Evolving Robots:
from Simple Behaviours to Complete Systems

Wei-Po Lee

Ph.D.
University of Edinburgh
1997

Abstract

Building robots is generally considered difficult, because the designer not only has to predict the interaction between the robot and the environment, but also has to deal with the ensuing problems. This thesis examines the use of the evolutionary approach in designing robots; the explorations range from evolving simple behaviours for real robots, to complex behaviours (also for real robots), and finally to complete robot systems — including controllers and body plans.

A framework is presented for evolving robot control systems. It includes two components: a task independent Genetic Programming sub-system and a task dependent controller evaluation sub-system. The performance evaluation of each robot controller is done in a simulator to reduce the evaluation time, and then the evolved controllers are downloaded to a real robot for performance verification. In addition, a special representation is designed for the reactive robot controller. It is succinct and can capture the important characteristics of a reactive control system, so that the evolutionary system can efficiently evolve the controllers of the desired behaviours for the robots. The framework has been successfully used to evolve controllers for real robots to achieve a variety of simple tasks, such as obstacle avoidance, safe exploration and box-pushing.

A methodology is then proposed to scale up the system to evolve controllers for more complicated tasks. It involves adopting the architecture of a behaviour-based system, and evolving separate behaviour controllers and arbitrators for coordination. This allows robot controllers for more complex skills to be constructed in an incremental manner. Therefore the whole control system becomes easy to evolve; moreover, the resulting control system can be explicitly distributed, understandable to the system designer, and easy to maintain. The methodology has been used to evolve control systems for more complex tasks with good results.

Finally, the evolutionary mechanism of the framework described above is extended to include a Genetic Algorithm sub-system for the co-evolution of robot body plans — structural parameters of physical robots encoded as linear strings of real numbers. An individual in the extended system thus consists of a brain(controller) and a body. Whenever the individual is evaluated, the controller is executed on the corresponding body for a period of time to measure the performance. In such a system the Genetic Programming part evolves the controller; and the Genetic Algorithm part, the robot body. The results show that the complete robot system can be evolved in this manner.

Acknowledgements

I am especially grateful to my supervisor, Dr John Hallam, whose patience and understanding has helped me to complete my project. His tolerance and encouragement is unforgettable.

There are a number of scholars and colleagues who have also helped me in various ways. I would like to thank Henrik Hautop Lund for his enthusiastic assistance and discussions, and Dr Peter Ross and Tim Taylor for their valuable suggestions.

I deeply appreciate Professor K. J. Fielding of University of Edinburgh, who has been the most affectionate friend of my family through these years in Edinburgh.

Finally I thank my parents and my wife Shu-Fang who are my strongest backup all the year round.

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research.

Contents

List of Figures

Chapter 1

Introduction

Building behaviour-based control systems that decompose the competence of an agent into multiple task-achieving control modules has now become a serious alternative to the traditional approach in robot design. This approach has been used to construct many real robots acting in real time in the real world. While it is also realised that building a robot by iteratively dealing with the robot-environment interaction is a tough job, especially when the number of modules increases with the complexity of tasks and environments, it would go beyond a designer's capability to construct all the modules and the relationships between them. The idea of *self-adaptation* is thus proposed as an attempt to make the robot learn to adapt to its environment by itself.

Evolutionary computation is a form of adaptation. It has been widely applied to the study of artificial life to evolve simulated creatures acting in simulated worlds where the creatures are demonstrated to be able to change their structures (internal or external) to find the form which best suits their environment. As a result, in the early 1990s, this idea was extended to synthesise situated creatures (i.e., robots) in the real world. This thesis addresses problems in the application of evolutionary algorithms to automate the generation of robot creatures. Topics include the evolution of robust and reliable behaviours, the behaviour transfer from simulation to reality, the scaling up problem, and the co-evolution of robot controller and body.

1.1 Robot Control

1.1.1 Classical Robot Control and the Problems

In the traditional study of artificial intelligence (AI), researchers have presumed that intelligent behaviours are obtained as the results of reasoning about the encoded knowledge and determining appropriate operations to react [Steels 93a, Malcolm *et al.* 89, Maes 92]. Robotics is a subfield of AI, and therefore inevitably inherits many of the principles and assumptions of traditional (classical) AI research. However, treating AI machines as symbol processing systems in classical AI does not succeed in building autonomous robots. The main reason is that in the classical approach, the operation of a robot is regarded as a so-called *sense-think-act* cycle [Malcolm *et al.* 89]. A situation in the environment is first sensed; based on the results, the robot begins to build a model of the world and then construct a plan to reason about what is the best way to act; after determining the operation, the robot then performs the operation in the real world. This approach has been criticised for a number of reasons. The first is that for each cycle the robot has to build a model and to construct a plan before taking any action. This is difficult to perform in practice because the world is continuously changing and unpredictable. For a world filled with uncertainties, it is unrealistic to build a world model for the robot in advance, and it is extremely time-consuming to update an internal world model at any particular time. These problems make the robot unable to act in real time in the real world. In fact, some working examples have shown that it is not necessary to build such an internal world model: when a robot is acting in the environment, only a small part of the environment is relevant to its actions; most of it is irrelevant and does not need particular attention.

The other problem is that, in a symbol processing system, the world is modelled in terms of symbols which are defined in advance, and the meaning of the symbols exists in the form of how they are related to each other and how they are processed. That is, the symbols are defined to be understandable (meaningful) for the designer (observer); they are meaningless to the robot itself. The lack of direct semantic coupling between the robot and its environment results in the problem of how to map the information from sensors to the internally-defined symbols.

1.1.2 Behaviour-Based Control and the Challenges

To overcome the problems caused by using classical AI methodology to build robots, Brooks has proposed his subsumption architecture as an alternative approach [Brooks 86]. Unlike the classical robot control which divides the task of implementing intelligent behaviour along functional lines, subsumption architecture performs behaviour decomposition in which the overall control mechanism is implemented as layers of behaviour modules in an incremental manner. Each module is a fairly simple computational machine and higher-level layers can subsume the role of low levels by suppressing their outputs. Later on, this distributed control idea was adopted by many researchers to build different robot systems (e.g.,[Payton 86, Arkin 89, Rosenblatt & Payton 89, Verschure *et al.* 92, Steels 93b]), and this type of control is now well-known as behaviour-based control. In behaviour-based control, the "think" phase has been removed: the robot does not model the world and construct the plan at each time step any more. The control system is now considered as a network of task-achieving control modules called behaviours in which the sensor information is connected directly to actions as in subsumption architecture. This approach not only results in a robot capable of acting in real time but also the performance of the overall control system does not depend on the weakest link as in the functional decomposition approach. In addition, behaviour-based control solves the symbol processing problem mentioned above: the world now becomes meaningful to the robot itself rather than the observer, because the robot can directly decide what to do by looking at the world and acquiring the relevant information through its own sensors. Figure 1.1 contrasts the decompositions of classical and behaviour-based control.

Although behaviour-based robots have been proven to be successful in acting in the physical world, the extension of this approach to design complete autonomous creatures for more complicated tasks still presents some challenges at different levels: *micro*, *macro*, and *multitude* levels, as the original proposer Brooks points out [Brooks 90]. The micro level challenge concerns the robustness of a single behaviour module; issues at this level include how to code a behaviour which is always capable of dealing with the complexity of real world by concentrating on what is relevant, to guarantee achieving a specific task. The macro level challenge is actually the problem of action selection; it

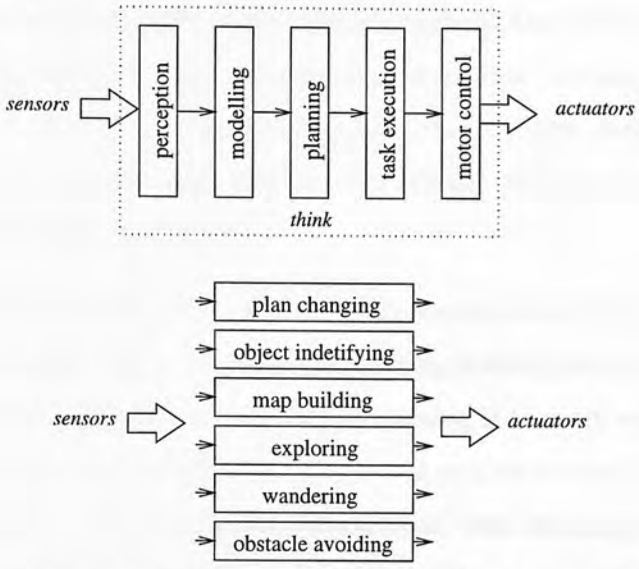

Figure 1.1: The architecture of classical robot control v.s. behaviour-based robot control (after [Brooks 86]).

concerns how to decide, for a system including multiple behaviours to handle a variety of situations, which behaviour or behaviours should be active at any particular time, to achieve various tasks. And the multitude level challenge is to consider how to deal with the interaction between such robots, once we can design a single physical robot operating successfully in a real world (i.e., once the problems at the above two levels are solved).

1.1.3 Robot Learning

As we can observe, the above problems are mainly derived from the fact that the robot must face unforeseen situations, such as using unreliable sensors and actuators to act in an incompletely known world, by using the control systems given by the designer. On the one hand, the designer must correct the control programs for the robot again and again by observing and analysing its behaviours, in order to achieve the task. On the other hand, the robot is responding to the environment by using the combination of its own sensors, actuators, and the designer's brain. In this way, the task is achieved only in the situations where the human designer's brain and the robot's body are fully integrated. This is not efficient for either of them.

One solution to cope with these problems is to adopt *learning* methods in which the robot is expected to self-improve its own behaviours to fit the different situations which it experiences, through the robot-environment interaction, without explicit programming. This is a more efficient strategy since now it is the robot itself which determines how to change its internal control structure to fully exploit its noisy sensors and actuators, to adapt to the environment.

Based on the amount of built-in structure (which is required to be pre-defined) in robot learning, Brooks and Mataric categorise the existing learning methods into strong and weak learning [Brooks & Mataric 93]. Strong learning is to learn everything; it begins with a control system that contains no behavioural module or any predefined structure – it learns the control system as a whole. In contrast, weak learning is to learn a control strategy given predefined capacities or structures. There is a tradeoff between strong and weak learning: the less built-in knowledge, the more difficult to learn; while the more predefined knowledge, the more restrictions in learning. How to find a suitable intermediate approach is case-dependent, relying on the designer's decision.

1.1.4 Learning by Evolution

Evolution-based algorithms, inspired by the Darwinian principle of selective reproduction of the fittest, are a specific kind of machine learning approach. They simulate the natural evolution process in which members of populations change their structures to adapt to the environment, in order to survive. In the last few years, the evolutionary algorithms have been proposed to synthesise robot control systems [Brooks 92, Cliff *et al.* 93], and this research field is called *evolutionary robotics*. The central idea of evolutionary robotics is that by defining a fitness function as the selective pressure, the control systems which drive the robot to perform the desired behaviour well will emerge over generations, through competition between members of populations. Since this approach is meant to aid the designer in programming robots, it must encounter those challenges in robot programming – dealing with the complex robot-environment interaction to create robust and reliable control systems, and scaling up to achieve more complicated tasks. In this dissertation, we intend to explore some specific issues in evolutionary robotics. Topics investigated include:

- How to employ evolutionary algorithms to evolve robust and reliable robot behaviours.
- How to evolve single behaviours for physical robots acting in the real environment.
- How to evolve complex behaviours in terms of dealing with the problem of behaviour selection by evolutionary algorithms.
- How to evolve a complete creature, including the control system and the corresponding robot body.

For these topics, we analyse the existing approaches, propose our methodology and conduct experiments for investigation and verification. Each topic is arranged in a different chapter in this thesis accordingly.

1.2 Evolutionary Computation

The theory of *natural selection* presented by Darwin in his most famous book *The Origin of Species* explains the cause of evolution: in any environment members of species struggle and compete for available resources, and it is those most adapted to their surroundings that survive [Darwin 59]. This theory profoundly influenced the early biologists and is currently playing an important role in the study of artificial intelligence. As more and more evidence shows, the key matter in AI research, intelligent computation, often requires adaptive search mechanisms which can dynamically adjust the searching direction according to the features of the problem itself. *Evolutionary Computation* (*Evolutionary Algorithm*) techniques, representing algorithms which simulate the natural evolution process on computers, were thus developed to construct more dynamic models of computational intelligence.

Evolutionary algorithms include genetic algorithms [Holland 75, Goldberg 89], evolutionary programming [Fogel *et al.* 66, Fogel 95], and evolutionary strategies [Rechenberg 73, Schwefel 81]. Although different in detail, all share the same concept of simulated evolution. In an evolutionary algorithm, a population of problem solutions is manipulated simultaneously (conceptually, at least). The performances of members of a set of candidate solutions to a specific problem are evaluated, then the new candidate solutions are generated by selecting current candidates and applying some operations on them. How to select current candidates to be the "parents" depends on a criterion which is generally based on the candidates' performance. The new candidate solutions are then evaluated and the same cycle continues until a certain termination criterion is met. As mentioned above, in this thesis, we will explore how to employ this approach for the design of robot systems and the considerable details of evolutionary algorithms are introduced in the next chapter.

1.3 Contributions of this Work

This thesis focuses on exploring the use of evolutionary computation techniques in designing robots. The major contributions of this thesis can be summarised as follows:

- *An evolutionary robotics framework is constructed for evolving reliable and robust controllers in relatively short time.* This framework includes two components: a task-independent Genetic Programming sub-system and a task-dependent evaluation sub-system. In our framework, a special circuit representation is designed for robot behaviour controllers. Successful results show the advantages of this representation: it enables the evolutionary system to efficiently evolve desired robot behaviours within a relatively small number of generations, using a relatively small population size (saving evolution time); it is easy to execute (also saving evolution time); and easy to translate to other languages for different robots. A series of experiments is also conducted to investigate how to evolve high performance controllers with modest computation cost.

- *The robot controllers evolved by our simulation can be transferred to real robots without losing performance.* The major goal of evolutionary robotics research is to produce controllers working on real robots. However, the use of simulation to evolve controllers within reasonable time makes this difficult to achieve. In this work, a specific simulator is developed and included in the evaluation sub-system to achieve both. The results prove that our GP framework, with its specific robot simulator, can evolve high quality controllers for real robots.

- *A methodology is presented to scale up our system to evolve hierarchical control architectures for more complex tasks.* It involves adopting the behaviour-based control architecture and evolving separate behaviour primitives and coordinators. This allows robot control systems for more complex tasks to be evolved in an incremental manner. The approach presented not only makes the fitness functions easy to define, but also the task easy to achieve. Therefore, the framework can be used to build (evolve) a behaviour library for a robot; and the behaviour controllers included can be arbitrarily combined by evolving coordinators to achieve different kinds of tasks.

- *The framework is extended to co-evolve robot controllers and morphologies.* Based on our observation that both controller systems and body plans can influence the behaviours of robots, a hybrid GP/GA approach is developed to co-evolve them. In the approach presented, the above evolutionary framework is extended to include a Genetic Algorithm sub-system for evolving robot morphologies. Experimental results show the promise of this approach: both control systems and robot bodies have adapted to the given environment to achieve the task. A series of experiments is also conducted to thoroughly explore the relationships between different parts of a robot body for a specific task.

This thesis aims to show that genetic programming can indeed be used to synthesise robot controllers in a reliable way, rather than to argue that it is the fastest way to create controllers or that it can produce the most compact controllers. Thus, there is no direct comparison in this thesis between the results of genetic programming methods and the results of methods such as hill-climbing or skilled human design. But the performance graphs do suggest strongly that genetic programming is very much better than random search; the evolved controllers do perform their intended tasks adequately in the real world; and the results of co-evolving the controllers and the physical parameters of the robot also suggest that the performance of a controller does depend on and exploit the characteristics of a specific robot. This suggests that the search for a reliable controller is therefore not a trivial task.

1.4 Overview of the Thesis

The previous sections briefly describe the motivation and summarise the contributions of this thesis. This section provides an overview of the contents and the structure of this thesis. After the chapter on evolutionary algorithms and the chapter on evolutionary robotics with a review of related work, there are chapters to explore evolutionary robotics arranged so that each chapter documents a stage of the exploration: from the evolution of simple robot behaviours to complex robot behaviours, and finally to complete robot creatures. The contents of each chapter are summarised as follows:

Chapter 2 gives a general introduction of evolutionary algorithms. It starts with explaining how an evolutionary algorithm works and then focuses on the techniques in one of the evolutionary algorithms, genetic programming, which is applied to evolve the robot control systems in the subsequent chapters.

Chapter 3 presents a literature review of evolutionary robotics. The review includes work using different forms of control mechanisms, such as neural networks, classifier systems, computer programs, as well as various experimental approaches, such as pure simulation, complete on-line evolution, and the simulation-reality transfer.

Chapter 4 describes the framework of our genetic programming system which can evolve simple behaviour modules in simulation, and studies the trade-off between the performance of the evolved results and the computational time. Experiments focus on seeking the most economic way which guarantees the success of the evolved robot behaviours with least computation resources.

Chapter 5 discusses the advantages and disadvantages of off-line and on-line methods in evolutionary robotics. It suggests using simulation to reduce evolution time and stresses the result must be verified on real robots. Experimental results demonstrate that the technique of simulation-reality transfer can be used to evolve successful behaviour controllers for real robots.

Chapter 6 analyses the difficulties in scaling up evolutionary robotics. It suggests that instead of handcoding the controller or evolving the controller as a whole, one can use a compromise approach which involves task decomposition by a human designer and the evolution of decomposed components by an evolutionary system. Experimental results show the promise of such a combination.

Chapter 7 discusses the evolution of a complete robot creature. It describes a hybrid GA/GP approach to co-evolve robot controller and body plan. Experiments are conducted to show how such an approach works, and to explore the relationships between the different components of a robot body.

Chapter 8 summarises and concludes the thesis. It also indicates some potential research directions in evolutionary robots.

Chapter 2

Evolutionary Algorithms

Evolutionary Algorithms are a kind of computer algorithm which simulate a natural evolution process to *evolve* solutions for problems. It has become increasingly popular to employ this kind of algorithm to solve problems in different domains. In this thesis, we intend to construct a system, based on the idea of simulated evolution, to automate the design of robots. Before detailing our research paradigm, we will introduce some evolutionary algorithms and the relevant techniques in this chapter. The first section explains the general concept of Evolutionary Algorithms and various forms of this kind of mechanism; the second section describes details of the specific genetic techniques used in this thesis; and then, different ways to parallelise the mechanisms of simulated evolution will be characterised in the last section as they can dramatically reduce execution time and enhance performance.

2.1 A Gentle Introduction

Evolutionary algorithms (evolutionary computation) represent the kind of algorithms which simulate the process of natural evolution to search for the fittest through selection and re-creation over the generations. Based on different philosophies, three main streams of research are currently used in evolutionary computation: *evolutionary strategies*, *evolutionary programming*, and *genetic algorithms*. Although sharing the same concept of simulated evolution, they are developed independently and emphasise the adaptive changes at different levels. Evolution strategies (ES) stress behavioural changes at the level of the individual; evolutionary programming (EP) emphasises behavioural changes at the level of species; and Genetic algorithms (GAs), including traditional genetic algorithms and recent genetic programming, emphasise the genetic operations at the chromosome level. We first explain the shared concept among these approaches, and then describe their individual concerns.

Evolutionary algorithms are population-based optimisation processes; they are based on the collective learning process within a population of individuals/species each of which represents a search point in the space of potential solutions for a given problem. In general, the process of an evolutionary algorithm mainly includes *initialisation*, *reproduction*, *re-creation*, and *selection*. The initialisation is usually to randomly select a set of initial points in the search space. The number of points (population size) varies from a small (e.g., less than 10) to a large number (e.g., several thousands), depending on the difficulty of the specific problem. By manipulating a population of potential solutions, an evolutionary algorithm can search various regions of the solution space simultaneously. Reproduction is an obvious way to propagate individuals/species; it is accomplished through transferring the genetic features of the members in the current generation to the next generation. Based on the fitness brought about by the environment, the members with better fitness are favoured to reproduce more often than the worse ones. Re-creation is a method to introduce variety for the individuals/species; it adjusts the genetic features which are inherited from their parent population. Different evolutionary algorithms involve different types of re-creation. For example, in evolutionary programming, mutation is strictly the only way to change the members' components; while in genetic algorithms, re-creation usually includes point mutation, and crossover (this is specially referred as recombination in GAs). Selection is the process in which different individuals/species compete for finite resources; in an evolutionary algorithm, this process can be simulated in an entirely random or stochastical way based on the fitness.

Thus, the outline of an evolutionary algorithm can be described as follows. An initial population of candidate solutions is generated at random; and each member is evaluated by a problem-dependent criterion which measures how good this member is for the specific problem. The result is quantified by a *fitness function* and is used as this member's survival fitness. The fitness function here is similar to a cost function used in other search-based techniques; it normally gives a real value to indicate how fit a candidate solution is for the problem. Essentially, an evolutionary algorithm is trying to maximise/minimise the value given by the fitness function. Once all the candidates in a population are evaluated, an evolutionary algorithm employs a certain selection scheme to select a subset of the current population to act as parents to generate a new population. There are various selection schemes used to choose parents. In general, the members with best fitness are preferred, but the relatively worse ones are not excluded, in order to maintain the diversity of the population. Therefore, the fitness corresponding to different members are regarded as the probabilities of survival and certain probability-based methods are used to choose parents. After that, an evolutionary algorithm applies a set of operators on the chosen parents to alter their features to form a subsequent population. As described above, the operations can be recombination or random re-generation, depending on what kind of algorithm (ES, EP, GA) is employed. The above cycle is an iteration (or generation) of an evolutionary mechanism; it is repeated until a certain termination criterion is met (e.g., a solution with a fitness better than the expected value appears; or the algorithm has been executed for a pre-defined number of generations). Figure 2.1 illustrates the general flow of an evolutionary mechanism.

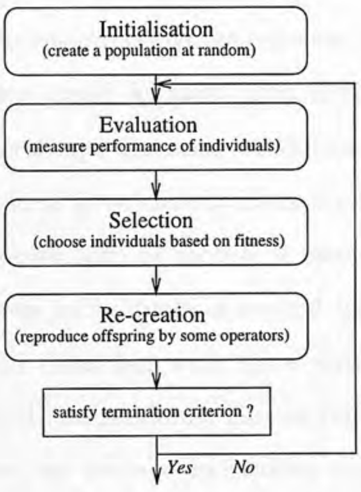
Initialisation
(create a population at random)
Evaluation
(measure performance of individuals)
Selection
(choose individuals based on fitness)
Re-creation
(reproduce offspring by some operators)
satisfy termination criterion ?
Yes
No

Figure 2.1: The general flow of a simulated evolution mechanism.

The above passage has explained the common concept of an evolutionary algorithm. As mentioned earlier, different evolution-based algorithms emphasise different evolution philosophies which lead to differences in their implementations. The individual characteristics of different evolutionary algorithms are briefly described below.

2.1.1 Evolution Strategies and Evolutionary Programming

The first kinds of algorithm in simulating evolution were independently adopted; they include evolutionary strategies by Schwefel and Rechenberg in Germany [Schwefel 81, Rechenberg 73], and evolutionary programming by Lawrence Fogel in the United States [Fogel *et al.* 66]. Unlike GAs, these kinds of evolutionary algorithm emphasise the behavioural link between parents and offspring.

In the ES model, a member of the population is considered as an individual which is constituted by a set of parameters: each component of an individual is viewed as a behaviour parameter rather than a gene. The evolution strategies aim at creating offspring whose behaviours are similar to their parents and at optimising the components of behaviours. Typically, the relation between different parameters of an individual is unknown. Thus, in an ES system, each parameter is represented as a real value which has no connection with others. As a result, an individual in ES is then a fixed-length, real valued string.

As described previously, evolutionary strategies model evolution at the level of the individual and emphasise the behavioural linkage between parents and offspring. Hence, two kinds of genetic operation could happen. The first involves only one parent; it creates a new individual by adding a Gaussian random variable with zero mean and a standard deviation (adaptable or pre-defined) to each component of the parent. The second involves different parents (two or more); it recombines the parameter sets of different parents to create new individuals in several ways. This operation is much like the crossover in standard GAs, but with more options because of its real value representation. For example, it can randomly choose the values from multiple parents to generate multiple children, or average individual components from parents by a specific weighting strategy. Some implementations also explore whether the selected parents should participate in the competition for survival. More details can be found in [Back 96, Back & Schwefel 96].

Similar to evolutionary strategies, evolutionary programming emphasises the behavioural linkage between parents and offspring. But EP models evolution at the level of species rather than at the level of individuals as ES. In EP, a population is regarded as composed of different species which compete with each other (a member in an EP population is regarded as a kind of species). An offspring is generally similar to its parent in the behaviour-level with only slight variation. Thus, different species (members) in the population are considered to have independent behaviour features and then are not allowed to mix with each other. In other words, the creation of a new offspring involves only one parent, and the offspring is derived from the selected parent with different forms of mutations.

EP often uses finite state machines as its representation because its original inventor, Lawrence Fogel, thinks that this kind of machine involves intelligent behaviour which requires the ability to predict environment conditions and to generate suitable responses for the given goal [Fogel *et al.* 66]. As mentioned earlier, no recombination is involved in EP; offspring machines are created by randomly mutating parent machines. Given that a finite state machine has a number of states, an initial state, a collection of transitions between the states, and an output associated with each transition, five possible modes of random mutation are used: adding a state, deleting a state, changing the initial state, changing a state transition, and changing an output symbol on a transition. The selection of mutation operation is based on the principle that the distribution of child structures would approximate a normal distribution around the parent. Details can be found in [Back 96, Fogel 95].

2.1.2 Genetic Algorithms and Genetic Programming

Genetic algorithms as they are known today were first developed by John Holland [Holland 75]. They are currently the most popular form in modelling evolution processes for optimisation problems. This model emphasises adaptive changes at the gene level; it treats a member in the population as the chromosome of an individual. A chromosome is constituted by a string of genes; each of them is regarded as carrying a genetic feature of an individual. An individual with better fitness is said to have some genetic features that are capable of solving the problem, and these features are expected to be propagated to the subsequent population by means of duplicating or recombining the gene sequence of parent populations.

In genetic algorithms, an individual is represented as a fixed-length string, often, but not necessarily, in the form of a bit string. The most important thing in designing a representation is to ensure that a potential solution in the search space for the problem to be solved can be expressed as a string of the developed representation. It often requires considerable knowledge and insight for the problem.

As other evolutionary algorithms, the GA is operated as an iteratively cyclic mechanism which includes a sequence of selecting parents and creating children, after the initialisation phase. Selection involves probabilistically choosing individuals from the current population as parents to generate offspring to form a new population. Different selection schemes have been proposed; they will be described in a later section. Unlike evolution strategies and evolutionary programming which rely on mutation to produce children individuals, in GAs, crossover is the major operator to create children. The operations of reproduction and mutation produce relatively small numbers of offspring; they are secondly operators. The operation of crossover is typically to recombine two parent individuals which are selected independently, to create two children individuals. The recombination is generally implemented as alternately copying gene sequences, separated by randomly chosen crossover points, from the selected parents. One or two point crossover is the method often employed in GAs. Figure 2.2 illustrates an example of creating new individuals by the use of one-point crossover.

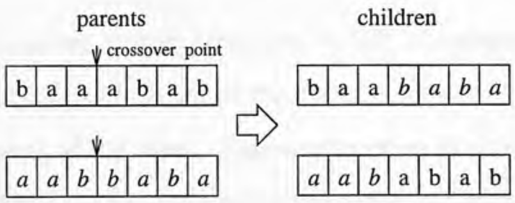

Figure 2.2: An example of one-point crossover in GA.

A variant of genetic algorithms, named *genetic programming*, was recently invented by John Koza, and it is popularity is currently increasing in the community of evolutionary algorithm research [Koza 92, Koza 94, Koza *et al.* 96a, Koza *et al.* 97]. Genetic Programming is similar to traditional genetic algorithms in its concept that the change mostly happens at the gene level, but is different from it in the representation and the implementation of the corresponding genetic operators. Hence, GP can be regarded as an extension of GA: it applies techniques used in GA to evolve dynamic-length tree structures rather than fixed-length strings as standard GA does. In GP, an individual is represented as a tree; this is inspired by the fact that a program in any computer programming language can be expressed as a parse tree with respect to the syntax of the language. As in GA, the tree-like individuals with better fitness are expected to pass their genetic features to the child population through the application of replication or recombination. Recombination in GP involves the swapping of subtrees. This new evolutionary approach has demonstrated its potential by evolving programs in a wide range of application, such as circuit design [Koza *et al.* 96b, Hemmi *et al.* 94], pattern recognition [Johnson *et al.* 94, Nguyen & Huang 94], computer animation [Sims 94b, Sims 94a], and signal processing [Alcazar & Sharman 96].

In this thesis, all of the experiments in evolving robot control systems are done based on GP. We will detail the techniques of this approach in a separate section.

2.2 Genetic Programming

GP is an extension of traditional GAs with the basic distinction that in GP the individuals are dynamic tree structures rather than fixed-length vectors. It aims to evolve dynamic and executable structures which are often interpreted as computer programs, to solve problems without explicit programming.

As in computer programming, a tree structure in GP is constituted by a set of non-terminals which are the internal nodes of the trees, and a set of terminals which are the external nodes (leaves) of the trees. The construction of a tree is based on the syntactical rules which extend a root node to appropriate symbols (non-terminal and/or terminals) and each non-terminal is extended again by suitable rules accordingly, until all the branches in a tree end up with terminals. The search space in genetic programming is the space of all possible tree structures which are composed of non-terminals and terminals.

As the original GP inventor John Koza mentions in his book [Koza 92], there are five major preparatory steps for one to apply GP to solve a problem. They include the determination of

- the terminal set
- the non-terminal set
- the fitness measure
- the parameters for controlling a run
- the termination criterion

The first two steps are about the genetic representation; they are to define the syntactic rules (organising the architecture of a tree) for the problem to be solved. These two steps correspond to the procedure of specifying the representation scheme for traditional GAs. The other three steps are about operating an evolutionary algorithm; similar steps can also be found in other evolutionary algorithms (i.e., ES, EP, GAs).

After the preparatory steps, which are the kernel events in the application of genetic programming, the evolutionary mechanism breeds individual tree-structures to solve problems by executing the following steps:

- Randomly generating a population in which each individual is constituted by the pre-defined non-terminals and terminals and is correct in syntax.
- Executing each tree-individual in the population and assigning a fitness value, based on the specified fitness function, to it to indicate how well this individual solves the target problem.
- Creating a new population by applying some operators on members of the old population which are chosen with probabilities based on their fitness. Duplication and recombination are two primary operators in GP; mutation is only the minor operator.
- Terminating the run and giving the result, according to the termination criterion specified previously.

The second and third steps are iterated until the termination criterion, which is normally defined as a certain number of generations or when a fitness threshold is met.

2.2.1 Representation

The basic structure of an individual in GP is a tree, so the first step to create a GP population is to define the sets of non-terminals (functions) and terminals that will comprise the evolving tree. As in a rewriting system in which a non-terminal will be substituted by a set of symbols, in GP, each function included in the function set must take a specified number of arguments for the corresponding branches in a tree. Terminals take no argument by definition. In this way, functions and terminals occupy the internal and external nodes of the trees, respectively; and the overall structure of a tree is determined by the number of arguments to the functions. For example, if a function set is defined as $\{+, -, *\}$ in which each of the functions has two arguments and performs the usual arithmetic operation, and if a terminal set is defined as {X, Y, $\Re$} in which X, Y are numerical variables and $\Re$ represents the set of real numbers, the typical tree individuals look like the ones in Figure 2.3. A tree is equivalent to a parse tree which most compilers construct internally to represent a given computer program. Thus, evolving individuals in this form is equal to manipulating computer programs genetically.

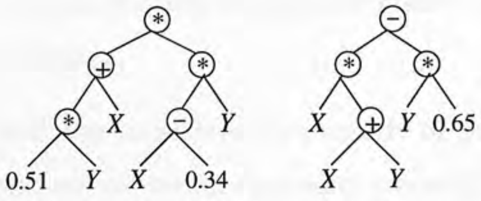

Figure 2.3: Examples of randomly generated tree individuals. The function set is $\{+, -, *\}$ and the terminal set is {X, Y, $\Re$}.

Defining Functions and Terminals

According to Koza (in [Koza 92]), there are two desirable properties in defining functions and terminals to form the tree individuals in GP. One is *sufficiency* which is to ensure that the defined functions and terminals are capable of expressing the solution to the problem. The other is the *closure* property which guarantees the consistency of the data value or type returned by a function or a terminal. That is, each function in the function set should be well defined to accept any combination of components (function or terminal) it may encounter.

Depending on the problem, defining a function set and terminal set which satisfy the property of sufficiency may be obvious or may require considerable insight. In some domains, the requirements for sufficiency are well-known. For example, for the task of evolving combinational logics to achieve some specific goal, the function set including logic operators and, or, not, is known to be sufficient for realising any boolean function; and the terminals can be defined to include the related input variables. However, sometimes it is not clear how to define these sets – some knowledge and understanding about the problem is necessary. The knowledge and understanding is not related to any specific theory at all but the problem itself.

As in traditional computer programming, each function and terminal in GP has an associated type, such as an integer or boolean. When a function or a terminal is called by others, it returns a value to the calling function and the types of the values passed through the tree must be consistent to guarantee the tree to be executable. Koza thus defines the closure property such that each of the defined functions is able to accept its own arguments with any value or data type that may possibly be returned by any function or terminals. With this property, the new offspring can be created by using the operator of crossover to swap subtrees at arbitrary points, and as a result the new trees will be still correct in syntax.

In general, a straightforward way to achieve the property of closure is to use a single return type for all the functions and terminals, and to carefully handle certain special situations such as calculating the result of a numerical variable divided by zero or the square root of a negative number. Typically, GP users have to pre-define some special operations, such as returning a constant in the former case or giving the square root of the absolute value of a negative variable in the latter case, to deal with cases like these.

Constrained Syntactic Structure

Although using a single return type makes GP easy to manipulate, it also constrains the development of a tree. For some problems, it is necessary to include different return types for the tree components. Koza thus introduces the concept of *constrained syntax structure* in his book [Koza 92] to allow components of different types to appear in the same tree. This is also one of the fundamental concepts of *strongly typed genetic programming* [Montana 95]. According to [Koza 92], a constrained syntax structure is a subtree in which the construction of this subtree is based on a set of problem-specific syntactic rules. In this case, the crossover is not allowed to happen at arbitrary points; it must be defined to enforce all syntactic restrictions which are introduced by the type constraints. This is usually implemented as selecting a subtree from one parent and determining its return type, and then choosing a crossover point in the second parent from only those with the same return type. In this way, swapping subtrees can then always guarantee the offspring to be correct in syntax.

2.2.2 Initialisation

As in other evolutionary algorithms, the initial phase in operating GP is to randomly generate a population of individuals. As GP uses dynamic tree structure as its representation, to create an initial population thus involves determining the length (depth) and the way of growing a tree.

In practice it is necessary to restrict the size of a tree when one is using GP to solve problems. Without a limitation on size, the tree evaluation will soon saturate the computational resources. In general, the limitation on tree size can be done by specifying a maximal depth or a maximal number of nodes for a tree. As noted in [Koza 92], any reasonable limitation on tree size will not be a factor affecting the development of trees since the number in the search space will still be extremely large. In fact, for most of the experiments in his book, Koza uses 6 and 17 as the default constraints on depths for trees in initialisation and after crossover, respectively. And he claims that the above values are enough to evolve expected solutions for most of the problems described in his books.

Two ways are normally used to generate trees; one is the *full* method and the other is the *grow* method. The former is to create a full tree in which the length of a path from root node to any terminal must be equal to the specified maximum depth value. And the latter is to allow the tree to grow arbitrarily but to meet the maximum depth constraint. In the latter case, the sizes and shapes of trees generated by the grow method can be quite different.

2.2.3 Fitness Measure

In genetic programming, to evaluate a tree individual is to execute its corresponding code in the environment of the particular problem. The fitness is measured in terms of how well this individual performs during the evaluation process. This is normally done by pre-defining a fitness function which quantitatively describes those matters relevant for the target task, and accumulating the quantities described during the evaluation to be the fitness.

In general, an individual is evaluated over a number of different situations which are called *fitness cases* in GP. Different fitness cases may represent a set of samples for different values of an independent variable, or a set of samples for different initial conditions of a system. For example, in the case of evolving robot controllers, the different fitness cases may represent different starting positions for the robot. The fitness is then measured as the sum or the average over the different cases. This is for evolving more general solutions. If the set of fitness cases are quite large, sometimes a random sampling technique is employed to save evaluation time [Koza 92] (We also conduct some experiments in Chapter 4 to verify the usefulness of random sampling techniques in evolving robots).

Since the size of an individual is variable in GP and the GP solutions tend to grow in length, those solutions with best fitness and short in length are preferred. Thus, it becomes common to include the *parsimony* pressure, which is measured by counting the nodes in a tree, in a fitness function to encourage shorter solutions. Some GP researchers are currently investigating different ways to inject this pressure to the evolution [Kinnear 93, Blickle 96].

2.2.4 Selection Methods

Evolutionary algorithms simulate natural evolution processes in which the fitter members of the population have higher probabilities of producing offspring genetically. Selection criteria are thus implemented in evolutionary algorithms to play the role of choosing the fitter members for the creation of offspring.

There are many different selection methods based on the fitness; they mainly include *fitness-proportional selection*, *rank-based selection*, *tournament selection*, and *local selection*. Fitness-proportional selection is the original selection method proposed for genetic algorithms by John Holland [Holland 75]. In this method, the probability of an individual being selected as a parent is directly proportional to its fitness value. It causes the selection probabilities to strongly depend on the scaling of the fitness values in the population. For example, if the worst and the best fitness values are 1 and 10 in a population respectively, the probability of the best individual being selected is then ten times than the worst one; however, if the worst fitness is 1000 and the best fitness is 1010, the probabilities of the best and the worst individuals being selected are almost identical. This undesirable property is due to the fact that fitness-proportional selection method is not translation invariant. Some scaling methods have been proposed to overcome this property (e.g., [Grefensette & Baker 89, Goldberg 89]).

Rank-based selection was first suggested by Baker as a way to eliminate the disadvantage of the fitness-proportional selection method [Baker 85]. In a rank-based selection scheme, the individuals in the population are sorted first, according to their fitness values; this requirement of global information results in implementations of this selection method being slower than those without sorting. The probability of an individual being selected as parent is now based on its relative rank in the population (rather than the numerical fitness value). Different types of rank-based selection methods have been developed; the most popular form is the one proposed in [Whitley 89] in which a *bias* factor is involved to control the selection pressure.

In tournament selection, a group of individuals is chosen randomly from the whole population and the individual with best fitness value is selected as a parent. The number of individuals in a group is the so-called tournament size. The tournament selection method has become more and more popular because its selection is based on rank and it only uses local information – no extra sorting on the fitness values of the whole population is required as in the rank-based selection method. The evolution experiments in this thesis employ tournament selection.

Local selection methods are typically used in massively parallel genetic algorithms (or cellular GAs, described in a later section) (e.g., [Collins & Jefferson 91b, Gorges-Schlenter 92]). In general, an algorithm using this type of selection arranges the individuals on a toroidal, two dimensional grid, with one individual at a grid position. Selection occurs locally at each grid position: the competition is among a small number of neighbouring individuals with this individual as centre. A typical implementation is that the parents chosen to produce a new individual at a certain grid position are the ones with the best fitness during a random walk starting from that position.

2.2.5 Creating Offspring

As in conventional genetic algorithms, reproduction and crossover are two primary operators in GP to create offspring. When reproduction happens, it involves only one parent tree which is selected from the parent population according to a certain selection criterion, and this parent tree is simply copied to be one of the new population members without any alteration.

Crossover is the major operator to create most of the offspring. It recombines two selected parent trees to generate two children. When this operation happens, two trees are chosen, based on their fitness, from the current population, then a randomly selected subtree is identified in each of them and the subtrees are swapped. Since the crossover points are randomly chosen, the sizes and shapes of the offsprings are generally different from their parents. A typical crossover operation is illustrated in Figure 2.4.

The operation of crossover is based on the belief that by recombining randomly chosen parts of those trees with better performance, we can produce new tree individuals which are even fitter for the problem. In GP, the terminals (leaf nodes) typically represent the input variables without involving specific functions; thus, in order to generate fitter

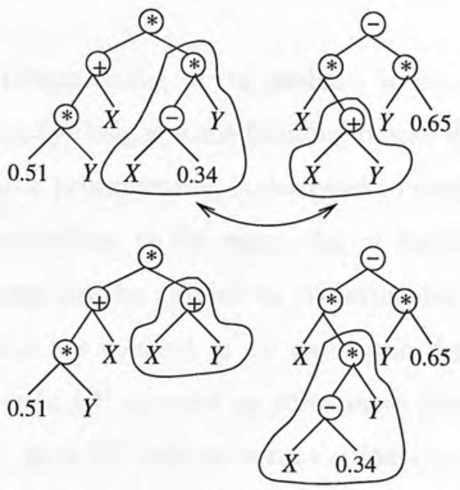

Figure 2.4: An example of crossover in GP. Two new trees are created by swapping subtrees of parents.

solutions, the GP user normally specifies higher probabilities on the internal nodes (functions) to allow the subtrees which carry certain functions to be exchanged more often than the terminals.

If the trees include constrained syntactic structures as described earlier, crossover must be performed carefully. As mentioned, two nodes of crossover points must have the same return type to guarantee the correctness of the produced offspring. In this thesis, we define some constrained syntactic structures on our tree representation; the details will be described in the corresponding chapters.

As in GAs, mutation only plays a minor role in creating a new population in GP. This operator happens on one tree; it picks a subtree from the selected parent, deletes it, and generates a new subtree at random to substitute for the removed one. Since the operation of swapping subtrees in crossover can provide a similar ability of regenerating a subtree to mutation, some GP work even does not use mutation at all. However, it is worth noticing that some recent ongoing work in GAs suggests that mutation may be more important than it is now considered (e.g., [Hinterding *et al.* 95]), and a similar trend is predictable in GP.

2.2.6 Automatic Defined Functions

In traditional computer programming, if the problem to be solved becomes complicated, the programmer usually designs some building blocks (i.e., subroutines) to simplify the programs. Genetic programming is designed to evolve tree-like programs as a way of automatic programming, so the same idea of building blocks in traditional computer programming can also be applied to the evolution paradigm. In his book [Koza 94], Koza introduces the concept of an *automatic defined function* (ADF) to tackle the scaling problem in GP in order to solve more complicated problems. According to his definition, an ADF acts as a subroutine that is dynamically evolved during a GP run and it may be called by a calling program which acts as a main program in a GP tree. Both the called ADF and calling main program are evolved simultaneously.

The simplest way to implement an ADF is to establish constrained syntactic structures for the individuals – they have a fixed architecture and crossover is restricted to happen only between the same type of ADFs or between main programs. Each tree individual contains multiple branches; some of them are ADFs and one (or more) is the main program. Figure 2.5 illustrates an example of a tree individual including ADFs and the main program.

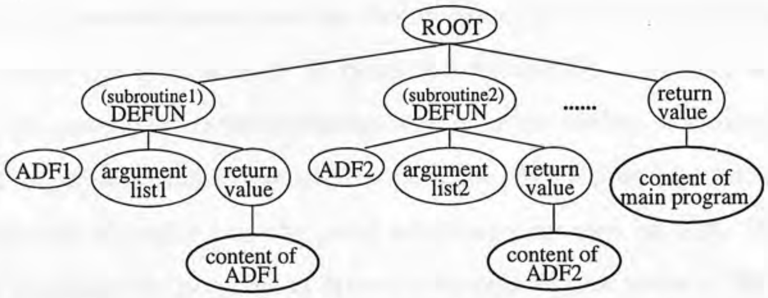

Figure 2.5: An example of a tree individual in GP with ADFs (after [Koza 94]).

In this figure, the DEFUN nodes represent the roots for subroutines (ADFs); they appear only to maintain the structure of such trees. The first branch of each subroutine stands for the name of this subroutine; it is used as a non-terminal symbol in the main program. The second branch of a subroutine is a list of arguments for this subroutine; they are typically localised dummy variables. The third branch contains the content of the subroutine; its return value is the evaluation result of this subroutine. All the names of ADF (e.g., ADF1, ADF2) are included in the non-terminal set of the main program. Whenever any of them is called in the main program, the actual values of its arguments in the main program are provided to substitute the dummy variables in the subroutine and then the corresponding ADF is evaluated once. The outputs of the whole tree are defined to be the values returned by the main program. During evolution, all the contents of ADFs and the main program are allowed to change: they are co-evolved. In this way, some useful information can be efficiently reused and then the performance of a GP system is enhanced. In [Koza 94], Koza has shown many examples in which GP with ADFs is able to solve more difficult problems and it solves problems faster than the plain GP. Similar concepts can also be found in [Angeline & Pollack 93b, Rosca & Ballard 94].

2.3 Parallelising Simulated Evolution

Although evolutionary algorithms have been proved to be promising search approaches and have been applied successfully to different problem domains, two of their inherent features must be improved in order to solve more difficult problems. The first is *premature convergence*. As is well known, one of the attractive features of an evolutionary algorithm is that it can quickly concentrate on searching promising areas of the solution space. But, this feature sometimes has the negative effect that the EA loses population diversity before the goal is met. In other words, the EA converges to local optima. Although the genetic operator, mutation, can offer the ability of maintaining diversity of population, it performs a destructive operation. With a high mutation rate, an EA can increase the diversity but the good solutions may also be lost. Thus, a method which can maintain the population diversity to explore new areas of the solution space without destroying the current results is desired.

The second feature which has to be improved is the computation time. As we know, an EA is a population-based approach; it has to evaluate all the population members and then can select the fittest to survive. Basically, the population size must be reasonably large in order to allow an EA to search the space globally, and the population size typically increases with increasing difficulty of problem. As a result, an inordinate amount of time may be required to perform all the evaluations for a hard problem.

Parallelising EAs was proposed by different researchers and has been proven to be a promising method to overcome both of the above problems. The parallelism is to divide a big population in a sequential EA into multiple smaller subpopulations which are distributed to separate processors and can then be evaluated simultaneously. According to the subpopulation size, the parallel EAs are categorised into two types: *coarse-grain* (e.g., [Tanese 89, Cohoon *et al.* 87, Koza & Andre 95]), and *fine-grain* (e.g., [Spiessens & Manderick 91, Gordon & Whitley 93, Baluja 93]). The characteristics of the two different types parallel EAs are described individually below.

