its own name (i.e., *S1* or *S7*). *S5* was not present.

- **Summary of the Settings**

Table C.1 presents a summary of the settings for the experiments:

Table C.1 – Summary of the settings

Parameter	Definition
Fitness Function:	+3 points every 1s moving forward –10 points each collision
Initial Fitness Value:	4096 points
Maximum Fitness Value:	4186 points
Generation Time:	Population fixed at the first generation: 30 seconds
Mutation Rate:	Not present
Speed Levels:	Fixed at maximum speed (Vmax=32)
Sensors Enable:	All sensors but *S5* can be enabled independently *S5* is permanently disabled
Navigation Controller:	Fixed Neural Network (m=4, n=7, neuron size=4 bits)

- **Results**

Table C.2 – The Best Sensor Configurations in 40 Tests

Sensor Configuration	Average Fitness Value	Sensor Configuration	Average Fitness Value
S8,0,S6,S4,S3,S2,S1	4144	S8,S7,S6,S4,S3,0,S1	4130
0,S7,S6,0,S3,0,S1	4143	0,0,0,0,S3,S2,S1	4129
0,S7,S6,0,S3,S2,S1	4142	0,0,0,0,0,0,S1	4128
0,S7,0,0,0,S2,S1	4141	S8,S7,0,0,0,S2,S1	4128
0,S7,0,0,S3,S2,S1	4140	S8,S7,S6,0,S3,S2,0	4127
0,S7,S6,S4,0,0,S1	4139	S8,S7,S6,S4,0,0,S1	4127
S8,S7,S6,S4,0,S2,S1	4139	0,0,0,0,0,S2,S1	4126
0,0,0,0,S3,0,S1	4138	S8,0,S6,S4,0,0,S1	4126
S8,0,0,S4,S3,S2,S1	4138	S8,S7,S6,S4,S3,S2,0	4125
S8,S7,S6,0,S3,0,S1	4138	0,0,0,S4,S3,S2,S1	4123
0,S7,0,S4,S3,0,S1	4136	0,0,S6,S4,S3,0,S1	4123
S8,0,0,S4,S3,S2,S1	4135	0,S7,S6,0,0,0,S1	4123
S8,S7,0,S4,0,0,S1	4135	S8,0,0,0,0,S2,S1	4122
0,0,0,S4,0,0,S1	4134	0,S7,0,S4,0,S2,0	4119
S8,S7,S6,S4,S3,S2,S1	4134	S8,0,0,S4,0,S2,S1	4118
0,S7,0,0,0,0,S1	4133	S8,0,0,0,S3,S2,S1	4113
0,0,0,S4,0,S2,S1	4132	S8,0,S6,S4,S3,0,0	4112
0,S7,S6,S4,S3,S2,S1	4132	S8,0,0,0,0,0,S1	4110
S8,S7,0,S4,S3,S2,S1	4132	S8,0,0,0,S3,0,S1	4107
S8,S7,0,S4,S3,S2,0	4132	S8,0,0,S4,S3,0,0	4106
S8,0,S6,S4,S3,0,S1	4131	0,0,0,0,0,0,0	4073
S8,0,S6,S4,0,S2,S1	4130		

C.2 - Data Relative to Chapter 7:

C.2.1 - Section 7.2: Experiment S1

The chart S1.1 presented in Figure 7.2 (Section 7.2) is just one of many experiments performed with the simulator configured as described in Experiment S1. In total, it was run 300 times for each mutation rate (0.1% and 0.5%), during 30,000 generations.

Mutation Rate:	Av. Best Fitness	Standard Deviation	Max. Fitness	Min. Fitness
6 Robots & 0.1%	4319.1	2.13177026	4322	4316
6 Robots & 0.5%	4308.2	2.29975844	4312	4305
50 Robots & 0.1%	4339.1	1.95874658	4342	4335
50 Robots & 0.5%	4336.4	2.01254863	4340	4331
100 Robots & 0.1%	4344.9	1.52356458	4348	4343
100 Robots & 0.5%	4341.8	1.62352418	4345	4338

Where:		
Where:	**Av. Best Fitness**	Average in 300 tests of the fitness of the best robot in generation 30,000;
	Standard Deviation	Standard Deviation for the Average in 300 tests of the fitness of the best robot in generation 30,000;
	Max. Fitness	Maximum fitness value obtained in generation 30,000 in 300 tests;
	Min. Fitness	Minimum fitness value obtained in generation 30,000 in 300 tests;

The charts S1.3 and S1.4 presented in Figures 7.4 and 7.5 (Section 7.2) are just one of many experiments performed with the simulator configured as described in Experiment S1. In total, it was run 300 times for each mutation rate (0.1%, 0.5%, 1%, 3%, 10%, 20%, 50%, and 80%), during 300 generations.