2.3.1 Coarse-Grain Models

A coarse-grain EA divides the whole population into a small number of subpopulations in which each subpopulation is expected to be evaluated by an independent EA in a separate processor. The subpopulations are kept relatively isolated from each other, so this kind of distributed EA is also called an *island* model EA. In this model, each subpopulation is manipulated by a sequential EA, and the selection and genetic operations are limited to happen only in the local subpopulations. A communication phase is introduced in island model EAs. This idea is that an EA periodically selects some promising individuals from each subpopulation and sends them to different subpopulations, according to certain criteria. For example, one can use the selected individuals to substitute the worst ones in the destination subpopulations. This operation is called migration and the selected individuals are called migrants. In this way, an EA has higher possibility to maintain population diversity and protect good solutions found locally.

Running a coarse-grain EA involves the determination of some parameters: the *topology* of the distributed system, the *migration rate*, and the *migration interval*. Topology defines the connections between different subpopulations; migration rate determines the number of individuals to be migrated from one subpopulation to others; and the migration interval controls how often the communication phase should happen. Most of the work in the study of coarse-grain EAs determines these parameters through empirical study; theoretical analysis is still needed.

2.3.2 Fine-Grain Models

The other type of parallelism is the fine-grain model in which an EA is implemented to be massively parallel: the originally large population is divided into a large number of subpopulations in which each of them includes only a small number of individuals. This model is designed to take advantage of machines with a large number of processors (1000 or more); in the ideal case, each individual is evaluated on a different processor. In the fine-grain model, the whole population is viewed as numerous small overlapping subpopulations in which each individual belongs to multiple subpopulations. The selection and mating are restricted to occur only between an individual and its localised neighbourhoods (i.e., those individuals within a certain range).

The network topology in a fine-grain type EA affects the performance of such a system profoundly. If the connectivity among the subpopulations is high, the local optimals will spread quickly to the entire population. This situation is more serious than coarse-grain models or sequential ones, because the population size in this model is quite small which makes it easy for the local optimals to dominate the subpopulations. How to constrain the interactions between subpopulations is yet to be investigated.

Although there has been some research work trying to compare the performance of coarse-grain and fine-grain models of parallelism, the results came out to be inconclusive [Cantu-Paz 95]. In [Baluja 93], the author prefers the fine-grain model, while in [Gordon & Whitley 93], the results favour the coarse-grain algorithms. Because of the lack of a connection machine for fine-grain model and the easy implementation of the coarse-grain model, we choose to use the latter in this thesis.

2.4 Summary

In this chapter, we have briefly introduced the concept of evolutionary computation, and its most popular forms including evolutionary strategies, evolutionary programming and genetic algorithms. We have also described more details about genetic programming, which is an extension of traditional genetic algorithms as well as the major approach used in this thesis for evolving robot control systems. Different methods to parallelise sequential evolutionary algorithms have then been characterised; they are essential for enhancing performance and reducing computation cost.

In the following chapters, the relevant techniques described in this chapter are employed to construct an island model evolutionary system to evolve robot systems for various tasks.

Chapter 3

Evolutionary Robotics

3.1 From Artificial Life to Evolutionary Robotics

Artificial life research has attracted much attention since the first two interdisciplinary workshops organised by Langton in 1987 and 1990 [Langton 89, Langton *et al.* 91]. According to Langton, artificial life research complements traditional biological science which concerns the analysis of living systems, by synthesising artificial systems that can exhibit the behaviours of natural living systems. In this field, evolutionary techniques have been widely applied to evolve systems with life-like behaviours. Yet, most of the work in synthesising artificial systems focused on evolving simulated organisms which live in grid worlds (e.g., [Wilson 85, Collins & Jefferson 91a, Koza 90]); there had been no attempt to apply this technique to evolve real agents (i.e., real robots) until 1991 when Brooks started to explore the problem of using artificial life techniques (i.e., evolutionary algorithms) to evolve programs to control real robots in the real world [Brooks 92]. Since then, some groups have been conducting research in this field which was later named Evolutionary Robotics.

In general, the process of evolving control systems for robots is similar to the traditional evolution-based work. It involves generating an initial population consisting of different control systems; evaluating each control system on a robot (simulated or real) to determine the corresponding performance; and applying genetic operators on the current robot population to create a new population, according to the fitness. Figure 3.1 shows a typical cycle of evolving control systems for robots.

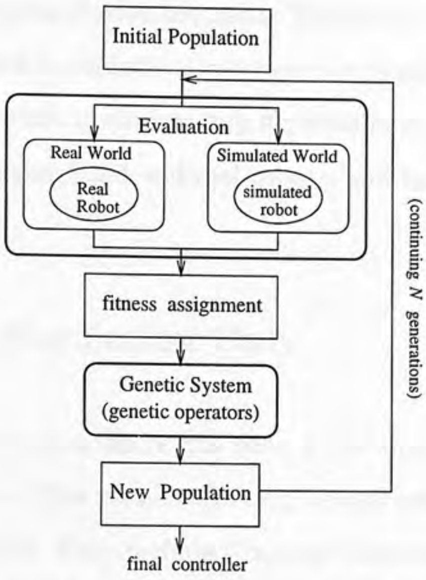

Figure 3.1: The general diagram of an Evolutionary Robotics system in which a controller could be a neural network, a classifier system, a computer program, etc; and the controller could be evaluated by a simulated robot in a simulated world or a real robot in the real world.

In theory, it sounds promising that one can look forward to seeing that the control system drives the robot to generate behaviours closer and closer to what we expect, over the generations and through the fitness improvement by genetic techniques; while in practice there are still a few problems to be solved before this idea comes true. From the point of view of designing a robot, the problems include the performance of the real robot, and the robustness and reliability of the evolved control systems; and as far as the GA is concerned, they also include the representation of a control system, the design of the fitness function, the convergence rate of the evolutionary system, the scalability of the evolutionary approach, etc. In the following sections, we will review some work in the Evolutionary Robotics literature, and then discuss relevant problems.

Since the research work on evolutionary robotics involves different representations for robot control systems (e.g. neural networks, control programs, classifier systems), different experimental approaches (e.g. evolving controllers entirely on real robots, or evolving the controllers in simulation first then transferring them to real robots), and different robot platforms, it will be clearer if the related work is reviewed according to certain categories. As we have mentioned in previous chapters, the final goal of the study of evolving robot controllers is to produce control systems working on real robots, no matter what kinds of representation are used. Therefore, in the following sections, we review some research work in evolutionary robotics in respect to the degrees of their involving real robots: the work involving only simulation is presented first, followed by the work involving both simulated and real robots, and finally the work conducted entirely on real robots.

3.2 Evolution in Simulation Only

As indicated in the above section, there has been a few work evolving simulated organisms in simulated worlds. This section will only review some well-known examples which are relevant to robotics. They include Koza and Reynolds' GP work ([Koza 91, Koza & Rice 92, Reynolds 93, Reynolds 94b]), and Ram's GA work ([Ram *et al.* 94]). The famous 3D simulation, involving the evolution of creatures' morphologies, by Sims ([Sims 94b, Sims 94a]) will be introduced in Chapter 7 because that chapter focuses on the investigation of co-evolving robot controllers and morphologies.

3.2.1 Koza's Work

Most of the work in Evolutionary Robotics encodes a control system to be a linear string and employs a standard Genetic Algorithm (GA) to search for the most suitable solution. In addition to the work applying GA to evolve robot controllers, there is some research work using Genetic Programming (GP) techniques to evolve robot controllers; for example, [Koza 91, Koza & Rice 92, Reynolds 93, Reynolds 94b]. Unlike the string representation in GA, a controller in GP has a tree-like structure. This encoding scheme allows the size of the genotype to be variable. It is an important feature for evolving control systems, because it provides complete freedom for the control architecture.

Koza was the first person to apply GP to evolve robot controllers. In his pioneering work [Koza 91], the author launched into evolving LISP-like programs to control a simulated robot to follow walls as Mataric's robot did [Mataric 90]. After examining Mataric's LISP code carefully, Koza picked up some useful terms to define his terminals and functions (non-terminals) which constituted his control programs. Those terms include the sensor information (sonar sensor distances, the shortest sensor distance, the minimum safe distance for the robot, and the edging distance); the motor actions (packed motor actions for moving and turning); and two other functions (one for connection, and one for conditional branch). As his result showed, the simulated robot was able to perform the wall following task in simulation.

In his later work [Koza & Rice 92], Koza applied a similar GP approach to evolve control programs for the box-pushing experiment which was previously conducted by Mahadeven and Connell [Mahadevan & Connell 91]. For this task, Koza used two additional conditional branches for two extra sensors to detect the situations of bumping and stuck, as originally suggested in [Mahadevan & Connell 91]. The evolved control program can push a box to the nearest wall.

As some roboticists remark [Brooks 92, Mataric & Cliff 95, Gomi & Griffith 96], there are certain questions in Koza's work. The major query is whether or not his approach can be extended to control embodied robots. His simulator is over-simplified and idealised: the robot has no physical size, the sonars have no bearings, and the sensors are capable of returning accurate distance information at any time, etc. Since the results of simulation were never verified on any real robot, the reliability and the performance of his results are thus doubted.

In [Mataric & Cliff 95], the authors also raised the problem of how to select the proper primitives (terminals and functions) in applying GP to evolve robot controllers. They commented that one may need expert knowledge to define primitives to avoid over-simplified or over-expansive design space, and then argued about the practicality of applying GP to evolve controllers. However, it is undeniable that the problem of how to construct an appropriate search space exists not only in GP, but also in all other search-based approaches. Like encoding proper information to define a suitable genotype in GA, or arranging proper network size, type, and architecture in the neural network approach, defining primitives in GP is a way of building knowledge to help the search rather than a burden to the designer. And, the importance of the domain knowledge needed in GP for solving a specific task is the same as in any other approach: the more we understand about the specific task, the more possibilities we can find some way to achieve it. In this work, we have explored how to use GP to evolve controllers to achieve different kinds of control tasks, and according to our experiences, employing GP to achieve tasks is not so difficult as is argued in [Mataric & Cliff 95].

3.2.2 Reynolds's Work

In addition to Koza's simulation, Reynolds has also conducted a series of experiments to evolve agents capable of avoiding obstacles by GP [Reynolds 93, Reynolds 94b, Reynolds 94a]. The major concern in his work is the steering direction; the author intended to evolve specific functions, which are designated to be constituted by arithmetic operators, such as $+$, $-$, $*$, $\%$, to fuse perceptual information from different distal sensors to control the agent's turning. However, like Koza's work, Reynolds's simulation used an over-simplified model for sensors and motors and he even assumed his simulated agent moved forward at constant speed except when turning. Again, all his experiments have the same limitation: they were only conducted in simulation but not on real robots.

3.2.3 Ram's Work

[Ram *et al.* 94] is an example of using GA to investigate the problem of robot control. Unlike all other work, however, they did not use GA to evolve controllers themselves but to optimise some control parameters (the corresponding weights) for some given behaviour controllers in order to control the robot's speed and steering. In their work, they gave the robot some pre-designed behaviour modules; each of them only focused on a certain task such as move-to-goal or obstacle-avoidance, and each could generate its own response to control the steering and speed of the robot. The actual direction and speed of the robot were controlled by the summed and normalised contributions from individual behaviour controllers through the corresponding control parameters (weights). These parameters were encoded as a string of floating point numbers and the GA was used to tune their values.

The task for the simulated robot was to navigate from a fixed starting position to a fixed goal position, and to avoid obstacles along the way in the meantime. During the evolution, the environments were changed randomly at each generation (i.e. re-arrangement of obstacles) to prevent the evolved parameter set from overfitting to some specific environments. The simulation results were successful. There were no further experiments on real robots provided, but the authors claimed that their simulation was based on a simulator which was previously demonstrated in other research work (e.g.,[Arkin 89]) in which a good correlation between simulation and real robot performance can be observed.

3.3 Work Involving Both Simulated and Real Robots

The most popular approach to achieve the purpose of evolving controllers for real robots is to evolve control systems in simulation and then to test them on real robots. Research work based on this approach so far includes those who evolve controllers for the miniature mobile robot Khepera [Nolfi *et al.* 94, Miglino *et al.* 96, Jakobi *et al.* 95, Lund & Hallam 96], for a more complex mobile robot Nomad 200 [Grefenstette & Schultz 94, Schultz *et al.* 96, Yamauchi & Beer 94], and for a walking robot [Gallagher & Beer 96].

3.3.1 Work by the Sussex Group

Cliff, Harvey, and Husbands conducted the first known experiments in the field of evolutionary robotics [Harvey *et al.* 92, Cliff *et al.* 92]. In their work, they advocated using recurrent neural networks as the robot controllers, and studied how to employ Genetic Algorithms to evolve such controllers based on a realistic simulator. They have also analysed the evolved networks in detail by some techniques and studied the effect of noise on the behaviour of the evolved controllers [Husbands *et al.* 95, Cliff *et al.* 93].

Their evolution framework SAGA (described in [Harvey 92, Harvey 93]) is different from traditional GA research: the length of genotype is variable. The development of SAGA was based on Harvey's idea that the complexity of the control systems is always unpredictable so the dimensions of the search space should not be fixed in advance. In addition, the evolution of control systems in their work is incremental: they believe that in order to solve a complex task, one should first define a sequence of tasks which lead to the target task gradually in complexity, and then define different selective pressures for those tasks to guide the evolution step by step. During evolution, the population evolved from a certain task will be slightly mutated and used as the initial population of its successor task. Since the length of the genotype is variable, some genes can no longer represent the same features after crossover. In order to minimise the variation of the features of genes, SAGA restricts the second parent to be the one with least difference to the first parent when the crossover is performed.

The application task in their simulation is to find a way to the centre of a circular room by the use of visual information. The experiment results showed that they were able to evolve dynamic recurrent neural network controllers to achieve the visually guided task in simulation.

After the Khepera robot was available in 1994, Jakobi developed a simulator for it to investigate techniques of evolving controllers in simulation and testing on the real robot [Jakobi 94, Jakobi *et al.* 95]. In his simulation, Jakobi applied elementary physics and basic control theory to construct an idealised mathematical model for the ambient light, reflected infra-red, and the wheel speeds. Several sets of experiments were performed and curve-fitting techniques were used to find the appropriate mappings from the simulated devices to real ones.

This simulator was then used to evolve controllers for the Khepera robot [Jakobi *et al.* 95]. Basically, the method used in [Jakobi *et al.* 95] was similar to those of [Harvey *et al.* 92, Cliff *et al.* 93]: Jakobi *et al.* used Harvey's SAGA system to evolve recurrent neural networks, except that the network parameters were restricted to be integers within the predefined intervals for the purpose of reducing search space. According to [Jakobi *et al.* 95], they have successfully evolved network controllers for the tasks of obstacle avoidance and light seeking. Also, when the evolved controllers were transferred to the real robot, it too can achieve the tasks.

The Sussex group has also investigated the importance of noise in the evolution experiments. After doing a series of experiments with different levels of noise, they concluded that in order to obtain the best match of behaviours for simulated and real robots, the noise injected in the simulation must be at the right level. They emphasised that over-high or over-low noise used in the simulation will lead the real robot to produce behaviours different from those in simulation.

3.3.2 Miglino, Lund, and Nolfi's Work

In investigating the approach of evolving controllers in simulation and testing on real robots, there has been some research work involving the use of a specific simulator built by the look-up table approach [Nolfi *et al.* 94, Nolfi & Parisi 95, Miglino *et al.* 96, Lund & Hallam 96]. In this kind of simulation, the sensor and motor responses were recorded before the evolution experiments, and then used in the later simulation. Experiments based on this kind of simulation all obtained successful matches between simulated and real robots. This is due to the fact that the sensor and motor tables were built through the robot itself and thus can precisely capture the characteristics of the devices.

In this work, the controllers to be evolved were feed-forward neural networks without any hidden layer. A direct encoding scheme was employed: networks with fixed structures were represented as linear strings of fixed length, and the GA was used to find the network parameters. The evolution experiments in this work did not involve crossover but relied mostly on mutation to generate the new populations. The best individuals of a certain generation were duplicated into several copies with slight mutation; these copies then formed a new generation. In [Nolfi *et al.* 94] and [Miglino *et al.* 96], the authors used the method described above to evolve controllers for an obstacle avoidance task; in [Lund & Hallam 96], the authors evolved exploration behaviour in which the robot was required to visit as much of a closed area as possible during a fixed period of time, and homing behaviour in which the robot had to move outward to explore the environment but came back to the charging station (indicated by an ambient light) if the simulated battery went low; and in [Nolfi & Parisi 95], a grasping behaviour was evolved in which the robot can explore the environment to find an object, grasp it, and move toward the wall to release it. All of the results showed that when the evolved controllers were downloaded to a real robot, the behaviours generated by the Khepera robot were similar to those observed in simulation.

In addition to evolving controllers in simulation and then testing them on real robots, Miglino *et al.* also conducted some experiments which allowed the last few generations of evolution to happen on the real robots [Miglino *et al.* 96]. In their experiments, they evolved controllers of obstacle avoidance in simulation as described above, and then continued the evolution on a real Khepera robot. The authors have shown that the population of controllers which were transferred to an on-line evolution process can rapidly improve their performance to a satisfactory level for the target task.

3.3.3 Dorigo and Colombetti's Work

Dorigo and Colombetti applied Genetic Algorithms to a learning classifier system in order to train robots to achieve specific tasks [Dorigo & Schnepf 93, Dorigo & Colombetti 94, Dorigo 95, Colombetti *et al.* 96]. In their work, a population was constituted by a set of classifiers (rules) in which each classifier had an associated strength indicating how good this classifier was, as far as the goal of the whole system was concerned; and the strength of each classifier was determined by a credit system. In their system, the GA plays the role of creating new classifiers to substitute the worst ones in the population, based on the strength of the individual classifiers. This is different from Grefenstette and Schultz's work (described in the next section): an individual in Grefenstette and Schultz's work is a rule set and the GA is employed mainly to create new rule sets by recombining old rule sets without changing the contents of rules, but an individual in Dorigo and Colombetti's work is a rule and GA is used to modify the content of a rule.

In their work, they have emphasised the importance of incremental learning through shaping to accelerate the learning process. To prove the correctness of the proposed methodology and the performance of their learning system, they have conducted some experiments for different tasks on different robots. One example provided in [Colombetti *et al.* 96] is that the robot had to move toward a light which was hidden behind a wall, and the sensors provided included light sensors, sonars, whiskers, and a virtual sensor which can give information if the robot turned. This task was decomposed into two subtasks: searching for light and approaching it. Both subtasks were learned separately, and the interaction between them was pre-designed as suppression: the robot performed light-approaching if light was seen; otherwise it performed light-searching. After a period of time of training in simulation, the learned controllers were transferred to a real robot and the learning strategy continued. As in [Nolfi *et al.* 94, Miglino *et al.* 96], this hybrid approach has the advantage that the rough control frame is learned in simulation to save time, and the on-line learning phase can give the best performance for the real robot in achieving target tasks.

3.3.4 Grefenstette and Schultz's Work

Grefenstette and Schultz have also conducted experiments on evolving controllers for mobile robots, by using the technique of simulation-reality transformation. Instead of using a miniature robot as the work described previously, they used a more complicated Nomad 200 mobile robot [Grefenstette & Schultz 94, Schultz *et al.* 96]. The controller used in their work was also different from others: it is a set of stimulus-response rules. Basically, the controller is to examine the stimuli which include the current sensor and environment conditions, and then to suggest responses, such as a translation or steering velocity command, to the robot according to these conditions. In order to resolve potentially conflicting motor commands suggested by different decision rules in the same rule set, they employed a learning system SAMUEL which was previously developed by Grefenstette (described in [Grefenstette *et al.* 90]) to evolve controllers of rule sets.

In their work, the authors took a more engineering view, and did not insist that all of the task must be learned from the very beginning. Thus they did not generate the initial population at random, but included a variety of rule sets from different sources such as hand-coded rule sets and their variants. This approach was reported to be able to evolve high performance controllers in less time than starting from a completely random initial population [Schultz & Grefenstette 94]. In [Grefenstette & Schultz 94], the task was to move from a starting position to a goal position without bumping any obstacle. The simulated robot with best performance completed the navigation task in 93.5 % of the tests and the evolved controller enabled the real robot to achieve the task with 86 % successful rate during 28 tests.

According to their latest report [Schultz *et al.* 96], the authors applied the same approach to evolve a shepherd robot. Here, both shepherd and sheep were Nomad 200 mobile robots, and the sheep was pre-coded to perform the behaviour of random movement whilst avoiding the obstacle (i.e. the shepherd robot). Consequently, what the shepherd must learn is how to move itself properly to affect the behaviour of the sheep robot to get it to move into a specific goal area within a limited amount of time. They reported that in the simulation, the best shepherd robot can reach a success level of 86 % from 100 episodes, and the real robot can have a 73 % success during the 27 tests (if the communication failures were ignored).

3.3.5 Yamauchi and Beer's Work

Instead of evolving controllers for pure reactive tasks, Yamauchi and Beer evolved recurrent neural networks for tasks which require the intergration of perceptions over time [Yamauchi & Beer 94]. In their work, recurrent neural networks were represented as strings as in the above work but each neuron was regarded as an indivisible unit. It cannot be separated by the crossover operator; only mutation can change the intrinsic parameters or thresholds in neurons.

The task they achieved is that the robot has to move along the sides of crates (as landmarks) for a period of time, and then identify the landmark as one of the two possible candidates. Since only one sonar signal was used at a time step, the controller has to integrate this information over a period of time to give the correct output. It means that the successful controller should have the ability to gradually set its internal parameters to appropriate values through its recurrent characteristics. An eight neuron network was evolved successfully in simulation and can correctly classify landmarks in 17 out of 20 tests after being transferred to a Nomad 200 mobile robot.

3.3.6 Gallagher and Beer's Walking Robot

In addition to that work evolving controllers for mobile robots, Gallagher and Beer have successfully evolved neural network controllers for a walking robot in simulation [Beer & Gallagher 92], and also transferred the evolved controllers to a real hexapod robot [Gallagher & Beer 96]. In their work, they used recurrent neural networks as controllers and employed a direct encoding scheme to encode all network parameters to a linear string. They then used a standard GA to determine the values of parameters.

The robot has six legs and each leg was controlled by a fully-interconnected network of 5 neurons. Each neuron also received an additional weighted input from the leg's angle sensor which gave the leg's angular position relative to the body. In order to simplify the problem and reduce the search space, the authors assumed that each neuron in a leg controller connected only to the corresponding neuron in each of the leg controllers adjacent to it, rather than all leg controllers being fully interconnected to make a full locomotion controller.

In their experiments, they evolved locomotion controllers under different conditions: with or without the sensory feedback from the angle sensor. They found that in both cases, successful controllers can be evolved, but the controller evolved from the case using sensor information did not work well if it lost this information; and the controller evolved from the case without sensor cannot take advantage of the sensor to perform better when it was available. In order to evolve a controller which can take advantage of sensor information if that was available and can still work well if that was unavailable, they eventually adopted the method of taking the average performance of the network with and without sensor information to be the fitness. Thus the evolved controllers were seen to work well in both cases.

Finally, they transferred their evolved controllers to a hexapod robot [Gallagher & Beer 96], and reported that the evolved controllers performed quite well in the real world. The success of transferring controllers from simulated to real robots could be due to the fact that they evolved controllers from different trials in which sensor information was not always available. This resulted in robust results which can be transferred to a real robot successfully.

3.4 Evolution on Real Robots

3.4.1 Floreano and Mondada's Work

Different from most of the evolutionary robotics work, the experiments of evolving control systems conducted by Floreano and Mondada happened entirely on a real robot without involving any simulation [Floreano & Mondada 94, Mondada & Floreano 95, Floreano & Mondada 96a]. In their work, the robot was attached to a host computer via a cable; all low-level processes (such as sensor reading and motor control) were performed by the on-board chips, while others (such as operations for neural networks or genetic algorithms) were operated by the host computer. In this way, controllers can be evolved through the process of the real robot interacting with the real world.

The controllers used in their experiments were multi-layer neural networks with fully recurrent connections between the hidden units. The network parameters, such as weights or thresholds, were encoded to a linear string and a standard GA was used to find the appropriate values for these parameters. In their first set of experiments, they evolved controllers for the task of navigating a robot in a simple maze. Since the work was purely reactive, no hidden layer was used in the network controllers: sensory neurons were connected directly to motor neurons. The results showed that the expected behaviour can be evolved through the on-line evolution procedure.

Floreano and Mondada applied the same approach to evolve a homing behaviour in which the robot was virtually recharged if it was in a specific area. The robot's battery was assumed to be discharged gradually during its life, so it must move to the charging station (indicated by a light) to extend its life cycle when the simulated battery went low. For such a task, 3-layer neural networks were used: one sensor input layer, one hidden layer, and one motor output layer; and the input sensors include infrared sensors, ambient light sensors, a charging station detector, and a battery level detector. The evolved behaviour was that the robot moved around the environment while avoiding the walls (safe-wandering) if the battery was high; and it moved toward the light once the battery status was lower than a critical value. After moving to the station and being charged virtually, the robot switched its behaviour to the previous wandering one.

In their most recent work [Floreano & Mondada 96b], they have begun to evolve learning systems which support learning after evolution, rather than the control systems which cannot be altered after evolution. In [Floreano & Mondada 96b], the authors did not encode the synaptic weight values of the neural networks to the genotypes, but encoded a set of parameters describing synapses' properties (including the synaptic types and the roles) and learning rules (including some possible hebbian learning rules and learning rates). In this way, a controller in their work is a learning system: during the life cycle of a network, the synapses gradually updated according to their own local learning rules and learning rates.

3.4.2 Cliff, Husbands, and Harvey's Work

After a series of experiments in simulation described above, Cliff, Husbands, and Harvey developed their Gantry robot to evolve control systems through real vision [Harvey *et al.* 94]. Unlike work by Floreano and Mondada in which all perception and motion operations were performed directly by the robot itself, work by Harvey *et al.* ([Harvey *et al.* 94]) used real vision but simulated motion. The robot was actually a camera with a mirror, and the mirror can be adjusted by a stepper motor. By adjusting the mirror, the robot can acquire the projected vision of the environment through the camera. However, the robot did not have wheels but was suspended from the gantry frame; the motor commands were translated to the movement of the gantry in the x and y directions by the stepper motors, together with appropriate rotation of the sensory apparatus.

They then employed Harvey's SAGA system to evolve recurrent neural networks to achieve a recognition task in which two targets (one triangle and one rectangle) were present in the environment and the robot had to move toward the triangle by using its vision. As described earlier, instead of evolving controllers for the target task directly, they employed an incremental evolution approach. They divided the whole evolution into different stages for a sequence of tasks with increasing difficulty and finally led to the target task. In the first stage, they evolved controllers for a forward-movement task; after a few generations, the task was changed to movement-toward-large-target, where the initial population for the latter task was the population evolved from the former task with slight mutation; then a similar method was used to evolve a population for a task of movement-toward-small-target and finally for the task of distinguishing-triangle-from-rectangle. The best controller in the final population was able to achieve the desired goal.

3.4.3 Work by AAI

Applied Artificial Intelligence (AAI) is a company which specialises in developing robots with subsumption architectures or other behaviour-based mobile robots. Since 1994, they began to be interested in evolutionary robotics. Their current approach is similar to Floreano and Mondada's work described above: adopting the on-line evolution. The task was expected to perform navigation in a maze which was organised as a hypothetical floor of an office building. The robot needed to visit each room and come back to the first room to complete the mail-delivering task. Since their environment was much more complicated than that in Floreano and Mondada's work, a hidden layer was introduced to the neural network controllers to increase the robot's capability in negotiating the environment. The other difference between theirs and others is that in their evolution experiments, the observers were allowed to comment on the robot's behaviours by pressing the G/B (good/bad) keys on the keyboard to increase/decrease the fitness score. According to their work [Gomi & Griffith 96], this interactive evaluation may make the evaluation process less objective, but it much better reflects the reality in evolution.

Their recent experiment is to set up the whole evolution process directly on the robot without using a host computer [Gomi & Griffith 96]. This work is currently in progress, and no further results have been reported yet.

3.5 Discussion

In this chapter, we have reviewed some of the related research work. As we can see, work in Evolutionary Robotics shared similar ideas in principle. It improved the performance of robot populations over the generations by the use of evolutionary techniques; and all of the reviewed work has provided preliminary results which support the idea of synthesising robot controllers through genetic evolution. However, depending on the research motivations behind the work, different authors explored the same topic from diverse points of view, thus the implementation details of individual evolutionary systems are quite different. For example, since Beer was concerned more about how a dynamical system works, he proposed to use dynamical recurrent neural networks as the internal representation and paid much attention to the analysis of the evolved systems. On the other hand, Nolfi took a more engineering view, and intended to solve more complex control tasks. He thus used a simpler control mechanism, a feedforward network without hidden units, as the robot controller. Our work focuses on investigating whether the evolutionary approach, in practice, can be used to design robots and on exploring how to evolve complex robot behaviours, so our major concerns are the following problems:

- *the performance of the real robot*: Since the main goal of Evolutionary Robotics research is to produce controllers for real robots, the performance of the real robot is then the most important issue to consider. If the performance evaluation for a control system happens completely on a real robot (i.e., on-line evolution), then the result should work on that robot without doubt. However, as pointed out by Mondada and Floreano, a homing behaviour took them about 10 days to evolve; and if they put obstacles in the environment, the evolution of controllers for the same task took 40 days. This makes this approach impractical in implementation. On the other hand, the pure simulation without involving any real robot, such as Koza's work, never reached the purpose of this research in practice. The approach used in [Nolfi *et al.* 94, Dorigo 95, Miglino *et al.* 96] seems more promising. It evolved controllers first in simulation and then transferred the evolved results to a real robot. If the performance of a real robot was not satisfactory, then the evolution process was continued on a real robot for a few generations to improve it. In addition, the robustness and reliability of the evolved control systems should also be taken into account. In other words, the evolved controllers must still work in spite of slight changes of the environment or the noisy response of the robot's sensors and motors. We will investigate the above problems in Chapter 4 and Chapter 5.

- *the scalability problem*: As this research means to aid the robot designer in developing robots, it must show its ability to evolve control systems to achieve more complex tasks. Harvey has proposed that to evolve controllers for complex tasks, one should not take any architecture in advance, but should define a sequence of tasks with increasing complexities to gradually lead to the target task and evolve controllers stage by stage for all the tasks in the sequence. In Dorigo and Colombetti's work, they proposed a more formal methodology called behaviour engineering to evolve modular behaviour networks. In Nolfi's work, the author claimed that after trying different architectures, he found that evolving feed forward networks as a whole was the best way. Instead of subjectively arguing which approach is better to offer the scalability, we would rather objectively analyse what kind of role a human designer plays in these approaches. In Harvey's approach, a designer has to take the job of defining a sequence of tasks with increasing complexities to guide the evolution; in Dorigo's approach, a designer has to decompose the target tasks; and in Nolfi's work, a designer has to try networks with different topologies and sizes to look for the most suitable one. Therefore, which approach to take completely depends on the designer's decision. Chapter 6 will furthermore analyse the difficulties a designer will have to face in using different approaches to achieve complex tasks.

- *the representation problem*: In the research of Evolutionary Robotics, the representation of the robot controller directly relates to the efficiency of an evolutionary system, since it determines the dimensionality of the search space and the evaluation time. From the above reviewed work, we can find that there have been a few different representations for controller used, and each of them has its own advantages and disadvantages. The controllers of recurrent neural networks have the characteristics of biological compatibility and noise resistance; however, the property of recurrence will extend the length of the chromosome which will inappropriately enlarge the search space and make the search difficult. In addition, the evaluation of dynamic neural networks is computationally expensive. For the work using classifier systems as controllers, the controllers can have the ability to learn but extra effort is needed to resolve the conflicts between different classifiers and to assign credit to individual classifiers. Using feed-forward networks as control mechanisms may be straightforward and simple; but this method may not have the potential to solve the problems involving internal states without introducing hidden layers, and it may need some effort in choosing suitable network structures and sizes, as mentioned above. Again, which representation to take entirely depends on the designer's concerns, such as the computation cost or biological similarity. In our work, we intend to develop a mechanism which is ideal for robot control, easy to evaluate, and changeable in length. We thus choose to adopt circuit networks as controllers; which is inspired by the circuit approaches in robot control (explained in the coming chapter). The details will be described in the next chapter.

Chapter 4

Evolving Reliable and Robust Behaviours

4.1 Introduction

The previous chapter has shown the growing interest in Evolutionary Robotics and reviewed some related research work. Although they have all reported their preliminary results of evolving controllers to achieve certain tasks, most of them focus on discussing what kind of control system, i.e., neural networks or high level languages, should be evolved to control a robot, and on using their own evolution-based systems to evolve the proposed control structures for a simple task such as obstacle avoidance to claim the validity of their approaches. In fact, in designing an adaptive robot, there are two important issues about control that have not been emphasised yet, which are:

- *reliability*: This concerns how well a system does what it was designed to do. It measures how well a robot system can repeat externally identical tasks, subject to internal variation (e.g., sensor/motor noise). For example, if a robot was trained in three different trials to achieve a task from three different starting positions, the reliability of this robot is the rate of success in testing this robot for a certain number of trials in which the robot always starts from one of the three positions used in training.
- *robustness*: This concerns how well a system does what it was not explicitly designed to do. It measures how well a robot can achieve a given task, subject to

external variation (e.g., introducing new objects or starting from new positions). A case of testing the robustness of a robot in the above example is to start the robot from a position different from the three positions used in training. Robustness is much more difficult to measure, but it is nonetheless an important concept in designing robots.

Some research work mentioned that their control systems were trained in multiple trials (e.g., [Cliff *et al.* 93, Floreano & Mondada 94, Miglino *et al.* 96]; this may encourage the reliability of the evolved controller. But there is no further evidence of how reliable their evolved solutions are, or whether the evolved solutions can achieve the specific task in a different environment.

In addition to the performance, we should also consider the computational cost, when applying the evolutionary approach to solve problems. Basically, the computational cost of an evolution-based system is measured in terms of the population size, the number of generations, the number of trials used to deal with an individual, and the cost of an individual trial. Therefore, to evolve a robot controller we must find a suitable point in the tradeoff between performance and cost in which a reliable and robust controller is evolved by modest computational effort.

In this chapter, we will first describe our genetic programming framework for robot evolution, including the development of the representation, the genetic operators, and the implementation details. The developed evolutionary system will then be used to evolve behaviour controllers. We will not only show the correctness of the evolved behaviour, but also conduct a series of tests to prove the reliability and robustness of the solution evolved from our GP system. In addition, a series of experiments will be arranged to explore how to evolve such a controller and the corresponding computational cost will be analysed. These experiments and analysis are helpful for us to understand the characteristics of the developed genetic system in evolving robot controllers. After that, we will also have an idea of how to choose appropriate values for the control parameters in this system to evolve robot controllers by the least effort. Once such a system is established and understood, it can then be used to evolve different behaviour controllers for different tasks described in the following chapters.

4.2 The Genetic Framework

The main flow of evolution in our genetic system is similar to the typical evolutionary approach; it mainly involves initialising a population at random and breeding them to produce the new population from generation to generation. The following subsections concern the main issues in developing our genetic system.

4.2.1 Representation

When using evolutionary computation techniques to solve a problem, the first important step is to choose the proper representation for an individual. On the one hand, a genetic representation must be able to express explicitly the features of the solution of the problem to be solved; on the other hand, it must be suitable to be manipulated by the genetic operators to obtain the solution. Therefore, one must study and understand the specific knowledge related to the problem in order to develop a proper representation.

In our work, we intend to apply the genetic approach to evolve a behaviour system for mobile robot control. Hence, we shall begin with understanding and analysing the characteristics of a behaviour controller, and then design a genetic representation for it to be evolved.

The Behaviour Controller

As described in the first chapter, to resolve the problem of robot control, Brooks introduced the behaviour-based control paradigm which advocates using the parallel decomposition of tasks to substitute the traditional vertical general-purpose function decomposition. In the behaviour methodology, the traditional world model is discarded from the control system because of the uncertainty of the world and the large amount of computational time needed to maintain the model. Thus, a behaviour controller will involve the least computation; it is a special-purpose computational module and only deals with the information directly related to itself. In general, such a computational module can be regarded as a finite state machine or automaton, which performs transduction in an embedded world. It has a stream of inputs; it also generates a stream of outputs to the world according to its inputs and internal states.

In principle, the behaviour controller modelled by this kind of machine can be categorised into two types, depending on whether internal states are involved. The first type are called *sequential* controllers, in which the current outputs are determined by the current inputs and the internal states. State transitions occur in this type of controller. It can be modelled as:

$$a_t = f(p_t, s_t)$$

$$s_{t+1} = g(p_t, s_t)$$

in which a_t is the output action from the machine to the environment at time t; p_t is the perception information from the environment at time t; and s_t, s_{t+1} represent the internal states of the machine at time t and $t+1$, respectively.

The second type are called *reactive* controllers, which can be viewed as the simplest form of the state machine: no internal state is involved. Thus, the output of a reactive system is completely determined by the current input; it can be modelled as:

$$a_t = f(p_t)$$

Reactive controllers are the most popular behaviour controllers built in the robot community nowadays. However, we should note that the behaviour controller need not be reactive and that not all tasks can be accomplished by reactive controllers.

The Circuit Model of a Behaviour Controller

From the above section, we have seen what constitutes a basic behaviour controller and how it functions. We are now going to discuss how to develop a representation which can both express explicitly such a computational module and be suitable to be manipulated by a genetic system.

A promising choice is the *circuit network* which has been proven to provide a finer-grained view to represent a behaviour controller. In the *circuit approaches* [Agre & Chapman 87, Rosenschein & Kaelbling 95, Rosenschein & Kaelbling 86], an agent (behaviour controller) exists in the form of digital hardware; and it is made up of two types of components, pure functions and delays, depending on what kind of tasks (reactive or sequential) it is achieving. Pure functions mean logic gates and delays correspond to the flip-flops or registers. The output of one component may be input to one or more other components, thus forming a network. Signals propagate through the network and sensing is thus linked to action. As is well known, any finite state transduction can be carried out by such a network.

Since this thesis focuses on evolving reactive controllers, we will only discuss how to represent controllers of that kind. The approach can be extended to evolve sequential ones with minor modification.

The genetic representation of our reactive controller is inspired by the logic representation in the circuit approaches. By duplicating and separating those components the outputs of which serve as inputs of multiple components and by introducing a dummy root node to connect the outputs of a circuit network together, we find it very straightforward to convert a circuit network to a circuit tree. Figure 4.1 shows an example. After structuring information from the environment and defining simple syntactic rules properly, we can use a genetic programming system to evolve the circuit trees. The details are described in the next section.

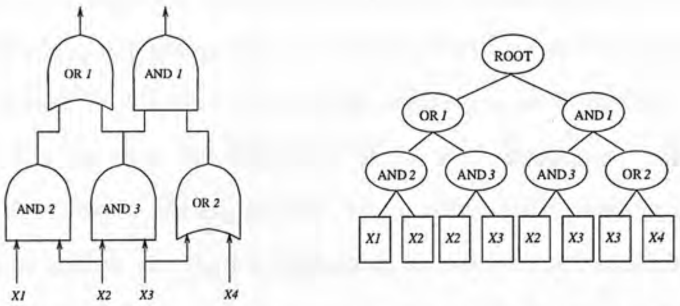

Figure 4.1: An example shows converting a circuit network to a tree.

Genetic Representation of Our Reactive Controllers

As described above, the world model has been excluded from a behaviour controller, and the internal states of a robot has been temporarily shelved for reactive tasks. Thus, the perception inputs always mean the sensor signals of a robot. In tasks of this kind, the robot continuously senses the environment to monitor the variation; it then determines its action according to the sensor responses: observing the difference between sensors (if they are comparable) or checking whether certain sensors reach some kind of threshold. Based on these observations, in the representation of our controller, we structure the perception information into *sensory conditionals* and connect them to the inputs of a logic circuit.

According to our design, structured sensor-conditionals involve comparing the responses of different sensors or comparing sensor response to numerical thresholds. For these purposes, both sensor responses and numerical thresholds are normalised to be between 0 and 1 inclusive. Thus a sensor conditional has a constrained syntactic structure; it exists in the form of X > Y, where X, Y can be any normalised sensor response or threshold which is determined genetically.

Depending on the characteristics of the specific tasks, different kinds of sensors will be required. In general, a sensor is defined to be associated with a value between 0 and 1 which indicates the angle between the direction in which the sensor is pointing and the robot's heading (the robot is assumed to be round and the sensors are positioned around the round body pointing radially outward; section 4.2.5 will describe the robot in detail). Thus, whenever a sensor is called/read in the control system, the normalised sensor response, in the direction indicated by the value associated with that sensor, is returned. For instance, a sensor with the value 0.3 will return the normalised sensor response in the direction 0.3 revolution (108 degrees) anti-clockwise, relative to the robot's heading. In this way, the sensor positions and directions are also allowed to be co-evolved if the sensors are adjustable. For a robot with fixed sensors, the values associated with the sensor are then constrained, subject to the availability of sensors. The experiments in this chapter use a robot with fixed sensors and those involving co-evolving sensors are described in a later chapter.

After organising our genetic representation, we can define *non-terminals* and *terminals* which constitute a circuit tree, for our genetic programming system. In general, three types of non-terminals, the dummy root node, the logic components and the comparator are defined as non-terminals for a reactive behaviour controller. The dummy root node is defined for collecting the main outputs of a control system for convenient manipulation by a genetic programming system; the logic components are defined to constitute the main frame of the controller to map the structured sensor information into appropriate actuator commands; and the comparator is defined to construct the sensor conditionals. As mentioned, because the elements in a sensor conditional can be normalised sensor response or numerical thresholds, both of them are defined as terminals. The tree representation of a typical controller is illustrated in Figure 4.2. In such a structure, the outputs of the subtrees of a circuit tree are interpreted as actuator commands to drive actuators.

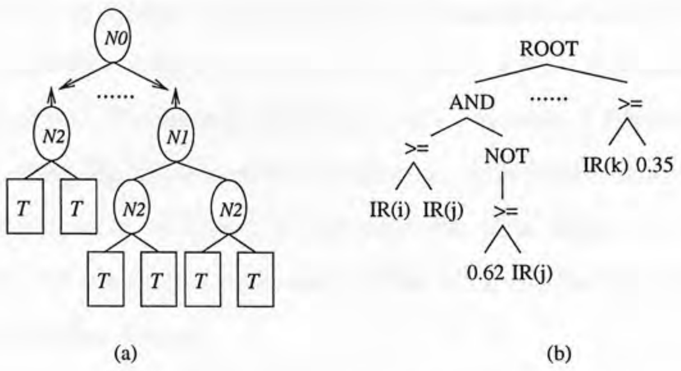

Figure 4.2: The structure of a typical controller. In this figure, *N0*, *N1*, *N2* represent the three types of non-terminals, root-node, logic components, and comparator, respectively. The terminal T can be a normalised sensor response or a threshold between 0 and 1 inclusive.

The above description has shown how we developed our genetic representation. To evolve instances to solve different control tasks, we will need to define different sensor terminals, depending on the requirements of the specific tasks. For example, we may define infra-red sensors as sensor terminals to detect walls and objects for an obstacle avoidance task; or define ambient light sensors as terminals to sense the light for a phototaxis task. The sections later in this chapter describing experiments will give the details of how to evolve behaviour controllers.

4.2.2 The Selection Scheme

In evolutionary systems, a selection scheme is used to choose individuals from the current population, and then the genetic operators are applied to these chosen individuals to produce offsprings. As introduced in the previous chapter, there are different types of selection schemes often used nowadays and they can be categorised into proportionate selection and ordinal-based selection. In general, ordinal-based selection schemes are preferred over proportionate selection schemes because of their translation invariance.

Tournament selection is one of the ordinal-based selection schemes and, according to [Miller & Goldberg 96, Blickle & Thiele 95], it satisfies the criteria of an ideal selection scheme: it is simple to code, easy to implement on both non-parallel and parallel architectures, robust to the presence of noise, provides adjustable selection pressure, and has no need for sorting. In our genetic system, tournament selection is used.

This selection scheme is chosen particularly for its robustness in using noisy fitness (see [Miller & Goldberg 96]), which is important in robot control. The noises from sensors and actuators and the environment uncertainty will cause noisy fitness estimates; and the technique of sampling fitness cases to assess an individual's fitness is used in our system (described in later sections) to increase run time speed, at the expense of decreased accuracy of the fitness evaluation. Thus, all these factors suggest the use of the tournament selection scheme.

4.2.3 The Genetic Operators

In any adaptive or learning system, the key issue for the system to generate better (fitter) individuals is the change of some structures of current individuals. In our evolutionary system, three different kinds of genetic operators, *reproduction*, *crossover*, and *mutation* are employed to create the new generation for this purpose. This section describes their operations.

- *reproduction*: This operation is the basic engine of the Darwinian natural selection and the survival of the fittest. It operates on only one parent and produces only one offspring. In our GP system, whenever this operation is performed, it picks up K individuals (tournament selection scheme) and copies the fittest into the next generation.
- *crossover*: This is the main scheme to generate new offspring by recombining the parents. It involves two parent individuals and produces two children. Whenever it happens, the selection scheme is performed twice to pick up two parents. Because of the tree representation of individuals in a genetic programming system,

the operation of crossover is to randomly swap subtrees of parents to form new trees. As described in last section, however, there are some constrained syntactic structures defined in our GP work, so the crossover must be restricted to protect this defined structure. If the selected crossover point in the first parent is the root node, the second crossover point must be a root node as well ; if the chosen crossover point in one parent is an internal node, then the crossover point in the other parent must be an internal node too; otherwise if the selected crossover point in the first parent is a terminal node, the crossover point for the second parent is restricted to be a terminal node.

- *mutation*: The mutation operation introduces random changes in structures of individuals in a population. Like reproduction, this operation only applies to one parent and produces one child. When it is performed, a randomly selected subtree in the parent is deleted and a new subtree is generated at random to substitute the deleted subtree to form a new offspring. A new tree must conform to the syntactic rules to maintain the correctness of structure.

4.2.4 Parallelism

In the previous chapter, we have introduced the idea that parallel genetic systems allow us to run several populations at the same time each in their own processors to speed up the process of evolution; and some research results have reported that parallel versions of genetic systems found better solutions than comparable serial ones, because their searching in multiple directions maintains the diversity of the population [Tanese 89, Cohoon *et al.* 91, Gordon & Whitley 93].

The approach in parallelising our GP system is the *island model* firstly proposed by Tanese [Tanese 89]. The main reason why we chose to implement this model is because of its easy implementation in software and hardware. The subpopulations in our distributed genetic system are configured as a binary n-cube; Figure 4.3 includes some examples of different n cubes. Migration will happen only between immediate neighbours, along different dimensions of the hypercube, and the communication phase is to send a certain number of the best individuals of each subpopulation to substitute the same number of worst individuals of its immediate neighbours at a regular interval. In our experiments, the communication phase happens every ten generations; this value is chosen because it was found to give the best performance in a small pilot study.

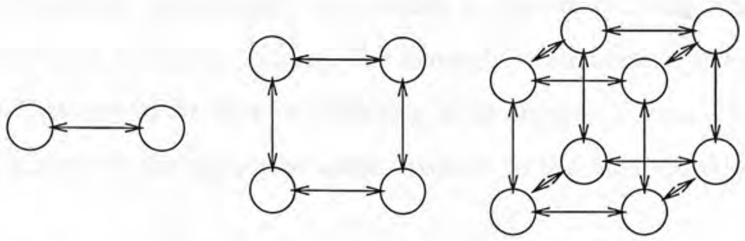

Figure 4.3: The n-cube models. From left to right: n = 1, 2, 3 respectively.

Although our parallel GP system is not implemented on real multiple processor hardware to speed up the evolution, the virtual parallelism of software does help in maintaining population diversity. Thus the performance of our GP system is enhanced. It will be shown in the experiment section.

4.2.5 Simulation

The main aim of this chapter is to establish a GP system and to provide an empirical study to evolve reliable and robust behaviour controllers by appropriate computational effort. In this stage, the simulator is a helpful tool for its easy control and lower time consumption. Thus, all the experiments in this chapter are done by the use of a physical-based simulator. It should be noted that the experiments conducted in this chapter are only for the above purposes; and the simulator here can be substituted by that of any particular robot (see Chapter 5), so that the evolved control systems can be downloaded to the real robot, like the results reported by [Jakobi *et al.* 95, Miglino *et al.* 96]. We will also show how to achieve the simulation-reality transformation without loss of performance in the next chapter. The rest of this subsection describes the simulated robot used in this chapter.

The Simulated Sensors and Motors

In general, agents can be equipped with different kinds of sensors for diverse purposes to achieve specific tasks. For instance, an agent can use infra-red sensors to acquire distance information for object detection or use ambient light sensors to measure the light intensity for direction distinction. In this simulation, we assume that the robot has a round physical body and the sensors are positioned around the body pointing radially outward. The general characteristic of a sensor is that it can only function within a certain distance and a certain bearing. For example, a simulated infra-red sensor can have a visual distance of 30 cm and a bearing of 20 degrees. Figure 4.4 illustrates the sensor arrangement of the simulated robot involved in the later experiments.

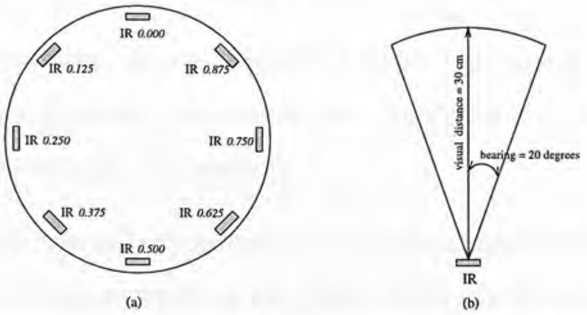

Figure 4.4: (a) The sensor arrangement of the simulated robot; (b) The sensor field of view.

In this simulator, the motion system of the vehicle is regarded as a process with natural dynamics, and is modelled by related first-order differential equations. Once the time constant of the motor system is specified, the system characteristics are determined; this system can then be used to convert motor commands with required speeds (Full/Half, Forward/Reverse), which are determined by the controllers, into actual motions. The angular velocity of each motor is firstly calculated, then the moving speed, the turning speed, and the new position of a vehicle are calculated by applying appropriate kinematic equations, with the specified wheel radius and wheelbase (the distance between two wheels).

In our implementation, the motor system is modelled by the following first-order differential equation:

$$\tau_i \frac{d\omega_i}{dt} + \omega_i = D_i \tag{4.1}$$

in which τ_i is the motor time constant; D_i is the demanded angular velocity (motor command); and ω_i is the actual angular velocity of motor i (i is left or right). This model shows that if we command motor i with a certain velocity D_i, it will actually revolve at a velocity ω_i. To calculate the actual angular velocity ω_i, we can reorganise equation (4.1) as:

$$\frac{d\omega_i}{dt} = \frac{D_i - \omega_i}{\tau_i}$$

and then perform integration to obtain ω_i. The simplest way for integrating the above first-order differential equation is to apply an Euler method (see [Press *et al.* 92]) to derive the formula for ω_i as:

$$\omega_i(t + \Delta h) = \omega_i(t) + \Delta h \times \frac{D_i - \omega_i(t)}{\tau_i} \tag{4.2}$$

in which $\omega_i(t)$ represents the angular velocity of motor i at time t, and Δh is a small time interval. In our simulation, we assume that the initial angular velocity $\omega_i(0)$ is zero and the time interval Δh is 0.1 second.

Once the angular velocities of both motors are obtained, they are then used to calculate the forward and rotating speeds of the robot. If the wheel radius of the robot is r_i and the wheel base is L as indicated in Figure 4.5, we can have the following set of equations which describe the relationships between the angular velocities ω_i of the motors and the forward speed v and rotating speed ω of the robot:

$$\begin{cases} r_l \times \omega_l = v + \omega \times \frac{L}{2} \\ r_r \times \omega_r = v - \omega \times \frac{L}{2} \end{cases} \tag{4.3}$$

By solving the above equations, we can now obtain v and ω as:

$$\begin{cases} v = \frac{r_l \times \omega_l + r_r \times \omega_r}{2} \\ \omega = \frac{r_l \times \omega_l - r_r \times \omega_r}{L} \end{cases} \tag{4.4}$$

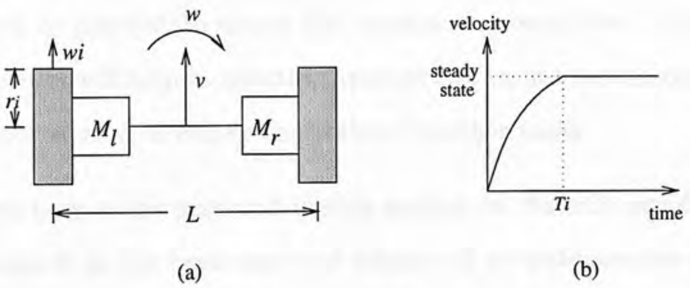

Figure 4.5: (a) Illustration of the relevant symbols of a motion system. M_i represents the motor i (with a time constant τ_i); ω_i is the actual angular velocity of motor i; r_i is the radius of wheel i; L is the length of the wheel base; and v and ω are the forward and turning speeds of the robot respectively.(b) The dynamic of a motor i; τ_i is the motor time constant.

Finally, with forward speed v and rotating speed ω, we can calculate the position (R_x,R_y) and the heading θ of the robot as:

$$\begin{cases} R_x(t+\Delta h) = R_x(t) + (\Delta h \times v \times cos(\theta)) \\ R_y(t+\Delta h) = R_y(t) + (\Delta h \times v \times sin(\theta)) \\ \theta(t+\Delta h) = \theta(t) + (\Delta h \times \omega) \end{cases}$$

In our simulation, a fine time-slice technique is used, therefore the above equations are arranged in an evaluation loop and are solved iteratively to indicate the position and heading of the robot at each time step.

Since the simulated robot is driven by two independent motors with separate speed commands, it is able to move forward, backward, turn right and left at any speed derivable from the command set. In addition, as we know, the real sensors and motors of the robot are noisy and unreliable. Hence, in order to make the simulation more realistic and to enhance the robustness of the evolved solutions, 5% random noise (+5% $\sim$ −5% uniformly) is injected to the perceptions and the motions.

4.3 Evolving a Reliable and Robust Controller

We have described the framework of our GP system which is built for evolving reactive behaviour controllers. In this section, we will use this system to evolve example controllers and then design an efficient criterion to evolve reliable and robust solutions. We will also conduct a series of experiments to analyse the corresponding computational costs and find out the proper scales (e.g., 10^2 or 10^3, etc.) for those control parameters, such as population size or the number of generations, of the developed GP system. The results will help in selecting appropriate values for the control parameters when this system is used to evolve controllers for other tasks.

The application task to be achieved in this section is obstacle avoidance. This task is chosen because it is the basic survival scheme of an autonomous robot. Obstacle avoidance is a general behaviour; a successful evolved controller should be able to keep a vehicle moving but without collision, no matter where it starts or how many time steps it moves. Moreover, the evolved controller should be able to achieve the task if the environment is slightly changed. The following subsections will describe the experiments and results.

4.3.1 Preparation for a GP Run

As introduced in Chapter 2, some preparatory steps are required before applying genetic programming to solve a problem, mainly including defining terminals and non-terminals to constitute the target trees, and determining the values of parameters for a GP system. Additionally, the outputs of a control tree here will be used to drive a robot to move, so they need to be interpreted to motor commands available for the particular robot used.

In the previous section, we have specified the general structure for a tree controller. To apply it to solve different control tasks, we need to define terminals and non-terminals for individual tasks according to their different requirements. For the task of obstacle avoidance here, a robot needs distal sensors to provide the distance information so the infra-red sensor IR is defined as the sensor terminal for this specific task. As mentioned previously, a sensor terminal is associated with a real number between 0 and 1 which indicates the direction the sensor is pointing. For a robot with fixed sensors, the value associated with a sensor terminal must be restricted in regard to the availability of sensor directions. In the following experiments, the simulated robot illustrated in section 4.2.5 is used so the available sensor directions are $\frac{n}{8}$, in which n is a integer between 0 and 7 inclusive.

To evolve a controller for the task of obstacle avoidance, we use one population of 100 individuals (this value is relatively small, compared to the population size 500 in [Koza 91] and 2000 in [Reynolds 94b] in which they also used the GP approach to solve a task with similar complexity). As mentioned, different genetic operators are applied to different numbers of individuals to produce offspring. In this work, the rates of reproduction, crossover, and mutation are 10%, 85%, and 5%, respectively. In addition, to prevent trees from growing without limit, the initial depth limit for a tree is 6 and the maximum depth after crossover and mutation is 13. These values are from [Koza 92] with minor modification. Since the technique of random sampling (only some of the defined fitness cases are used to evaluate an individual, see experiment section for explanation) will be used, we do not use the *best-so-far* solution (the solution with

Terminal Set:	IR, $\Re\{0..1\}$
Function Set:	PROG, AND, OR, NOT, XOR, >
number of populations:	1
population size:	100
number of generations:	50

Table 4.1: The key features of the problem of evolving an *obstacle-avoidance* robot described in the text.

tree	*motor command*		*motion*
output	*left*	*right*	*description*
0 0 0	slow-forward	slow-forward	forward at low speed
0 0 1	slow-forward	fast-forward	forward and turn to left
0 1 0	fast-forward	slow-forward	forward and turn to right
0 1 1	fast-forward	fast-forward	forward at high speed
1 0 0	slow-reverse	slow-reverse	backward at low speed
1 0 1	slow-reverse	fast-reverse	backward and turn to left
1 1 0	fast-reverse	slow-reverse	backward and turn to right
1 1 1	fast-reverse	fast-reverse	backward at high speed

Table 4.2: The interpretation of tree outputs to motor commands. The motion description for each command (including the left and right commands) is based on that the robot starts from a static state and then continuously executes that command for a few time steps.

best fitness value during the whole run) as most of the GP work but designate the best individual appearing in the *last* generation as the final solution. The number of generations is 50 in the experiments to follow. Table 4.1 summaries the key features of the problem of evolving an *obstacle-avoidance* robot.

Because the output of the tree controllers will be used to drive a robot to move, we need to convert these outputs to motor commands for the robot used. For simplicity, the tree controllers in the experiments of this chapter are defined to have only three subtrees. The output of the first subtree is interpreted as the direction of revolution of both motors: if the output is 0, the controller will command both motors to revolve forward, otherwise it commands both motors to revolve backward. The outputs of the second and the third subtrees determine the speeds of the left and the right motor, respectively: 0/1 represents half/full speed. Table 4.2 shows the mapping of the tree outputs and motor commands.

4.3.2 Fitness Measure

In evolving a controller, to evaluate an individual is to execute the controller on a robot for a given period of time and to measure the performance according to a certain criterion (fitness function). As mentioned, a fine time-slice technique is used in this simulation and each time step will last 100 ms in the experiments to follow. At each time step, the controller is executed once and the output is used to drive the robot to move; then the corresponding fitness is calculated. The performance of a controller is thus defined to be the accumulated fitness of an individual during the given time steps.

A direct way to formulate the behaviour of obstacle avoidance is to describe its goal as keeping the robot as safe as possible at each time step. To achieve this, each robot individual is virtually equipped with eight IR sensors[1], which point towards $-\frac{3}{4}\pi$, $-\frac{1}{2}\pi$, $-\frac{1}{4}\pi$, 0, $\frac{1}{4}\pi$, $\frac{1}{2}\pi$, $\frac{3}{4}\pi$, and π relative to the heading of the robot, and is trained to keep the responses of these IRs as low as possible. The term *max_response*, meaning the maximum of the eight IR responses, is introduced in the fitness function for this purpose. In order to keep it safe, the robot is punished whenever it begins getting dangerous (that is, the *max_response* is larger than 0), and the higher this value, the larger the penalty. In addition, in order to avoid the degenerate situation in which a robot sticks at a certain position, or the situation in which a robot spins, an agent is encouraged to move straight, and discouraged from rotation. Thus, the fitness function is defined as a penalty function:

$$f = \sum_{j \in Cases} \sum_{t=1}^{T} P_j(t)$$

where

$$P_j(t) = [\alpha \times max\_response(t) + \beta \times (1 - v(t)) + \gamma \times w(t)]$$

in which T is the number of time steps in a single trial; $Cases$ is the set of fitness evaluations done on this individual (i.e., fitness cases); if the robot collides with an obstacle at $t_c < T$, say, the trial stops but the robot is penalised with $P_j(t) = P_j(t_c)$ for each remaining time step; $max\_response(t)$ is the largest IR response at current

[1] The eight IRs mentioned here are used for fitness assessment only: they are independent from those sensors evolved as parts of the controllers. After the process of evolution, these eight IRs are removed.

time step (it is time-dependent); $v(t)$ is the normalised forward speed (backward is regarded as negative); $w(t)$ is the absolute value of the normalised rotating speed (it is always positive); and α, β, γ are the corresponding weights expressing the relative importance of the above three criteria (they are robot-dependent; in the following experiments the values 0.2, 0.2, 0.6 are used for α, β, γ, respectively, because this combination was found in preliminary testing to give best performance). This would keep a robot safe and moving forward as straight as possible.

4.3.3 Experiments and Results

One of the reasons for our applying evolution techniques to synthesise controllers is to enable the robot itself to deal with the robot-environment interaction. Thus, to be familiar with different environment situations, the robot needs to be evolved from different trials. One way to do this is to define a sample set to represent the environment situations for the specified task and then use the samples as fitness cases to evolve controllers. For the task of obstacle avoidance here, a sample (fitness case) means a starting position, and a trial is that the robot starts from a new position and then its controller is executed for 500 time steps. In order to evolve a general solution, different samples should be distributed over and cover the whole environment. In this work, 25 different starting positions were defined and included in a sample set.

In the first set of experiments, each individual was evaluated 25 times, one time on each of the pre-defined fitness cases. Each time the robot started from a different position and was executed for 500 time steps. The fitness was calculated as described in the above subsection.

Our aim is to evolve a reliable and robust solution capable of achieving a specified task in different situations. To prove whether the evolved controller is reliable and robust, we must test it thoroughly. In our experiments, an evolved controller was tested 1000 times, using different starting positions, random effects of sensor and motor noises, and varied environments; and each test lasted 10000 time steps. An evolved controller was then claimed to be able to achieve the task reliably and robustly if it did not bump any obstacle during the 10000 time steps moving, in at least 990 test cases (99% success).

M	no	$P_s(\%)$
25	C_0	99.9
	C_1	99.6
	C_2	99.8
	C_3	99.5
	C_4	99.4
	C_5	99.2
	C_6	99.8
	C_7	99.6
	C_8	99.2
	C_9	99.4

Table 4.3: The testing results of evolved controllers. M is the number of trials for an individual and P_s is the proportion of tests in which no collision happened (based on 1000 tests).

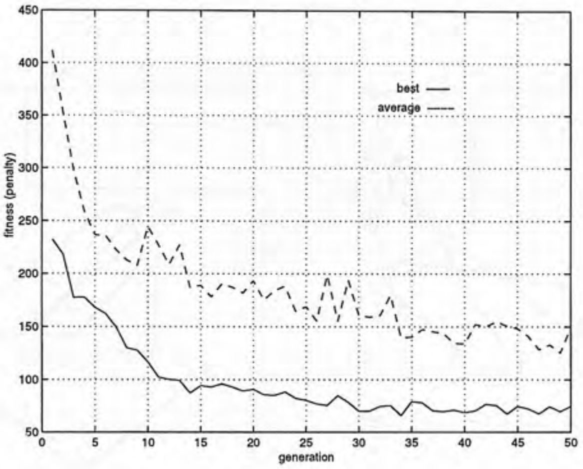

Figure 4.6: The best and average fitness (penalty) of each generation for an example run.

To be more objective, ten independent runs were conducted for the same task and each evolved controller was tested exactly as described above. The testing results are listed in Table 4.3 in which P_s represents the percentage of tests in which no collision happened. As shown in Table 4.3, all of the evolved controllers are reliable and robust, and this table also shows the reliability of our genetic system. Figure 4.6 shows the best and average fitness at each generation for a typical run and the evolved solution from this run is (the constants in PROG are what Koza calls ephemeral constants – they are generated in the initial population or by mutation):

```
(PROG
    (> IR 0.000 IR 0.500)
    (AND (OR (AND (> IR 0.000 IR 0.625) (> IR 0.000 IR 0.500)) (> 0.48 IR 0.625))
          (> IR 0.875 IR 0.000))
    (> IR 0.000 IR 0.500))
```

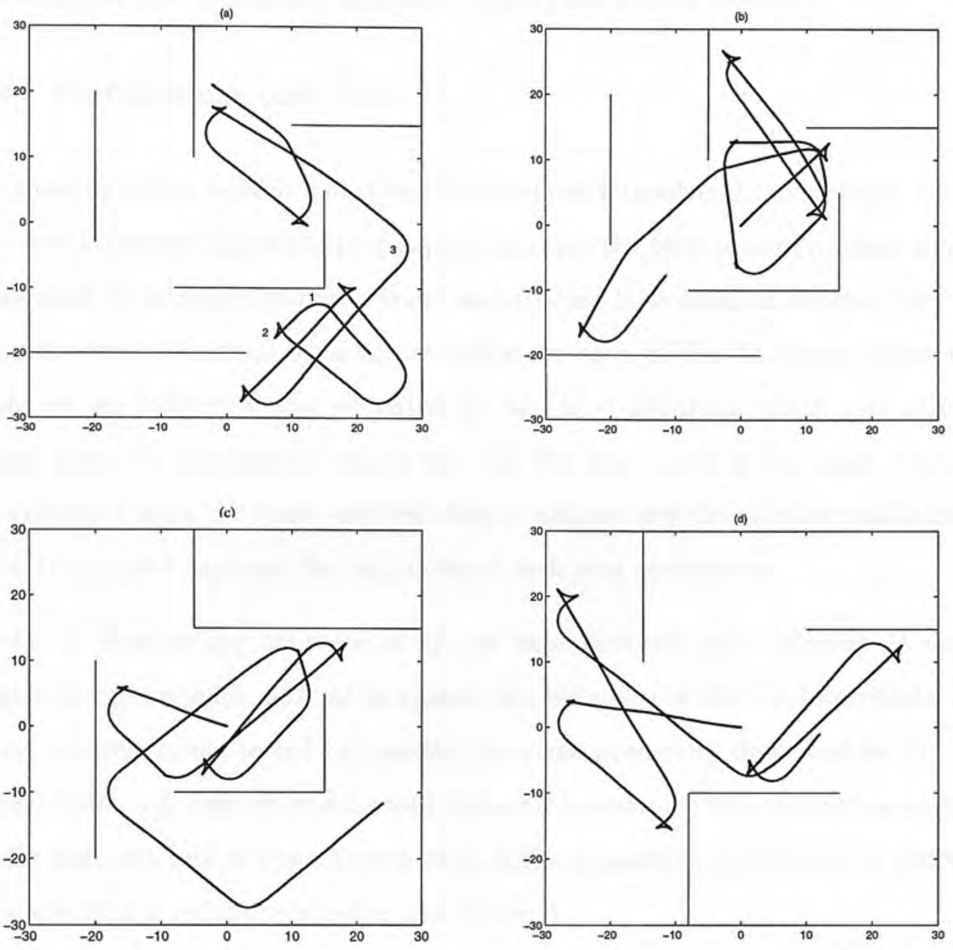

Figure 4.7: Four examples of emergent behaviours, in which the robot started from different situations (different start positions in (a)(b), or different environments in (c)(d)).

Figure 4.7 illustrates some of the behaviours generated by the above controller: (a)(b) are two examples in which the robot started from two different positions in the environment used in the evolution; (c)(d) show the trajectories of the robot when it performed the same controller in different environments. To analyse the emergent behaviour, we use Figure 4.7(a) as an example. From Figure 4.7(a) we can see that the vehicle always kept moving forward. Yet when it sensed any obstacle, some special behaviours can be observed: in situation 1, the vehicle slowed down and approached the obstacle, then it sped up to move away from the obstacle; in situation 2, it slowed down and moved backward to leave the obstacle, and then sped up to turn away from the obstacle; and in situation 3, it did not slow down but simply sped up to leave the obstacle. As a reactive agent, how it behaved depended entirely on what it sensed.

4.3.4 Performance and Cost

After showing that a reliable and robust solution can be evolved from multiple trials, we conducted a series of experiments to explore whether the technique of random sampling can be used to decrease evolution time, and studied how different sample sizes affect the performance. Instead of being evaluated on each of the 25 fitness cases in the sample set, an individual was evaluated for M ($M < 25$) trials which were randomly sampled from the pre-defined sample set. All the individuals in the same population were evaluated from the same sampled start positions; and the starting positions were randomly sampled again at the beginning of each new generation.

In order to observe the influence of M, we experimented with different M and did 10 independent runs for each M to reduce the influence of the random effects. Each evolved controller was tested by exactly the same procedure described in the above section. Table 4.4 lists the results and Figure 4.8 contrasts the number of successful runs for different sets of experiments with different sample sizes (where a 'successful' run means that a reliable controller was evolved).

M	*no*	$P_s(\%)$	*type*	M	*no*	$P_s(\%)$	*type*
	C_0	89.8	2		C_0	99.8	1
	C_1	97.5	2		C_1	87.6	2
	C_2	4.2	2		C_2	61.6	2
	C_3	60.9	2		C_3	85.3	2
1	C_4	99.8	1	10	C_4	99.0	1
	C_5	99.2	1		C_5	99.4	1
	C_6	80.3	2		C_6	99.8	1
	C_7	99.7	1		C_7	99.2	1
	C_8	0.0	2		C_8	99.7	1
	C_9	15.9	2		C_9	99.5	1
	C_0	99.2	1		C_0	99.3	1
	C_1	68.4	2		C_1	99.4	1
	C_2	99.8	1		C_2	99.4	1
	C_3	99.6	1		C_3	99.3	1
5	C_4	45.5	2	15	C_4	82.4	2
	C_5	99.4	1		C_5	99.8	1
	C_6	99.2	1		C_6	99.1	1
	C_7	99.5	1		C_7	99.8	1
	C_8	72.4	2		C_8	99.1	1
	C_9	65.8	2		C_9	99.3	1

Table 4.4: The testing results of evolved controllers for different sample size M.

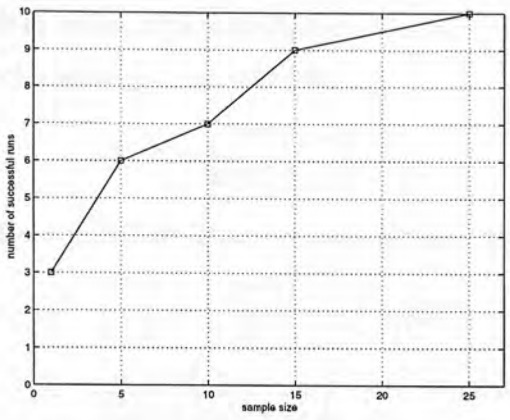

Figure 4.8: A figure contrasts the number of successful runs for different sets of experiments with different sample sizes (10 runs were done for each sample size). The case with sample size 25 represents the set of experiments in which 25 fixed fitness cases were used (see Table 4.3).

According to P_s, which is the percentage of testing cases without collision, the evolved controllers in Table 4.4 are categorised into two types: the controllers with very high P_s ($P_s \geq 99\%$) are classified as type 1; and the others are classified as type 2. From this table, we can see that the larger the sample size, the higher the probability that a type 1 controller is evolved.

Although a larger sample size used in an evolutionary experiment can result in a higher probability of obtaining a successful controller, it also implies longer evolution time. To investigate this tradeoff, we suggest using a "try until success" strategy, i.e., we run single experiments until we succeed, then stop. This can also be stated recursively: our strategy is to run an experiment and if it succeeds, stop; otherwise we use our strategy again. This recursive formulation allows us to calculate the expected computational cost of the strategy (x, say) given the chance of success of a single run and the cost of a single run:

$$x = C \times P + (C + x) \times (1 - P)$$

in which C is the computational cost of running a single experiment by this strategy; and P is the probability of success of a single experiment (we assume the experiments are probabilistically independent). By solving the above equation, we find

$$x = \frac{C}{P}.$$

Now, suppose we have conducted $(M + N)$ independent single experiments; M of them succeeded and N of them failed. The distribution of the probability of success P can then be described as [Gradshteyn & Ryzhik 80]:

$$f(P) = \frac{P^M(1-P)^N}{B(M+1, N+1)}$$

in which $B(\cdot,\cdot)$ is the Beta function. Thus, we can calculate the expected cost of x as:

$$\begin{aligned} E_P[x] &= \int_0^1 \frac{C}{P} \times \frac{P^M(1-P)^N}{B(M+1,N+1)} dP \\ &= \frac{C}{B(M+1,N+1)} \int_0^1 P^{M-1}(1-P)^N dP \\ &= C \times \frac{B(M,N+1)}{B(M+1,N+1)} \end{aligned}$$

By the definitions $B(m,n) = \frac{\Gamma(m)\Gamma(n)}{\Gamma(m+n)}$ and $\Gamma(x+1) = x\Gamma(x)$, we can simplify the above result to:

$$E_P[x] = C \times \frac{M+N+2}{M+1}$$

According to the results of different sets of runs, for the strategy of using 25 fitness cases, we have $M = 10$ and $N = 0$; if we assume the cost of conducting a run by this strategy is C, then the expected cost of this strategy is $C \times \frac{12}{11}$ ($\sim 1.09C$). Similarly, for the strategy of using 15 fitness cases, we have $M = 9$, $N = 1$, and the cost of a single run $\frac{15}{25}C$, so the expected cost of this strategy is $\frac{15}{25}C \times \frac{12}{10}$ ($= 0.72C$). This shows that running evolutionary experiments by the latter is in fact more efficient (in terms of expected cost) than the former.

As indicated, we have implemented an island model to support the virtual parallelism for our evolutionary system. To investigate whether it can improve the performance of the GP system, we used two populations of 50 individuals to replace the original single population of 100 individuals and then repeated the experiment. The sample size remained as 15 in this set of runs. The results show that all of the ten independent runs can evolve successful controllers for the obstacle-avoidance task. It suggests that the parallel model does enhance the performance of our GP system. We can also estimate the expected cost for this experimental strategy, as for the above two sets of experiments. Here, M is 10, N is 0, and the cost of a single run is $\frac{15}{25}C$, so we can calculate the expected cost of this strategy as $\frac{15}{25}C \times \frac{12}{11}$ ($\sim 0.65C$). This shows that by using this strategy (two populations of 50 individuals and 15 randomly sampled

pop_size	*num of pops*	*fitness cases*	*num of type 1 controller*	*expected cost*
100	1	25	10	$1.09C$
100	1	15	9	$0.72C$
50	2	15	10	$0.65C$

Table 4.5: The comparison of results from different strategies. In this table, C is the computational cost of conducting a single run by the strategy of population size 100 and 25 fitness cases.

fitness cases) one can evolve successful controllers in all of the ten runs by relatively less computational effort. The comparison of three different strategies is listed in Table 4.5.

4.4 Summary and Discussion

4.4.1 Summary

In this chapter, we have described the framework of our genetic programming system in which a circuit-tree is proposed to represent a behaviour controller. This system has been used to evolve the control system for an application task. We not only showed the correctness of the evolved behaviour but also verified the reliability and robustness of the evolved controllers. To verify the reliability of the developed GP system, we have run it a number of times for the same experiments.

We have also investigated how to use a more efficient criterion to evolve this kind of solution. In particular, the trade-off between the performance and the computation cost has been analysed. The results suggest that although the more fitness cases sampled and used in one generation, the higher probability a successful solution is evolved, it also takes a longer time; one should analyse the computational cost of obtaining a high performance solution to choose an efficient strategy. Experiments were conducted to find out the suitable scale of different control parameters and to enhance the performance of our evolutionary system by a parallel model.

4.4.2 The Control Parameters

When a genetic mechanism is operated to solve a problem through evolution, there are some parameters, which control the operation of such a mechanism, needing to be determined. In the work of evolving robot controllers, the values for these parameters are task-dependent – they depend on the difficulty of the given task. Though there is no theoretical evidence supporting whether the values of the parameters we obtained above can directly be used for other tasks, from the above experiments, we have been able to capture the characteristics of the developed GP system in applying it to evolve robot controllers and therefore the results obtained from the above obstacle avoidance task will certainly help in setting the initial values of the control parameters in achieving different control tasks. For instance, we may use the developed GP framework, with a population size 50 to evolve controllers for a different task (e.g., light seeking), rather than with a population size 500 as in Koza's work [Koza 91, Koza & Rice 92] or Reynolds' work [Reynolds 93, Reynolds 94a] in evolving robot controllers by GPs. In fact, the following chapters will show that the strategies derived from this chapter can be used, with slight variations, to evolve controllers for a variety of tasks successfully.

Based on the experimental results, we can estimate the appropriate scales of the parameters for evolving robot controllers by our GP system. They are summarised as follows:

- population size and number of populations: Both of them are decided by the difficulty of the problem to be solved. As the experimental results suggest, population size 50 may be a proper choice (for both evolving successful controllers efficiently and obtaining relatively more successful runs) for the task of which the difficulty is similar to that of obstacle avoidance; and two to four populations should be able to evolve controllers to achieve reasonably complicated tasks.

- number of generations: In all the experiments, we observed that the GP system converged to a stable state within 30 generations. Therefore, the number of generations (50) used in the above experiments should be suitable.

The above parameters play important roles in evolution-based systems. In addition to them, the values of other minor parameters used in the above experiments are similar to those of a general genetic system. They include:

- probabilities of genetic operators: We have mentioned in the previous chapter that in a GP system, crossover is the main scheme active while mutation is not. So the probability of operating crossover is relatively higher than others. The probabilities 10%, 85%, and 5% for operators reproduction, crossover, and mutation respectively are derived directly from [Koza 92].
- depth constraints: The depth limit for the initial trees is 6, and the maximum depth allowed after crossover and mutation is 13. The latter is slightly smaller than those default values in [Koza 92].
- The rate of migration is 12% and the migration interval is 10 generations in our island model GP system. These values are inspired by [Tanese 89].

The values of minor parameters seem promising when they were used in our GP system to evolve controllers. Thus, they will be the default values in employing our GP system to evolve different behaviour controllers for different kind of tasks. The related experiments are described in the coming chapter.

Chapter 5

Evolving Controllers for Real Robots

5.1 Introduction

In the last chapter, we have described our genetic framework in detail and have verified that it can be employed to evolve high performance controllers for mobile robots. Even though an experimental model has been established and an efficient approach to evolve controllers has been empirically explored, all the previous experiments were conducted in simulation. However, the main purpose of developing an Evolutionary Robotics methodology is to produce control systems which can actually work on real robots. Therefore, after the computational framework of the genetic system has been established and proved, we now need to investigate whether this approach works on a real robot.

There are two kinds of experimental approaches applied in Evolutionary Robotics for evolving control systems, namely on-line and off-line approaches. The former indicates that all of the evaluations are performed on real robots; while the latter involves the use of simulation. Before choosing one of them, we need to consider the tradeoff between the performance of real robots and the consumed time to evolve controllers. In the on-line approach, there is no need to verify the performance, because it evolves control systems directly on the real robot and in the real environment. However, an evolution-based approach requires the evaluation of populations of robots over a number of generations, so the on-line methodology will then involve evaluating the populations of controllers one by one on the real robot in the physical environment. As far as the time consumed is concerned, this method is not economic, which makes it unrealistic as a tool for developing robots.

On the other hand, off-line evolution is much quicker in obtaining the result, though there exists the problem of the simulation-reality gap in this kind of approach [Brooks 92, Smithers 94]. If it is difficult to remedy the disadvantage of the unreasonable time-consumption of the on-line approach, an alternative is to adopt the off-line approach and to bridge the gap between simulated and real robots. This seems to be an inevitable step in the study of Evolutionary Robotics: before exploring the Evolutionary Robotics approach further, we need to study how to bring it to reality.

In this chapter, we will first analyse the time consumed in developing control systems by the evolutionary approach, and discuss the need for, and the role of, simulation. The advantages and disadvantages of different methods of simulation will also be compared from different perspectives. In addition, we will describe briefly how to build a simulator which takes the dynamics of the real world into account, as suggested in [Nolfi *et al.* 94, Miglino *et al.* 96, Lund & Hallam 96]. To prove that the evolutionary approach can evolve working real robots, in the experimental section we will choose a specific simulator to evolve various behaviour controllers for different tasks and then transfer them to real robots. The results show that using the methodology developed in the last chapter and a realistic simulator, one can transfer the evolved controller from simulation to reality successfully: the behaviours observed in the real robot are extremely good as expected.

5.2 The Use of Simulation

5.2.1 The Need and the Role of Simulation

The use of simulation nowadays seems inevitable in developing methodologies of robot learning or evolution, because of their time-consuming characteristics. In the field of Evolutionary Robotics, some researchers have briefly analysed and discussed the time-consumption of this kind of approach [Mataric & Cliff 95, Lund & Hallam 96]. Based on their analysis, the estimated computational cost of running a single Evolutionary

Robotics experiment is

$$N \times T \times \mu \times I \times G$$

where N is the number of trials in which the robot is evaluated (for the purpose of measuring the fitness objectively from some uncertain factors such as the arrangement of objects in the environment, the initial conditions of the robot, and the random noise of sensors and motors, etc); T is the number of time steps to evaluate a controller in a single trial; μ is the time taken by the robot to perform a single action; I is the number of individuals in the population(s); and G is the number of generations for running the evolution system. If we do not use simulation but perform on-line evolution as [Floreano & Mondada 94, Floreano & Mondada 96a, Mondada & Floreano 95], the time for evolving a controller can easily rise up to a few weeks even if the task to be achieved is only to avoid obstacles in a fairly simple environment, as reported in [Floreano & Mondada 94]. The above estimation includes only the time of the actions performed by the robot, but not the time for other processing, such as dealing with selection and crossover in a genetic system, executing controllers, reading/sending information from/to sensors/actuators, etc., though the latter tends to be small compared to the former. It shows that the time taken to evolve control systems for the real robot on-line is far beyond what we can afford.

From the evolution time estimated above, we can observe that the time μ for a robot to perform a single action determines the difference of duration of run in on-line and in off-line approaches. In the on-line evolution, μ is measured as real time and cannot be sped up; whereas in simulation, it means only a few instructions executed and is machine dependent – the faster the machine we use, the sooner the result comes out. In general, for a task as "obstacle avoidance", off-line evolution is about $10^2 \sim 10^3$ times faster than the on-line approach, on a single user SPARC-5 machine. Therefore, simulation becomes necessary for researchers to speed up building and prototyping their evolutionary robotics systems. Though some researchers have argued that once vision is involved, the simulation of vision will become a serious computational burden of off-line evolution [Cliff *et al.* 93, Harvey *et al.* 94], the speed of vision processing should be improvable in some ways, such as the hardware implementation of vision chips.

Although using simulation can save us a dramatic amount of time in developing evolution systems and evolving controllers for autonomous robots, it must be noted that simulations should be considered only as scientific models rather than substitutes for physical robots. The results of simulation must be verified on real robots. In the next section, we will examine the issue of transferring evolved controllers from simulation to reality.

5.2.2 The Gap between Simulated and Real Worlds

In the study of building control systems for real robots to act in the real world, there have been a number of roboticists warning us of the danger in the use of simulation in the design of robots (i.e., [Brooks 92, Smithers 94]). Their reasons include the following:

- The use of over-simplified robot simulators: Some researchers studying robot control have been using simulators of abstract models rather than the physical-based ones which are modelled carefully from the real robots. Research results based on the kind of abstract models could not, normally, be verified on real robots, and thus cannot result in any conclusion in practice.
- The difficulty of simulating the actual dynamics of the real world: It is very hard to capture the characteristics of the real sensors and actuators. Devices like sensors may not return clean readings when acting in the real world. Therefore, the sensing and actuation differences between real and simulated worlds will probably lead the control systems working well in simulation to fail on real robots even if physical-based simulators are used.

Before exploring the above problems, we should recall the conclusion of the last chapter – the evolved controllers were able to achieve the desired behaviours when they were tested in a new/changed world (though in simulation). It provides evidence that if the difference between a new world W_i and the world used in evolution W_e is within a certain range (in which case we say that they are compatible), then the controllers evolved from W_e still work reliably in W_i. Similarly, if we consider the real world as one of the W_i, and try to keep the simulated world used in the evolution compatible to (not necessarily identical with) the real world, then a controller evolved from the simulated world should be able to work well when it is delivered to the real world.

The description above reveals that what we need is a simulator compatible with, rather than identical to, the real world. Instead of simulating all the details of dynamics, we only need to capture the characteristics of the real world at a certain level, especially when we are only expecting to see a real robot with behaviour which is qualitatively rather than quantitatively similar to the simulated ones when performing the evolved controller. Therefore, though it has been argued that to simulate the detail of a physical world is probably not possible, this does not mean that the transfer from simulation to reality is extremely hard to achieve. We will demonstrate this in the experiments that follow.

5.2.3 The Comparison of Different Kinds of Simulators

There have been two kinds of simulators widely used in the community of Evolutionary Robotics: one is the mathematical modelling approach [Cliff *et al.* 93, Jakobi *et al.* 95, Michel 95, Lee *et al.* 96], and the other is the look-up table approach [Nolfi *et al.* 94, Miglino *et al.* 96, Lund & Hallam 96, Lee *et al.* 97a, Lee *et al.* 97b]. The former one describes the responses of sensors and the dynamics of actuators by a set of equations; while the latter one first records the responses of each sensor to each object in the environment through the robot's sensors themselves and records the effects (such as displacement in distance or angle) of the actuators for each allowable command before the simulation, and then accesses the sensor/actuator tables in the simulation. Before deciding which approach to take, we have an analysis of their advantages and disadvantages classified and listed below.

- The look-up table method gives more reliable responses of devices: As mentioned above, the mathematical modelling method models the responses of sensors and actuators by mathematical equations. For the convenience of modelling, this approach always assumes that devices with the same type have the same characteristics. However the hypothesis is not true. In [Miglino *et al.* 96], the authors have shown significant differences between different sensors, even if they belong to the same type and are exposed to the same external stimulus. The differences are due to the different intrinsic characteristics of sensors, and are difficult to take account of in the mathematical modelling method. However, if one ignores these intrinsic characteristics of sensors, the performance may be reduced when the evolved system is transferred to a real robot. The look-up table method does not have this disadvantage because of its direct sampling of the device-environment responses.

- The mathematical modelling method is computationally more expensive than the look-up table method when the simulator is running: Because of the use of equations to model the sensor and actuator responses, the mathematical modelling method has to solve those equations once at each time step. In contrast, the look-up table method does not have to do so because the sensor and motor tables are built before the simulation starts, and it only takes constant time to access these tables when the simulator is running. This makes the latter method faster than the former one. This situation is particularly apparent when the environment becomes relatively complicated where the number of equations are increased and consequently the simulation is slowed down. For a typical Evolutionary Robotics experiment such as evolving obstacle avoidance behaviour for a Khepera robot, employing the look-up table approach is at least 3 times faster than the use of the mathematical modelling approach [Lund & Hallam 96].

- The look-up table method is not scalable in constructing the tables of sensor and actuator responses: Since this kind of simulation is based on the robot's own sampling of sensor and motor responses, all these responses must be recorded before the simulation starts. Although the procedure of sampling can be done automatically [Nolfi *et al.* 94, Miglino *et al.* 96], it is still computationally expensive when the environment becomes more complicated. Furthermore, if there are objects with non-symmetrical shapes, we will need to sample the responses of sensors from different angles. In addition, robots like Lego vehicles do not have precise motor systems as does the Khepera robot, so when building tables for that kind of robot it may be necessary to repeat the sampling procedure more times and then take the average values as the response [Spagocci 96]. In the above situations, the building of a simulator of this kind is quite time-consuming. This disadvantage does not apply to the mathematical modelling method since the equations can be easily added to the simulator in that approach.

Although both of the above methods have their own disadvantages, each has shown its capabilities to evolve controllers which can be transferred to real robots successfully [Jakobi *et al.* 95, Miglino *et al.* 96]. From their reports, however, we observe that the look-up table approach gives the best match of behaviours in simulation and reality: by measuring real sensor and motor responses, one can then build a simulator with the least gap to the reality. In the next section, we will briefly describe how to use the look-up table technique to develop a simulator; and in the following sections, we will show some examples of using this kind of simulator to evolve controllers for a real robot.

5.3 The Real and Simulated Robots Used in this Chapter

As mentioned earlier, the approach used in this chapter is to evolve controllers in simulation and then to verify them on a real robot. Building a simulator for the real robot is thus involved. In this section, we will first describe the robot to be used in the later experiments and then explain how we construct a simulator for it through the look-up table method.

5.3.1 The Khepera Robot

The robot used to test the evolved controllers is the miniature mobile robot *Khepera* [Mondada *et al.* 93] (Figure 5.1). It is designed by LAMI[1] and has been widely used in the study of autonomous robots, in particular, in the field of Evolutionary Robotics. It is designed in a modular manner. Different kinds of modules and sensors can be added easily without the need for changing software. This flexibility allows the researchers to use the most suitable configuration to conduct their own experiments. The main characteristics of this robot are described below and the details can be found

[1] Laboratory of Microcomputing, Swiss Federal Institute of Technology

in [Mondada *et al.* 93] or the home page of Khepera[2].

The Khepera robot is cylindrical with a diameter of 55 mm, a height of 30 mm, a weight of 70 g, and is supported by two wheels and two small Teflon balls. Its two wheels are driven by two DC motors with incremental encoders (12 pulses per mm of robot displacement) and both motors can revolve forward and backward. The robot is equipped with eight infra-red proximity sensors; six sensors are positioned on the front and the other two on the back. These sensors can also be used in a different mode to measure ambient light around the robot. For identification purposes, the sensors are numbered from 0 to 7 as illustrated in Figure 5.1.

In the internal design, the robot uses a Motorola 68331 controller with 256 Kbytes of RAM and 512 Kbytes of ROM to manage all the input-output routines, and can communicate with a host computer via a serial port. Thus, the robot can be controlled by the host computer via a cable or can run the control program on its own control chips. The former allows the robot to exploit all the computational power and storage capabilities available in the host computer; and the latter provides the ability to test an evolved controller by downloading it to the real robot and running it on the on-board processor. In the experiments that follow, the controllers evolved from simulation will be downloaded to the robot to evaluate the performance without cable connection.

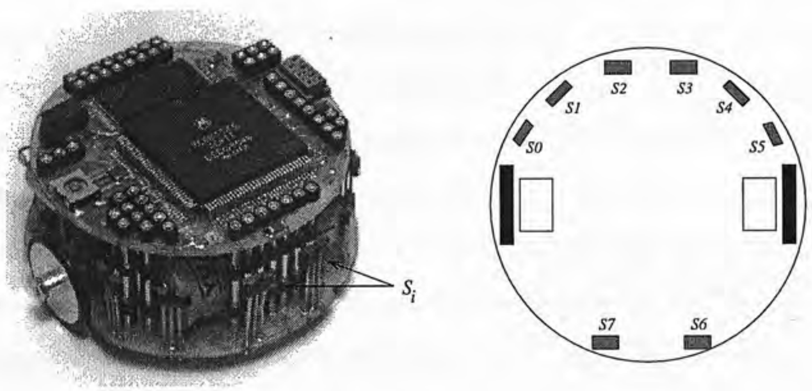

Figure 5.1: The Khepera miniature robot and its sensor arrangement. In the right figure, a sensor S_i can function as an infra-red or an ambient light sensor.