Mutation Rate:	Av. Best Fitness	Standard Deviation	Max. Fitness	Min. Fitness
0.1%	4216.4	2.59058123	4219	4211
0.5%	4228.2	1.75119007	4230	4225
1%	4198.1	2.72641401	4202	4194
3%	4192.1	2.51440296	4195	4188
10%	4181.5	3.20589734	4185	4177
20%	4166.8	2.394438	4170	4163
50%	4162.2	4.18462795	4169	4157
80%	4160.3	3.77270902	4165	4152

Where: **Av. Best Fitness**	Average in 300 tests of the fitness of the best robot in generation 300;
Standard Deviation	Standard Deviation for the Average in 300 tests of the fitness of the best robot in generation 300;
Max. Fitness	Maximum fitness value obtained in generation 300 in 300 tests;
Min. Fitness	Minimum fitness value obtained in generation 300 in 300 tests;

C.2.2 - Section 7.3: Experiment S2

The chart S2.1 presented in Figure 7.7 (Section 7.3) is just one of many experiments performed with the simulator configured as described in Experiment S2. In total, it was run 300 times for each mutation rate (0.1% and 0.5%), during 30,000 generations.

Mutation Rate:	Av. Best Fitness	Standard Deviation	Max. Fitness	Min. Fitness
0.1%	4300.5	2.12132034	4304	4292
0.5%	4326.4	1.71269768	4339	4330

Where: **Av. Best Fitness**	Average in 300 tests of the fitness of the best robot in generation 30,000;
Standard Deviation	Standard Deviation for the Average in 300 tests of the fitness of the best robot in generation 30,000;
Max. Fitness	Maximum fitness value obtained in generation 30,000 in 300 tests;
Min. Fitness	Minimum fitness value obtained in generation 30,000 in 300 tests;

The chart S2.2 presented in Figure 7.8 (Section 7.3) are just one of many experiments performed with the simulator configured as described in Experiment S2. In total, it was run 300 times for each mutation rate (0.1%, 0.5%, 1%, and 10%), during 300 generations.

Mutation Rate:	Av. Best Fitness	Standard Deviation	Max. Fitness	Min. Fitness
0.1%	4225.9	2.13177026	4229	4222
0.5%	4251.1	1.85292561	4253	4248
1%	4239.9	2.33095117	4243	4236
10%	4190.4	3.16929715	4194	4185

Where:		
	Av. Best Fitness	Average in 300 tests of the fitness of the best robot in generation 300;
	Standard Deviation	Standard Deviation for the Average in 300 tests of the fitness of the best robot in generation 300;
	Max. Fitness	Maximum fitness value obtained in generation 300 in 300 tests;
	Min. Fitness	Minimum fitness value obtained in generation 300 in 300 tests;

C.2.3 - Section 7.4: Experiment S3

The Experiment S3 presented in Figure 7.10 (Section 7.4) is just one of many experiments performed with the simulator configured as described in Experiment S3. In total, it was run 300 times for mutation rate of 0.5%, during 9000 generations.

Mutation Rate:	Av. Best Fitness	Standard Deviation	Max. Fitness	Min. Fitness
0.5%	4353.9	1.59513148	4356	4349

Where:		
	Av. Best Fitness	Average in 300 tests of the fitness of the best robot in generation 9000;
	Standard Deviation	Standard Deviation for the Average in 300 tests of the fitness of the best robot in generation 9000;
	Max. Fitness	Maximum fitness value obtained in generation 9000 in 300 tests;
	Min. Fitness	Minimum fitness value obtained in generation 9000 in 300 tests;

C.2.4 - Section 7.5: Experiment S4

The chart S4.1 presented in Figure 7.11 (Section 7.5) is just one of many experiments performed with the simulator configured as described in Experiment S4. In total, it was run 300 times for Sexual and Asexual reproduction, during 3700.

Reproduction:	Av. Best Fitness	Standard Deviation	Max. Fitness	Min. Fitness
Sexual	4352	1.01846171	4356	4352
Asexual	4356	0	4356	4356

Where: **Av. Best Fitness**	Average in 300 tests of the fitness of the best robot in generation 3700;
Standard Deviation	Standard Deviation for the Average in 300 tests of the fitness of the best robot in generation 3700;
Max. Fitness	Maximum fitness value obtained in generation 3700 in 300 tests;
Min. Fitness	Minimum fitness value obtained in generation 3700 in 300 tests;

The chart S4.2 presented in Figure 7.12 (Section 7.5) is just one of many experiments performed with the simulator configured as described in Experiment S4. In total, it was run 300 times for Sexual and Asexual reproduction, during 600.