[2] http://www.lamiwww.epfl.ch/Khepera/

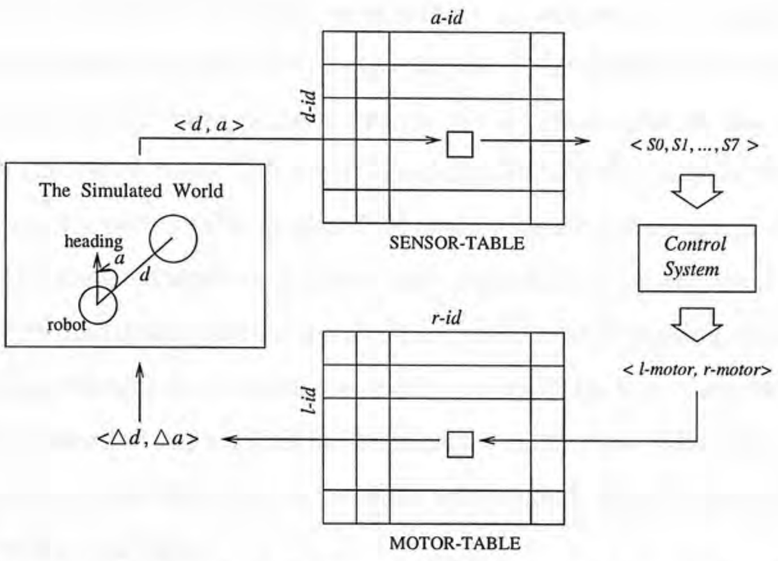

Figure 5.2: The sensing-motion cycle of the look-up table approach in simulation. In this figure, d and a represent the relative distance and angle of the robot to an object, respectively; Δd and Δa mean the changes in distance and angle when the robot performs the motor command $< l-motor, r-motor >$ produced by the control system for one time step.

5.3.2 Building a Simulator from Real Sensor and Motor Response

The basic idea of building sensor tables is to sample the robot's sensor responses to different types of objects in the environment. To take the samples, the robot is placed in front of an object, which could be a wall, an obstacle, or a light source, and then its sensor responses are recorded at different distances. For a certain distance, the robot is turned 360 degrees in steps of 2 degrees; at each step its eight sensor responses are recorded 10 times and then the averages are used to construct sensor tables. The motor table is constructed in a similar manner. For the motors of the robot, there are some speed settings available which enable the robot to move and turn. By setting allowable speeds to the two independent motors separately, we can measure and record the displacement, including distance and angle, as the entries of the motor table.

After these tables are built, they are used in the simulation. The sensing-motion cycle of the look-up table approach is illustrated in Figure 5.2. At each time step in the simulated world, the relative distance and angle between the simulated robot and a certain object are firstly calculated and indexed, and then all sensor responses of an entry are read from the sensor table according to the indices of distance and angle.

This is done for each relevant object, to construct a complete set of simulated sensor responses. These sensor responses are used as inputs to the control system. Then motor commands, that are the corresponding outputs of the control system, are converted to the indices of the motor table (the actual mapping between the outputs of a controller and the indices of a motor table is described in the experiment section). According to the indices, the displacements of distance and angle caused by the revolution of the motors are read and then used to update the position and heading of the robot in simulation. The changes in distance and angle produced by the motor table are used in simulation, instead of any explicit model of motor movement. The sensor and motor tables are used to avoid the need to simulate sensors and motors explicitly; they are not needed with a real robot.

Since the responses of sensors and motors are recorded by the robot itself, the simulator based on them can be very precise. The experiments in the later sections of this chapter will show that the controllers evolved from the simulation can be successfully transferred to a real Khepera robot in which very similar behaviours on simulated and real robots can be observed. However, due to the fact that the displacements of distance and angle generated by each motor command (actually a pair of commands, one for left motor and one for right motor) are obtained by isolated measurement,[3] sometimes the motor command chaining effect in reality makes the behaviours of simulated and real robot differ. Section 6.5 will provide an example in which the controller evolved by simulation cannot be transferred to a real robot successfully.

5.4 From Simulation to Reality

As mentioned, the final aim of Evolutionary Robotics is to evolve controllers which work on real robots. Once we have shown that the developed GP system can evolve control systems working well in simulation, the next step is to examine whether the evolved controllers can work on real robots. In this section, we intend to use our GP system with the simulator described in the above section to evolve controllers to

[3] for each command, the robot starts from a static situation and executes the same command for 100 time steps, and then the average displacements of distance and angle are used as the effect of a single step motion.

grouped outputs	*index (in simulation)*	*command (in reality)*
0 0 0	3	-2
0 0 1	2	-4
0 1 0	1	-6
0 1 1	0	-8
1 0 0	4	2
1 0 1	5	4
1 1 0	6	6
1 1 1	7	8

Table 5.1: The grouped outputs and their corresponding motor activities. In this table, *index* means the index for accessing the motor look-up table in simulation, and *command* means the actual value sent to the Khepera's motor in reality.

achieve various tasks in simulation first, then transfer them to a real Khepera robot to evaluate the corresponding performance.

According to the experience learned from the last chapter, in the following experiments, multiple populations are used to enhance the performance of the evolution. Also we define a training set of fitness cases which are sampled randomly at each generation to evaluate the controllers for multiple trials during the run, in order to evolve reliable and robust controllers. The best individual that will appear in the last generation is designated as the final solution.

Since the controllers we intend to evolve in this chapter are all monolithic and reactive, the inputs of the controllers are from sensors and the outputs are used to drive motors directly. In our design, each controller has six outputs which are organised into two groups: the first three and the last three outputs are decoded as motor commands to control the left and right motors respectively. In each group, the first output is interpreted as the revolving direction (1/0 for forward/reverse) and the other two are decoded as different revolving speeds. Thus, there are 8 different motor commands available for one motor, which means there are in total 8$\times$8 different combinations of moving speeds and turning angles for the simulated and real robots in the experiments below. In simulation, the groups of outputs are used as indices of the motor tables; while in reality, they are converted to real motor commands to control a real Khepera robot. Table 5.1 lists the interpretations of different outputs for the motors. In all the following experiments, each motor command will last 200 ms for both simulated and real robots.

Terminal Set:	IR, $\Re\{0..1\}$
Function Set:	PROG, AND, OR, NOT, XOR, >=
number of populations:	2
population size:	50
number of generations:	40
fitness cases:	3 fitness cases are used; they are randomly sampled from 5 pre-defined fitness cases at each generation

Table 5.2: The key features of the problem of evolving an *obstacle-avoidance* robot.

5.4.1 Obstacle Avoidance

The first behaviour controller we would like to evolve is one that can fulfill the task of *obstacle avoidance.* In fact, we have evolved controllers for the same task in the previous chapter. Here we attempt to evolve the controller to show that using a specific simulator associated with a physical robot, one can transfer the controller evolved from the specific simulator to the corresponding real robot successfully. In other words, we can expect similar behaviours on the real and simulated robots.

In this experiment, the robot is restricted to use IR sensors to detect objects, so only IRs and the numerical thresholds are defined as terminals to our GP system. The fitness function defined here is also the same as the one used in section 4.3.2, which is

$$f = \sum_{t=1}^{T}[\alpha \times max\_response(t) + \beta \times (1 - v(t)) + \gamma \times w(t)]$$

where T, $max\_response(t)$, $v(t)$ and $w(t)$ have the same definitions as the ones described in section 4.3.2; α, β, γ are 0.15, 0.3, and 0.55 (determined by preliminary testing), respectively; and the final fitness of an individual is the summation of fitness from different trials (fitness cases).

To evolve an obstacle avoidance controller, we used two populations of 50 individuals. The number of time steps for evaluating a controller in a single trial was 300, and the number of generations was 40. For this task, 5 different positions and orientations were defined in the training set; and 3 of the pre-defined 5 positions were sampled randomly at each generation to be the initial positions of different trials. Table 5.2 summaries the key features of the problem of evolving an obstacle-avoidance robot here. To evaluate this evolution experiment objectively, we conducted 10 independent runs for the same task; 8 of them evolved controllers with the expected behaviour. In

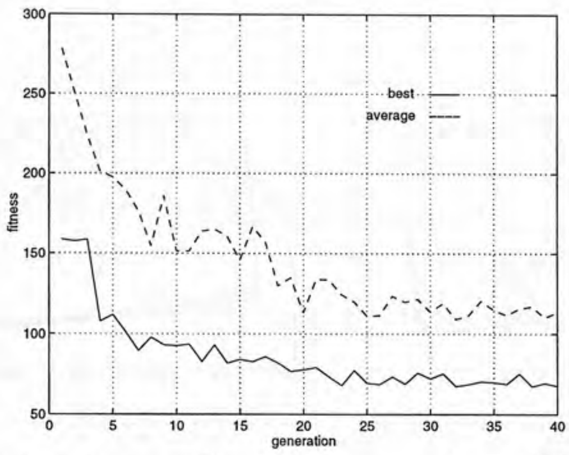

Figure 5.3: The best and average fitness during a run for evolving an obstacle avoidance controller.

general, the evolution has converged to a stable state within 20 $\sim$ 30 generations. A typical example of how the fitness converged is shown in Figure 5.3, and the controller evolved from this example is

```
(PROG
    (AND (XOR (>= 0.68 IR1)(>= IR4 0.22))(AND (>= 0.04 IR2) (XOR (>= 0.68 IR1)
            (>= IR3 0.22))))
    (NOT (AND (>= IR4 0.43)(>= IR2 0.43)))
    (>= IR6 0.55)
    (>= 0.96 IR4)
    (>= IR0 IR0)
    (AND (AND (>= 0.09 IR3)(>= IR5 0.97))(AND (XOR (>= 0.09 IR1) (>= IR4 IR5))
            (>= 0.09 IR3))))
```

Some of the obstacle avoidance behaviours generated by the above controller are demonstrated in Figure 5.4. As can be seen, the simulated robot moved straight forward most of the time, but the robot was able to move away from the obstacles whenever it approached to them. We can also analyse the behaviour in terms of the sensor and motor activities. Figure 5.5 shows the related sensor and motor information corresponding to the behaviour in Figure 5.4(a). We can observe that when the robot approached the object, the *max_response* tended to be high (Figure 5.5(a)), and this caused the robot to slow down (Figure 5.5(b)) and rotate (Figure 5.5(c)) to avoid the object. From the qualitative and quantitative observations above, it has been shown that the robot was able to achieve the specified task in simulation.

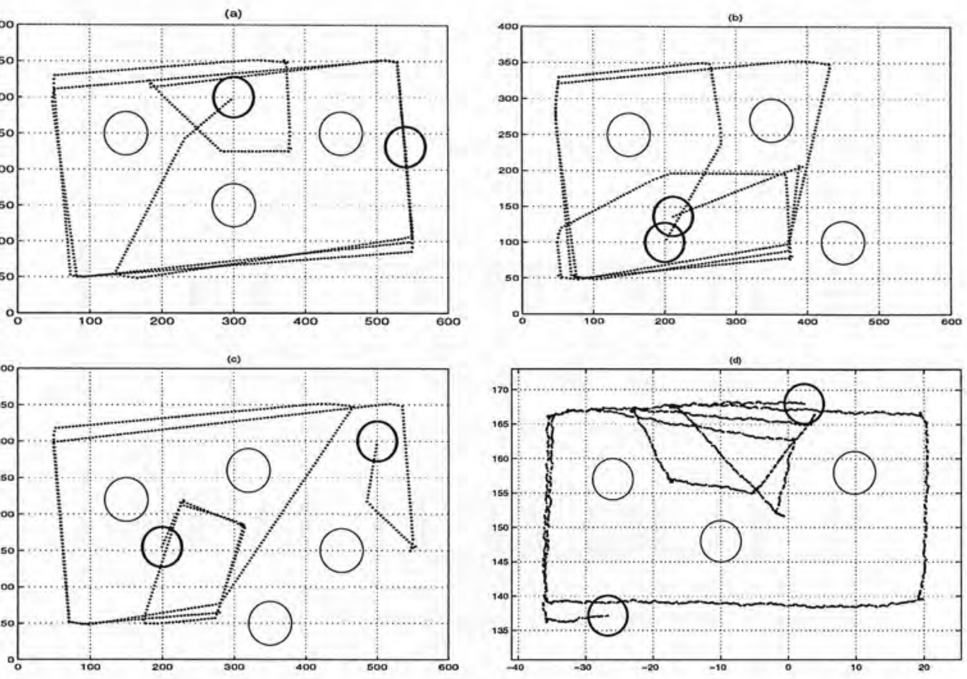

Figure 5.4: The obstacle avoidance behaviours of the simulated (a)(b)(c) and real (d) robots. In these figures, the darker circles are the robots. The figure for the real robot is obtained by setting LEDs on the tops of the robot and obstacles, and using a video tracking system to record their trajectories.

As mentioned earlier, the main aim of Evolutionary Robotics is to evolve controllers for real robots. Once the simulated robot can achieve the task, we then transferred the controller evolved from the simulation to a real Khepera robot. This downloaded controller was tested ten times for slightly different conditions and the robot can achieve the tasks in all of the tests. The typical obstacle avoidance behaviour produced by the real robot is presented in Figure 5.4(d). This figure is obtained by setting LEDs on the tops of the robot and obstacles, and using a video tracking system to record their trajectories [Lund *et al.* 96]. The behaviour of the real robot is very similar to the behaviour of the simulated robot, which shows that we have been able to evolve control system for real robots.

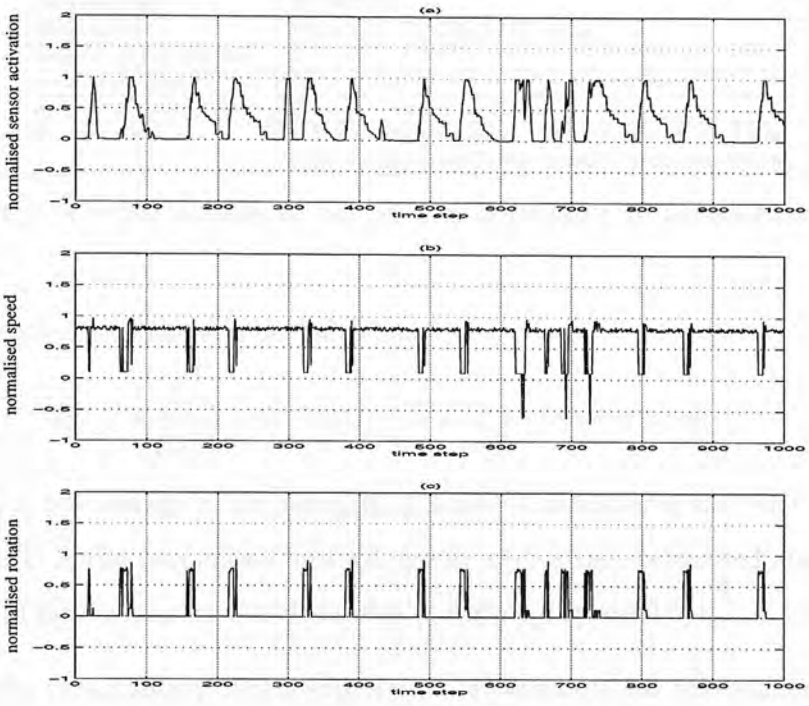

Figure 5.5: The relevant sensor and motor activities corresponding to the Figure 5.4(a). (a) is the maximum of the normalised sensor activations; (b) and (c) are the normalised speed and turning rate of the simulated robot.

5.4.2 Box-Pushing

The task of *box-pushing* is that the robot should keep pushing a box forward as straight as possible. To achieve such a task, the robot needs to use its IR sensors to acquire perceptual cues for the location of the box. Therefore, we define two kinds of terminals, IRs and numerical thresholds, for our GP system to evolve controllers capable of achieving this task.

The fitness function can be formulated through the quantitative description of the expected behaviour, which is to keep the activation value of its front IR sensor high, the robot moving forward, and the speed difference between the two motors low. The pressure from keeping the front IR sensors with high activation values is to reinforce the robot to head toward a box, and the pressure from keeping the robot moving forward with low speed difference is to encourage the robot to move straight and to prevent it from getting stuck in front of a box. The combination of these will lead to a pushing-forward behaviour. Thus, the fitness function for evolving a behaviour

Terminal Set:	IR, $\Re\{0..1\}$
Function Set:	PROG, AND, OR, NOT, XOR, >=
number of populations:	2
population size:	50
number of generations:	40
fitness cases:	6 fitness cases are used; they are randomly sampled from 8 pre-defined fitness cases at each generation

Table 5.3: The key features of the problem of evolving an *box-pushing* robot.

controller of box-pushing can be defined as:

$$f = \sum_{t=1}^{T}[\alpha \times (1 - s(t)) + \beta \times (1 - v(t)) + \gamma \times w(t)]$$

where $s(t)$ is the average of the normalised sensor activations of the front sensors IR*2* and IR*3*; $v(t)$ is the normalised forward speed; $w(t)$ is the normalised absolute speed difference of two motors at each time step t; and α, β, γ are 0.6, 0.2, 0.2 here.

To prove the repeatability of this experiment of evolution, ten independent runs were conducted. For each run, two populations of 50 individuals were used and each individual was evaluated for 150 time steps during a single trial. The GP system was run for 40 generations. The key features of the problem of evolving a *box-pushing* robot are listed in Table 5.3. For this task, the training set was defined to include eight different cases in which each case was a different starting position and orientation around the box. During the run, six cases from the pre-defined ones were randomly sampled at each generation to evaluate the performance of controllers. With the parameters above, nine out of ten runs can produce successful controllers which were able to achieve the box-pushing task all the time from different starting positions and orientations in simulation. Figure 5.6 shows how the fitness curve converged in one of the successful runs, and the controller evolved from this example is:

```
(PROG
    (OR (>= 0.13 IR0)(>= 0.13 IR0))
    (>= IR1 IR1)
    (XOR (>= IR3 IR7)(OR (>= 0.13 IR0)(>= 0.13 IR0)))
    (>= IR1 0.36)
    (>= IR1 IR1)
    (XOR (>= 0.13 IR0)(>= IR2 0.13)))
```

The typical box-pushing behaviour of the simulated robot is illustrated in the left figure

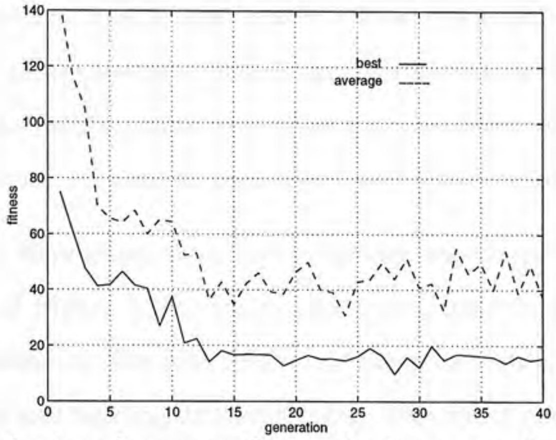

Figure 5.6: The best and average fitness during a run for evolving a box-pushing controller.

of Figure 5.7. As in the above section, we can analyse the related sensor and motor information to examine whether the robot fulfills the fitness requirements, when it is performing the evolved controller. As described, the robot is expected to have high activations from the front IRs, which implies the robot keeping heading toward the box. Figure 5.8(a) presents the average activation of the two front IRs, and it shows that the robot was able to achieve the requirement. In addition, Figure 5.8(b) and Figure 5.8(c) present the left and right motor commands generated by the evolved controller, corresponding to the box-pushing behaviour in Figure 5.7(a). We can see that both motors revolved at the same and relatively high speed most of the time; while at some specific time steps, the left motor revolved reversely at high speed and

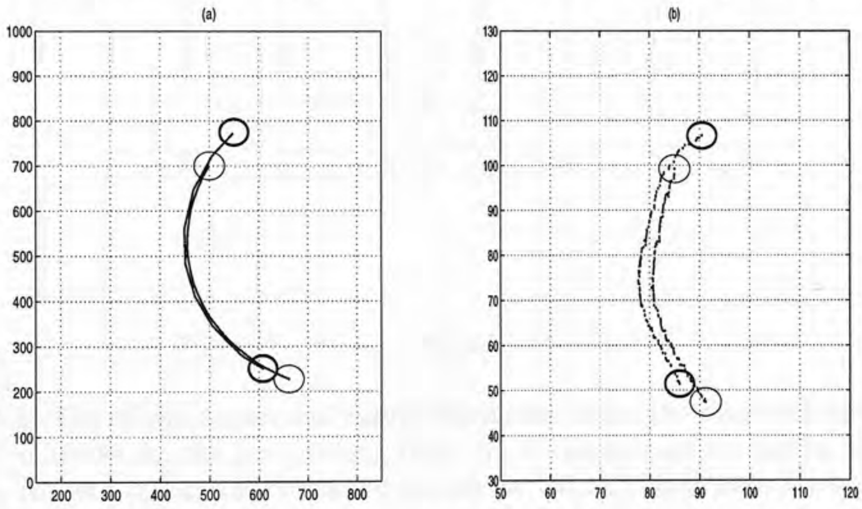

Figure 5.7: The trajectories of simulated (a) and real (b) robots when they are pushing a box (the darker circles represent the robots and the boxes are pushed from top to bottom).

the right motor revolved even faster forward. These periodical changes were to produce prompt turns to drive the robot back from path deviations to head towards the box again. The figures of motor commands show that the robot can satisfy the other fitness requirements: moving forward at high speed with little turning.

After evolution in simulation, the above controller was transferred to a Khepera robot. The right figure of Figure 5.7(b) shows the typical behaviour of the real robot. This controller was tested on the real robot ten times and each time it started from an arbitrary position and heading around the box. The robot always generated consistent behaviour: it turned to face the box and then approached and pushed it. We can also see that the real robot recovered from deviations as it did in simulation. Although the robot did not contact and push the box exactly at its mid-position, it did achieve the goal reliably: continuously pushing the box forward.

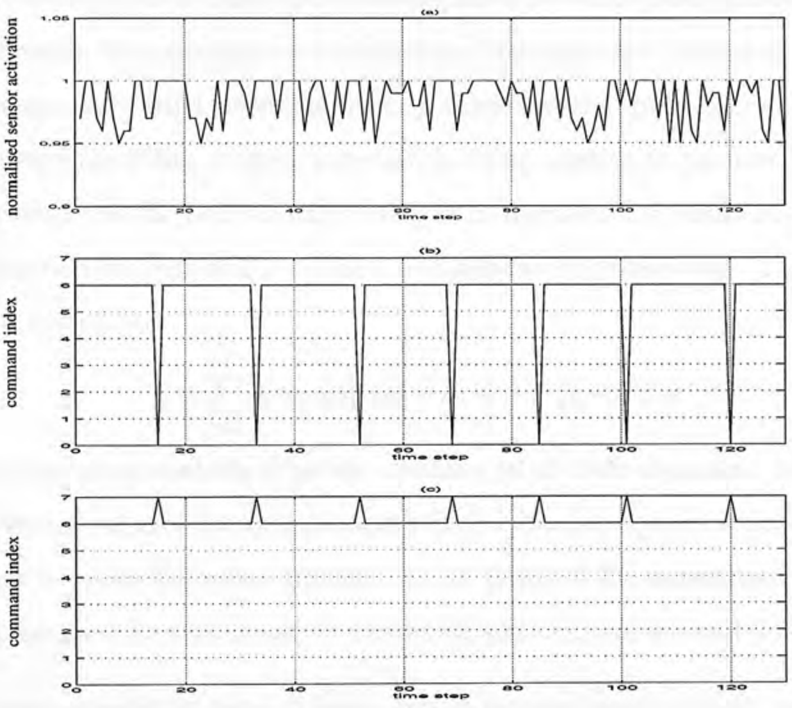

Figure 5.8: The related sensor and motor information when the robot is performing the evolved controller for the box-pushing task. (a) is the average activation of the front two IRs; (b) and (c) are the command indices for left and right motors respectively.

Terminal Set:	IR, $\Re\{0..1\}$
Function Set:	PROG, AND, OR, NOT, XOR, >=
number of populations:	2
population size:	50
number of generations:	40
fitness cases:	6 fitness cases are used; they are randomly sampled from 8 pre-defined fitness cases at each generation

Table 5.4: The key features of the problem of evolving a *box-side-circling* robot.

5.4.3 Box-Side-Circling

The task of *box-side-circling* is that the robot needs to keep moving forward and circling along the sides of a box. As it does in the box-pushing task, the robot should use its own IRs to capture the location of the box. Thus, terminals for evolving a controller to achieve this task are the same as those in the box-pushing task: IRs and numerical thresholds.

Again, we should define a fitness function to guide the evolutionary process, and it can be done through the quantitative description of the expected behaviour: to keep the left side sensor IR*0* with a certain activation value and the speed positive. The former is to encourage the robot to keep a certain heading relative to the box and a certain distance away from the box; and the latter is to reinforce the robot moving forward. The combination of these will produce a box-side-circling behaviour. Thus the fitness function is defined as:

$$f = \sum_{t=1}^{T}[\alpha \times abs(s(t) - k) + \beta \times (1 - v(t))]$$

where abs is a function which gives the absolute value of its argument; $s(t)$ is a normalised activation value of the specific sensor IR*0*; k is a pre-defined constant indicating the distance between the robot and the box, in terms of the normalised sensor range; v is the normalised forward speed of the robot; and α, β are 0.8 and 0.2 here.

Ten runs were conducted with different initial populations to obtain more objective results. For each run, two populations of 50 individuals were used and the number of generations was 40. Table 5.4 lists the main features of the problem of evolving a robot able to circling around a box. Eight different fitness cases were pre-defined; each case represented a different starting position and heading for the robot. At each generation, 6 from the 8 were sampled at random to evaluate the performance of each controller,

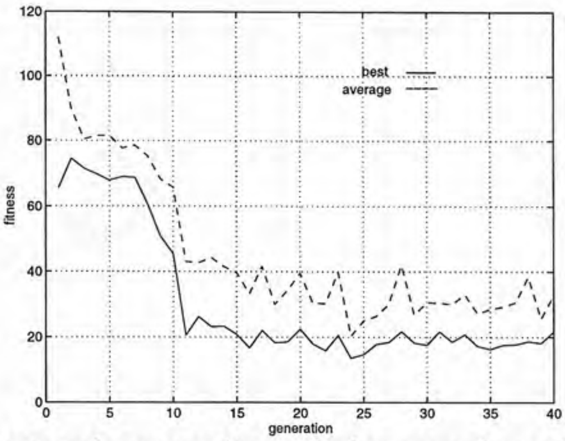

Figure 5.9: The best and average fitness during a run for evolving a box-side-circling controller.

and each trial lasted 150 time steps. With the parameters specified, all of the ten runs can evolve successful solutions. Figure 5.9 shows the converged fitness curve for one of the runs, which indicates that the expected controller can be found rapidly. The controller evolved from this run is

```
(PROG
    (NOT (>= IR6 IR1))
    (>= IR1 0.78)
    (OR (>= IR5 IR1)(>= IR3 IR3))
    (OR (>= 0.32 IR3)(>= IR5 IR4))
    (>= IR4 IR3)
    (OR (>= IR5 IR1)(>= IR6 IR3)))
```

Figure 5.10(a) presents the evolved box-side-circling behaviour of the simulated robot, which shows that the task was achieved successfully in simulation. We can also observe the related sensor and motor information to examine if the evolved robot satisfied the fitness requirements. In general, we can see that the activations of the specific sensor IR*0* were quite close to the pre-defined threshold 0.9 (Figure 5.11(a)); both motors kept revolving forward at constant but different speeds (Figure 5.11(b) and (c)), and that caused the robot to move forward with constant turning rate counter-clockwise. We can also observe that at certain time steps, noise caused the right motor to slow down and led the robot to turn slightly toward the right hand side (the activation value of IR*1* in Figure 5.11(a) dropping from a certain value down to zero indicates this). But this kind of deviation was recovered soon due to the slowing down of the left motor.

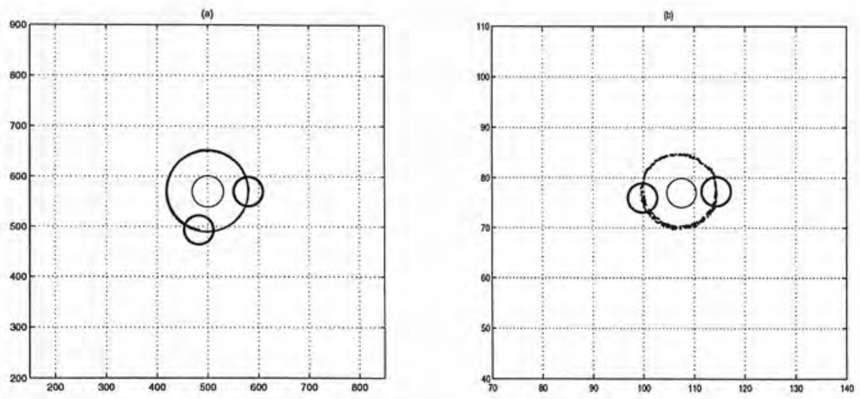

Figure 5.10: The box-side-circling behaviours of simulated (a) and real (b) robots. In this figure, the darker circles represent the robots.

After the heading was adjusted to the appropriate direction, the robot kept circling the box with constant speed and turning rate again. These figures demonstrate that the robot was able to achieve the task by satisfying the fitness requirements.

The evolved controller was then transferred to the real robot, and the typical behaviour of the real robot is demonstrated in Figure 5.10(b). We tested this evolved controller ten times by putting the real robot around the box with an arbitrary heading each time. In all the tests, the robot showed similar behaviour: it performed turning to adjust its heading first and then moved along the side of the box. From the testing results, we can conclude that the robot is able to achieve the specified task reliably.

5.4.4 Exploration

This task is that the robot needs to wander safely in an enclosure and visit as much of the enclosed space as possible. It can be described quantitatively as that the space is divided into some grid squares and the robot must visit as many squares as possible during a fixed period of time. There are different ways to achieve this task. For instance, it can be achieved by the use of a map to provide location information to the robot. With location information, the robot can realise its own location and the locations of those squares that have been visited already. It can then head to those squares which have not been visited according to the records of the map. There is also another kind of strategy without a map; in other words, there will not be location information available. In such a strategy, the robot does not know where

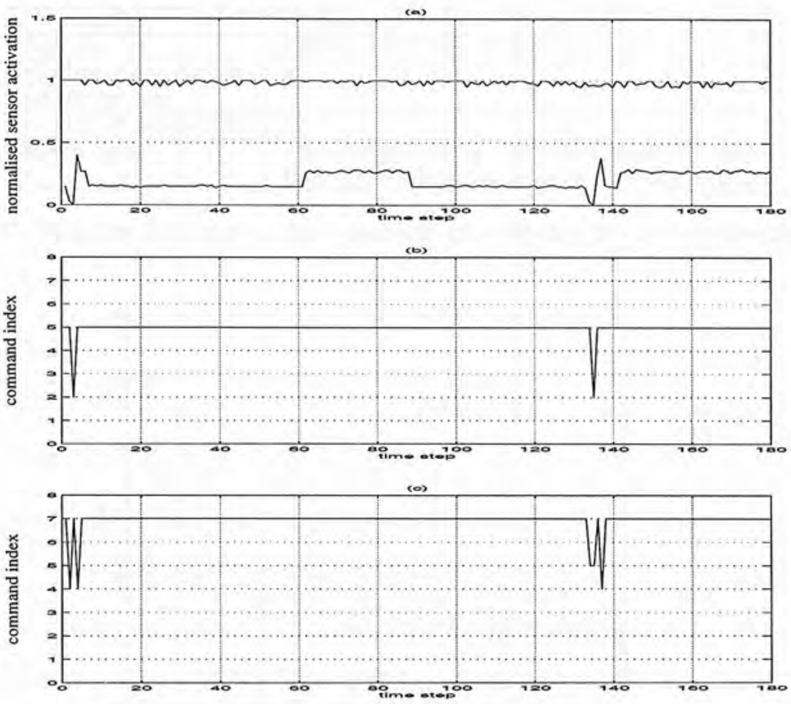

Figure 5.11: The relevant sensor and motor activities for the behaviour shown in Figure 5.10(a). (a) includes the activations of sensor IR*0* (higher) and IR*1* (lower); (b) and (c) are the commands indices for left and right motors respectively.

it is and which squares it has not visited. To carry out the exploration task, the robot needs to determine its turning angle carefully in the situations when it senses the boundary of the enclosure. Actually, it has been shown that without using a map, the exploration task is achievable by a controller with or without internal states [Miglino *et al.* 94, Lund & Hallam 96]. In the experiment below, we intend to evolve a reactive controller to explore an space without using location information.

Unlike the experiments presented above, the fitness measurement for this task is not the sum of penalties over each time step but rather can only be assigned after a complete trial. The main concern for the fitness here is to minimise the number of squares which have not been visited while an extra pressure on the speed is added to encourage the robot to move forward in exploring. Thus the fitness function is defined as:

$$f = \alpha \times (1 - P) + \beta \times (1 - Avg)$$

in P is the proportion of the space visited, i.e., $\frac{visited-grids}{total-grids}$; Avg is the average speed of the robot during a complete trial; and α (= 0.85), β (= 0.15) are the corresponding

Terminal Set:	IR, $\Re\{0..1\}$
Function Set:	PROG, AND, OR, NOT, XOR, >=
number of populations:	2
population size:	50
number of generations:	40
fitness cases:	4 fitness cases are used; they are randomly sampled from 6 pre-defined fitness cases at each generation

Table 5.5: The key features of the problem of evolving a *safe-exploration* robot.

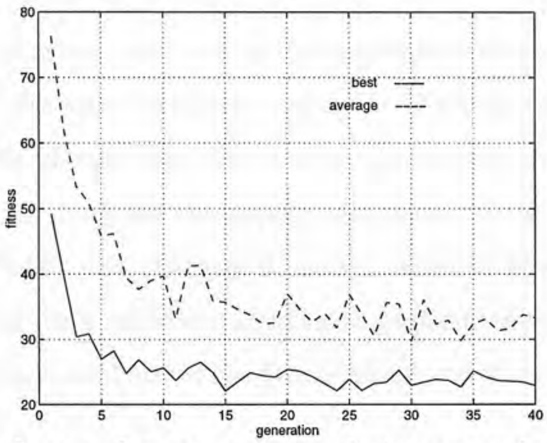

Figure 5.12: The best and average fitness during a run in evolving an exploration controller.

weights. The enclosure in the exploration experiment here is a square of 50 × 50 cm and each grid square is 5 × 5 cm.

As described above, the controller to be evolved is reactive and there is no location information provided here, so the robot must fully exploit its IR sensors to determine the turning angle carefully to achieve this task. Since IR sensors are the only mechanism for providing perception cues, the terminals for the exploration task are then defined to include IRs and numerical thresholds as in other tasks. To evolve a controller for exploration, two populations of 50 individuals were used and the GP system was run for 40 generations as before. The main features of this problem are summaried in Table 5.5 For this task, 6 different starting positions were defined in the training set and 4 of them were sampled randomly at each generation as different trials to evaluate controllers. Each individual was evaluated for 1000 time steps in each trial. Figure 5.12 shows the typical converging fitness curve for evolving controllers of exploration. It indicates the fast and smooth converging behaviour of the evolution. The evolved controller corresponding to this experiment is:

```
(PROG
    (>= IR7 IR7)
    (>= IR7 IR7)
    (>= IR7 IR5)
    (NOT (OR (>= IR4 0.17)(>= IR1 0.97)))
    (>= 0.72 IR6)
    (AND (NOT (>= 0.17 IR7))(NOT (>= 0.72 IR4))))
```

The exploration behaviour produced by this controller is demonstrated in Figure 5.13(a), which shows that the robot is able to visit most of the specified arena during a fixed period of time. We should note that it is not important how the robot moves when it does not sense anything but the appropriate match of the turning angle (when the robot senses the wall) and the way it moves (when it does not sense anything) is nevertheless crucial for a reactive controller to perform exploration. As we can see in Figure 5.13(a), a successful match has been evolved and it enabled the robot to achieve the task.

Like the previous experiments, the evolved controller was downloaded to the real Khepera after the simulation. The behaviour observed from the real robot is presented in Figure 5.13(b). This behaviour is very similar to the one produced by the simulated robot in the simulated environment. Once again, it shows a successful example that we are able to evolve a behaviour controller by our GP system in simulation, and then transfer it to a real robot.

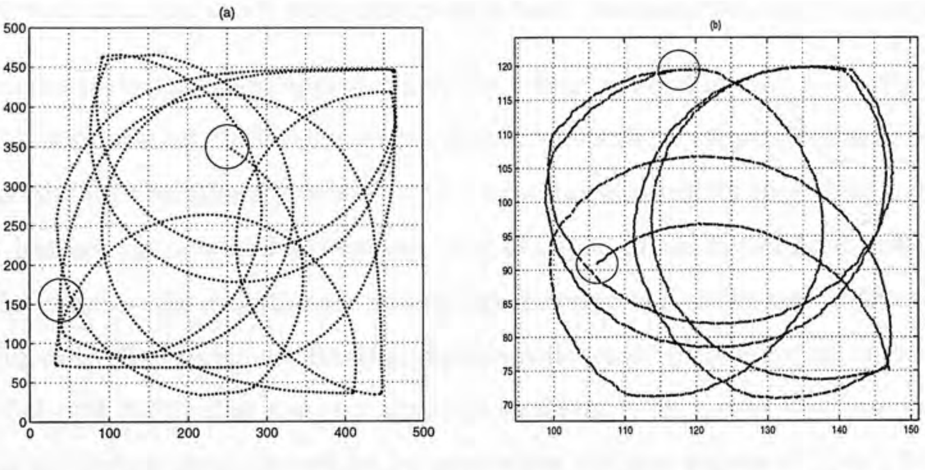

Figure 5.13: The exploration behaviours of simulated (a) and real (b) robots.

5.5 Summary and Discussion

In this chapter, we have analysed the evolution time of evolving control systems for robots, explained the main reason for using simulation, and compared the advantages and disadvantages of different approaches to simulation. To prove that our GP system can evolve controllers for real robots, we also used it, with a simulator built by look-up table techniques, to evolve various behaviour controllers for different tasks and then transferred the evolved controllers to a real robot. Experimental results show that the behaviours produced by the real robot look very similar to the ones observed in simulation.

Compared to other research work using GP techniques to evolve robot controllers, our work is more efficient: the population size used in our work is relatively small and the solution is relatively succinct. In our experiments, we evaluated 100 individuals at each generation and can evolve reliable and robust controllers, while in [Koza & Rice 92], the author used population size of 500 to evolve box-pushing controllers which have not been proved to be reliable. It means that he may have to increase the population size if reliable solutions are required. Furthermore, in [Reynolds 94b], the author even needed a population size of 2000 to evolve a satisfactory controller. In addition, in our work, the evolved controllers (not simplified) are constituted by only 30 $\sim$ 50 nodes (terminals and non-terminals) while the controllers evolved from Koza's work include a few hundred nodes. And most importantly, we have shown that our evolved controllers work well on a real robot while others have been demonstrated only in simulation.

Since the evolution-based approach involves a large amount of evaluations for individuals, it is important to develop more efficient methods to reduce the time consumed, especially for evolutionary robotics. Our experimental results show that our GP system has several potential advantages. For example, in the application tasks achieved in this chapter, the evolutionary system can converge to a stable and sufficient solution within only 30 generations. Besides, our controllers only involve logical operators, such as AND OR NOT, that are very simple to evaluate. This means that our approach is computationally cheap; especially, in cases where the first argument of an AND is false, or the first argument of an OR is true, there is no need to evaluate the other argument.

In addition, because our controllers are composed of very basic logic components, they can be easily compiled to custom hardware such as FPGAs to speed up the evaluation in controlling a robot.

Although we have provided some successful examples of evolving controllers for real robots, the tasks achieved are however not particularly difficult. We should further investigate how to develop control systems to achieve more complicated tasks by the evolutionary approach. Shall we still evolve the controller as a whole, or shall we use the developed techniques to evolve behaviour competences first and then integrate them by an evolutionary approach in a hierarchical way? We will explore this problem of scalability in the next chapter.

Chapter 6

Evolving Control Systems for Complex Tasks

As shown in the reviewed work and our experiments, evolutionary techniques seem promising for synthesising control systems for robots. Yet, the tasks achieved so far, such as obstacle avoidance or light seeking, are relatively simple. To help the human designer develop control systems which are difficult to handcode, we have to resolve the problem of how to scale up the evolution-based approach in controlling robots. This chapter will discuss, from different points of view, some of the difficulties one will encounter for evolving controllers to accomplish complex tasks, and conduct experiments to tackle the problems of applying evolutionary techniques to tasks with increasing complexities.

6.1 The Scale-up Problem in Evolutionary Robotics

In Chapter 3, which reviews related work, we have mentioned that different research works have proposed various ways for extending the methods they describe, for evolving control systems for more difficult tasks. For example, Harvey developed an incremental evolution system in which the fitness function is gradually changed at different stages to guide the evolution of control systems [Harvey 93]; Gruau used an indirect encoding scheme to reduce the search space [Gruau 95]; while Colombetti and Dorigo used a shaping technique which involves task decomposition to solve more difficult tasks [Colombetti *et al.* 96]. This section will first examine some potential problems in scaling up the evolutionary approach to achieve complex tasks, and then analyse the pros and cons of some possible ways to overcome these problems.

The purpose of evolutionary robotics is to investigate how to develop robot control systems by evolutionary techniques. We thus should analyse the scaling up problem from two different points of view: one is about the problem domain itself – robot control; and the other is about the applied approach – the evolutionary computation technique. The former is to analyse the properties of the control architectures involved which are directly related to robot control; while the latter is to examine what kinds of difficulties will arise in operating the evolutionary techniques when the tasks become complicated. The following two subsections will discuss the scale-up problem from the two points of view.

6.1.1 About Control Architectures

As shown in the reviewed work, two kinds of control architectures are used to support different evolutionary systems: *centralised* (*monolithic*) and *distributed* ones. Centralised architectures here refer to the ones that do not have explicit control structures. They are evolved as a whole: all different kinds of sensor candidates are provided to an evolution system, which is expected to choose necessary sensors and to perform sensor fusion to integrate all information from the chosen sensors in order to generate appropriate control outputs. On the other hand, distributed architectures here refer to those similar to the behaviour-based systems. They have explicit structures, including some independent control modules and coordination modules. In general, work employing a distributed architecture also involves the decomposition of the target task, in which a task is divided into some subtasks recursively and a distributed architecture is designed to fit the decomposed result in structure. In this kind of architecture, each distributed control module only takes responsibility for its corresponding subtask, thus its sensory-motor cycle is relatively clear and simple: only that sensory information directly related to this specific subtask is involved. Figure 6.1 illustrates the typical aspects of the two different control architectures.

From the point of view of controlling a robot, the centralised architecture has some disadvantages. Firstly, it has the problem of sensory bottleneck: all sensor data must

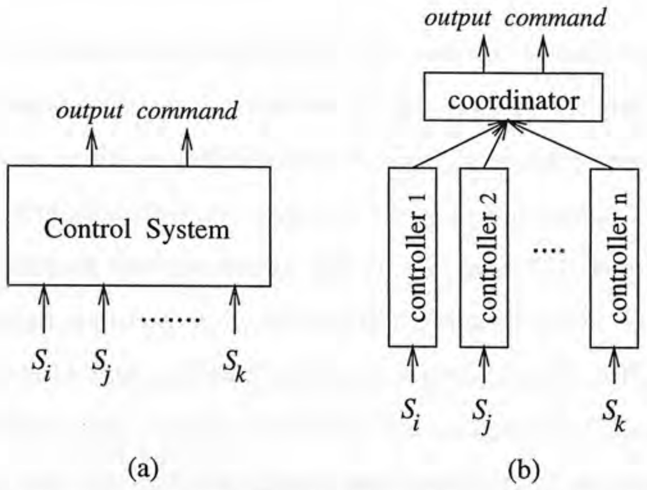

Figure 6.1: The general design of centralised (a), and distributed (b) control architectures. In this figure, S_i, S_j, S_k could be any kind of sensory information, such as sonars, infra-reds, cameras, etc.

be collected, handled, and integrated; this may degrade the performance of the control system. Besides, to perform sensor fusion for all sensory information is difficult because the information from different sources, such as cameras, sonars, maps, and so on, is normally not amenable to integration. In addition, the centralised architecture may have the problem that the entire control system will cease to function if only one portion of the system breaks; it does not have the characteristic of graceful degradation. These considerations indicate that using the approach of evolving controllers as a whole will have to solve the difficult sensor fusion problem by evolutionary techniques, and the evolved control systems may have the performance problem as mentioned above.

A potential problem may also arise from the system-design level, if one intends to evolve control systems as a whole to achieve complex tasks. In general, achieving a more complicated task implies designing a relatively complex control system which normally includes the efforts of different designers. It means that each individual designer may use his own approach to develop parts of the whole system independently. Under such circumstances, a control system without an explicitly distributed architecture may lead to difficulties in the integration of individual subsystems. Furthermore, the design of a centralised architecture is not flexible, because one cannot easily replace parts of the overall control system to fit in a new task; every time we must start from scratch. Thus, from the system-design point of view, evolving control systems as a whole may not be preferred.

In contrast, the distributed architecture has some potential advantages for scaling up for complicated tasks. In a distributed architecture, the perceptual processing is distributed across multiple independent modules, and every module only deals with the sensory information directly related to its particular need. This not only reduces the sensory bottleneck but also allows each control module to employ the most suitable representation and approach with the least restriction. Because the control architecture is structured as a behaviour-based system, the performance of the overall system will degrade gradually, even if some of the devices or control strategies do not function properly. And, with the explicitly distributed architecture, an overall system would be easily integrated from different subsystems which could be designed independently; it can also be easily maintained. Therefore, from the point of view of designing systems for robot control, the explicitly distributed control architecture seems to have better scaling capability to achieve tasks with high complexity.

6.1.2 About Evolutionary Computation

In contrast to discussing the scale-up issue from the robot control point of view, we should also consider the difficulties associated with the applied approach itself. Generally speaking, the evolutionary approach is a kind of search-based approach in which genetic operators, such as reproduction, crossover, and mutation, are used and expected to find a satisfactory solution in a space; and the dimensionality of this space is determined by the length of the chromosome. For instance, in the binary encoding scheme, a chromosome with length n indicates that the evolution techniques are expecting to find the appropriate solution from a space with 2^n candidates. Thus, when the length of the chromosome is reasonably increased to match the increase in task complexity, the solution space will grow exponentially and lead the search to be more and more difficult. This is particularly apparent in work which uses recurrent neural networks as control systems since the characteristic of recurrence leads the length of chromosome to increase quadratically and then enlarges the search space even faster. In fact, in [Beer & Gallagher 92, Yamauchi & Beer 94], the authors have reported their failure in trying to evolve recurrent neural networks from scratch for some control tasks. After defining modular structures and specifying some connections in advance to simplify the problems, they can then evolve the desired behaviours successfully.

Increasing task complexity also introduces difficulty in defining fitness functions to guide the search direction during evolution. In general, an increase of task complexity implies a higher-level goal to achieve, which almost always involves the interaction of multiple subgoals. For a complex task, directly defining a fitness function at the higher-level is relatively straightforward and simple but it makes the task difficult to be achieved. On the contrary, defining fitness functions at lower-level is more difficult while it makes the task more achievable. For example, in the work [Nolfi & Parisi 95], the authors have shown that, in their grasping task, if the fitness function was simply defined as the number of objects grasped and deposited correctly, then the desirable behaviour could not be evolved successfully. This is due to the fact that during the earlier generations none of the individuals can achieve the complete task; this results in equally bad fitness for all the populations (all scored zero) and made all the control systems indistinguishable in performance. On the other hand, in the same example, if lower-level subgoals were introduced to the fitness function, such as rewarding the behaviours of recognising objects and picking objects up, the performance of controllers became more distinguishable and then the target task could be achieved. Manipulating fitness at lower levels can assist the evolutionary system to converge; however defining an appropriate fitness function at a lower level is never easy because it has to deal with the multiple subgoals simultaneously. Further, this kind of difficulty will grow with an increase in task complexity.

6.1.3 Some Ways to Achieve Complex Tasks

In order to evolve control systems for complex tasks, some researchers began to investigate how to mediate the above problems in two directions. One is to develop advanced evolutionary techniques, such as indirect encoding schemes or co-evolution of higher level representations, to enhance the performance of the evolutionary system. The other is to construct a smooth pathway between the higher level goal and the initial state to make the goal easier to achieve; this includes incremental evolution or task decomposition. This section will discuss these approaches in turn.

Indirect Encoding Schemes

In order to enhance the performance of the evolutionary system, some research work endeavours to develop indirect encoding schemes to reduce the search space by shortening the length of the chromosome. In general, the genotype in this kind of encoding scheme specifies some developmental rules (or operations) which are used to construct the corresponding phenotype as the control system (e.g. [Nolfi *et al.* 94, Gruau 95, Sims 94b]). The genetic operators are then applied to the genotype in order to evolve the appropriate developmental rules to construct satisfactory control systems. Figure 6.2 illustrates the concept of using an indirect encoding scheme to evolve controllers. Since the number of rules used to develop specific controllers can be much smaller than the number of units (bits) within a chromosome in a traditional direct encoding approach, the search space will become correspondingly smaller and the genetic search may then become relatively easy. This provides more possibilities to evolve control systems for complex tasks. [Gruau 95] is a successful example, in which he developed a modular encoding scheme to evolve neural networks controllers for a simulated hexapod walking robot. The genotype in his work specifies a sequence of graph-rewrite operations to develop a neural network from a single neuron. A similar approach has been followed by Koza to evolve analog circuits as food-foraging controllers for a simulated lizard [Koza *et al.* 96b]. The other successful example is the work done by Sims [Sims 94b, Sims 94a]. He developed a graph-based grammar to evolve 3D rigid-part creatures. The genotype in his work is structured as a direct graph of nodes and connections, and the instructions and information contained in the nodes are used to grow the creature for walking, jumping, and swimming behaviours in a virtual world.

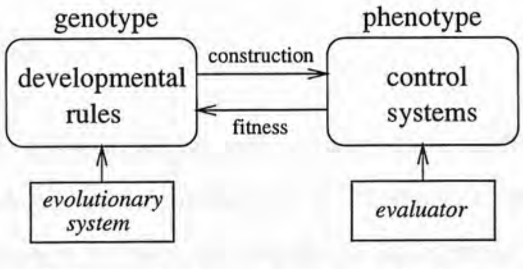

Figure 6.2: The design of the evolutionary system which uses an indirect encoding scheme to evolve control systems.

Although the indirect encoding scheme has shown its potential in evolving control systems to achieve complex tasks for robot control, it requires specific techniques; for example, designing appropriate developmental rules and the operations to construct the corresponding control system. This requirement of specific techniques may make this approach difficult to use in practice.

Higher Level Representation

The other way in reducing the search space and improving the performance of an evolutionary system is the use of higher level representation. The main idea of this kind of approach is to co-evolve some useful building blocks along with the main program body of a genetic representation. The relationship between the co-evolved building blocks and the main program body is much like the one between subroutines and the main program in the conventional computer programming. In this kind of approach, each building block has an unique name and only one copy is maintained in the genetic representation. The building blocks appear in the main program body in terms of their names (symbols) rather than their contents. Figure 6.3 illustrates this concept. In this way, the size of the genetic representation becomes relatively small and the corresponding search space is therefore reduced. The building blocks also allow useful grouped information to be reused easily; this enhances the power of the evolutionary approach. Automatic Defined Functions in [Koza 94] and the Genetic Library Builder in [Angeline & Pollack 93a, Angeline & Pollack 93b] are two typical approaches in the co-evolution of higher level representation. However, research work of this kind has been focusing only on exploring the power of artificial evolution. As for how to apply it to the robot control domain, it may need further investigation.

Incremental Evolution

In contrast to the above two approaches which concentrate on developing techniques to assist the genetic search, an alternative way is to construct an easier fitness landscape for the evolutionary system to help the search in an indirect manner. Incremental evolution is an example of this approach. The main idea of incremental evolution is that, instead of directly challenging the target task with a relatively high complexity,

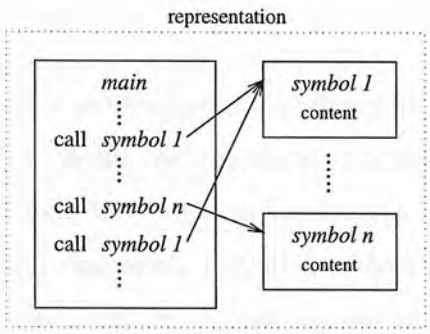

Figure 6.3: The representation of an evolutionary system which co-evolves the main program body together with building blocks.

one can define a sequence of tasks with increasing complexity to lead to the target task gradually, and then evolve control systems for each intermediate task in the predefined sequence. During the process of this kind of evolution, the initial population for a certain task is the one evolved for its predecessor task with slight mutation. Figure 6.4 illustrates the general design of an incremental evolutionary system. Since the complexity of the task is increased gradually, the higher level goal will then become more achievable through a smoother pathway. The concept of incremental evolution has been applied to evolve robot controllers, and the result was promising (e.g., [Harvey *et al.* 94, Harvey *et al.* 96]); however, as the authors themselves pointed out in [Harvey *et al.* 96], how to define a sequence of tasks with increasing complexity is another kind of challenge. It is particularly difficult to define such a task sequence if the final task includes different inconsistent goals (for example, in [Jakobi 94], the author tried to use the evolved obstacle avoidance controllers to constitute an initial population to evolve controllers which fulfill both obstacle avoidance and light seeking behaviours, but he did not succeed). In addition, it is also difficult to follow this kind of approach to evolve distributed control architectures.

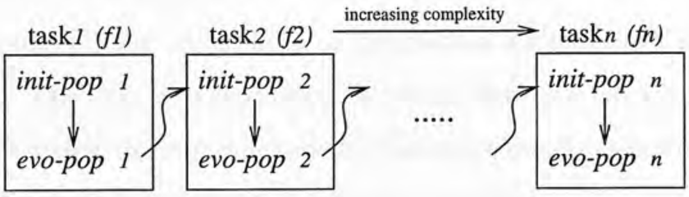

Figure 6.4: The aspect of incremental evolution in which there is a fitness function defined in each task.

Task Decomposition

Another way in constructing a more achievable pathway for higher-level tasks is to take a more engineering view and adopt the divide-and-conquer problem-solving methodology. In this kind of approach, the designers break tasks from complex (higher-level) down to simple (lower-level) recursively and then achieve the tasks in the reverse sequence. How to decompose a task of course depends on the designers' experience, and human designers are generally quite good at that. The tasks are arranged to be achieved in a sequence of increasing complexity and, at each level, the control systems are evolved on top of the ones evolved at lower levels. Hence, fitness functions will become easier to define and the tasks will be easier to achieve (the fitness function of a certain level task can be defined simply as the goal at this level, to reduce the difficulty in embedding the lower-level subgoals into it; and evolving control systems on top of other lower-level controllers can exploit their corresponding control skills to achieve the current goal). In addition, each subtask only needs to deal with the perceptual information directly related to it, which also makes the tasks easier to achieve.

Actually, the concept of this kind of approach is much like behaviour-based control, which has been successfully and widely used in the robot community for building autonomous robots (e.g.,[Brooks 89, Malcolm *et al.* 89, Pfiefer & Scheier 96, Steels 93b]); while the main difference is that the approach here employs evolutionary techniques to *evolve* new behaviours and behaviour coordinators, which have been the main burdens in hand-coding systems, rather than to *handcode* them. By the use of evolutionary techniques, the human designer can concentrate on the system level design and let the evolutionary system take care of the implementation details. In addition, since in this approach the tasks are decomposed in the horizontal way proposed in [Brooks 86], the corresponding control architectures will be explicitly distributed and then fully exploit all the advantages of distributed architectures as analysed in the above section. The task decomposition technique has also been applied to other robot learning domains; examples include Dorigo and Colombetti's work in evolutionary learning [Dorigo & Colombetti 94, Dorigo 95, Colombetti *et al.* 96], and Mataric and Lin's work in reinforcement learning [Mataric 94, Lin 94].

We are particularly interested in investigating ways to reduce the load of robot programmers and in evolving distributed architectures for complex tasks. Because, at the present stage, the task-decomposition technique seems to be the most direct way to achieve these, we will concentrate on investigating how to use this approach, with the previously developed GP system, to evolve control modules and coordinators to achieve complex tasks in the rest of this chapter.

6.2 Task Decomposition and Integration

6.2.1 Two Types of Task Decomposition

There are various ways to decompose a complex task into some simpler subtasks, and, in general, they can be categorised into two types: *flat* and *hierarchical* ones. The former only decomposes the overall task once into two levels; while the latter decomposes the task into multiple levels recursively. In the flat type decomposition, the designer considers the target task as the interaction of a set of lower-level subtasks which are related to the target task and relatively simple to solve. All the subtasks are independent and at the same level. They have their own goals, and each of them will be achieved by a separate controller, which generates its own output according to the sensory input directly relevant to the subtask itself. Those independent outputs from the separate controllers will be combined appropriately by an arbitrator in order to generate the final output to achieve the overall task. A typical flat type decomposition is illustrated in Figure 6.5.

The other type of decomposition divides tasks in two dimensions; it not only decomposes the target task into some simpler subtasks as the flat type decomposition does, but also decomposes subtasks into even simpler sub-subtasks in a recursive manner. Thus, this kind of decomposition will result in a hierarchical structure consisting of different levels of tasks. With respect to the top-down decomposition described above, the tasks in the hierarchical structure are achieved in the bottom-up sequence. For each of the lowest level tasks — the tasks that are not decomposed into any others — there is an independent controller to achieve its own goal; and the outputs from these are merged to achieve relatively higher-level tasks. As in the flat decomposition, a cer-

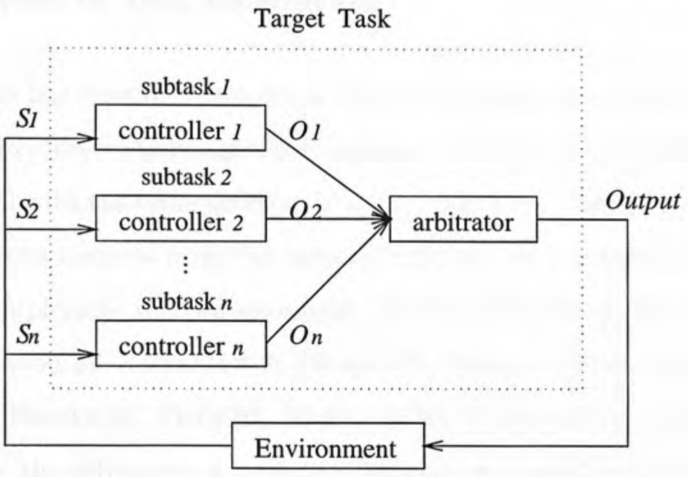

Figure 6.5: A typical example of flat decomposition and its corresponding control architecture.

tain task composed of some sub-tasks needs a kind of arbitrator to combine multiple outputs from the sub-tasks involved. After a control system is assembled from a set of lower-level controllers by an arbitrator, it can be used as a building block to construct other controllers for even higher-level goals. The final goal can then be achieved, step by step, in this way. Figure 6.6 illustrates a typical example of hierarchical task decomposition.

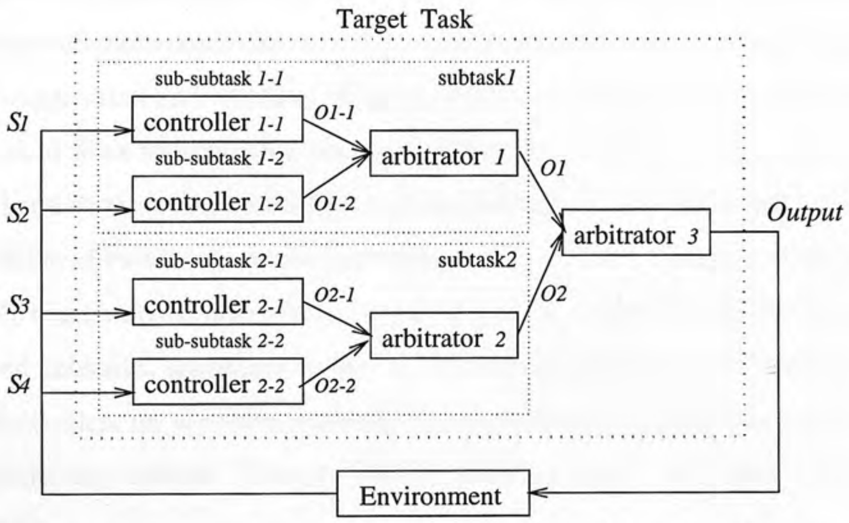

Figure 6.6: A typical example of hierarchical decomposition and its corresponding control architecture.

6.2.2 Two types of Task Integration

The above section has described two types of task decomposition and their corresponding control architectures. Although they organise their structures differently, both of them need to deal with the same problem of *action selection*. This is to specify a way to combine the various outputs from the involved subtasks in a coherent manner. There are two ways to implement such an arbitrator, namely *command fusion* (e.g., [Steels 94, Rosenblatt & Payton 89, Rosenblatt & Thorpe 95, Pfiefer & Scheier 96]) and *command switching* (e.g., [Brooks 86, Firby 94, Blumberg 94, Simmons 94]). In the first way, command fusion, the arbitrator is a certain function of the outputs from the involved subtasks (the *weighted sum* is the most popular function); it makes all of the subtasks able to contribute to the current outputs simultaneously. If this type of integration is taken, the evolutionary approach is then used to tune the weights for the corresponding commands. Figure 6.7(a) presents the general form of this type of arbitration.

The second way, command switching, operates in a winner-take-all fashion; only one of the output commands from the involved subtasks is chosen to take over the control at any given moment, according to certain sensory stimuli. This kind of arbitration is particularly useful in achieving higher-level tasks defined by multiple sequential goals. For example, the task of requiring the robot to look for a charging station to recharge itself among different rooms can be achieved by combining two outputs from the subtasks *safe-exploration* and *reaching-charging-station*, in the way of command switching: once the robot finds the charging station the controller *reaching-charging-station* is in charge; otherwise the other controller, *safe-exploration*, is. The mechanism supporting the arbitration of switching can be regarded as a multiplexer, and can be implemented by a simple reactive controller whose output is used to trigger one of the outputs from the involved subtasks, according to the environmental conditions. As with other independent controllers for separate subtasks, the controller for arbitration can be evolved by an evolutionary system. The general form of this type of arbitration is shown in Figure 6.7(b).

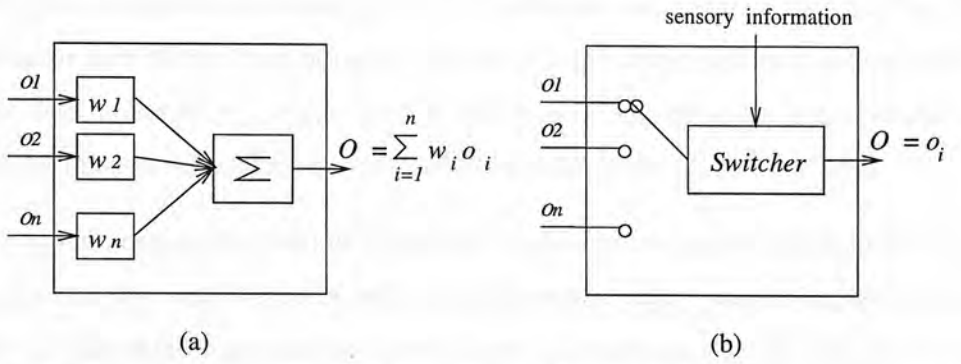

Figure 6.7: Two typical ways to implement an arbitrator. (a) The current output is the weighted sum of it inputs; (b) the current output of an arbitrator is one of its inputs.

6.3 Evolving Hierarchical Task-achieving Controllers

After describing the general ways for decomposing and integrating tasks, we shall concentrate on employing the task decomposition and integration techniques to evolve distributed control systems to achieve complex tasks. In this section, we will explain the features of the control architecture associated with task decomposition used in our work, and then describe the application task which will be undertaken later to prove our approach.

6.3.1 Control Architecture

Since we will decompose tasks in a hierarchical way, the control system is organised in multiple layers. After decomposition, the architecture of our control system looks like a behaviour-based system; it includes a set of *behaviour primitives* and *behaviour arbitrators.* Here, a behaviour primitive is a reactive controller with the representation described in the previous chapter; it involves the lowest level sensory-motor control. Unlike the priority network in the subsumption architecture [Brooks 86], a behaviour arbitrator here is not hardwired in advance; it is also treated as a reactive controller and implemented as a switcher as in Figure 6.7(b). The behaviour arbitrator has the same structure and representation as the primitive; the only difference between them is that the output of a primitive is used to control the motors and the output of an arbitrator is used to activate one of the involved subcontrollers. Thus, in a similar manner to a reactive planner in [Firby 92, Firby 94] or a conditional sequencer in [Gat 92, Gat 94], an arbitrator here allows the binding between environmental conditions and activations of lower level behaviours to take place at run time. This provides adaptiveness not only at the lower level sensory-motor control but also at the behaviour level.

Depending on whether the computing system used supports parallel computation, the control flow in the control system with the above architecture can be implemented as bottom-up (if parallel computation is supported) or top-down (if not). For bottom-up flow, all reactive controllers are active and they are run in parallel. The behaviour primitives send outputs to the arbitrators as their inputs, and each arbitrator selects one of its inputs, according to the environmental stimuli, as its output and then sends this value to higher-level arbitrators. In this way, the output of the highest-level arbitrator will be the output of the overall control system. In contrast, for top-down flow, all the control modules are passive. At each time step, the highest level arbitrator invokes one of its subcontrollers to be in charge of the control, according to certain sensory information. If the invoked subcontroller includes an arbitrator, this arbitrator will be evaluated first and its output can then be used to activate another controller. This process continues until a control primitive at the lowest level is invoked and drives the actuators. Because our system does not support parallel computation, top-down flow is used. Figure 6.8 illustrates the general architecture of our control systems.

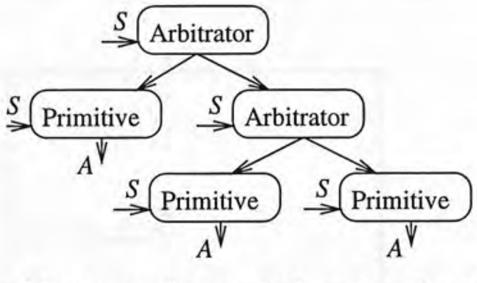

Figure 6.8: The general architecture of a control system in which an arbitrator is implemented as shown in Figure 6.7(b). S and A represent the sensors and actuators related to a certain control work.

6.3.2 The Application Task

In the following experiments, we will follow the approach described above to develop control systems for a moderately difficult box-pushing task. In this task, the robot is required to explore the given arena in order to find a box; once it detects the box, it is then required to push the box toward a goal position indicated by a light source. Figure 6.9 illustrates the environmental setup for this application task.

There have been different versions of box-pushing tasks accomplished by reinforcement learning [Mahadevan & Connell 91] and genetic programming [Koza & Rice 92]. However, those tasks are simpler than ours because their robots were only asked to push a box to any wall rather than to a specific position, which in fact can be done without any deliberate strategy.

The task to be achieved is difficult for the following reasons. First of all, the robot is round, so that it only contacts the box at one point while pushing it, and the box tends to slide and rotate unpredictably when the pushing force exerted by the robot is not directed straight through the centre of the box. Therefore, the robot has to adjust its own position occasionally in order to push the box forward. Furthermore, as there is no particular restriction on the initial relative positions of the robot, the box, and the ambient light, the robot can approach and detect the box at any position and orientation around the box; in such circumstances, the robot needs to deliberately move to a proper position in order to perform an efficient push to satisfy the final goal.

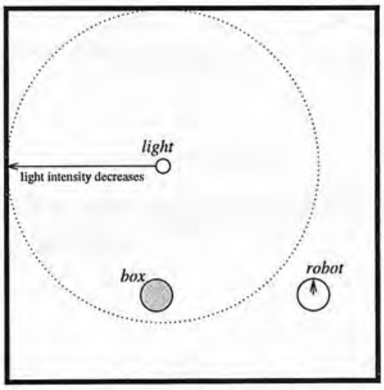

Figure 6.9: Illustration of the application task. The robot starts from an arbitrary position within a closed area; it has to find the box (placed within the area brightened by the light) and push the box toward the light centre.

6.3.3 Task Decomposition

To accomplish this task, we decompose it into two subtasks, *exploration* and *push-box-toward-light*. The former is to control the robot to explore the given arena in order to find the box without bumping into a wall; and the latter, to push the found box to the light centre. As mentioned above, when the robot finds the box, it has to move to a proper position before pushing it, so the task *push-box-toward-light* is decomposed again into two lower-level subtasks, *box-pushing* and *box-side-circling*. The goal of *box-pushing* is to keep the robot pushing a box forward, while the goal of *box-side-circling* is to keep the robot moving along the side of a box in order to provide the opportunity for the robot to move to suitable positions for pushing. Each of the atomic subtasks is controlled by a separate behaviour primitive, and the different subcontrollers for the same task are merged by an arbitrator, which is implemented as a switcher to coordinate the relevant subcontrollers. Figure 6.10 shows the decomposition results and the design of the corresponding architecture for the target task. After decomposition, the genetic programming system developed previously is used to evolve both behaviour primitives and arbitrators.

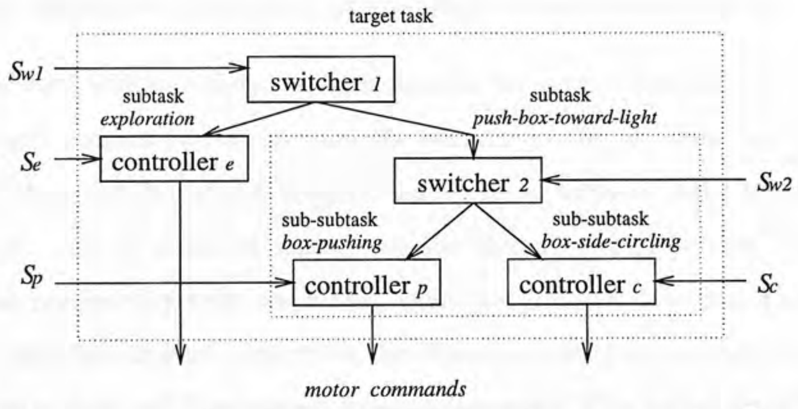

Figure 6.10: The decomposition and integration of the target task. S_i indicates the sensory information relevant to $task_i$.

6.4 Experiment I

6.4.1 Hardware Limitations

In the previous chapter, we have evolved separate controllers for the three lower-level subtasks, *exploration, box-pushing, and box-side-circling.* The following experiments will then concentrate on evolving the arbitrators for the hierarchical control architecture. In order to accomplish the task *push-box-toward-light*, the arbitrator will need infra-red sensors and ambient light sensors to detect the box and the light respectively, so that it can manipulate the relevant two lower level controllers appropriately. The ambient light sensors must be positioned higher than the box, to ensure they can detect the light even in the situation that the box is between the robot and the light.

On the other hand, for the overall task, the other arbitrator will require certain perceptual information, which can be organised as some sensory conditionals for the robot to recognise the box, to determine when to perform *exploration* and when to perform *push-box-toward-light.* For this purpose, we define a kind of virtual sensor *DR*, which can give the normalised reading difference between a pair of upper and lower infra-red sensors. (The sensor pair here means two sensors pointing at the same direction but with different heights: one is higher and the other is lower than the box.)

A straightforward way to satisfy the requirements for both arbitrators is to duplicate Khepera's eight sensors on its top (and the sensors on the top must be higher than the box) so that the duplicated sensors can serve as ambient light sensors for the first arbitrator and as infra-red sensors for the second arbitrator (see Figure 6.11). However, the preliminary tests show that when the robot was within a certain area around the bulb, the infra-red sensors on the Khepera robot was seriously disturbed by the normal bulb light and thus cannot function properly. This will cause difficulties in verifying the simulation results on the real robot. Even so, we can still study how our approach works by using a simulation in which we assume that there are eight extra sensors on the top of the simulated robot as described above. Before the simulation, we constructed the light look-up table using the existing sensors to sample the light emitted from a real 25 Watt bulb, which was masked by wrapping a piece of paper around it and suspended 11 cm above the ground. In the later simulation, the responses for the assumed ambient light sensors will be obtained by accessing this light table.

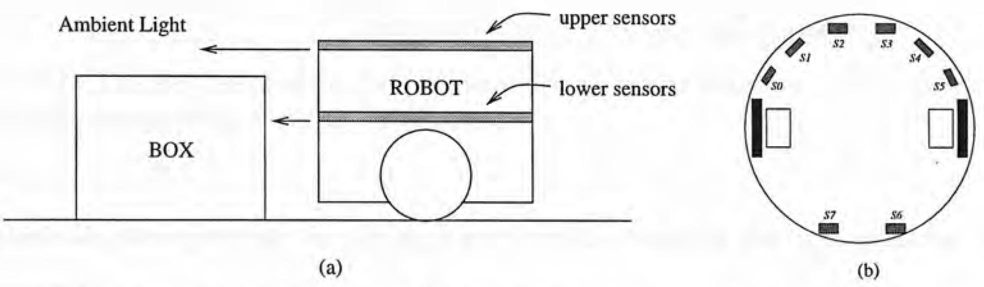

Figure 6.11: (a) An ideal way to satisfy the sensor requirements for both arbitrators is to duplicate a set of eight sensors on the top of the robot; (b) the sensor arrangement - a sensor S_i can function as an infra-red or an ambient light sensor.

6.4.2 Evolving an Arbitrator for the *push-box-toward-light* Task

As mentioned above, an arbitrator is still implemented as a reactive controller; its inputs are from the sensors and its outputs are used to trigger other controllers. For the arbitrator here, two kinds of sensors — infra-red (IR) and ambient light (LDR) — are needed to detect the locations of the box and the light, so both kinds of sensors and the numerical thresholds are defined as terminals to the GP system to construct the structured sensory conditionals for the arbitrator. Since there are only two sub-controllers involved, the arbitrator is designated to have a single output to activate them: if the output is 0, then the controller for subtask *box-pushing* dominates the control; otherwise the controller for subtask *box-side-circling* does. During the experiment, the two subcontrollers will be frozen and only the arbitrator will be evolved.

In the simulation, the box was placed 22 cm away from the light centre, and the robot was expected to push the box as close as possible to the centre of the area brightened by the light. Instead of measuring the distance between the goal position and the final position of the box at the end of a complete trial, we calculated the summation of the distance between the goal position and the box at each time step to reinforce the robot to push the box straight toward the light. Thus the fitness function is defined as:

$$F = \sum_{t=1}^{T} D_{b,l}(t)$$

Terminal Set:	IR, LDR, $\Re\{0..1\}$
Function Set:	PROG, AND, OR, NOT, XOR, >=
number of populations:	4
population size:	50
number of generations:	40
fitness cases:	6 fitness cases are used; they are randomly sampled from 8 pre-defined fitness cases at each generation

Table 6.1: The key features of the problem of evolving arbitrators to coordinate the controllers *box-pushing* and *box-side-circling*.

in which $D_{b,l}(t)$ represents the distance between the box and the light source at each time step t.

As in the experiments of the previous chapters, each individual was evaluated in multiple trials and then the average fitness was taken as the corresponding fitness, in order to enhance the reliability and the robustness of the evolved arbitrators. In this experiment, we pre-defined 8 positions and orientations around the box to be a training set, and 6 of them were randomly sampled at each generation as the starting positions to evaluate the performance of each arbitrator. Each trial lasted 500 time steps and each step was a complete cycle: both the arbitrator and the activated controller were executed once. The robot did not need so many time steps to push the box to the goal position, but it was necessary to evaluate an arbitrator a bit longer to prevent the box being pushed away after it was pushed to the goal position. In a single run, 4 populations of 50 individuals were used and the evolutionary run lasted 40 generations. Table 6.1 summaries the key features of the task of evolving arbitrators here. To confirm the reliability of the experiment, ten independent runs were conducted, and the results showed that in 8 of the 10 runs successful arbitrators were evolved, which were capable of generating proper behaviour sequences to achieve the task. One of the evolved arbitrators looks like:

(PROG
(OR (OR (>= 0.62 LDR*5*)(OR (OR (>= IR*6* LDR*3*) (>= 0.62 LDR*7*)) (AND (>= LDR*6* LDR*7*)(>= LDR*3* LDR*7*)))) (AND (>= LDR*3* LDR*7*)(OR (OR (OR (>= LDR*5* LDR*3*)(>= LDR*3* 0.62)) (>= IR*6* LDR*7*))(AND (NOT (>= IR*6* LDR*4*))(>= LDR*3* LDR*7*))))))

Figure 6.12 illustrates, step by step, the typical behaviour of the simulated robot. As can be seen, the arbitrator first activated the primitive *box-side-circling* to move the robot along the side of a box. Once the robot reached an appropriate position in which the box was between the light and the robot itself, the control was then switched to the other primitive, *box-pushing*, to drive the robot to push the box forward. The *box-side-circling* and the *box-pushing* primitives were activated again in the same order if the pushing path deviated. After the box was pushed to the goal position, the arbitrator continuously activated the primitive *box-side-circling* to make the robot circle the box in order to prevent it pushing the box away from the goal position. From Figure 6.12, we can see that the box was successfully pushed to almost the centre of the bright area.

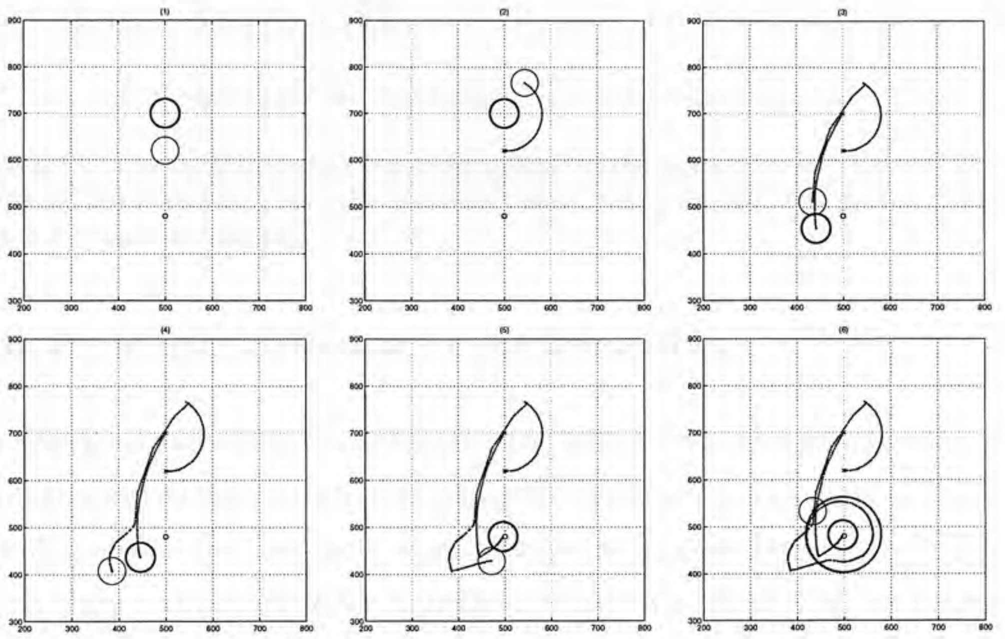

Figure 6.12: The behaviour sequence of the robot during a typical test: (1) The initial positions of the box (dark circle), the light (smallest circle) and the robot; (2) the robot moved along the side of the box; (3) pushing the box forward; (4) circling again to an appropriate position; (5) pushing the box again to the goal position; (6) continuously circling after the box has been pushed to the goal position.

We can also examine whether the task decomposition performed has been exploited in achieving the higher-level goal by observing the output sequence of the arbitrator: if the sequence is separated as periods of consistent activation, then the performed decomposition is confirmed to be helpful; otherwise it may be not. Figure 6.13 demonstrates the output sequence corresponding to the behaviour shown in Figure 6.12. According to this figure, the evolved arbitrator was able to generate periods of quite consistent activation, except the short oscillating period, after the box was pushed to the goal position. The reason for the appearance of such a period is probably that the combination of pushing and circling behaviours may be the best way for the robot to leave the box without touching it (to prevent pushing the box away from the goal position). This figure in fact indicates that the relevant lower-level sub-controllers have been fully exploited.

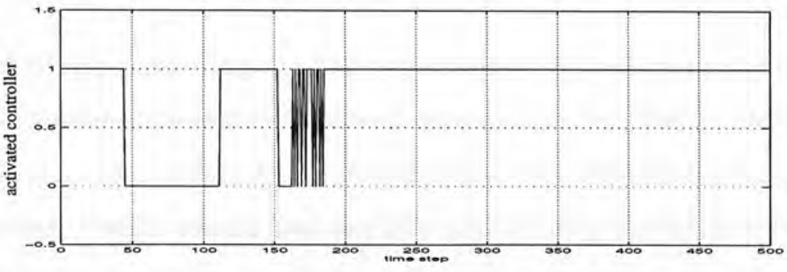

Figure 6.13: The output sequence corresponding to the behaviour in Figure 6.12. In this figure, the y-axis indicates the controller which was activated: 0 is for *box-pushing* and 1 is for *box-side-circling.*

6.4.3 Evolving an Arbitrator for the Overall Task

After evolving an arbitrator to combine two pre-evolved lower-level primitives, we can regard the above integrated control system, including one arbitrator and two primitives, as a building block, and then evolve a new arbitrator to combine this building block and the previously evolved *exploration* controller to achieve the overall task. As described in section 6.4.1, this arbitrator will need perceptual information which can be used to recognise the appearance of the box, in order to generate proper output sequences to coordinate the two control systems involved. Therefore, the virtual sensor *DRs* and the numerical thresholds are defined as terminals for our GP system to evolve the desired arbitrator. Since this arbitrator is to coordinate only two control systems, it is designated to have one output: if the output is 0, the controller for *exploration* is activated; otherwise the controller for *push-box-toward-light* is. Again, the controllers to be combined are frozen and only the arbitrator is evolved.

The goal here is the same as the above one: to push the box as close as possible to the specified position, so the fitness function can be defined as above – to accumulate the distance between the box and the goal position at each time step. However, the

Terminal Set:	DR, $\Re\{0..1\}$
Function Set:	PROG, AND, OR, NOT, XOR, >=
number of populations:	2
population size:	50
number of generations:	40
fitness cases:	6 fitness cases are used; they are randomly sampled from 8 pre-defined fitness cases at each generation

Table 6.2: The key features of the problem of evolving arbitrators to coordinate controllers *exploration* and *push-box-toward-light.*

criterion of simply measuring the fitness function for a fixed period of time as before cannot give an objective evaluation here because of the fact that in different trials the robot could start from different positions and then take different numbers of time steps to find the box – which means that the time periods used to push the box to the light will be different. Therefore, in this experiment, the robot is given an extra period of time (800 time steps at most, which should be enough for the robot to visit most of the given arena if it performs *exploration* as expected) to find the box; and the fitness value is accumulated for a fixed period of time which starts from the moment the robot finds the box (or the end of the time period for looking for the box – in the case where the robot does not find the box). Thus, the fitness function is defined as:

$$f = \sum_{t=k+1}^{k+T} D_{b,l}(t)$$

in which $D_{b,l}(t)$ is the distance between the box and the goal position at time t; k is the time when the robot finds the box (or the end of the time period for looking for the box); and T is the fixed period of time for fitness measurement.

Again, the arbitrators were evolved from multiple trials and the entire evolutionary procedure was repeated ten times (to decrease the influence of random effects) to prove the corresponding correctness. For each run, two populations of 50 individuals were used and the GP system was run for 40 generations, as in most of the previous experiments. The main features of this task are summaried in Table 6.2. For this task, 8 positions were pre-defined and 6 of them were sampled at random at each generation to be the starting positions of different trials. Each individual was evaluated for 500 time steps during a trial and each time step was a complete cycle from the highest-level arbitrator to lowest level controllers. With the specified parameters above, all of the ten runs can evolve successful solutions which are capable of generating a coherent

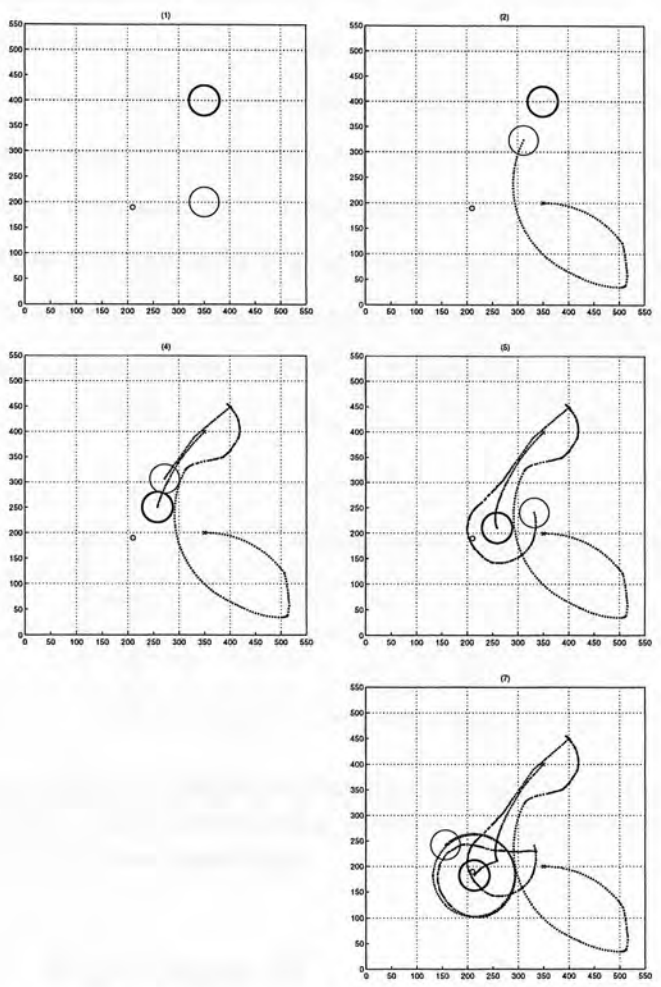

Figure 6.14: The behaviour sequence of the robot: (1) the initial conditions; (2) the robot wandered around the environment before finding the box; (3)~(7) the robot continuously performed the building block controller *push-box-toward-light* to achieve the task.

output sequence to activate the involved sub-controllers appropriately. One of the evolved arbitrators is:

(PROG

(OR (OR (>= DR*0* 0.16)(>= DR*4* 0.16)) (AND (XOR (>= DR*0* DR*4*)(OR (>= DR*3* 0.58) (>= DR*1* DR*4*)))(XOR (>= DR*0* DR*4*) (NOT (>= DR*4* 0.16))))))

The typical behaviour of the robot, when performing the whole control system, is shown in Figure 6.14. From these figures, we can see that the arbitrator firstly kept activating the controller *exploration* to drive the robot to explore the given environment and to avoid the walls. Once the robot found the box, the arbitrator began to activate the other controller, *push-box-toward-light*, according to the sensory stimuli. Since the arbitrator was able to activate this controller continuously after the robot found the box, the overall task can then be achieved successfully. In fact, the performance of the arbitrator above can also be observed from Figure 6.15, which shows the output sequence generated by this arbitrator during the test shown in Figure 6.14. It clearly indicates that this arbitrator can keep activating the controller *exploration* before the box was found, and after finding the box it can activate the other controller *push-box-to-light* consistently to achieve the target task.

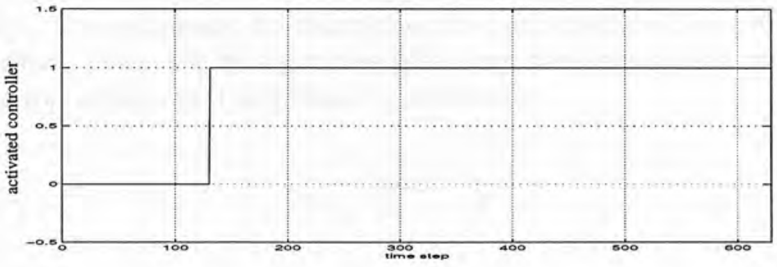

Figure 6.15: The output sequence corresponding to the behaviour in Figure 6.14. In this figure, the y-axis indicates which controller was activated: 0 is for *exploration* and 1 is for *push-box-toward-light.*

6.5 Experiment II

The experiments and their results presented in the last section have shown that complicated tasks with higher-level goals can be achieved by evolving arbitrators to coordinate lower-level behaviour controllers. Yet, as explained in the last section, due to hardware limitations, it is difficult to verify the performance of the evolved arbitrators on a real Khepera. In this section, we design new experiments, based on the available hardware, to examine if an evolved arbitrator can work on a real robot.

For the task *push-box-toward-light*, we construct a new light source — a LED ring — which does not disturb the function of the original infra-red sensors on the Khepera robot, and then deploy the camera module, which includes a light detector and is currently available for the Khepera robot, on top of our robot to detect the LED ring (the original ambient light sensors do not respond to the light emitted by LEDs). Figure 6.16 shows the new experimental equipment.

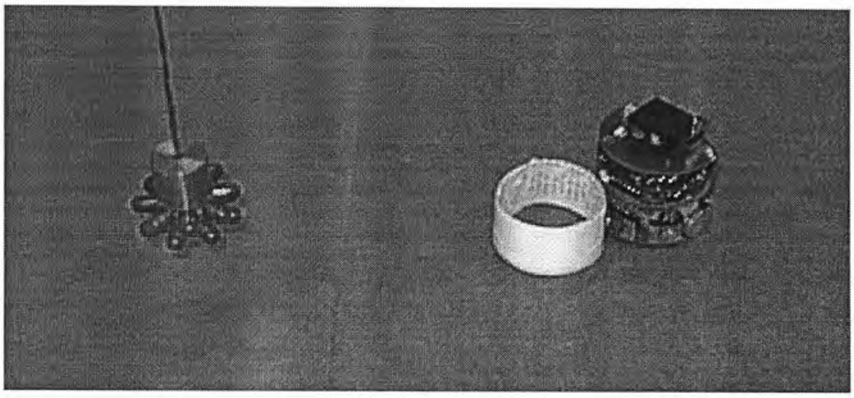

Figure 6.16: The equipment for examining the performance of an evolved arbitrator on a real robot. From left to right: the LED ring, the box and the robot. The black box on the top of the robot includes a light detector.

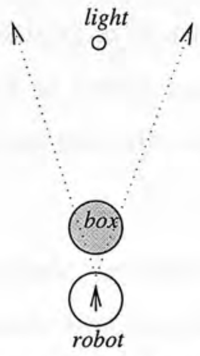

Figure 6.17: The desired situation in which the robot heads toward the box and detects the light at the same time.

To achieve the task of *push-box-toward-light* in the expected way, the robot must be able to move to a position where it can sense the box by its front IR sensors and detect the light at the same time, as illustrated in Figure 6.17. In previous experiments, there were eight ambient light sensors to detect the light in eight different directions, so the robot can create the desired situation by circling around the box to the specific position indicated in Figure 6.17 and then performing the pushing behaviour to face the box. However, there is only one light detector available now, so it is in fact impossible to have the above situation by performing the pushing and circling behaviour controllers evolved previously. Thus, we will have to evolve a new circling behaviour which can circle around the box and create the desired situation. To satisfy these requirements,

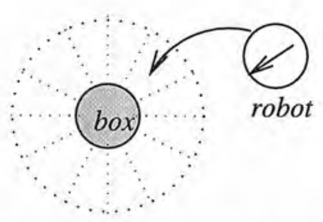

Figure 6.18: An example for explaining how the fitness is defined for evolving the new circling behaviour. The dotted line indicates how the space around the box is divided into small regions for fitness measurement.

we define the fitness function for evolving the new controller as:

$$f = M - P$$

in which M is the number of regions into which the zone around the box has been divided (shown in Figure 6.18, M is 12 in that case); and P is the number of visited regions, where a region is defined as having been visited if the robot faces the box (i.e., the average response of the two front IRs is larger than a specified value) in that region.

To evolve the new circling behaviour, we defined the infra-red sensor IRs and the numerical thresholds as the terminals. In the experiment, 2 populations of 50 individual were used and 40 generations were run. The key features of the problem of evolving a new circling robot are summaried in Table 6.3. We also pre-defined 8 positions around the box as a set of fitness cases and 6 from them were sampled randomly at each generation as the starting positions to evaluate each controller. Like the earlier experiments, ten independent runs were conducted; all of them can evolve controllers which satisfy the requirements perfectly (with zero penalty/fitness) in simulation. Two different kinds of behaviours, shown in Figure 6.19(a) and (b), were observed, though

Terminal Set:	IR, $\Re\{0..1\}$
Function Set:	PROG, AND, OR, NOT, XOR, >=
number of populations:	2
population size:	50
number of generations:	40
fitness cases:	6 fitness cases are used; they are randomly sampled from 8 pre-defined fitness cases at each generation

Table 6.3: The key features of the problem of evolving a new circling robot.