Population:	Av. Best Fitness	Standard Deviation	Max. Fitness	Min. Fitness
6 Robots	4271	2.01846171	4275	4267
8 Robots	4302	2.03546816	4305	4298
10 Robots	4318	1.96587146	4321	4316
15 Robots	4323	1.82664546	4326	4319
25 Robots	4339	2.04882256	4341	4336
35 Robots	4343	1.62585825	4345	4341
45 Robots	4343	1.82325496	4346	4341
60 Robots	4349	1.53585212	4352	4347
80 Robots	4356	0	4356	4356
100 Robots	4356	0	4356	4356

Where: **Av. Best Fitness**	Average in 300 tests of the fitness of the best robot in generation 600;
Standard Deviation	Standard Deviation for the Average in 300 tests of the fitness of the best robot in generation 600;
Max. Fitness	Maximum fitness value obtained in generation 600 in 300 tests;
Min. Fitness	Minimum fitness value obtained in generation 600 in 300 tests;

C.2.5 - Section 7.6: Experiment S5

The chart S5.1 presented in Figure 7.13 (Section 7.6) are just one of many experiments performed with the simulator configured as described in Experiment S5. In total, it was run 300 times for each mutation rate (0.5%, 1%, 3%, and 15%), during 600 generations.

Mutation Rate:	Av. Best Fitness	Standard Deviation	Max. Fitness	Min. Fitness
0.5%	4301.5	3.02765035	4306	4297
1%	4332.3	2.83039063	4336	4328
3%	4322.7	2.58413966	4326	4318
15%	4301.9	3.21282154	4304	4295

Where:		
Where:	**Av. Best Fitness**	Average in 300 tests of the fitness of the best robot in generation 600;
	Standard Deviation	Standard Deviation for the Average in 300 tests of the fitness of the best robot in generation 600;
	Max. Fitness	Maximum fitness value obtained in generation 600 in 300 tests;
	Min. Fitness	Minimum fitness value obtained in generation 600 in 300 tests;

C.2.6 - Section 7.7: Experiment S6

The chart S6.2 presented in Figure 7.15 (Section 7.7) are just one of many experiments performed with the simulator configured as described in Experiment S6. In total, it was run 300 times for each mutation rate (0.5%, 1%, 3%, and 15%), during 600 generations.

Mutation Rate:	Av. Best Fitness	Standard Deviation	Max. Fitness	Min. Fitness
0.5%	4323.6	3.33999335	4326	4318
1%	4341.3	2.71006355	4344	4337
3%	4331.3	2.71006355	4334	4327
15%	4289.5	3.27448045	4292	4283

Where: **Av. Best Fitness**	Average in 300 tests of the fitness of the best robot in generation 600;
Standard Deviation	Standard Deviation for the Average in 300 tests of the fitness of the best robot in generation 600;
Max. Fitness	Maximum fitness value obtained in generation 600 in 300 tests;
Min. Fitness	Minimum fitness value obtained in generation 600 in 300 tests;

C.3 - Data Relative to Chapter 8:

C.3.1 - Section 8.2: Experiment E1

The chart Summary of Experiment E1 presented in Figure 8.3 (Section 8.1) is just one of many experiments performed with the real evolutionary system configured as described in Experiment E1. In total, it was run 12 times for the black box controller, during 200 generations. Far less experiments were performed here than with using the simulator in Chapter 7. The reason for this is that it takes more than three hours to perform only one real experiment with the duration of the generations set to one minute.

Controller:	Av. Best Fitness	Standard Deviation	Max. Fitness	Min. Fitness
Black box	4666.948	3.68920588	4671	4660

Where: **Av. Best Fitness**	Average in 12 tests of the fitness of the best robot in generation 200;
Standard Deviation	Standard Deviation for the Average in 12 tests of the fitness of the best robot in generation 200;
Max. Fitness	Maximum fitness value obtained in generation 200 in 12 tests;
Min. Fitness	Minimum fitness value obtained in generation 200 in 12 tests;

C.3.2 - Section 8.3: Experiment E2

The chart Summary of Experiment E2 presented in Figure 8.6 (Section 8.2) is just one of many experiments performed with the real evolutionary system configured as described in Experiment E2. In total, it was run 12 times for the black box controller, during 200 generations. Far less experiments were performed here than with using the simulator in Chapter 7. The reason for this is that it takes more than three hours to perform only one real experiment with the duration of the generations set to one minute.

Controller	Av. Best Fitness	Standard Deviation	Max. Fitness	Min. Fitness
Neural Network	4690.235	3.20538458	4696	4982

Where:	**Av. Best Fitness**	Average in 12 tests of the fitness of the best robot in generation 200;
	Standard Deviation	Standard Deviation for the Average in 12 tests of the fitness of the best robot in generation 200;
	Max. Fitness	Maximum fitness value obtained in generation 200 in 12 tests;
	Min. Fitness	Minimum fitness value obtained in generation 200 in 12 tests;