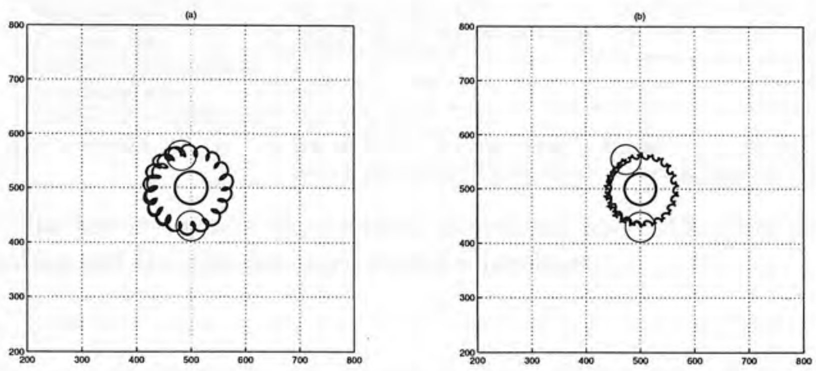

Figure 6.19: Two different circling behaviours generated by the simulated robot.

both of them can achieve the same goal.

Both kinds of circling controllers were transferred to a real robot. The result shows that only the behaviour presented in Figure 6.19(a) can be transferred to reality successfully, and the typical trajectory of the real robot is demonstrated in Figure 6.20. According to our investigation, the reason for the unsuccessful transfer of the behaviour shown in Figure 6.19(b) is that to generate this behaviour the robot has to switch its speed, rapidly and regularly, between positive and negative, and this results in considerable differences between simulated and real robots – each entry in the motor table used in simulation is obtained by performing a certain motor command for a period of time and then taking the average value; the values used in simulation are thus quite different from the actual speed values obtained from real motors if the motors switch their revolving directions rapidly forward and backward.

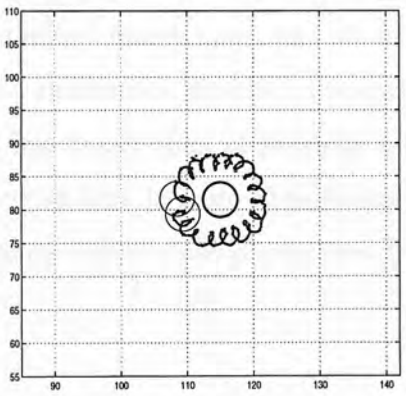

Figure 6.20: The trajectory of the real robot when it performed the evolved controller with the behaviour shown in Figure 6.19(a)

Terminal Set:	IR, LDR, $\Re\{0..1\}$
Function Set:	PROG, AND, OR, NOT, XOR, >=
number of populations:	4
population size:	50
number of generations:	40
fitness cases:	6 fitness cases are used; they are randomly sampled from 8 pre-defined fitness cases at each generation

Table 6.4: The key features of the problem of evolving new arbitrators to coordinate the *box-pushing* and the new *box-side-circling* controllers.

The main purpose for doing the experiments in this section is to verify if we can evolve an arbitrator working on a real robot, so we can simply choose one of the transferable circling controllers (i.e., the ones that can generate the kind of circling behaviour shown in Figure 6.19(a)) as a behaviour primitive for the *box-side-circling* task and use the previously evolved box-pushing controller as the other primitive, then focus on evolving new arbitrators to coordinate these two primitives to achieve the *push-box-toward-light* task for a real robot. The goal here is the same as previous *push-box-toward-light* task: pushing the box as close as possible to the light centre. Thus, the fitness function is defined as:

$$f = \sum_{t=1}^{T} D_{b,l}(t)$$

in which $D_{b,l}(t)$ represents the distance between the box and the centre of the LED ring at each time step t, as in the last section.

To evolve the new arbitrators, we define the infra-red sensor IRs and the light detector LDR as sensor terminals for our GP system, and the control parameters used here are the same as the ones described in section 6.4.2: four populations of 50 individuals; 40 generations; eight pre-defined fitness cases and six of them sampled randomly at each generation. Table 6.4 summaries the main features of this problem. The time to evaluate an individual was longer than in previous experiments (2000 time steps) because the circling behaviour here is more time-consuming. Ten independent runs were done and nine of them evolved successful solutions. One of the evolved arbitrators looks like:

```
(PROG
(OR (OR (>= LDR0 0.11)(>= LDR0 IR3))(>= 0.89 IR3)))
```

in which LDR*0* is the only light detector on the top of the robot and pointing forward.

Figure 6.21(a) is the typical behaviour generated by the simulated robot. The robot firstly circled around the box and then pushed it toward the light. As can be seen, the robot occasionally performed the circling controller to adjust its position for further pushing. After pushing the box to the region around the light, the robot generated the mixed behaviour of pushing and circling – it carried on pushing the box but kept it close to the light. The reason for the generation of this combination is probably that there is only one light detector available now and this combined behaviour may be the best way to achieve the task. After being evolved in simulation, the above arbitrator was transferred to a real Khepera robot to verify its performance at coordinating the lower-level controllers. The typical behaviour of the real robot is presented in Figure 6.21(b). The behaviour of the real robot is similar to the one in simulation.[1] (The trajectories of the robot and the box could not be recorded because of the fact that all three cables, for the LED ring, the robot, and the LED on the top of the box, tangled together when the box was pushed around the LED ring). It shows that the evolved arbitrator can successfully manage other lower-level controllers to achieve the specified task on a real robot.

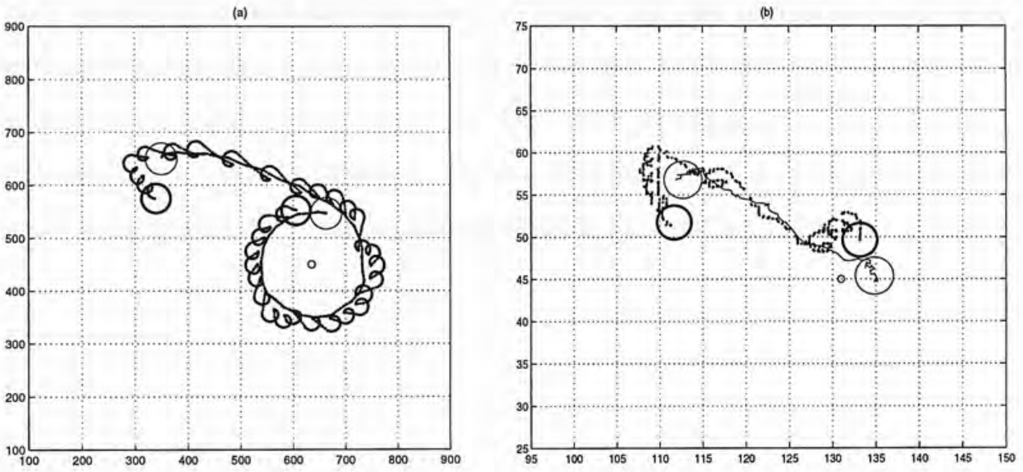

Figure 6.21: The typical behaviours of the simulated (a) and real (b) robots, when they used the evolved arbitrator to coordinate two lower-level controllers for the *push-box-toward-light* task. In both figures, the darker circles are robots and the smallest circles indicate the centre of the LED ring. In the right figure, the solid and dotted lines are the trajectories of the box and the robot, respectively.

[1] In fact, we have mentioned in Chapter 5, due to the environmental uncertainty, such as the nature light from the windows, only qualitatively similar behaviours are expected between the simulated and real robots.

6.6 Summary and Discussion

In this chapter, we have analysed, from the points of view of evolutionary computation and robot control, some difficulties and relevant problems in applying evolutionary algorithms to evolve robot control systems for complex tasks. We have also discussed different ways, mostly involving the development of more advanced evolutionary techniques and the construction of more achievable pathways, to achieve complex tasks by evolutionary approaches. Because we are particularly interested in investigating methods to reduce the load of robot programmers and in evolving behaviour-based control architectures for complex tasks, and because the approach involving task decomposition seems to be the most direct way to achieve these at present, we therefore chose to focus on using this approach to evolve behaviour arbitrators to coordinate lower-level controllers in achieving complex tasks. To verify our approach, we have evolved arbitrators to manage the lower-level controllers involved in a moderately complex box-pushing task as an example. Successful results have been obtained.

By top-down task decomposition and bottom-up controller evolution, our approach not only makes the fitness functions easy to define, but also makes the tasks easy to achieve. In fact, the results have shown that evolving an arbitrator is as straightforward and simple as evolving a primitive: a relatively small population size was used and the desired results were quickly obtained. These points indicate that our approach has the potential to be scaled up to evolve control systems for complex tasks.

Chapter 7

Co-evolution of Robot Brains and Bodies

The above chapters have explored the application of the evolutionary computation approach to develop robot control systems, in monolithic and hierarchical ways. In practice, however, not only the controller but also the robot body itself can affect the behaviour of a robot. For a given command of rotation, for instance, a robot with a relatively small wheel base turns more than one with a large base. Hence, the robot body itself should, ideally, also adapt to the task that we want the evolved robot to do. In this chapter, a hybrid GP/GA approach is presented to evolve complete robot systems — including controllers and robot bodies — to achieve fitness-specified tasks. In order to assess the performance of the developed approach, it is used to evolve simulated agents, with their own controllers and bodies, to do obstacle avoidance in the simulated environment. Experimental results show the promise of this co-evolution approach. In addition, the importance of co-evolving controllers and robot bodies is emphasised, and the relationships between different body parameters will also be explored and discussed in this chapter.

7.1 Introduction

In nature, it is reasonably clear that the structure and abilities of a creature's body are closely related to the kinds of behaviours it can produce. For instance, the hand of a man is formed for grasping, that of a mole for digging. And of course these patterns need to be operated by certain skillful strategies, in order to generate the special behaviours which are beneficial for the creature. In general, any modification of a creature must be profitable in some way to the modified form, and it is often affected by changes to other correlated parts of the organisation – be they changes to physical parts or the strategies operating the parts. That is, the control system and the morphology co-evolve together over the generations.

Typically, however, in evolutionary robotics research, the robot is always assumed to have a fixed body (unchangeable physical structure) and the evolutionary algorithm is applied to evolve control systems on this fixed body to achieve the tasks specified. Although some researchers occasionally mention co-evolving the morphology of the robot as well as the controller, their work is limited to co-evolving the placements or acceptance angles of sensors (e.g., [Harvey *et al.* 94, Reynolds 94b]).

The main difference between this chapter and other work (including our previous chapters) is that, this chapter considers co-evolving the structure of a mobile robot, which has not yet been taken into account in the literature as far as we can discern (with one exception – Karl Sims's work, described in next section). In most of the previous work, the authors change just their controllers if the performance of a robot is not satisfactory; but our approach in improving the performance of a robot is to change not only the control system but also the robot body itself: they are co-evolved [Lee *et al.* 96, Lund *et al.* 97]. Due to the considerations of practicality and computational cost, our way in co-evolving morphologies is to extract the determining structural parameters of a robot body, and then to apply the evolutionary algorithm to decide these values, before constructing the physical body of a robot in reality. In investigating the co-evolution of robot controllers and bodies, experiments include synthesising the complete robot system to achieve a task, verifying that both control system and physical structure have co-adapted to the specified task, and exploring the relationships between different parts of a robot body.

7.2 Related Work in Co-evolving Morphology

As mentioned, there is little work reported in the literature on applying evolutionary algorithms to co-evolve morphologies with control systems. Moreover, the term "morphology" in most of the existing work involves sensors only, but not the physical characteristics, such as the wheel base or the actuator time constant, of a robot. The only exception is the work developed by Karl Sims [Sims 94b, Sims 94a], in which the complete morphology of a 3D creature is co-evolved with the control system. This section gives a brief review of the related work.

7.2.1 Co-evolving Sensors

In the Evolutionary Robotics literature, the most typical study in co-evolving morphology is to co-evolve sensors with the controller. In Sussex's work (i.e., [Cliff *et al.* 92, Cliff *et al.* 93, Harvey *et al.* 94]), the robot is assumed to have a round body and to be equipped with two adjustable photoreceptors. The acceptance angles of the sensors and their positions relative to the axis of the robot's heading are coded as a bit string with fixed length, then the GA acts on this bit string as well as the one coding the controller. Thus both controller and the visual morphology are co-evolved. In their work, the number of sensors is pre-defined, so only the angles and positions of the defined sensors are evolvable.

The other work related to sensor evolution in robotics is [Balakrishnan & Honavar 96]. In this work, both the placement and the range of the sensor are co-evolved with the controller. However, this work is oversimplified. The simulated robot occupies a square in a grid world in which the motion of the robot and the response of sensors are measured in terms of numbers of grid squares. In the simulation, the robot has eight sensors, each with two associated attributes: one indicates the direction (from the eight directions surrounding the robot in the grid world) which the sensor is pointing; and the other indicates the range over which the sensor can detect obstacles (in terms of grid squares). For instance, a robot could have a sensor looking to the *north* and detect obstacles within *three* grid squares. Both kinds of attribute are coded as parts of the bit string with the controller and co-evolved by a traditional GA system. As described above, this work is over-simplified in simulating sensors and actuators; it thus cannot be transferred to a real robot.

Another related work of co-evolving sensor morphology is reported in a series of papers by Reynolds [Reynolds 93, Reynolds 94b, Reynolds 94a]. The author uses Genetic Programming techniques to determine the steering direction of a turtle vehicle from sensor information. Basically, there is a special non-terminal defined in his GP work serving as the sensor; whenever this non-terminal is called, it returns distal information along a specific direction indicated by the argument of the sensor. In his earlier work [Reynolds 93], the argument can be any expression satisfying the defined syntax but he soon found that the argument must be constrained to a numerical value to make his GP system work [Reynolds 94b, Reynolds 94a]. This sensor representation is similar to ours; however there is no particular genetic operation acting on the sensor in all his work. And most importantly, in his work, the sensor information is used to calculate a steer angle by means of symbolic regression (i.e., a technique that discovers a symbolic equation to describe a set of data). Arithmetic operators, such as $+$, $-$, $*$, % are used as symbols in his work which makes his evolved results unreliable and difficult to understand.

7.2.2 Evolving Complete Agents

The work reviewed above only involves co-evolving sensors with control systems, and no other physical structure is included. This section introduces work in evolving a complete agent.

The most famous work on this research topic is the system developed by Karl Sims [Sims 94b, Sims 94a]. It is inspired by Lindenmayer's L-system [Lindenmayer 68] which was used to describe the development of plants. In Sims's work, a creature is actually composed of 3D rigid-parts and acts in a virtual poly-world. Sims added some neural components which perform particular functions, to the nodes which specify the body of an agent. By designing a graph-based grammar to grow the nodes, he can then co-evolve the control system and the physical body. However, his work is more suitable for animation in computer graphics rather than for the design of a physical robot system since the resulting morphologies of 3D rigid parts are probably too difficult to construct in practice.

Apart from this, Dellaert and Beer have tried to evolve a complete agent – including brain and body [Dellaert & Beer 94, Dellaert & Beer 96]. They developed a biological-inspired developmental model which divides a single cell into multiple cells with different types to eventually produce a complete organism. Unlike Sims's work, their cells specify the wiring of the control architecture and update rules. Thus, when the morphology of the agent has settled and cell divisions no longer occur, the control system develops on top of the arrangement of cells, according to the rules specified by the types of cells. Their model is relatively complicated and computationally expensive. Also, the morphology in their work is actually not the physical structure but the arrangement of cell types, so evolution is mainly exploring the relative placement of sensors, actuators, and the control system.

7.3 Simulation

In Chapter 5, we described the reasons why roboticists warn of the danger of using simulations to develop robots. But, as indicated, the simulation may be inevitable in developing methodologies of learning or evolution because of their heavy time consumption. The research work presented in this chapter particularly needs the use of simulation — this work involves the evolution of morphology and it is difficult to expect the physical body of a robot to vary through on-line evolution without human intervention. A more reasonable way is to co-evolve the robot body with the corresponding brain (i.e., the control system) in more realistic simulations, and then construct a robot system according to the evolved result.

The simulator used in this chapter is the one described in Chapter 4; it uses mathematical models to simulate sensors and motors. However, in this work, the structural parameters of the robot are not pre-defined; they remain as variables and are determined by an evolutionary system. The relevant details will be described in a later section.

7.4 Evolving Complete Robot Systems

7.4.1 System Overview

Our genetic system is a hybrid of Genetic Programming and Genetic Algorithms. An individual in this genetic system consists of a controller and a robot body, treated as $< brain, body >$, in which a brain is a tree-like controller as described in previous chapters, and a body is quantitatively specified by a string of real numbers. The GP part of this system evolves the tree-like controller and the GA part evolves the floating point string. The typical structure of an individual is shown in Figure 7.1.

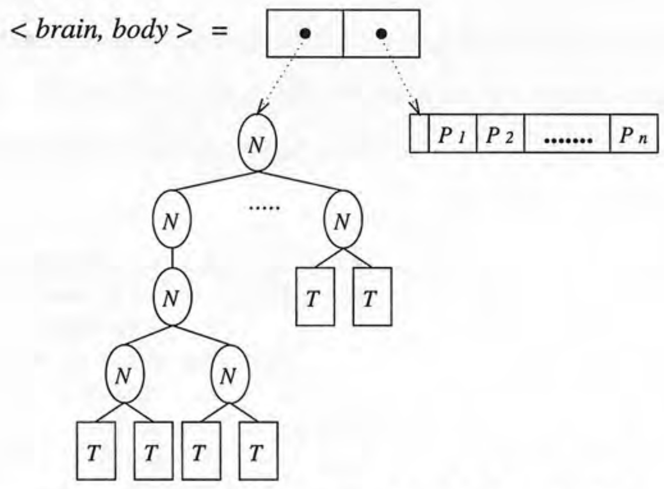

Figure 7.1: The structure of an individual defined in this work: in the tree structure, a node with an N/T is a non-terminal/terminal node; in the string representation, P_i is a real number.

The main flow of our hybrid system is described in Algorithm *main_flow*; it is similar to those of evolutionary robotics systems without co-evolving body plans. After an environment has been given and a goal formulated as a fitness function, an initial population is created at random. But here, each individual has its own brain and body. To evaluate an individual is to execute the brain on the corresponding robot body for a period of time, in the given environment, and to measure the performance. An individual's survival probability is determined by how well the controller performs on the corresponding body, to fit the evaluation criterion (fitness function). Like a conventional evolutionary system, after evaluating each individual, a certain selection method is employed to choose parent individuals and then genetic operations are applied on them to create child individuals. As in the previous experiments, the *tournament selection* method is used here.

Genetic operations, such as *reproduction*, *crossover*, and *mutation*, are applied to the current population to create new individuals. The reproduction operation simply copies the selected parent individuals, without changing the controllers or the bodies, into the next generation. The crossover or mutation operation is allowed to take place on the brain(s) or the body(bodies) at random. Due to the special structure of an individual here, however, the crossover is constrained to occur on both brains or both bodies of the involved parents in order to maintain the correctness of the structures.

Because the representations of the brain part and the body part are different, this work requires separate crossover and mutation operations for the tree expressions and the linear strings. Related techniques of GP and GA are applied independently for the brains and the bodies. The details are described in the following subsections.

```
Algorithm main_flow(num_gens, pop_size)
      create an initial population $P_0$;
      evaluate($P_0$);
      for ($g$ = 1 to num_gens)
            repeat
               choose a genetic operation;
               if (reproduction)
                  select a parent;
                  copy(parent);
               else if (crossover)
                  choose brain or body;
                  if (brain)
                     select parents;
                     GP_crossover(parent1, parent2);
                  else if (body)
                     select parents;
                     GA_crossover(parent1, parent2);
               else if (mutation)
                  choose brain or body;
                  if (brain)
                     GP_mutation(parent);
                  else if (body)
                     GA_mutation(parent);
            until (a new population $P_g$ is created);
            evaluate($P_g$);
      end;
end;
```

7.4.2 Evolving Brains with Sensors – The GP Part

Although sensors are physical parts of a robot body, they are closely associated with the controller: the information received from them is directly connected to the controller. Thus, they can be considered, functionally, as parts of the controller, and directly co-evolved with it. This section explains how this is implemented; all other parts of a robot body are left and discussed in the next section.

The Brain with Sensors

A "brain" here means a reactive controller which controls the associated body. The genetic representation of a reactive controller is the circuit tree used in previous chapters: it includes a dummy root node, different kinds of logic components as internal nodes, and sensors and thresholds as leaf nodes. As mentioned in Chapter 4, the robot is assumed to be round and the sensors are positioned around the round body pointing radially outwards; and each sensor symbol has an associated value to indicate the direction it is pointing at, relative to the robot's heading anti-clockwise. However, in all previous experiments, the position candidates for sensors were predefined, subject to the fixed robot structure; while there is no pre-defined positions for sensors here because we intend to co-evolve them with the controllers. That is, the value associated with a sensor was previously restricted to be one of the pre-defined values; but it is now allowed to be any value, which is determined genetically, between 0 and 1 inclusive in this chapter. Figure 7.2 illustrates an example of the robot's brain.

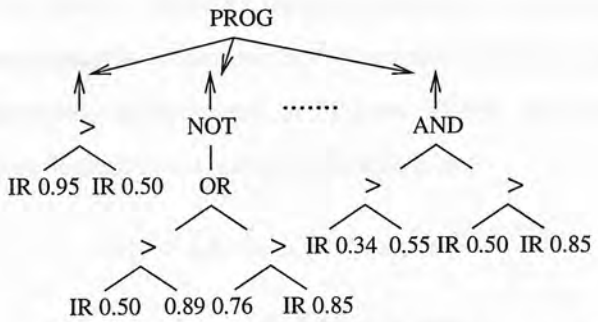

Figure 7.2: Diagram of a typical controller.

Genetic Operations

For the operation of reproduction, the genetic system simply copies individuals, without changing the brains or bodies, into the next generation. The crossover and mutation for the tree controllers are the same as the ones we have described in Chapter 4, except that there is an extra operation, *averaging*, introduced here for averaging the values associated with the sensor terminals – this is to provide a way to shift sensors gradually. Thus, if the crossover points on parent controllers are both sensor terminals, the operation of averaging or swapping could happen randomly.

When the crossover or mutation occurs to the brains of the individuals, their corresponding bodies are not changed. This means, a changed brain with an unchanged robot body is put together to constitute a new individual in the next generation.

7.4.3 Evolving Robot Bodies – The GA Part

As indicated earlier, an agent is made up of a brain and a body; both can affect the behaviour. The performance of an agent is measured by how well the task is achieved by executing the brain on the corresponding body. In this section, we will describe how to specify a robot body and to employ the genetic approach to evolve such a body.

The Robot Body

In order to evolve a robot body we need to analyse what constitutes a body and extract the crucial elements, which affect the behaviour of a robot profoundly, from the structure of a robot. Without losing generality, we use the robot described in section 4.2.5 as an example to explain how to evolve robot bodies. As indicated in the same section, the robot is composed of motors, wheels, and sensors; and its motion can be modelled as formula (4.2) and (4.4), which are

$$\omega_i(t + \Delta h) = \omega_i(t) + \Delta h \times \frac{D_i - \omega_i(t)}{\tau_i}$$

$$v = \frac{r_l \times \omega_l + r_r \times \omega_r}{2}$$

$$\omega = \frac{r_l \times \omega_l - r_r \times \omega_r}{L}$$

where w_i is the actual angular speed of motor i (i = left or right); v and ω are the forward and turning speeds of the robot; and D, τ, r, L are the motor command, motor time constant, wheel radius, and wheel base, respectively. From the first formula, we know that, after the motor command D_i is determined by a robot brain (controller) at each time step, it is used to estimate the actual angular speed w_i for motor i, based on the corresponding motor time constant τ_i; and the larger the time constant, the longer the motor takes to reach the demanded value – the time constant determines the acceleration of the motor and affects the response of the robot. From the second formula, we can see that the forward speed of the robot is determined by the wheel radius r_i, once the angular speed of the motor is available; and the larger the radius, the faster the robot moves – the wheel radius determines the maximum and minimum forward speeds of the robot, for a certain motor command. Also, we can observe, from the third formula, that the rotating (turning) speed of the robot is determined by both the wheel radius r_i and wheel base L. In addition to the above, the size of a robot (the diameter of the body, if we assume the robot is round) is important as well, and it is task-oriented: a small size robot might be suitable for an obstacle avoidance task, while a larger robot might be advantageous for pushing a box. In this work, to evolve a robot body, in fact, means to decide these crucial *structural parameters* of a robot genetically.

In our system, the structural parameters (except, as we have seen, the sensor placement) are arranged as a linear string, in which each position is a real number representing the value of the corresponding parameter. In addition, due to hardware limitations and performance considerations, each structural parameter has its own lower and upper bounds; when we build a robot, the value of each structural parameter must be between its bounds. Thus, a robot body can then be expressed as

$$P_1 P_2 .....P_n$$

where

$$Min(P_i) \leq P_i \leq Max(P_i) \ ; \ 1 \leq i \leq n$$

For this linear string representation, a Genetic Algorithm is employed to determine the value of each structural parameter P_i within its own range.

Genetic Operations

Two-point crossover and one-point mutation operations are used to create new body strings. The crossover operation here, like the standard one, involves two parents and two crossover points. But its function is slightly different from the standard one: it is defined to perform operations of *swapping* or *averaging* randomly, for each pair of P_i between the two chosen crossover points. The intention for the use of averaging is to change the structural parameters gradually (for fine tuning) [Janikow & Michalewicz 91, Davis 91]. The mutation operation randomly picks a P_i for the selected parent and substitutes it with a re-generated random number which satisfies its upper and lower bounds, to generate a new string.

7.5 Experiments in Co-evolving Brains and Bodies

In this experiment, we intend to co-evolve robot controllers and bodies to achieve an obstacle avoidance task (the one described in previous chapters) for the evaluation of the developed approach. The experiments are arranged in three phases: in the first phase, we concentrate on how to co-evolve the controller and body for an individual to achieve the specific task; in the second phase, we investigate the importance of the appropriate brain-body coupling; and in the third phase, we explore the relationships between different body parameters. This section describes the experiments and results of the first phase; the other two are examined and analysed in the following sections.

7.5.1 Interpretation of the Control Output

For simplicity, a tree controller was defined to have only three subtrees as in Chapter 4: the output of the first subtree was interpreted as the direction of revolution of both motors; and the outputs of the second and the third subtrees determined the speeds of the left and the right motor, respectively. The mapping between the output of a controller and the motor command was listed in Table 4.1.

7.5.2 The Specification of a Robot Body

The structural parameters we hope to evolve in this work are *motor time constant*, *wheel base*, *wheel radius*, and *body size*. As mentioned previously, each structural parameter has its own limitations. In this experiment, the value of the motor time constant was restricted to lie between 0.5 and 2.5 second; the lower bound and upper bound of the wheel radius were 1.0 cm and 3.0 cm; the value of body size was limited to be in the range 10 cm to 25 cm; and the wheel base was constrained to be no larger than the body size. These values were chosen to approximate the device constraints of Lego robots which provide the possibility of body reconstruction.

7.5.3 Experiments and Results

The fitness function was the same as in the previous obstacle avoidance task (see Chapter 4 and Chapter 5). At each time step, the controller was executed once and the output was used to drive its body to move; and then the corresponding fitness was calculated. As in the experiments described in previous chapters, the accumulated fitness of an individual during the given time steps represented its performance.

The control parameters for a single run were: two populations of 50 individuals; 25 pre-defined fitness cases and 15 from them were sampled randomly at each generation; 500 time steps for a single evaluation of an individual; and 40 generations for a run. In addition, the testing procedure for an evolved robot was the same as in the earlier experiments. As usual, ten runs were conducted. With the above experimental setup and the constraints of structural parameters indicated in section 7.5.2, eight of the ten runs evolved successful robot systems. Figure 7.3 illustrates an example of how the evolutionary system converged, when co-evolving robot controllers and body plans. The robot system evolved from this example is indicated below and its typical behaviours are shown in Figure 7.4 (tests with different starting positions) and Figure 7.5 (tests with changed environments).

```
(PROG
    (> IR0.97 IR0.52)
    (OR (OR (AND (> IR0.97 IR0.19) (> IR0.97 IR0.21)) (> IR0.97 IR0.60))
```

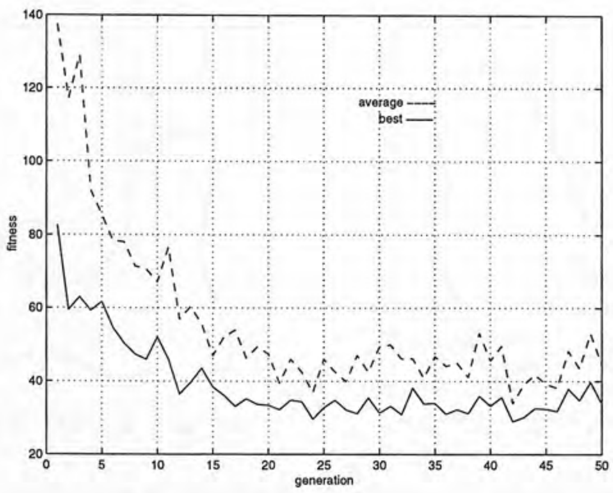

Figure 7.3: An example of the behaviour of the evolutionary system, when it was used to co-evolve robot brains and bodies to achieve an obstacle avoidance task.

```
        (> IR0.21 IR0.23))
    (> IR0.83 IR0.71))
```

time constant: 0.68
wheel base: 10.53
wheel radius: 1.56
body size: 10.53

7.5.4 Analysis of the Evolved Robot

The trajectories presented in Figure 7.4 and Figure 7.5 show that the evolved robot has been able to achieve the behaviour specification described by the fitness function. In addition, the extensive tests also prove the reliability and robustness of the evolved robot. We can furthermore analyse the evolved robot to understand how it achieves this task.

From the evolved controller and the sensor arrangement, we can see that if there is no obstacle appearing around the robot, i.e., no response from any sensor, the outputs of the three subtrees are all 0 and the robot will move straight forward. If the robot senses an obstacle, how it avoids it depends on whether the front sensor IR*0.97* is the first one to detect it. If it is, both motors will revolve backward because the output of the first subtree is 1. In this situation, the second output will also be 1 since the response of the front sensor IR*0.97* will be larger than the other backside sensor IR*0.60* which

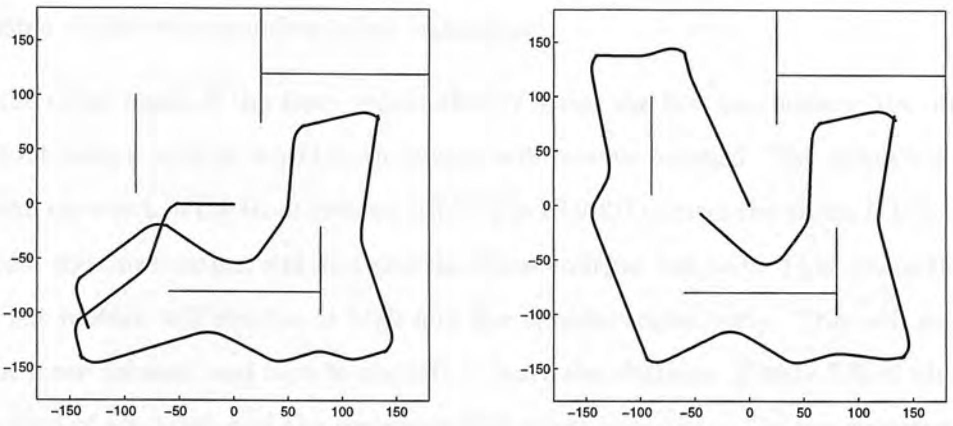

Figure 7.4: Two examples of emerged behaviours, in which the robot started from different situations.

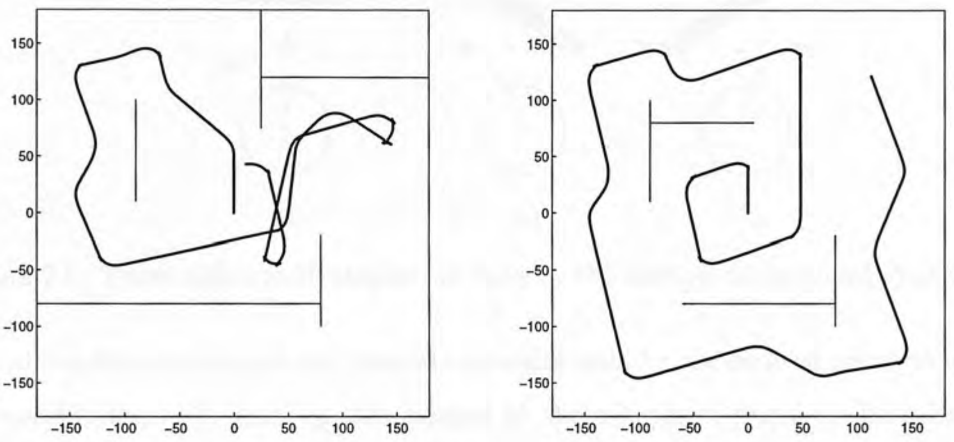

Figure 7.5: Two examples of testing the evolved agent in changed environments.

is close to the rear sensor IR*0.52*. This will cause the left motor to revolve at high speed. And, because the front sensor IR*0.97* is the first one that senses the obstacle, both sensors IR*0.83* and IR*0.71* will not have responses at that time. This will cause a zero output from the third subtree, which means the right motor will revolve at low speed. Thus, the robot will move backward to the right hand side. When the front sensor does not sense the obstacle any more, which causes both the first and second outputs to turn to 0, the right hand side sensor IR*0.83* will be the sensor closest to the obstacle, and this will cause the third output to turn to 1. This means the left and right motors will revolve forward at low and high speeds respectively, and the robot will move forward with a left turn to leave the obstacle. Figure 7.6.(a) shows an example of the corresponding robot behaviour.

On the other hand, if the front sensor IR*0.97* is not the first one sensing the obstacle, the first output will be 0 and both motors will revolve forward. The revolving speeds depend on which of the front sensors (IR*0.83* or IR*0.21*[1]) senses the obstacle first. If it is IR*0.83*, the third output will be 1 and the second output will be 0. This means the right and left motors will revolve at high and low speeds, respectively. This will make the robot move forward and turn to the left to leave the obstacle. Figure 7.6(b) illustrates this kind of situation and the corresponding robot behaviour. On the contrary, if the sensor IR*0.21* senses the obstacle first, the robot will move forward and turn to the right to leave the obstacle. This is shown in Figure 7.6(c).

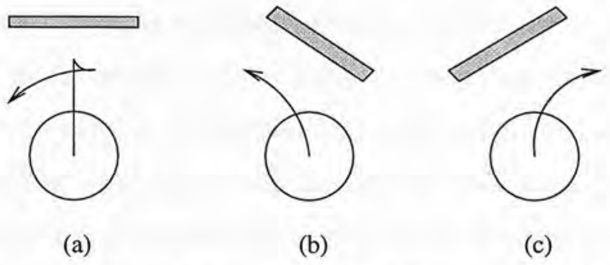

Figure 7.6: Three different strategies are used by the evolved robot to avoid obstacles.

The above descriptions are the general strategies used by the evolved robot to achieve the specific task. To sum up, the sensors at three kinds of positions have been co-evolved with the controller to generate three ways to achieve the task: a front obstacle will cause the robot to move backward and turn to the right first, then move forward and turn to left to avoid obstacles; a left front obstacle will cause the robot to move forward with right turning; and a right front obstacle will cause the robot to move forward with left turning to avoid the obstacle. In general, the last two ways are similar to the strategy of Braitenberg's "fear" vehicle [Braitenberg 86]. However, his vehicle will get stuck in front of a symmetric obstacle (shown in [Floreano & Mondada 94]), which will not happen on our evolved vehicle because of the extra avoiding strategy of moving backward and turning to the right. In addition, we notice that the evolved

[1] Although IR*0.23* is one of the front sensors, it will not be the sensor which senses the obstacle first since it is slightly behind the other sensor IR*0.21* when the robot is moving forward.

robot has a small size and fast motor response (small motor time constant) which are helpful in performing the task of avoidance. The issue of the body parameters will be thoroughly explored in the later sections.

7.6 The Importance of Appropriate Brain-Body Coupling

We have shown that the controller and the robot body can be co-evolved to achieve the fitness-specified task. In order to investigate how the evolved controller relies on the co-evolved body, we tested the evolved controller on different robot bodies which were designed according to various combinations of structural parameters chosen from each of their pre-specified ranges.

For each robot body, we tested it with the evolved controller 100 times. Each time it started from a new position and was allowed to move 10000 time steps, like the testing procedure described in the above section. Table 7.1 shows the results. Each entry in Table 7.1 represents the number of successes (the cases when the robot did not bump any wall during the 10000 time steps) from the 100 test cases for a certain robot body. From Table 7.1, we find that the combination where the evolved controller was executed on the co-evolved robot body (marked with an asterisk) has the highest success rate (actually it is 100%, as reported in the previous section). The inappropriate brain-body couplings cannot achieve the task perfectly. This demonstrates that both the brain and body of a robot have participated in the evolutionary process and have adapted to the specific task.

7.6.1 Further Investigation

Some results in Table 7.1 attract our attention. For the case a (indicated by the mark a at the corresponding number), for instance, the number of successes increases dramatically, compared to case b. We can explore the reasons by observing the behaviours emerging from the two pairs of brains and bodies.

In case b, the robot always bumped the wall because of the inappropriate enlargement of the body size and wheel base. But if the wheel base was enlarged more, the robot became more difficult to rotate, especially with the relatively small wheels which slowed

body-size	*wheel-base*	*wheel radius* 1.56	2.34	3.12	*wheel radius* 1.56	2.34	3.12
10.53	1.05	83	74	75	86	79	64
	5.26	96	83	70	84	74	71
	10.53	100*	99	97	96	86	81
15.79	1.05	64	61	52	71	59	42
	5.26	76	56	74	52	45	27
	10.53	89	95	92	95	64	51
	15.79	90	88	92	94	92	84
21.05	1.05	41	33	26	52	46	34
	5.26	51	24	11[d]	31	13	6
	10.53	30	53	65[c]	91	34	17
	15.79	24[b]	28	27[e]	67	75	68
	21.05	87[a]	19	30	62	59	41

Table 7.1: The results of testing the evolved controller on different robot bodies. The *motor time constants* of the left table and right table are 0.68 and 1.35, respectively. (the number with mark $*$ is the evolved solution)

down the motion of the robot. This caused the robot to get stuck easily — it oscillated forward and backward with a little turning at one particular position, so it was safe in most of the test cases. The typical behaviours of case *a* and case *b* are shown in Figure 7.7.1 and Figure 7.7.2.

The other example we investigated is case *c*, whose performance is much better than cases *d* and *e*. After examining their behaviours carefully, we found that there were actually two types of failure caused by certain kinds of robots which bumped the wall easily. The first was a robot with a large body, small wheel base, and large wheels. A large body inevitably increased the bumping probability; the small wheel base with large wheels made the behaviour of the robot unstable: it was easy to turn at high speed. Consequently, it often drove the robot to bump the wall suddenly. Figure 7.7.3 shows this situation.

The second was a robot with a large body, wide wheel base, and large wheels. As in the first situation, a large body increased the probability of the robot bumping into the wall, while the wide wheel base with large wheels, drove the robot forward faster with a small turning rate. This resulted in the robot bumping the wall in most of the test cases although it could detect the wall efficiently and tried to move away from the wall. Figure 7.7.4 shows this situation.

According to our observations, if the wheel base of the robot became larger, failure of the first kind decreased, but failure of the second kind increased. In case *d*, most of the failures were because of the first situation but in case *e* most of the failures belonged to the second situation. Considering both situations which cause bumping behaviour together, we found case *c* happened to be the best case with a small number of failures for both situations. So its performance is better than case *d* and case *e*.

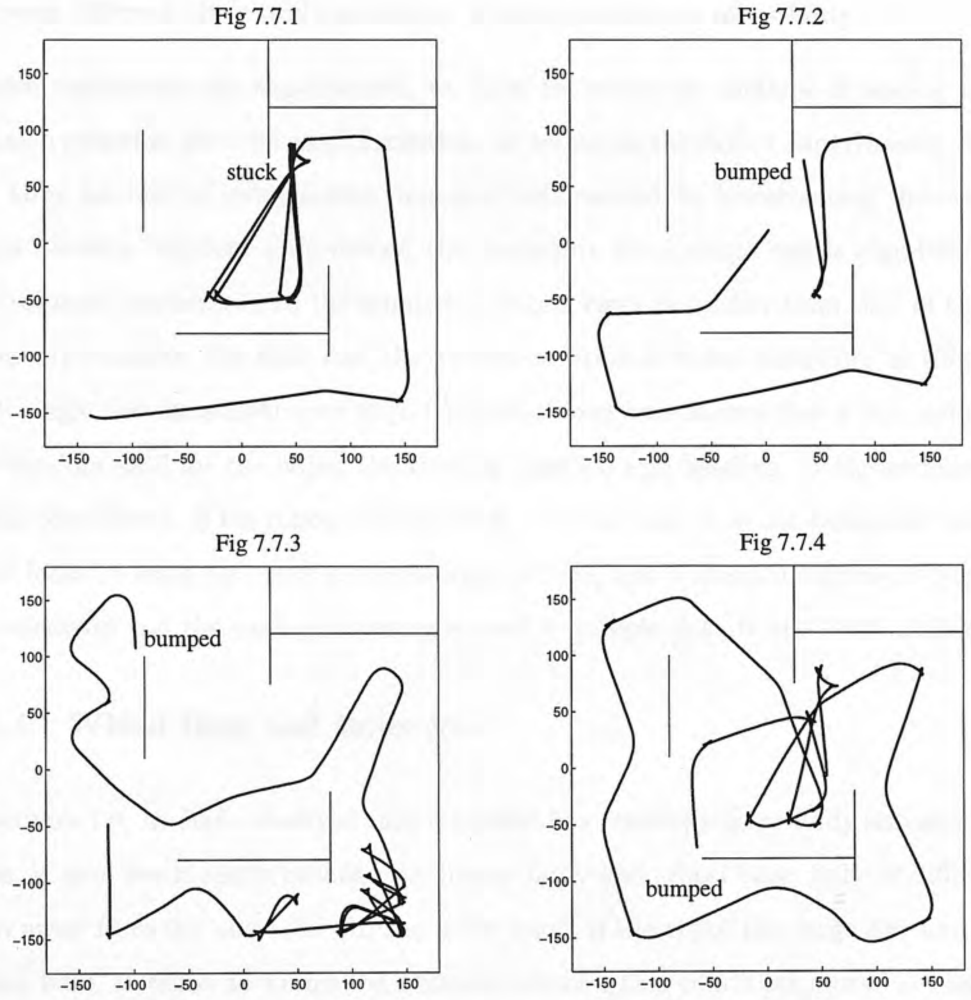

Figure 7.7: Some faults caused by the inappropriate brain-body couplings (see text for explanation)

7.7 Exploring the Relationships between Different Body Parameters

In the above experiments, it has been shown that the controller and the body of a robot can be co-evolved by an artificial evolution system. And it has been proven that not only the brain but also the body was involved in the evolutionary process and adapted to the given task, by testing the evolved brain on different bodies and analysing the faults that happened with inappropriate brain-body couplings. In this section, a series of experiments is arranged to explore whether there exist relationships between different structural parameters which constitute a robot body.

Before conducting the experiments, we have to decide the method of testing and to define a criterion for a successful solution, as we did in the earlier experiments. Due to the large amount of independent runs and tests needed for investigating the relationships between different parameters, the procedure for a single run is simplified: the environment is simpler and the number of fitness cases is smaller than that of the previous experiments. For each run, the evolved solution is tested randomly 20 times and each single test lasts 5000 time steps (A random test here means that a test could vary the random seed for the noise, the starting position and heading, or the arrangement of the obstacles.). If the robot, evolved from a certain run, does not bump any obstacle in at least 19 tests, i.e., with a success rate of 95%, this corresponding run is regarded as successful and the evolved solution is used as sample data in the later analysis.

7.7.1 Wheel Base and Body Size

In section 7.6, we have observed that if a robot has relatively large body size and wheel base, it gets stuck easily because the bigger body and wheel base make it difficult to turn away from the obstacle. On the other hand, if the robot has large size and small wheel base, it tends to bump the obstacle because this condition makes it unstable. Therefore, we will first investigate what kind of combination of body size and wheel base is most beneficial to a robot system for this task.

To do so, we conducted 15 independent runs. For each run, the motor time constant and wheel radius of all robot populations were fixed, and only the wheel base, body size and controller were allowed to be co-evolved. The fixed motor time constant and wheel radius meant that all robots had the same inherent limitations in motor response and speed, so that we could find out what kind of combination of wheel base and body size would be the most suitable solution for the robot to turn away from obstacles and survive in the specific environment.

For all 15 runs, the motor time constant, the wheel radius and the sensor range were fixed to 0.8 sec, 2.0 cm and 30 cm, respectively. The body size was constrained to be between 10 cm and 30 cm, and the wheel base was restricted to be no larger than the body size.

Qualitative Observation

Figure 7.8 shows the distribution of the body sizes and wheel bases of the evolved robots from the successful runs. From Figure 7.8, we can see that the robots with relatively small body sizes are the fittest to do the task of obstacle avoidance. The other observation we can make from Figure 7.8 is that the wheel base tends to be equal to the body size. In other words, there is an almost linear relation between these two parameters. From the point of view of robot behaviour, these results suggest that a small size robot is relatively safe in performing obstacle avoidance, and the body-sized wheel base results in a relatively small turning angle which keeps the robot moving stably. From the point of view of evolution, these results also satisfy the selective pressures formulated in the fitness function: maximising the factor of safety and minimising the factor of turning.

We can also observe how the wheel base and body size varied during the process of co-evolution to understand the tendency of these two variables to converge. Figure 7.9 provides a typical example to demonstrate the variation of the wheel base, body size and the corresponding fitness of the best individuals, during the 50 generations. From Figure 7.9(a) we can see that the two variables tended to converge to approximately the same value which is almost the lower bound of the body size (i.e., 10 cm). By contrasting the converging behaviours of the two variables to the fitness curve (Figure 7.9(a) and (b)), we can find that in the first 10 generations, both wheel base and body size converged rapidly to the relatively appropriate value, and this resulted in a dramatic

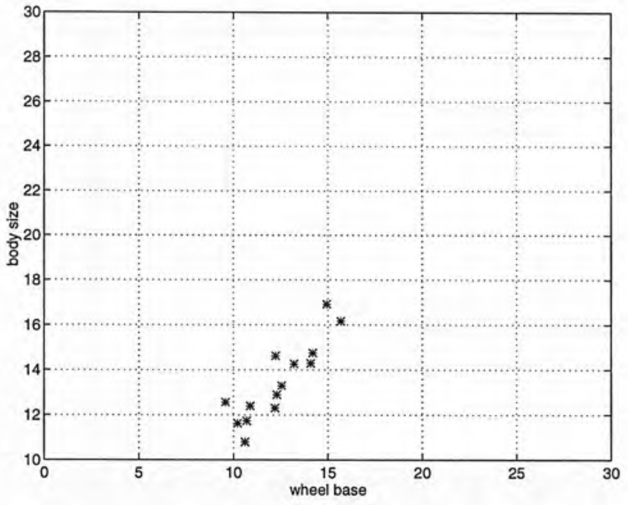

Figure 7.8: The distribution of the body size and wheel base of the evolved solutions from the successful runs.

improvement of the fitness. From generation 10 to generation 20 approximately, there was no apparent variation in the values of wheel base and body size so the fitness of best individual remained more or less the same. After 20 generations, the body size converged slightly toward the value of the wheel base; and this caused the slight but visible change in fitness curve.

Quantitative Analysis

We have observed a *positive* correlation between the wheel base and body size from Figure 7.8. To be more certain, we can also analyse, quantitatively, their correlation by performing the statistical computation to derive the correlation coefficient between them. For two variables x and y, Pearson's correlation coefficient $r_{x,y}$ can be obtained by:

$$r_{x,y} = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2 \sum(y_i - \bar{y})^2}}$$

According to this, we can calculate the correlation coefficient $r_{base,size}$ as 0.8694 (based on 14 sample data). This clearly quantifies the relation between the wheel base and body size. Furthermore, using Fisher's Z transform [Cohen 95], we can show that the chance of obtaining this value from uncorrelated data (i.e., $r_{x,y} = 0$) is less than 0.05.

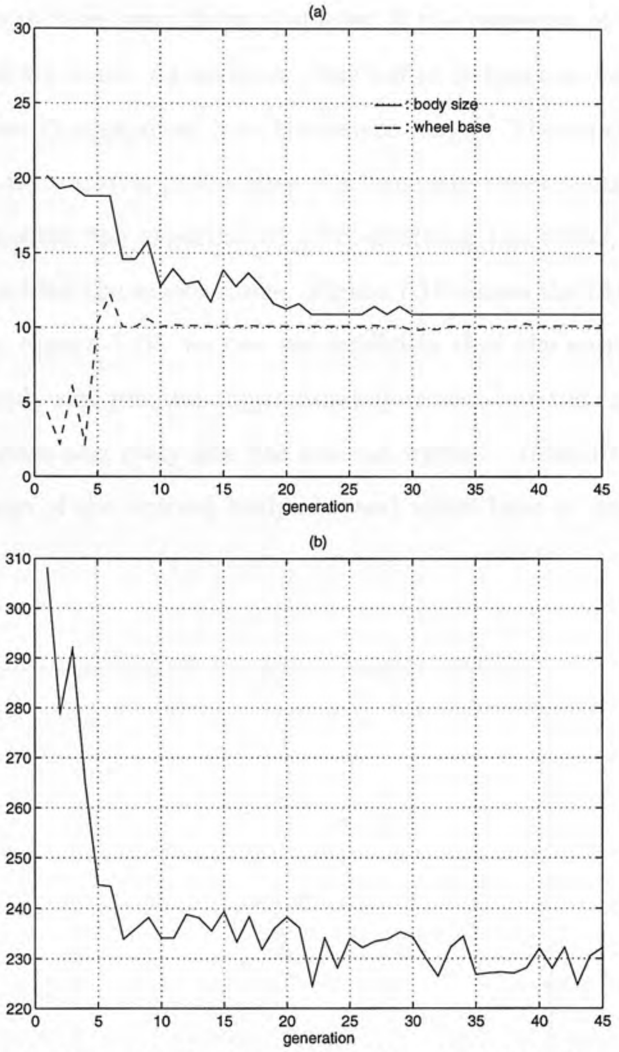

Figure 7.9: (a) The body size and wheel base of the best individual at each generation; (b) the corresponding fitness curve.

The Effect of the Sensor Range

The above sections have provided both qualitative and quantitative descriptions of the relation between the wheel base and body size. However, as we can observe from Figure 7.8, the scales of the wheel base and body size are distributed in the range between 10 and 15 cm. Although it is reasonable that small size robots are safer than bigger ones to achieve the task of obstacle avoidance, it raises a question: what constrains the range of coverage of the evolved wheel base and body size?

In achieving the task of avoiding obstacles, a robot with a relatively large wheel base and body size will have a relatively smaller turning rate and therefore need a relatively longer buffer distance to turn away from obstacles, if the response of the motors and the approaching speed are fixed. As we know, this buffer distance is determined by the characteristics of a sensory equipment (i.e., the sensor range). Therefore, if the range of the sensors increases, the survival probability of a large size robot should also increase. To prove this, we repeated the experiments of co-evolving the wheel base, body size and controller, but doubled the sensor range. Figure 7.10 shows the distribution of the evolved robots. From Figure 7.10, we can see explicitly that the correlation between the wheel base and body size remains approximately linear, but the range of coverage of the evolved wheel base and body size has become wider — from 10 to 25 cm. This shows that the coverage of the evolved body size and wheel base is constrained by the sensor range.

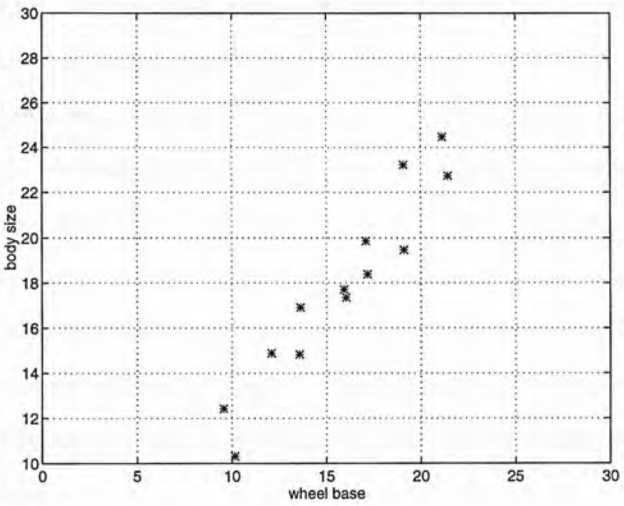

Figure 7.10: The distribution of the evolved body size and wheel base after the sensor range was doubled.

7.7.2 Motor Time Constant and Wheel Radius

From formula (4.4), we know that if the wheel base of a robot is fixed, the forward and turning speeds of this robot are proportional to the wheel radius. Hence, it seems that a robot with a large wheel radius has the advantages of moving forward faster, if no obstacle is sensed, and of producing larger turning angles to avoid a detected obstacle. On the other hand, formula (4.2) tells us that the motor time constant determines the response of a robot. Therefore, if a robot does not have appropriate motor time constants to control its motion well, the larger wheel radius may not be beneficial in achieving the desired task.

The second series of experiments are therefore to explore whether there is a relationship between the motor time constant and wheel radius. To do so, the robot populations are constrained to have a fixed body size and wheel base. Consequently, all the robots have the same probability of bumping into the wall and their turning speeds will be proportional to the wheel radii as described above. In this way, we can then observe what kind of relationship will emerge between the motor time constant and wheel radius, if both of them are co-evolved with the controller.

Qualitative Observations and Quantitative Analysis

In this experiment, 15 runs were conducted in which the motor time constant and the wheel radius were constrained to be between 0.5 and 5.0 sec and between 1.5 and 5.0 cm, respectively. Figure 7.11 shows the distribution of the evolved results. From Figure 7.11, we find that all of the evolved motor time constants are very close to the lower bound and the wheel radius tends to be relatively large. This suggests that to achieve the specified obstacle avoidance task well, the robot needs motors that can respond quickly to sufficiently control the speeds of moving forward and turning while frequently accelerating and decelerating. Besides, the evolutionary system always chooses a relatively large wheel radius for the robot since that gives the robot inherent superiority in action.

We can also observe how the time constant and wheel radius converged during the evolutionary process to understand the dependency between them. Figure 7.12 provides a typical example in which Figure 7.12(a) shows the time constant and wheel radius of the best individual during the evolution, and Figure 7.12(b) is the corresponding fitness curve.

From Figure 7.12(a), we find that the time constant explicitly converged to the lower limit, and the value of the wheel radius seemed to be determined by the time constant: it increased only if the time constant decreased. In Figure 7.12(a), the time constant

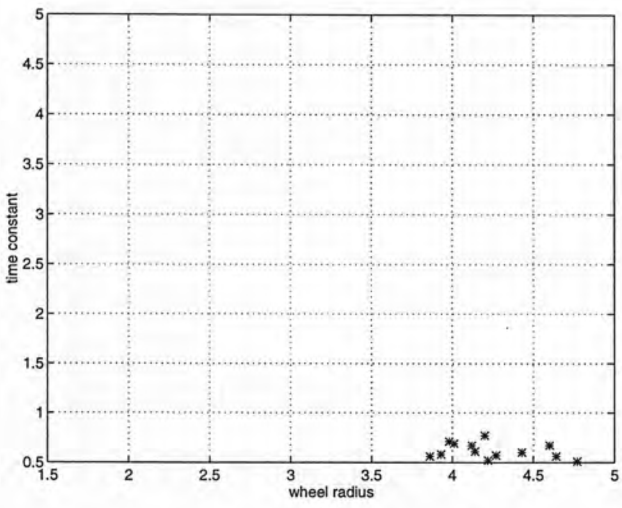

Figure 7.11: The distribution of the co-evolved motor time constant and wheel radius.

clearly moved downwards in the first 10 generations and the wheel radius moved upwards. The time constant remained the same between generation 10 to generation 20 so there was no obvious variation on the wheel radius. After that, the time constant moved to an even lower value between generation 20 to generation 25. A few generations later, the value of the wheel radius rose up again and then reached a stable value since there was no subsequent significant changes in the time constant. Comparing Figure 7.12(b) to Figure 7.12(a), we find that the performance was continuously improved whenever the time constant moved downwards and the wheel radius moved upwards. This clearly illustrates the dependency between the time constant and wheel radius during the evolution.

Again, we can perform statistical calculations to derive the correlation coefficient $r_{time,radius}$ between the motor time constant and wheel radius for the above evolved results as -0.3144 (based on 13 sample data). For concreteness, we can do the Z test to obtain Z score as 0.9942 (less than 1.65) which indicates that the correlation of the sample data is not significantly different from $r_{time,radius} = 0$ (unrelated).

The Effect of Motor Time Constant

The above section has described the dependency between the motor time constant and wheel radius. To concretely study the influence of the motor time constant, we

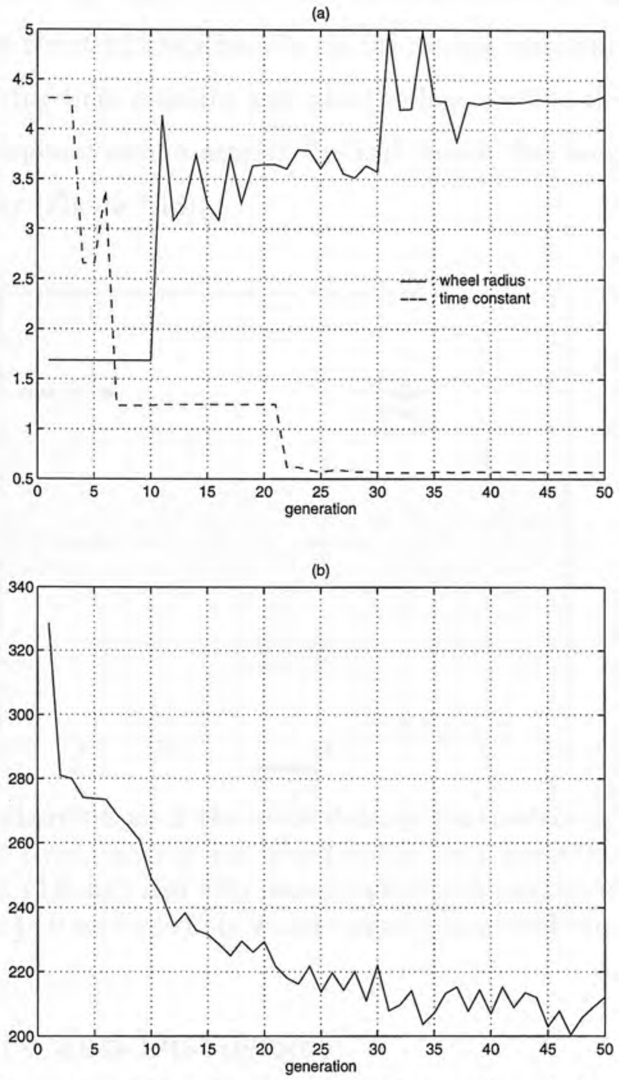

Figure 7.12: (a) The motor time constant and wheel radius of the best individual at each generation; (b) the corresponding fitness curve.

fixed it to higher values, which implies slowing down the motor response, and then co-evolved the wheel radius with the controller. Two sets of runs, in which the motor time constants were fixed to 2.5 and 4.0 seconds respectively, were conducted. Again, the evolved solution of each run was tested as previous experiments and only those results from successful runs are presented in Figure 7.13. The previous experimental results of co-evolving both time constants and wheel radius are also presented for comparison. In addition, the rate of successful runs for three different criteria are illustrated in Figure 7.14. The results show that for those robots with fixed slow motor responses (larger time constants), small wheel radii are better choices, in order to control the motion well to achieve the task. Under such circumstances, the achievement of the specific task for the robot will rely heavily on the control systems. In comparison to the case of co-evolving time constant and wheel radius, we find that the robots with fixed slow motor response cannot achieve the task easily: the rate of successful runs decreases apparently (Figure 7.14).

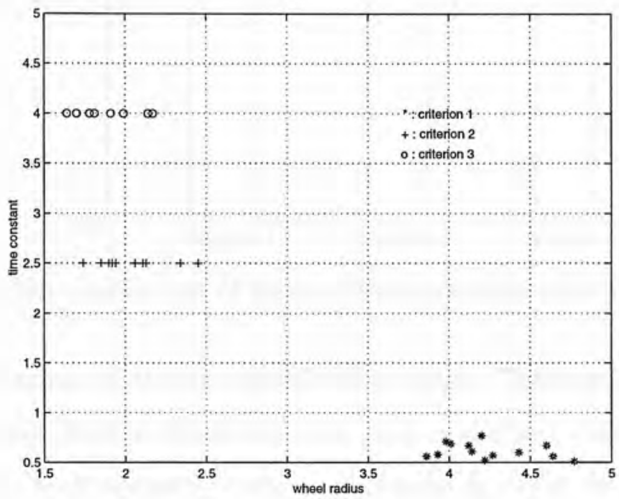

Figure 7.13: The distribution of the evolved body parameters in which criterion 1 co-evolved both the time constant and wheel radius with controllers; criterion 2 used fixed time constant (2.5 sec) and only wheel radius was co-evolved; criterion 3 used fixed time constant (4.0 sec) and only wheel radius was co-evolved.

7.8 Summary and Discussion

In this chapter, a hybrid approach of Genetic Programming and Genetic Algorithms has been presented to co-evolve reactive controllers and their corresponding robot bodies to achieve a fitness-specified task. In our approach, the crucial structural parameters of a physical robot are extracted and arranged as a linear string of real numbers to represent a robot body. The GP part of the system is used to evolve the tree-structure of a controller and the GA part of the system is applied to determine the string of structural parameters. Experimental results have shown the promise of the developed approach.

In addition, we have also analysed the importance of appropriate brain-body coupling in designing a robot system. The evolved controllers can achieve the task perfectly

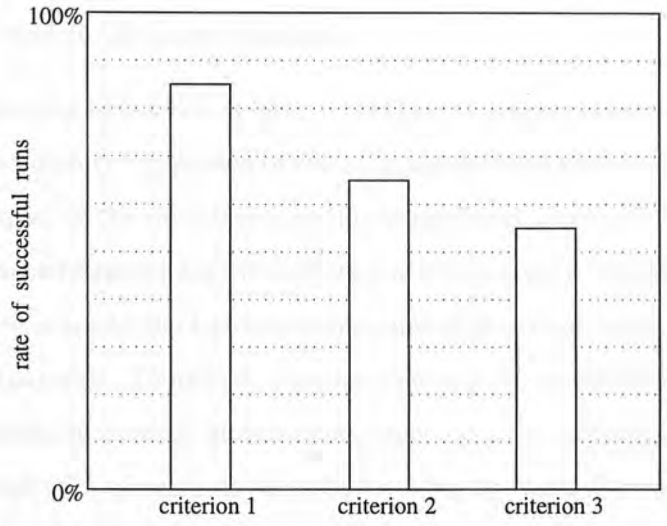

Figure 7.14: The comparison of three different criteria described in Figure 7.13.

only when performing on the co-evolved robot bodies. This means, in the evolutionary process, the robot bodies themselves also play important roles, because controllers and bodies have both adapted to the environment in order to achieve the task. For simple tasks, a human designer may be able to design a robot body according to his prediction of the difficulty of the task, and then design (or evolve) the controller. But for more complicated tasks, co-evolving controllers and morphologies for robot systems may provide a potential alternative.

Finally, we have explored the relationships between different body parameters and investigated what scales of different parameters will be most beneficial to the task of obstacle avoidance. The results have suggested that a small robot is relatively safe; the robot will move stably if its wheel base is about the same as its body size; and a robot with relatively large wheel radius has the inherent superiority of being able to move faster, but it needs motors that can respond fast to control its motion well. We have also analysed the details of the evolutionary runs to better understand the tendency of different parameters to converge.

Some aspects of future work are worth mentioning. Following our work done in simulation, it would be interesting to build a real robot according to the evolved results, and download the evolved controller to it to evaluate the performance. The information obtained from observing the performance difference between the simulated and real worlds can be used to improve the simulator and to make the approach of co-evolving controllers and robot bodies more practical.

Another issue that can be studied is how to combine the co-evolution approach presented in this chapter with the approach of evolving hierarchical task-achieving controllers described in Chapter 6. As we can see, in the hierarchical approach, all the behaviour primitives and the arbitrators are evolved separately, and only the lowest-level primitives are directly relevant to the hardware structure of the robot (only they are directly connected to actuators). Therefore, one possible way to co-evolve a coherent robot body for a hierarchical control architecture, may be to co-evolve all the behaviour primitives involved with a common robot body, after the task decomposition (in such a case, an individual will include a few controller trees and a string of body parameters). After that, one may then evolve arbitrators, as described in Chapter 6, to coordinate different lower-level sub-controllers on the fixed but co-evolved robot body plan to achieve higher-level goals.

Chapter 8

Summary and Conclusions

8.1 Summary and Conclusions

In the first chapter, we have briefly described the main reason why the behaviour-based approach has become a serious alternative to the traditional approach in designing robots. As pointed out by its inventor, however, there are still challenges at different levels, if this approach is to be extended to design complete autonomous robots for more complicated tasks. These mainly include how to code a robust and reliable behaviour controller for a single robot (the micro level); how to deal with the action selection problem to negotiate inconsistencies in actions between different behaviour controllers (the macro level); and how to coordinate multiple robots to achieve the desired group behaviours (the multitude level). None of these are trivial, nor are they easy to achieve. In this thesis, we have examined the use of an evolution-based approach to design robots – especially focusing on applying evolutionary algorithms to synthesise control systems for a single robot (i.e., the first two challenges). The explorations range from evolving simple behaviours for real robots, to complex behaviours (also for real robots), and finally to complete robot systems – including controllers and body plans.

8.1.1 The Construction of an Evolutionary Robotics Framework

In order to evolve controllers for real robots, we have developed an evolutionary robotics framework. It includes two components: a task independent Genetic Programming sub-system which operates as a mechanism for evolution, and a task dependent evaluation sub-system in which the desired robot behaviour is quantified as a fitness function and each controller is executed for a period of time to determine its corresponding performance (fitness). In general, there are two major concerns in such a system: the time consumed to obtain a solution, and the performance of the evolved solution. In other words, researchers in this field have been trying to evolve high performance controllers and to minimise the time consumed. We will draw conclusions along these lines.

The Time Issue

Our analysis has shown that experiments in evolving robots are extremely time-consuming for the following reasons:

1. The evolutionary system must be run for a certain number of generations;
2. Each population member must be evaluated;
3. Each controller is evaluated for multiple trials to prevent overfitting results;
4. Each single trial lasts a certain number of time steps;
5. At each time step, the controller is executed once;
6. At each time step, the actuators are driven to move.

Apparently, the first, the second and the fifth items rely on the evolutionary system used (system-dependent) and they are particularly related to the genetic representation of the controller – a well-designed representation will make it easy for the system to converge to a good solution, with a relatively small population size; the number of trials (the third item) for evaluation is system-independent and may be reduced by certain ways (described below); the number of time steps (the fourth item) is task-oriented and not related to the system used; and the last item is also system-independent but it can be reduced dramatically by the use of simulation. We will firstly examine our work from the point of view of the system-dependent items and then the system-independent ones.

In this thesis, we have designed a special representation for reactive behaviour controllers, by carefully analysing their characteristics. With regard to the computational cost, our representation has some advantages. Firstly, it is succinct and only involves logic operators that are very simple to execute. This means that our evolutionary system is computationally cheap, compared to others, owing to the fact that each controller must be executed once at each time step as mentioned above. The execution of our logic tree is especially quick in cases where the first argument of an AND is false or the first argument of an OR is true, as there is no need to execute the other argument. In addition, because our controllers are composed of basic logic components, they can be easily compiled to custom hardware such as FPGAs for further improvements in speed. Secondly, our representation has been proven to capture the characteristics of the reactive controllers well; this makes our evolutionary system able to evolve the controllers for the desired behaviours efficiently within a relatively small number of generations, using a relatively small population size. All these points mean that less computation is needed in our system, compared to others. Thirdly, we do not distinguish genotypes from phenotypes in our system, so that there is no extra computation cost in converting a chromosome to its corresponding controller before a controller is executed. This also saves a certain amount of time.

In order to reduce the time consumed in evaluating each controller for multiple trials (which is system-independent) without losing the performance of the evolved results, we have used a random sampling technique in our work to evolve the controller from a large number of trials, while evaluating each controller only a relatively small number of times. Different sets of experiments for evolving different robot behaviours have been done, and the results have shown that such an approach can certainly decrease the evaluation time while retaining the performance of the evolved controllers.

The other system-independent way to reduce the time consumed in running an evolutionary robotics experiment is to use simulation. As indicated in Chapter 4, the actual time for a simulated/real robot to perform a single action determines the difference in evolution time between the on-line and off-line approaches. In this thesis, we have chosen to use the approach of evolving controllers in simulation and then to transfer the evolved results to a real robot for performance verification. Our framework has been used to evolve controllers for real robots to achieve a variety of simple tasks successfully, such as obstacle avoidance, safe-exploration, and box-pushing.

The Performance Issue

Two kinds of performance have been emphasised in this thesis; one is about the robustness and reliability of the evolved controllers; the other is the reliability of our evolutionary system. The former concerns whether the evolved controllers can still work well when subject to different environmental situations and random noise; and the latter concerns whether high performance controllers can still be evolved using different random seeds. In our experiments, we have shown that by introducing suitable random noise on sensors and motors, and evolving controllers from multiple trials, controllers with very high performance (gauged by a thorough testing procedure) can be obtained. In addition, by carefully constructing simulators to capture the characteristics of the real world, the controllers evolved from simulation have been transferred to a real robot without losing performance. Additionally, for each task, at least ten runs have been conducted to prove the reliability of our evolutionary framework.

8.1.2 Achieving Complex Tasks

After verifying that our system can evolve high performance controllers for real robots with relatively low computational effort, we discussed different ways to evolve control systems for complex tasks. They can be categorised into two different strategies: the first has been focussed on developing advanced evolutionary techniques to enhance the capability of a genetic system; and the second has stressed the construction of a more achievable pathway in an incremental manner to evolve controllers. In this thesis, we have proposed a methodology, which can be regarded as the second kind, to scale up our system to evolve controllers for more complicated tasks.

Because we aim to reduce the load of robot programmers and are particularly interested in evolving behaviour-based control systems, our approach involves task decomposition (in which a behaviour-based architecture is adopted and a target task is decomposed to fit that architecture), and the evolution of separate behaviour primitives and arbitrators for coordination. This allows robot control systems for more complex skills to be evolved in an incremental way. As indicated in Chapter 6, by evolving arbitrators to coordinate lower-level controllers for complex tasks, the job of defining fitness functions becomes more straightforward and simple, and the tasks become easier to achieve. Moreover, the resulting control systems can be explicitly distributed, understandable to the system designer, easy to maintain, and perform like a behaviour-based system. We have employed this approach to evolve control systems to achieve a moderately complex task, in which a robot has to explore a given environment to find a box without bumping the walls, and then push the box to a goal position indicated by a light source. Experimental results have shown that this approach can evolve control systems for the target task well.

8.1.3 Evolving Complete Systems

Based on the observation that the physical body of a robot can also affect its behaviour, we have extended our Genetic Programming framework to include a Genetic Algorithm sub-system for the co-evolution of robot controllers and body plans, in the last set of experiments. In our work, a robot body means the structural parameters of a physical robot, such as wheel base and wheel radius; and it is encoded as a linear string of real numbers. Therefore, an individual in the hybrid system consists of a controller and a body; and to evaluate the performance of such an individual is to execute the controller on the corresponding body. The GP part of the hybrid system is used to evolve controllers; and the GA part, the robot body.

In addition to evolving complete robot systems to achieve the specified task, we have conducted a series of experiments to verify that the controllers and bodies have both adapted to the given environment to achieve the task. Furthermore, we have also explored the relationships between different body parameters for the specified task. Experimental results have shown the promise of our co-evolution approach. As pointed out in the same chapter (Chapter 7), for more complicated tasks, co-evolving controllers and morphologies for robot systems may provide a potential alternative to other approaches, since both controllers and morphologies can offer their own opportunities for adaptive advantage.

8.2 Future Research

This thesis has provided some concrete evidence to verify that evolutionary computation approaches can be used to evolve simple behaviour controllers for real robots, and has shown, in detail, the possibility of extending this kind of approach to evolve control systems for more complex tasks and to evolve complete robot systems. Based on the work presented, some directions for further research can be suggested.

The first concerns simulation. As indicated in this thesis, the simulator plays a very important role in conducting research in evolutionary robotics. It is system-independent and can reduce the time consumed for the experiments dramatically. Indeed, a simulator must be easy to build and computationally fast, as pointed out by Mataric and Cliff [Mataric & Cliff 95], in order to evolve controllers for complex tasks. Recently, Jakobi in the University of Sussex has proposed to minimise the simulation (i.e., paying attention to what is relevant) to try to fulfill these requirements [Jakobi 97]. In addition to that, there are still other possible ways to speed up the simulation. One is the development of a massively parallel system on real machines in which each controller can be run on a simulated robot on a separate processor; this can reduce the time for running a single experiment linearly, subject to the population size. The other is to code the simulator by the hardware description language VHDL, and compile the simulator to reconfigurable circuits (i.e., FPGA); this can make the simulation much faster than any other technique (results could be obtained in a matter of seconds).

Another useful direction for research is to conduct more experiments to evolve control systems for more complex tasks. Although we have successfully evolved a hierarchical task-achieving control system for a moderately complex task, it will be worth using this approach to evolve more control systems for different kinds of tasks, and for even more complex tasks, to examine the generality of this approach. Particularly, it will be interesting to extend our system to evolve controllers for sequential tasks which involve internal states in the control mechanisms. For this, it will be necessary to introduce some memory components, such as flip-flops, to our circuit trees to participate the evolution.

For the long term goal, building complete autonomous robots should also include the construction of the physical bodies of robots. As indicated, we have developed a promising approach for the co-evolution of robot controllers and body plans, but the experiments currently involve simulation only. Consequently, more research into the realisation of this approach in the real world is to be undertaken. To achieve this, a precise simulator for the robot to be used will be required, because the physical structure of the robot can now adapt to the environment to fulfill the task.

Bibliography

[Agre & Chapman 87] P. Agre and D. Chapman. Pengi: an implementation of a theory of activity. In *Proceedings of AAAI-87*, pages 268 – 272. Morgan Kaufmann, 1987.

[Alcazar & Sharman 96] A. I. E. Alcazar and K. C. Sharman. Some applications of genetic programming in signal processing. In *Late breaking papers at the Genetic Programming 96 Conference.* 1996.

[Angeline & Pollack 93a] P. J. Angeline and J. B. Pollack. Co-evolving high level representation. In *Proceedings of Artificial Life III*, pages 55 – 72. Addison Wesley, 1993.

[Angeline & Pollack 93b] P. J. Angeline and J. B. Pollack. Evolutionary module acquisition. In D. Fogel, editor, *Proceedings of the Second Annual Conference on Evolutionary Programming.* Morgan Kaufmann, 1993.

[Arkin 89] R. Arkin. Motor schema-based mobile robot navigation. *International Journal of Robotics Research*, 8(4):92 – 112, 1989.

[Back & Schwefel 96] T. Back and H.-P. Schwefel. Evolutionary computation: An overview. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 20 – 29. 1996.

[Back 96] T. Back. *Evolutionary Algorithms in Theory and Practice.* Oxford University Press, 1996.

[Baker 85] J. E. Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of the First International Conference on Genetic Algorithms*, pages 101 – 111. San Mateo: Morgan Kaufmann, 1985.

[Balakrishnan & Honavar 96] K. Balakrishnan and V. Honavar. On sensor evolution in robotics. In *Proceedings of International Conference on Genetic Programming.* MIT Press/Bradford Books, 1996.

[Baluja 93] S. Baluja. Structure and performance of fine-grain parallelism in genetic search. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 155 – 162. San Mateo: Morgan Kaufmann, 1993.

[Beer & Gallagher 92] R. D. Beer and J. C. Gallagher. Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior*, 1(1):91 – 122, 1992.

[Blickle & Thiele 95] T. Blickle and L. Thiele. A Comparison of Selection Schemes used in Genetic Algorithms. TIK-Report-11, Computer Engineering and Communication Networks Lab., Swiss Federal Institite of Technology, 1995.

[Blickle 96] T. Blickle. Evolving compact solutions in genetic programming. In *Proceedings of Parallel Problem Solving from Nature IV*. Springer-Verlag, 1996.

[Blumberg 94] B. Blumberg. Action selection in hamsterdam: Lessons from ethology. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 108 – 117. MIT Press/Bradford Books, 1994.

[Braitenberg 86] Braitenberg. *Vehicles: Experiments in Synthetic Psychology.* MIT Press, 1986.

[Brooks & Mataric 93] R. A. Brooks and M. Mataric. Real robots, real learning problems. In J. H. Connell and S. Mahadevan, editors, *Robot Learning.* Kluwer Academic Publishers, 1993.

[Brooks 86] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14 – 23, 1986.

[Brooks 89] R. A. Brooks. A robot that walks; emergent behaviors from a carefully evolved network. *Neural Computation*, 1(2):365 – 382, 1989.

[Brooks 90] R. A. Brooks. Challenges for complete creature architectures. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 434 – 443. MIT Press/Bradford Books, 1990.

[Brooks 92] R. A. Brooks. Artificial life and real robots. In *Proceedings of the First European Conference on Artificial Life*, pages 3 – 11. MIT Press/Bradford Books, 1992.

[Cantu-Paz 95] E. Cantu-Paz. A summary of research on parallel genetic algorithms. IlliGAL Report No.95007, University of Illinois at Urbana-Champaign, 1995.

[Cliff *et al.* 92] D. Cliff, P. Husbands, and I. Harvey. Evolving visually guided robots. In *From Animals to Animats II: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 374 – 383. MIT Press/Bradford Books, 1992.

[Cliff *et al.* 93] D. Cliff, I. Harvey, and P. Husbands. Explorations in evolutionary robotics. *Adaptive Behavior*, 2(1):73 – 110, 1993.

[Cohen 95] P. R. Cohen. *Empirical Methods for Artificial Intelligence.* MIT Press, 1995.

[Cohoon *et al.* 87] J. P. Cohoon, S. U. Hegde, W. N. Martin, and D. Richards. Punctuated equilibria: a parallel genetic algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 148 – 154. San Mateo: Morgan Kaufmann, 1987.

[Cohoon *et al.* 91] J. P. Cohoon, W. N. Martin, and D. S. Richards. A multi-population genetic algorithm for solving the k-partition problem on hyper-cubes. In *Proceedings of International Conference on Genetic Algorithms*, pages 244 – 248. San Mateo: Morgan Kaufmann, 1991.

[Collins & Jefferson 91a] R. J. Collins and D. R. Jefferson. Ant-farm: Towards simulated evolution. In *Proceedings of Artificial Life II*, pages 579 – 601. Addison-Wesley, 1991.

[Collins & Jefferson 91b] R. J. Collins and D. R. Jefferson. Selection in massively parallel genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 249 – 256. San Mateo: Morgan Kaufmann, 1991.

[Colombetti *et al.* 96] M. Colombetti, M. Dorigo, and G. Borghi. Behavior analysis and training: A methodology for behavior engineering. *IEEE Trans. on Systems, Man, and Cybernetics*, 26(6):365 – 380, 1996.

[Darwin 59] C. Darwin. *The Origin of Species.* John-Murray, 1859.

[Davis 91] L. Davis, editor. *Handbook of Genetic Algorithm.* Van Nostrand Reinhold, 1991.

[Dellaert & Beer 94] F. Dellaert and R. D. Beer. Toward an evolvable model of development for autonomous agent synthesis. In *Proceeding of Artificial Life IV*, pages 246 – 257. MIT Press/Bradford Books, 1994.

[Dellaert & Beer 96] F. Dellaert and R. D. Beer. A developmental model for the evolution of complete autonomous agents. In

From Animals to Animats 4: Proceedings of International Conference on Simulation of Adaptive Behavior, pages 393 – 401. MIT Press/Bradford Books, 1996.

[Dorigo & Colombetti 94] M. Dorigo and M. Colombetti. Robot shaping: Developing autonomous agents through learning. *Artificial Intelligence*, 71(2):321 – 370, 1994.

[Dorigo & Schnepf 93] M. Dorigo and U. Schnepf. Genetic_based machine learning and behavior_based robotics: A new synthesis. *IEEE Trans. on Systems, Man, and Cybernetics*, 23(1):141 – 153, 1993.

[Dorigo 95] M. Dorigo. Alecsys and the autonmouse: Learning to control a real robot by distributed classifier systems. *Machine Learning*, 19(3):209 – 240, 1995.

[Firby 92] R. J. Firby. Building symbolic primitives with continuous control routines. In *Proceedings of the First International Conference on AI Planning Systems.* Morgan Kaufmann, 1992.

[Firby 94] R. J. Firby. Task networks for controlling continuous processes. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 49 – 54. Morgan Kaufmann, 1994.

[Floreano & Mondada 94] D. Floreano and F. Mondada. Automatic creation of an autonomous agent: Genetic evolution of a neural-network driven robot. In *From Animals to Animats: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 421 – 430. MIT Press/Bradford Books, 1994.

[Floreano & Mondada 96a] D. Floreano and F. Mondada. Evolution of homing navigation in a real robot. *IEEE Transactions on Systems, Man and Cybernetics*, 26(3):396 – 407, 1996.

[Floreano & Mondada 96b] D. Floreano and F. Mondada. Evolution of plastic neurocontrollers for situated agents. In *From Animals to Animats IV: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior.* MIT Press/Bradford Books, 1996.

[Fogel 95] D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence.* IEEE Press, 1995.

[Fogel *et al.* 66] L. Fogel, A. Owens, and M. Walsh. *Artificial Intelligence through Simulated Evolution.* John Willey and Sons, 1966.

[Gallagher & Beer 96] J. C. Gallagher and R. D. Beer. Application of evolved locomotion controllers to a hexapos robot. *Robotics and Autonomous Systems*, 19:95 – 103, 1996.

[Gat 92] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of AAAI-92*, pages 809 – 815. 1992.

[Gat 94] E. Gat. Robot navigation by conditional sequencing. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 1293 – 1299. 1994.

[Goldberg 89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[Gomi & Griffith 96] T. Gomi and A. Griffith. Evolutionary robotics – an overview. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 40 – 49. 1996.

[Gordon & Whitley 93] V. Gordon and D. Whitley. Serial and parallel genetic algorithms as function optimizers. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 177 – 183. San Mateo: Morgan Kaufmann, 1993.

[Gorges-Schlenter 92] M. Gorges-Schlenter. Comparison of local mating strategies in massively parallel genetic algorithms. In *Proceedings of Parallel Problem Solving from Nature II*, pages 553 – 561. 1992.

[Gradshteyn & Ryzhik 80] I. S. Gradshteyn and I. M. Ryzhik. *Table of Integrals, Series, and Products (corrected and enlarged edition, prepared by A. Jeffrey).* Academic Press, 1980.

[Grefensette & Baker 89] J. Grefensette and J. E. Baker. How genetic algorithms work: A critical look at implicit parallelism. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 20 – 27. San Mateo: Morgan Kaufmann, 1989.

[Grefenstette & Schultz 94] J. Grefenstette and A. Schultz. An evolutionary approach to learning in robots. In *Proceedings of Machine Learning Conference, Workshop on Robot Learning.* 1994.

[Grefenstette *et al.* 90] J. Grefenstette, C. L. Ramsey, and A. Schultz. Learning sequential decision rules using simulation models and competition. *Machine Learning*, 5(4):355 – 381, 1990.

[Gruau 95] F. Gruau. Automatic definition of modular neural networks. *Adaptive Behavior*, 3(2):151 – 183, 1995.

[Harvey 92] I. Harvey. Species adaption genetic algorithms: the basis for a continuing saga. In F. J. Varela and P. Pourgine, editors, *Proceedings of the First European Conference on Artificial Life*, pages 346 – 354. MIT Press/Bradford Books, 1992.

[Harvey 93] I. Harvey. Evolutionary robotics and saga. In *Proceedings of Artificial Life III*, pages 299 – 326. Addison-Wesley, 1993.

[Harvey *et al.* 92] I. Harvey, P. Husbands, and D. Cliff. Issues in evolutionary robotics. In *From Animals to Animats II: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 364 – 373. MIT Press/Bradford Books, 1992.

[Harvey *et al.* 94] I. Harvey, P. Husband, and D. Cliff. Seeing the light: Artificial evolution, real vision. In *From Animals to Animats: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 393 – 401. MIT Press/Bradford Books, 1994.

[Harvey *et al.* 96] I. Harvey, P. Husband, and D. Cliff. Evolutionary robotics: the sussex approach. *Robotics and Autonomous Systems*, 1996.

[Hemmi *et al.* 94] H. Hemmi, J. Mizoguchi, and K. Shimohara. Development of evolution of hardware behaviors. In *Proceedings of Artificial Life IV*, pages 371 – 376. MIT Press/Bradford Books, 1994.

[Hinterding *et al.* 95] R. Hinterding, H. Gielewski, and T. C. Peachey. The nature of mutation in genetic algorithms. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 65 – 72. San Mateo: Morgan Kaufmann, 1995.

[Holland 75] J. Holland. *Adaptation in Natural and Artificial Systems.* The Michigan University Press, 1975.

[Husbands *et al.* 95] P. Husbands, I. Harvey, and P. Husbands. Circling in the round: State space attractors for evolved sighted robots. *Robotics and Autonomous Systems*, 15(1):83 – 106, 1995.

[Jakobi 94] N. Jakobi. Evolving sensorimotor control architectures in simulation for a real robot. School of Cognitive and Computing Science, University of Sussex, 1994.

[Jakobi 97] N. Jakobi. Half-baked, ad-hoc and noisy: Minimal simulation for evolutionary robotics. In *Proceedings of* *the Fourth European Conference on Artificial Life.* MIT Press/Bradford Books, 1997.

[Jakobi *et al.* 95] N. Jakobi, P. Husbands, and I. Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In *Proceedings of Third European Conference on Artificial Life*, pages 704 – 720. Springer-Verlag, 1995.

[Janikow & Michalewicz 91] C. Z. Janikow and Z. Michalewicz. An experimental comparison of binary and float point representations in genetic algorithms. In *Proceedings of the Fourth International Conference on on Genetic Algorithms*, pages 31 – 36. San Mateo: Morgan Kaufmann, 1991.

[Johnson *et al.* 94] M. P. Johnson, P. Maes, and T.Darrell. Evolving visual routines. In *Proceedings of Artificial Life IV*, pages 198 – 210. MIT Press/Bradford Books, 1994.

[Kinnear 93] K. E. Kinnear. Generality and difficulty in genetic programming: Evolving a sort. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 287 – 294. San Mateo: Morgan Kaufmann, 1993.

[Koza & Andre 95] J. R. Koza and D. Andre. Parallel genetic programming on a network of transputers. Technical Report CS-TR-95-1542, Stanford University, 1995.

[Koza & Rice 92] J. R. Koza and J. P. Rice. Automatic programming of robots using genetic programming. In *Proceedings of AAAI-92*, pages 194 – 201. 1992.

[Koza 90] J. R. Koza. Evolution and co-evolution of computer programs to control independently-acting agents. In *From Animals to Animats: Proceedings of International Conference on Simulation of Adaptive Behavior*, pages 366 – 375. MIT Press/Bradford Books, 1990.

[Koza 91] J. R. Koza. Evolving subsumption using genetic programming. In *Proceedings of the First European Conference on Artificial Life*, pages 110 – 119. MIT Press/Bradford Books, 1991.

[Koza 92] J. R. Koza. *Genetic Programming: on the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[Koza 94] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, 1994.

[Koza *et al.* 96a] J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors. *Genetic Programming 1996: Proceedings of the First Annual Conference.* MIT Press, 1996.

[Koza *et al.* 96b] J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane. Toward evolution of electronic animals using genetic programming. In *Proceedings of Artificial Life V.* MIT Press/Bradford Books, 1996.

[Koza *et al.* 97] J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors. *Genetic Programming 1997: Proceedings of the Second Annual Conference.* MIT Press, 1997.

[Langton 89] C. Langton, editor. *Artificial Life : Proceedings of the Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems.* Addison-Wesley, 1989.

[Langton *et al.* 91] C. Langton, J. Farmer, S. Rasmussem, and C. Taylor, editors. *Artificial Life: Proceedings of Artificial Life II.* Addison-Wesley, 1991.

[Lee *et al.* 96] W.-P. Lee, J. Hallam, and H. H. Lund. A hybrid gp/ga approach for co-evolving controllers and robot bodies to achieve fitness-specified tasks. In *Proceedings of IEEE International Conference on Evolutionary Computation.* 1996.

[Lee *et al.* 97a] W.-P. Lee, J. Hallam, and H. H. Lund. Applying genetic programming to evolve behaviour primitives and arbitrators for mobile robots. In *Proceedings of IEEE International Conference on Evolutionary Computation.* 1997.

[Lee *et al.* 97b] W.-P. Lee, J. Hallam, and H. H. Lund. Learning complex robot behaviours by an evolutionary approach. In A. Birk and J. Demiris, editors, *Learning Robots: A Multi-Perspective Exploration (to appear).* 1997.

[Lin 94] L.-J. Lin. Scaling up reinforcement learning for robot control. In *Proceedings of International Conference on Machine Learning,* pages 182 – 189. Morgan Kaufmann, 1994.

[Lindenmayer 68] A. Lindenmayer. Mathematical models for cellular interactions in development. *J. of Theoretical Biology,* 18:280 – 315, 1968.

[Lund & Hallam 96] H. H. Lund and J. Hallam. Sufficient neurocontrollers can be surprisely simple. Research Paper No.824, Department of Artificial Intelligence, University of Edinburgh, 1996.

[Lund *et al.* 96] H. H. Lund, E. V. Cuenca, and J. Hallam. A simple real-time mobile robot tracking system. Technical Paper

No.41, Department of Artificial Intelligence, University of Edinburgh, 1996.

[Lund *et al.* 97] H. H. Lund, J. Hallam, and W.-P. Lee. Evolving robot morphology. In *Proceedings of IEEE International Conference on Evolutionary Computation.* 1997.

[Maes 92] P. Maes. Behavior-based artificial intelligence. In *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 2 – 10. MIT Press/Bradford Books, 1992.

[Mahadevan & Connell 91] S. Mahadevan and J. Connell. Automatic programming of behavior based robots using reinforcement learning. In *Proceedings of AAAI-91*, pages 768 – 773. 1991.

[Malcolm *et al.* 89] C. Malcolm, T. Simthers, and J. Hallam. An emerging paradigm in robot architecture. In *Proceedings of Intelligent Autonomous Systems*, pages 545 – 564. 1989.

[Mataric & Cliff 95] M. Mataric and D. Cliff. Challenges in evolving controllers for physical robots. Technical Report CS-95-184, Department of Computer Science, Brandeis University, 1995.

[Mataric 90] M. Mataric. A distributed model for mobile robot environment learning and navigation. AI-TR 1228, MIT AI Lab., 1990.

[Mataric 94] M. J. Mataric. Reward functions for accelerated learning. In *Proceedings of International Conference on Machine Learning*, pages 181 – 189. Morgan Kaufmann, 1994.

[Michel 95] O. Michel. An artificial life approach to the synthesis of autonomous. In *Artificial Evolution: European Conference.* Springer, 1995.

[Miglino *et al.* 94] O. Miglino, K. Nafasi, and C. E. Tayler. Selection for wandering behaviour in a small robot. *Artificial Life*, 2(1):101 – 116, 1994.

[Miglino *et al.* 96] O. Miglino, H. H. Lund, and S. Nolfi. Evolving mobile robots in simulated and real environments. *Artificial Life*, 2(4):417 – 434, 1996.

[Miller & Goldberg 96] B. L. Miller and D. E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex System*, 9(3):193 – 212, 1996.

[Mondada & Floreano 95] D. Mondada and D. Floreano. Evolution of neural control structures: Some experiments on mobile robots. *Robotics and Autonomous Systems*, 16:183 – 195, 1995.

[Mondada *et al.* 93] F. Mondada, E. Franzi, and P. Ienne. Mobile robot miniaturation: A tool for investigation in control algorithms. In *Proceedings of the Third International Symposium on Experimental Robotics.* 1993.

[Montana 95] D. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199 – 230, 1995.

[Nguyen & Huang 94] T. Nguyen and T. Huang. Evolvable 3d modeling for model-based object recognition systems. In Kinnear, editor, *Advances in Genetic Programming*, pages 459 – 475. MIT Press, 1994.

[Nolfi & Parisi 95] S. Nolfi and D. Parisi. Evolving non-trivial behaviors on real robots: An autonomous robot that picks up objects. In *Proceedings of the Fourth Congress of the Italian Association for Artificial Intelligence.* Spring-Verlag, 1995.

[Nolfi *et al.* 94] S. Nolfi, D. Floreano, O. Miglino, and F. Mondada. How to evolve autonomous robots: Different approaches in evolutionary robotics. In *Proceedings of Artificial Life IV*, pages 190 – 197. MIT Press/Bradford Books, 1994.

[Payton 86] D. W. Payton. An architecture for reflexive autonomous vehicle control. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 1838 – 1845. 1986.

[Pfiefer & Scheier 96] R. Pfiefer and C. Scheier. Sensory-motor coordination: the metaphor and beyond. *Robotics and Autonomous Systems, special issue in Practice and Future of Autonomous Agents*, 1996.

[Press *et al.* 92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 1992.

[Ram *et al.* 94] A. Ram, R. Arkin, G. Boone, and M. Pearce. Using genetic algorithms to learn reactive control parameters for autonomous robotic navigation. *Adaptive Behavior*, 2(3):277 – 304, 1994.

[Rechenberg 73] I. Rechenberg. *Evolutinsstrategie: Optimierung Technischer Systemenach Prinzipiender Biologischen Evolution.* Frommann-Holzboog Verlag, 1973.

[Reynolds 93] C. W. Reynolds. An evolved, vision-based behavioral model of obstacle avoidance behavior. In *Proceedings of Artificial Life III*, pages 327 – 346. Addison Wesley, 1993.

[Reynolds 94a]	C. W. Reynolds. The difficulty of roving eyes. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 262 - 267. 1994.
[Reynolds 94b]	C. W. Reynolds. Evolution of corrider following behavior in a noisy world. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 402 - 410. MIT Press/Bradford Books, 1994.
[Rosca & Ballard 94]	J. P. Rosca and D. H. Ballard. Hierarchical self-organization in genetic programming. In *Proceedings of International Conference on Machine Learning*, pages 251 - 258. Morgan Kanfmann, 1994.
[Rosenblatt & Payton 89]	J. K. Rosenblatt and D. W. Payton. A fine-grain alternative to the subsumption architecture for mobile robot control. In *Proceedings of IEEE International Joint Conference on Neural Networks*, pages II.317 - 323. 1989.
[Rosenblatt & Thorpe 95]	J. K. Rosenblatt and C. Thorpe. Combining multiple goals in a behavior-based architecture. In *Proceedings of IEEE International Conference on Intelligent Robots and Systems*, pages 136 - 141. 1995.
[Rosenschein & Kaelbling 86]	S. J. Rosenschein and L. P. Kaelbling. The synthesis of digital machines with provable epistemic properties. In *Proceedings of Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 83 - 98. Morgan Kaufmann, 1986.
[Rosenschein & Kaelbling 95]	S. J. Rosenschein and L. P. Kaelbling. A situated view of representation and control. *Artificial Intelligence*, 73:149 - 174, 1995.
[Schultz & Grefenstette 94]	A. C. Schultz and J. J. Grefenstette. Improving tactical plans with genetic algorithms. In *Proceedings of IEEE International Conference on Tools for AI*, pages 328 - 334. 1994.
[Schultz *et al.* 96]	A. C. Schultz, J. J. Grefenstette, and W. Adams. Roboshepherd: Learning a complex behavior. In *Proceedings of RoboLearn-96*, pages 105 - 113. 1996.
[Schwefel 81]	H.-P. Schwefel. *Numerical Optimization of Computer Models.* John Wiley and Sons, 1981.
[Simmons 94]	R. Simmons. Structured control for autonomous robots. *IEEE Trans. on Robotics and Automation*, 10(1):34 - 43, 1994.

[Sims 94a] K. Sims. Evolving 3d morphology and behavior by competition. In *Proceedings of Artificial Life IV*, pages 28 – 39. MIT Press/Bradford Books, 1994.

[Sims 94b] K. Sims. Evolving virtual creatures. In *Computer Graphics, Annual Conference Series (SIGGRAPH'94)*, pages 15 – 22. 1994.

[Smithers 94] T. Smithers. On better robots make it harder. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 54 – 72. MIT Press/Bradford Books, 1994.

[Spagocci 96] S. Spagocci. Evolving neurocontrollers and body plans of a Lego robot. Master thesis, Department of Artificial Intelligence, University of Edinburgh, 1996.

[Spiessens & Manderick 91] P. Spiessens and B. Manderick. A massively parallel genetic algorithm: Implementation and first analysis. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 279 – 285. San Mateo: Morgan Kaufmann, 1991.

[Steels 93a] L. Steels. The artificial life root of artificial intelligence. In L. Steels and R. Brooks, editors, *The Artificial Life Route to Artificial Intelligence.* Lawrence Erlbaum Associates, 1993.

[Steels 93b] L. Steels. Building agents out of autonomous behavior systems. In L. Steels and R. Brooks, editors, *The Artificial Life Route to Artificial Intelligence.* Lawrence Erlbaum Associates, 1993.

[Steels 94] L. Steels. A case study in the behavior-oriented design of autonomous agents. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 445 – 452. MIT Press/Bradford Books, 1994.

[Tanese 89] R. Tanese. Distributed genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 434 – 439. San Mateo: Morgan Kaufmann, 1989.

[Verschure *et al.* 92] P. Verschure, B. Krose, and R. Pfeifer. Distributed adaptive control: the self-organization of structured behaviors. *Robotics and Autonomous Systems*, 9:181 – 196, 1992.

[Whitley 89] D. Whitley. The *genitor* algorithms and selection pressure: Why rank-based allocation of reproduction trial is best. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116 – 121. San Mateo: Morgan Kaufmann, 1989.

[Wilson 85] S. W. Wilson. Knowledge growth in an artificial animal. In *Proceedings of the First International Conference on Genetic Algorithms.* San Mateo: Morgan Kaufmann, 1985.

[Yamauchi & Beer 94] B. Yamauchi and R. D. Beer. Integrating reactive, sequential, and learning behaviors using dynamical neural networks. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 382 – 391. MIT Press/Bradford Books, 1994.

Appendix A

Publications

- A Hybrid GP/GA Approach for Co-evolving Controllers and Robot Bodies to Achieve Fitness-specified Tasks. In *Proceedings of IEEE International Conference on Evolutionary Computation.* Nagoya, Japan, 1996.
- Applying Genetic Programming to Evolve Behaviour Primitives and Arbitrators for Mobile Robots. In *Proceedings of IEEE International Conference on Evolutionary Computation.* Indianapolis, USA, 1997.
- Evolving Robot Morphology. *Proceedings of IEEE International Conference on Evolutionary Computation.* Indianapolis, USA, 1997.
- Learning Complex Robot Behaviours by an Evolutionary Approach. To appear in A. Birk and J. Demiris, editors, *Learning Robots: A Multi-Perspective Exploration.* World Scientific Publishers.

A Hybrid GP/GA Approach for Co-evolving Controllers and Robot Bodies to Achieve Fitness-Specified Tasks

Wei-Po Lee John Hallam Henrik H. Lund
Department of Artificial Intelligence
University of Edinburgh
Edinburgh, U. K.
{weipol,john,henrikl}@aifh.ed.ac.uk

***Abstract*— Evolutionary approaches have been advocated to automate robot design. Some research work has shown the success of evolving controllers for the robots by genetic approaches. As we can observe, however, not only the controller but also the robot body itself can affect the behavior of the robot in a robot system. In this paper, we develop a hybrid GP/GA approach to evolve both controllers and robot bodies to achieve behavior-specified tasks. In order to assess the performance of the developed approach, it is used to evolve a simulated agent, with its own controller and body, to do obstacle avoidance in the simulated environment. Experimental results show the promise of this work. In addition, the importance of co-evolving controllers and robot bodies is analyzed and discussed in this paper.**

I. INTRODUCTION

The behavior-based approach has been proposed as a methodology for building autonomous robots. Although many successful robots have been built based on this approach, increasing robot complexity makes the design difficult. Consequently, the evolutionary approach was advocated to provide some kind of design automation for building behavioral modules [3][4].

The first work proposing to use the genetic approach to synthesize programs for robot control is [6]. Due to being overly simplified, however, it has been criticized as being not complicated enough to control a real robot [3]. Another example of using a GP approach to evolve control programs is given in a series of papers by Reynolds. In the final version [8], the author applied arithmetic operations, such as $+$, $-$, $*$, %, and the conditional operation "iflte" to calculate a single output value from sensor values and interpreted it as the steering direction. The robot is then assumed to move for a fixed forward distance in the steering direction.

In our work, the structure of a control program differs from theirs. Our control programs are defined at the logic level: a control program is much like a boolean network, which maps *conditional structures* (constructed on sensor information) into appropriate motor commands. Instead of using compound robot actions, such as moving forward 1 foot or turning left 30 degrees, we use the outputs of the boolean network to directly drive left and right motors to revolve at different speeds. This results in a vehicle moving forward, backward, turning left, right smoothly and continuously.

The other difference between this work and all the others is that this work considers co-evolving the structure of a mobile robot, which has not yet been taken into account in the literature so far as we can discern. In previous work, the authors change just their controllers if the performance of a robot is not satisfactory. But our approach to improve is to change not only the controller but also the robot body itself because both affect the robot's behavior. Although Sims has evolved creatures with controllers and morphologies [9], his creatures are actually constituted from rigid parts which are not practical in the real world. Our way is to extract the determining parameters in designing a robot body then to apply the evolutionary algorithm to decide these values. The main goal of this work is then to investigate how to evolve the controllers and the robot bodies together to achieve a specified task.

II. THE SIMULATION

Agents can be equipped with different kinds of sensors for different tasks. In our current implementation, the agents are restricted to use only infra-red sensors (IRs) to acquire distance information. We assume that the robot has a physical round body and the IRs are positioned around the body pointing radially outward. The characteristic of an IR sensor is that it can only sense objects within a certain distance and a certain bearing. The visual distance is 30 cm and the bearing is 20 degrees in our simulation.

In this simulator, the motion system of the vehicle is regarded as a process with natural dynamics. It converts motor commands with required speeds (Full/Half, Forward/Reverse) into actual motions, and is modeled by related first-order differential equations. Once the time constant of the process is specified and the speed commands are determined by the control program, the rotational velocity of each motor can be calculated. Then the moving speed, the turning speed, and the new position of a vehicle are calculated by applying appropriate kinematic equations, with the specified wheel radius and wheelbase. The vehicle is driven by two independent motors with separate speed commands. This results in a vehicle able to move forward, backward, turn right and left at any possible speed. In order to make the simulation more realistic and to enhance the robustness of the evolved solutions, 5% random noise ($+5\% \sim -5\%$ uniformly) is injected to the perceptions and the motions.

III. GENETIC IMPLEMENTATION

A. System Overview

Our genetic system is a hybrid of Genetic Programming [6] and Genetic Algorithms [5]. An individual in this genetic system consists of a controller and a robot body, treated as $< brain, body >$, in which a brain is a tree-like program and a body is described by a string of real numbers. The GP part of this system evolves the tree-like program and the GA part evolves the floating point string. The aspect of an individual is shown in Figure 1.

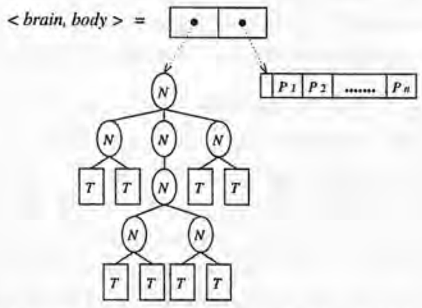

Fig. 1. The aspect of an individual defined in this work: in the tree structure, a node with the N/T is a non-terminal/terminal node; in the string representation, P_i is a real number.

Given an environment and a goal formulated as a fitness function, an initial population is created at random. Each individual has its own brain and body. To evaluate an individual is to execute the brain on the corresponding simulated robot body for a period of time and to measure the performance. The probability of the survival of an individual is then determined by how well the controller performs with the corresponding body to fit the evaluating criteria. After evaluating each individual, a certain selection method is employed to choose parent individuals and genetic operations are applied on them to create children individuals. In this work, the *tournment selection* method, which picks K individuals at random and chooses the fittest as the parent, is used.

Like other genetic work, genetic operations, such as *reproduction*, *crossover*, and *mutation*, are applied to the current population to create new individuals. The reproduction operation simply copies the selected parent individuals, without changing the controllers or the bodies, into the next generation. The crossover or mutation operation is allowed to to take place on the brain(s) or the body(bodies) at random. Due to the special structure of an individual here, however, the crossover is constrained to occur on both brains or both bodies from the involved parents in order to maintain the correctness of the structures.

Because of the different representations of the brain and the robot body, this work requires separate crossover and mutation operations for the tree expressions and the linear strings. Related techniques of GP and GA are applied independently for the brains and the bodies. The details are described in the following sections.

B. Evolving Brains with Sensors

Although sensors are parts of a robot body, they are closely associated with the control programs. Thus they are directly co-evolved with the control programs as described in this section. All other parts of a robot body are left and discussed in the next section.

B.1 Interpretation of the Control Program

A "brain" here means a *reactive* control program which controls the corresponding body. As we know, a reactive controller can be considered as a combinational logic system in which the output is determined completely by the current input states at each time step. It is well accepted that any combinational system can be described by a *boolean network* and a boolean network can be converted into a *boolean tree* [1][2], so that we define our control programs at logic level and use logic components, such as AND, OR, NOT, to map structured sensor conditionals into motor commands. Genetic Programming techniques are used to evolve such control programs.

We define a control program organized by three subtrees, and a non-terminal PROG is defined as a dummy root node to connect the three subtrees. The output of the first subtree is interpreted as the revolving direction of the left and right motors: if the output is 0, the control program will command both motors to revolve forward, otherwise it commands both motors to revolve backward. The evaluated results of the second and the third subtrees determine the speeds of the left and the right motor: 0/1 represents full/half speed. Three basic logic operations, AND, OR, and NOT, are defined as non-terminals to constitute the main frames of the three subtrees and map different combinations of structured sensor information (described below) into motor commands.

According to our program design, structured sensor-conditionals involve comparing the values from different sensors or comparing the sensor values to the numerical values as thresholds. Different kinds of sensors are required to build different behavioral modules. We use the task *obstacle avoidance*, which is the application task used in the experiment later, as the example to explain how the sensor information is structured.

We assume the agent uses simulated IR sensors to acquire distance information in this task. The characteristic of an IR sensor is that it can only sense the obstacle within a certain distance. Based on this, the sensor information from a certain sensor is converted to a signal indicating how safe the agent is in the direction which the sensor is pointing. So a safety signal can be 1, indicating that there is no obstacle within the maximum distance which an IR sensor can sense, a value between 0 and 1, which is the ratio of sensed distance to the maximum distance an IR sensor can sense, or 0, indicating that the agent has bumped against an obstacle.

We then define the symbol IR as one of the terminals. In this work, we intend to co-evolve sensor positions with the controller, so that there are no pre-defined position candidates for IR sensors. The agent is allowed to acquire distance information from IR sensors in *any* direction it wants. Each terminal IR is defined to be associated with a value between 0 and 1 which indicates the angle between the direction in which the IR is pointing and the agent's heading. In this way, whenever a terminal IR is evaluated, converted distance information (safety signal), in the direction indicated by the value associated with it, is returned. For instance, a terminal IR with a value 0.5 will return the converted sensor information in the direction 0.5 revolution (180 degrees) relative the agent's heading.

The other terminal defined in this task is a numerical value between 0 and 1 inclusive: these are used as thresholds. Thus a structured sensor-conditional in this work is defined as the *constrained syntactic structure* X > Y, where X, Y can be any terminal. The symbol '>' is a non-terminal which performs the comparison operation and returns a boolean true/false according to the result.

To sum up, a reactive program here includes three boolean subtrees. The evaluation results of the boolean trees are interpreted as motor commands to drive left and right motors directly. A typical control program is shown and illustrated in Figure 2.

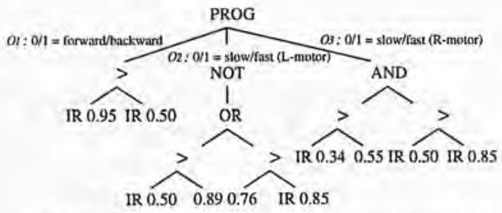
PROG
O1 : 0/1 = forward/backward
O2 : 0/1 = slow/fast (L-motor)
O3 : 0/1 = slow/fast (R-motor)
>
NOT
AND
IR 0.95 IR 0.50
OR
>
>
>
>
IR 0.34 0.55 IR 0.50 IR 0.85
IR 0.50 0.89 0.76 IR 0.85

Fig. 2. The diagram of a typical control program.

B.2 The Genetic Operations

As we mentioned previously, the genetic system simply copies individuals, without changing the brains or bodies, into the next generation for reproduction. But separate crossover and mutation operations are required for tree structures (controllers) and the string of real numbers (robot bodies). The crossover and mutation operations for controllers are described as follows and those for robot bodies will be described in the next section.

The *crossover* operation involves swapping two subtrees from the parent control programs. Because there are some *constrained syntactic structures*, such as X > Y, defined in this work, the crossover operation must be restricted. If the selected crossover point in the first parent is the root node, the second crossover point must be a root node as well (this means two individuals exchange their brains to become new individuals); if the chosen crossover point in one parent is an internal node, then the crossover point in the other parent must be an internal node too; otherwise if the selected crossover point in the first parent is a terminal node, the crossover point for the second parent is restricted to be a terminal node. In the last case there is an additional operation, *averaging*, that will possibly occur. If the types of the two terminals are different, i.e., a IR and a numerical constant, the system swaps them as described. But if the two terminals are the same type then *swapping* or *averaging* would occur randomly. The latter averages the two values associated with IRs or the two numerical constants. The *mutation* operation deletes a subtree at a randomly selected point and re-creates a random subtree to substitute it in the selected individual.

When the above genetic operations occur to the brains of the individuals, their corresponding bodies are not changed. This means, a changed brain with an unchanged robot body is put together to constitute a new individual in the next generation.

C. Evolving Robot Bodies

An agent is made up of by a brain and a body; both can affect the behavior. The performance of an agent is measured by how well the task is achieved by executing the brain on the corresponding body. In this section, we describe how to represent a robot body and to employ the genetic approach to evolve such a body.

In order to evolve a robot body we need to analyze and extract the determining elements, which affect the behavior of a robot profoundly, from the structure of a robot. In mobile robot design, for instance, there are some determining elements such as the wheel radius, the width of the wheel base, the time constant of the motion system, the body size (the diameter of the body, if we assume the robot body is round), and positions (with orientations) of the sensors, etc. The wheel radius affects the speed of the robot and determines the maximum and minimum moving speed for the specified motor commands; the width of the wheel base determines the turning rate of a robot; the time constant affects the response of the robot and determines the acceleration of the robot; the size of a robot body should be task-oriented: to avoid obstacles it may need to be smaller but to push boxes it may need to be larger; and the positions and orientations of the sensors allow the robot to acquire the perceptual information it needs. To evolve a robot body, in fact, means to decide these determining *structural parameters* of a robot genetically.

The structural parameters can be arranged as a linear string, in which each position is a real number representing the value of the corresponding parameter. Due to hardware limitations and performance considerations, each structural parameter has its lower bound and upper bound. When we build a robot, the value of each structural parameter must be between its bounds. Thus a robot body can then be expressed as

$$P_1 P_2 .....P_n$$

where

$$Min(P_i) \leq P_i \leq Max(P_i) \ ; \ 1 \leq i \leq n$$

For the linear string representation, a Genetic Algorithm can be employed to determine the value of each structural parameter P_i in its range.

Two-point crossover and one-point mutation operations are used to create new body strings. The crossover operation here, like the standard two-point crossover, involves two parents and two crossover points for parents. But its function is slightly different from the standard one: it is defined to perform operations of *exchanging* or *averaging* at random. The *exchanging* operation exchanges the P_i between two crossover points, but the *averaging* operation averages the corresponding P_i for the two parent strings instead. The mutation operation randomly picks a P_i for the selected parent and substitutes it with a re-generated random number, which satisfies its upper and lower bounds, so generating a new string.

IV. EXPERIMENTS AND RESULTS

In the experiment, we define "obstacle avoidance" behavior as the application task to evaluate the developed approach. The experiment is arranged in two phases. In the first phase, we concentrate on how to evolve an individual to move without collision; and in the second phase, we investigate the importance of the appropriate brain-body coupling.

A. Fitness Measures

As mentioned before, to evaluate an individual is to execute the control program on the corresponding robot body for a given period of time and to measure the performance according to certain criteria (fitness function). In this work, a fine time-slice technique is used. At each time step, the control program is evaluated once and drives its body to move; then the corresponding fitness is calculated. The accumulated fitness of an individual during the given time steps is then used to measure the performance.

An obstacle avoidance behavior means that an agent can keep moving without collision. A straightforward way to formulate this is to keep it as safe as possible at each time step. To achieve this, each agent is equipped with eight IR sensors, which point towards $-\frac{3}{4}\pi$, $-\frac{1}{2}\pi$, $-\frac{1}{4}\pi$, 0, $\frac{1}{4}\pi$, $\frac{1}{2}\pi$, $\frac{3}{4}\pi$, and π relative to the heading of the agent, and is trained to keep the safety signals from these IRs as close to 1 as possible. (The eight IRs mentioned here are used for fitness assessment only: they are independent of those sensors evolved as parts of the controllers. After training, these eight IRs are removed.) The term *minimum_safety*, which is the minimum of the eight safety signals, is used in the fitness function for this purpose. In order to keep it safe, the vehicle is punished whenever it begins getting dangerous (that is, the *minimum_safety* is less than 1), and the lower this value, the larger the penalty. In addition, in order to avoid the degenerate situation when an agent sticks at a certain position, or the situation when an agent spins, an agent is encouraged to move straight at high speed, and discouraged from rotation. Thus, the fitness function is defined as a penalty function. For an individual I,

$$Fitness(I) = \sum_{j \in Cases} \{\sum_{i=1}^{k} penalty(t_i) + (N-k) * penalty(t_k)\}_j$$

where

$$penalty = [\alpha * (1 - minimum\_safety) + \beta * (1 - v) + \gamma * w]$$

In this function, $Cases$ is the set of fitness evaluations done on this controller (fitness cases), k is the time the vehicle hits an obstacle, N is the given number of time steps that the obstacle avoidance behavior should last for, v is the normalized forward speed (backward is regarded as negative), and w is the normalized rotating speed. This would keep a vehicle safe and moving forward as straight as possible.

B. Evaluation and Testing

In this experiment, we trained the robot to avoid obstacles using the evolutionary procedure then tested the evolved pair of controller and robot body to examine the performance. Before training, we defined a training set including M starting positions. For a certain generation, each individual was trained to move from C starting positions, which were chosen randomly from the predefined training set, and ran for N time steps for each start position. The cumulative fitness is designated as the fitness of an individual. In our experiment, M was 30, C was 15 and N was 500.

The structural parameters we hope to evolve in this work are *time constant*, *wheel base*, *wheel radius*, and the *body size*, but each structural parameter has its own limitations. In this experiment, the value of time constant was restricted between 0.5 and 2.5 second; the lower bound and upper bound of wheel radius was 1.0 cm and 3.5 cm; the value of body size was limited from 10 cm to 25 cm; and the wheel base was constrained to be not larger than the body size.

A simple *island* model GA[10] is implemented in this work. It allowed us to use multiple populations to maintain diversity. We used two populations of 40 individuals each. The number of generations was 50 and the best individual appearing in the *last* generation is designated as the final solution. The best individual evolved from this procedure and its typical behavior is shown in Figure 3.

In order to examine the performance of the evolved solution, it is necessary to test it in different test cases. In our testing procedure, the evolved individual was tested 100 times, to control for random effects of perceptual and motor noise. Each time it moved from a new starting position with a given orientation and was allowed to move for 10000 time steps. The evolved best individual did not bump any obstacle in all the 100 test cases.

```
( PROG
   (> IR 0.52  IR 0.97)
   (OR  (OR  ( AND  (>  IR 0.19  IR 0.97)
                    (>  IR 0.21  IR 0.97))
              (> IR 0.60  IR 0.97))
        (> IR 0.23  IR 0.21))
   (> IR 0.71  IR 0.83))

time constant  :  0.68
wheel base     : 10.53
wheel radius   :  1.56
body size      : 10.53
```

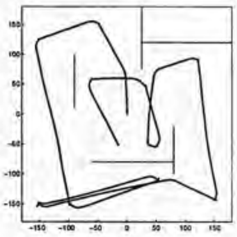

Fig. 3. The evolved solution(including the control program and the structural parameters); and the emergent behavior.

body-size	*wheel-base*	*wheel radius* 1.56	2.34	3.12	*wheel radius* 1.56	2.34	3.12
10.53	1.05	83	74	75	86	79	64
	5.26	96	83	70	84	74	71
	10.53	100*	99	97	96	86	81
15.79	1.05	64	61	52	71	59	42
	5.26	76	56	74	52	45	27
	10.53	89	95	92	95	64	51
	15.79	90	88	92	94	92	84
21.05	1.05	41	33	26	52	46	34
	5.26	51	24	11[d]	31	13	6
	10.53	30	53	65[c]	91	34	17
	15.79	24[b]	28	27[e]	67	75	68
	21.05	87[a]	19	30	62	59	41

In addition, we tested the evolved agent in somewhat different environments to examine whether it still had reasonable performance. In 10 different tests, the evolved agent did not have any collision in all of the test cases. Figure 4 illustrates two examples.

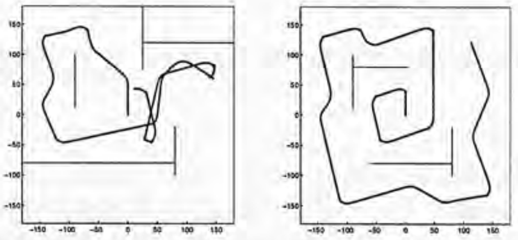

Fig. 4. Two examples of testing the evolved agent in changed environments.

C. The Importance of Appropriate Brain-Body Coupling

We have shown that the controller and the robot body can be co-evolved to achieve the behavior-specified task. In order to investigate how the evolved control program relies on the co-evolved body, we tested the evolved program on different robot bodies which are the various combinations of structural parameters available in their own ranges.

For each robot body, we tested it with the evolved control program 100 times. Each time it moved from a new starting position and was allowed to move 10000 time steps, like the testing procedure described in the above section. Each entry in Table 1 shows the number of successes (cases when the robot did not bump any wall during the 10000 time steps) from the 100 test cases for a certain robot body. From Table 1, we find that the combination where the evolved controller was executed on the co-evolved robot body (marked with an asterisk) has the highest success rate (actually it is 100%). The inappropriate brain-body couplings can not achieve the specified task perfectly. This demonstrates that the evolved controller relies on the co-evolved robot body.

C.1 Further Investigation

Some data in Table 1 attracts our attention. For the case *a* (indicated by the mark *a* at the corresponding number), for instance, the number of success increases dramatically, comparing to case *b*. We can explore the reasons by observing the behaviors emerging from the two pairs of brains and bodies.

In case *b*, the robot always bumped the wall due to the inappropriate enlargement of the body size and wheel base. But if the wheel base is enlarged more, the robot became more difficult to rotate, especially with the relatively small wheels which slow down the motion of the robot. This causes the robot to get stuck easily — it oscillates forward and backward with a little turning at a certain position, so it is safe in most of the test cases. The typical behaviors of case *a* and case *b* are shown in Figure 5.1 and Figure 5.2.

The other example we investigated is case *c*, whose performance is much better than cases *d* and *e*. After examining their behaviors, we found that there are actually two types of failure caused by certain kinds of robots which bump the wall easily. The first is a robot with a large body, small wheel base, and large wheels. A large body inevitably increases the bumping probability; the small wheel base with large wheels makes the behavior of the robot unstable: it is easy to turn at high speed. Consequently, it often drove the robot to bump the wall suddenly. Figure 5.3 shows this situation.

The second is a robot with a large body, wide wheel base, and large wheels. As in the first situation, a large body increases the bumping probability for the robot but the wide wheel base with large wheels, on the contrary, drives the robot forward faster with small turning rate. This results in the robot bumping the wall in most of the test cases although it can detect the wall efficiently and tries to move away from the wall. Figure 5.4 shows this situation.

As we can observe, if the wheel base of the robot becomes larger, failure of the first kind decreases, but failure of the second kind increases. In case *d*, most of the failures are because of the first situation but in case *e* most of the failures belong to the second situation. Regarding the two situations which cause the bumping behavior together, we find case *c* happens to be the best case with a small number failures for both situations. So its performance is better than case *d* and case *e*.

V. CONCLUSION AND FUTURE WORK

In this paper, we have developed a hybrid approach of Genetic Programming and Genetic Algorithms to co-evolve a reactive control program and its corresponding robot

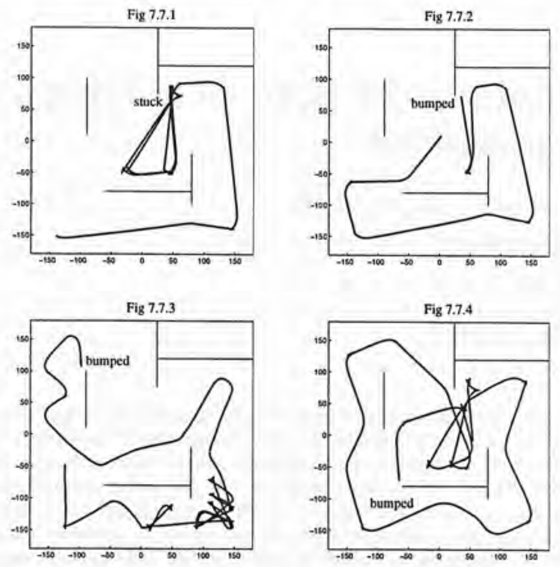

Fig. 5. Some faults caused by the inappropriate brain-body couplings (see text for explanation)

body to achieve the specific behavior. A boolean-tree has been well-defined to represent a control program and the determining structural parameters of a physical robot are extracted and arranged as a linear string of real numbers to represent a robot body. The GP part of the system is used to evolve the tree-structure of a control program and the GA part of the system is applied to determine the string of the structural parameters. Experimental results have shown the promise of the developed approach.

In addition, we have also analyzed the importance of appropriate brain-body coupling in designing a robot system. The evolved controller is successful only performing in the co-evolved robot body. This means, in the evolutionary process, the robot body itself also plays an important role because they both have adapted to the environment in order to achieve the task. For the simple tasks, a human designer may be able to design a robot body according to his prediction of the difficulty of the tasks, then design (or evolve) the controller. But for more complicated tasks, co-evolving controllers and morphologies for robot systems may provide a potential alternative.

Some aspects of future work are important. First of all, because our work is done in simulation, it is necessary to build a real robot (Lego-like) and download the evolved controller to it to observe the performance. Although there are gaps between simulated and real worlds, research [7] has shown that we could sample the real sensor data and the robot motion in the real world to build a more realistic simulator to develop evolutionary systems. Our future work also involves integrating these considerations into our simulator. Finally, since our experiment is focused on a certain behavior, we can go on to consider more difficult behaviors. Based on what we learned from this, we shall furthermore see if we can successfully evolve more new combinations of controllers and robot bodies for different tasks.

References

[1] S. B. Ackers. Binary Decision Diagrams. In *IEEE Transactions on Computers*, C-27(6), p509-516, 1978.

[2] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. In *ACM Computing Survey*, 24(3), p293-318, 1992.

[3] R. A. Brooks. Artificial Life and Real Robots. In *Proceedings of the First European Conference on Artificial Life*, p3-10, 1991.

[4] D. Cliff, I. Harvey, P. Husbands. Explorations in Evolutionary Robotics. In *Adaptive Behavior*, 2(1), p71-104, 1993.

[5] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA, 1989.

[6] J. R. Koza. *Genetic Programming: on the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

[7] O. Miglino, H. H. Lund, S. Nolfi. Evolving Mobile Robots in Simulated and Real Environments. To appear in *Artificial Life*, 1996.

[8] C. W. Reynolds. Evolution of Corridor Following Behavior in a Noisy World. In *Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, p402-410, 1994.

[9] K. Sims. Evolving 3D Morphology and Behavior by Competition. In *Proceedings of Artificial Life IV*, p28-39, 1994.

[10] R. Tanese. Distributed Genetic Algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, p434-439, 1989.

Applying Genetic Programming to Evolve Behavior Primitives and Arbitrators for Mobile Robots

Wei-Po Lee John Hallam Henrik Hautop Lund
Department of Artificial Intelligence
University of Edinburgh
Edinburgh, U. K.
{weipol, john, henrikl}@dai.ed.ac.uk

***Abstract*— Behavior-based approach has been successfully applied to design control systems of robots. This paper presents our work, based on evolutionary algorithms, to program behavior-based robots automatically. Instead of handcoding all the behavior controllers or evolving an entire control system for an overall task, we suggest our approach at the intermediate level: it includes evolving behavior primitives and behavior arbitrators for a mobile robot to achieve the specified tasks. To examine the developed approach, we evolve a control system for a moderate complicated box-pushing task as an example. We first evolved the controllers in simulation and then transferred them to the Khepera miniature robot. Experimental results show the promise of our approach and the evolved controllers are transferred to the real robot without loss of performance.**

I. Introduction

Behavior-based design has become a main alternative to traditional robot design nowadays. A number of behavior-based robots have demonstrated the performance of this approach [1; 16]. In general, a behavior-based system can be considered as a behavior network consisting of some behavior modules. To design such a system is to design individual behavior modules first and then design some kind of coordination mechanism to manage the interaction between them. This approach has been proven successful in that its robots can deliver real-time performance in a dynamic world, by creating tight coupling between perceptions and actions for each behavior. However, increasing robot and task complexity makes the design difficult. Consequently, an automatic design process using artificial evolution is advocated to synthesize the controllers.

Some researchers have used evolutionary algorithms to evolve robot controllers in the forms of LISP-like programs, neural networks, or classifier systems (i.e., [2][4], see [12] and [5] for a brief survey). Unlike behavior-based architectures, most of their work has no explicit pre-defined modular structures for the control systems, and they expect to create a comprehensive evolutionary system to evolve a single overall control system for a robot. The application tasks achieved in their work, such as obstacle avoidance, exploration, light seeking, are mostly too simple to promise that their work can be scaled up to achieve other more difficult tasks.

In this paper, we present our approach at the intermediate level between handcoding a behavior-based system and evolving an overall control system. It takes the behavior-based architecture as the control scheme and evolves the individual components, including *behavior primitives* and *behavior arbitrators*, by employing an evolutionary approach. This will take the advantages of both but mediate their disadvantages. In our work, the controllers are defined at the logic level: a controller is much like a boolean network, which uses *conditional structures* (constructed on sensor information) to select appropriate motor commands or activate a behavior primitive. A genetic programming system is implemented to evolve this kind of boolean controller.

To prove our approach, we describe how we employ it to undertake an application task, in which a robot is required to push a box toward a specific position indicating by a light source. This task is fairly complex compared to the ones achieved in previous Evolutionary Robotics experiments. Due to the time-consuming characteristic of on-line evolution, we use the technique of evolving control systems in simulation and then transferring to a real robot. Experimental results show that the robot can achieve the specified task reliably.

II. Real and Simulated Robots

The robot we used to test the evolved controllers is the miniature mobile robot *Khepera* [14] (Fig.1). The Khepera robot has a diameter of 55 mm, a height of 30 mm, and a weight of 70 g. Its two wheels are driven by two DC motors with incremental encoder and they can revolve forward and backward. The robot is equipped with eight infra-red proximity sensors (which can serve as ambient light sensors as well): six sensors are positioned on the front and the other two on the back. The sensors are numbered from 0 to 7 and the layout of the arrangement of the sensors is illustrated in Fig.1. The robot can be controlled by the host computer via a cable or run the control program on its own control chips. In the later experiments, the controllers evolved from simulation are downloaded to the robot to test the performance without the cable connection.

As mentioned previously, we use the approach of evolving in simulation and testing on the real robot. Some research work has shown that the gap between simulated and real world can be bridged by sampling the real world through the sensors and motors of the robot itself to build the simulator, and the simulation time can be reduced significantly by using the look-up tables constructed from the sampled data of sensors and motors [13; 15; 10]. In this work, we use this approach to build the tables to record the responses of infra-red sensors to a box, the responses of the ambient light sensors to the 25 Watt light source, and the motion of the robot at different commanded speeds. These tables are then used in simulation to evolve the control systems.

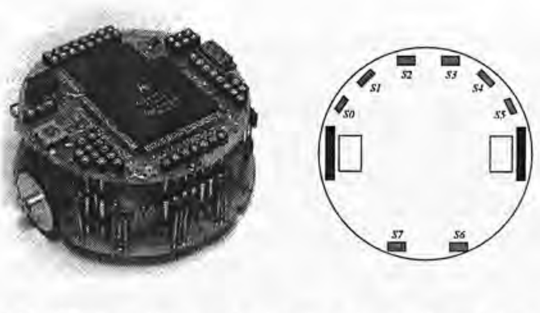

Fig. 1. The Khepera robot and its sensor arrangement. In the right figure, a sensor S_i can be an infra-red or an ambient light sensor.

III. Evolving Control Systems

A. Control Architecture

Like the general behavior-based system, the architecture of our control system includes a set of behavior primitives and the behavior arbitrators to coordinate them. However, unlike the subsumption architecture [1], for instance, we do not hardwire a priority network in advance. A behavior arbitrator in our control system is treated as a reactive controller: it has the same structure as the primitives and the only difference is that the output of a primitive is used to control the motors and the output of an arbitrator is used to activate a behavior primitive. Thus, similar to reactive planners [3], the arbitrators allow the binding between environment conditions and activations of lower level behaviors happening at the run time. This offers adaptiveness not only at the lower level sensor-motor control but also at the behavior level. Fig.2 illustrates the general architecture of our control systems.

B. The Application Task

To prove the developed approach, we use it to achieve a moderately difficult box-pushing task, which is to push a box toward a goal position indicated by a light source. There have been different versions of box-pushing tasks accomplished by reinforcement learning [11] and genetic programming [7]. However, those tasks are simpler than ours because their robots are only required to push a box toward any wall, which can be done without any deliberate strategy.

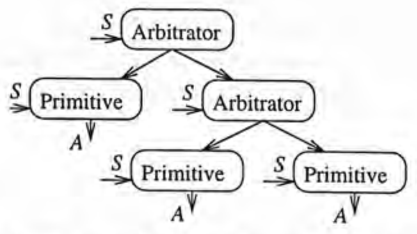

Fig. 2. The general architecture of a control system. S and A represent the sensors and actuators needed for a certain control work.

This task is difficult for several reasons. Since the robot only contacts the box at a point while pushing it, the box will tend to slide and rotate unpredictably when the pushing force exerted by the robot is not straight through the center of the box. Thus the robot has to adjust its position occasionally in order to keep pushing the box forward. Furthermore, as there is no restriction on the initial relative positions of the robot and the box, the robot needs to move to a proper position deliberately to push a box to satisfy the final goal. In learning such a task by a monolithic kinds of reactive controller, some failure could happen often: for instance, the robot keeps pushing a box but away from the light, or the robot itself moves toward the light without pushing the box.

To accomplish this task, we decomposed it into two other subtasks, *box-pushing* and *box-side-circling*. The goal of *box-pushing* is to keep the robot pushing a box forward, while the goal of *box-side-circling* is to require the robot moving along the side of a box. Each of them is controlled by a separate behavior primitive. In addition, a behavior arbitrator is involved to arrange the executing sequence of the behavior primitives. A genetic programming system is implemented to evolve both behavior primitives and arbitrators.

C. The Genetic System

The main evolutionary flow in this work is similar to the typical evolutionary approach. Given an environment and a goal formulated as a fitness function, an initial population is generated at random. After evaluating each individual, a certain selection scheme is used to choose parent individuals and some genetic operators are applied on them to create children individuals. In our implementation, the *tournament* selection scheme is used.

Three genetic operators, *reproduction*, *crossover*, and *mutation*, are applied to create a new generation. Because there are some constrained syntactic structures defined in this work, the crossover and mutation operations must be constrained, as explained in [8], to preserve the required structures.

In order to maintain the diversity and reduce the computation cost, an *island model* ([17]) GP system is implemented. Instead of using a single large population, it involves many small subpopulations, with an occasional exchange of useful information among the subpopulations. The migration policy used in this work is to copy a certain percentage of the best individuals in each subpopulation to substitute the same number of worst individuals in its neighbor subpopulations after a specified migration interval.

C.1 Representation

In our genetic system, the controller is reactive and is considered as a combinational logic system in which the output is determined completely by the current input states at each time step. In general, three types of non-terminals, the dummy root node, the logic components and the comparator, are defined for evolving the controllers. For the convenience of manipulating a GP system, a dummy root node is defined to connect some subtrees; the logic components are defined to constitute the main frame of the subtrees; and the comparator is defined to construct *sensor conditionals* which are described in the section below. The output of these subtrees are interpreted as actuator commands if a controller represents a behavior primitive, or used to activate other controllers if it is a behavior arbitrator.

According to our program design, structured sensor-conditionals involve comparing the values from different sensors or comparing the sensor values to the numerical values as thresholds. Sensors and thresholds, which are between 0 and 1 inclusive, are defined as terminals in our GP system. Depending on the characteristics of the specific tasks, different kinds of sensors are required and defined for evolving different behavioral modules. In general, a sensor terminal is defined to be associated with a value between 0 and 1 which indicates the angle between the direction in which the sensor is pointing and the robot's heading. Thus, whenever a sensor terminal is evaluated, the normalized sensor information, in the direction indicated by the value associated with it, is returned. For instance, a sensor terminal with a value 0.5 will return the converted sensor information in the direction 0.5 revolution (180 degrees) relative the robot's heading. In this way, the sensor positions and directions can be co-evolved with the controllers. The details are described in our previous research work [8]. In this work, we intend to use a fixed structure robot to verify our approach, so the sensors are fixed and named as S*0* to S*7*. The sensor arrangement is shown in Fig.1.

Thus, a structured sensor-conditional in this work is defined as a *constrained syntactic structure* X >= Y, where X, Y can be any terminal. The logic components then play the roles of mapping the structured sensor information into appropriate output. The aspect of a typical controller is illustrated in Fig.3.

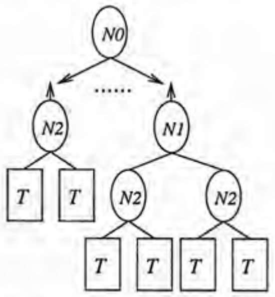

Fig. 3. The aspect of a typical controller. In this figure, $N1$, $N2$, $N3$ represent the three types of non-terminals, root-node, logic components, and comparator, respectively. The terminal T can be a normalized sensor response or a threshold between 0 and 1 inclusive.

C.2 Terminals and Non-terminals

As described earlier, there are three controllers, two behavior primitive and one arbitrator, involved in the application task. To apply a GP system to evolve these controllers, we need to define terminals and non-terminals for each of them. For both primitives, we defined infra-red sensor IR as the sensor terminal to provide distal information for the robot to locate the position of the box. In addition, numerical values are defined as the other kind of terminal, to be used as thresholds. For the behavior arbitrator here, both infra-red sensor IR and ambient light sensor LDR are defined as terminals to provide the perception clues of box and the target position. As in other controllers, numerical values are also defined as terminals to be the thresholds.

In our work, different control tasks need different kinds of sensor terminals but they use the same non-terminals. The non-terminals for evolving all the primitives and arbitrators are defined to include the root node PROG, the logic components AND, OR, NOT, XOR, and the comparator >=.

IV. Experiments and Results

In order to evolve a robust controller, we train a controller in multiple trials. For each task, we defined a training set of cases which are sampled randomly at each generation for training the controllers during the run. The best individual appearing in the last generation is designated as the final solution.

To evaluate a controller is to execute this controller for a given number of time steps and to measure its performance by certain criteria (fitness function). The accumulated fitness of a controller during the given time steps represents its performance. In the following experiments, a fitness function is defined as a penalizing function which needs to be minimized. The rest of this section describes the evaluation criteria for each control work and presents the results.

A. Evolving Primitive Box-Pushing

Box-pushing is to keep a robot pushing a box as straight as possible. It can be described as to keep the activation value of its front IR sensor high; the robot moving forward; and the speed difference of two motors low. The pressure from keeping the front distal sensor with high activation value will reinforce the robot to approach and face a box, and the pressure from keeping robot moving forward with low speed difference is to encourage the robot move straight and prevent it from getting stuck in front of a box. The combination of the these can lead a pushing forward behavior. The fitness function for a controller C is defined as:

$$f(C) = \sum_{t=1}^{T} \alpha * (1 - s(t)) + \beta * (1 - v(t)) + \gamma * w(t)$$

in which $s(t)$ is the average of normalized sensor activations of the front sensors IR*2* and IR*3*; $v(t)$ is the normalized forward speed; and the $w(t)$ is the normalized speed difference of two motors at each time step t.

Since this controller is a primitive, the output is grouped as two sets and converted to motor commands to drive two motors: the first three and the second three subtrees are decoded as motor commands for left and right motor, respectively. In our experiments, each motor command lasted 200 ms for both simulated and real robots.

For a certain run, two populations of 50 individuals were used and each individual was evaluated for 150 time steps. The evolution process lasted 50 generations and the best individual appearing in the last generation is:

```
(PROG
(OR (>= 0.13 IR0)(>= 0.13 IR0))
(>= IR1 IR1)
(XOR (>= IR3 IR7)(OR (>= 0.13 IR0)(>= 0.13 IR0)))
(>= IR1 0.36)
(>= IR1 IR1)
(XOR (>= 0.13 IR0)(>= IR2 0.13)))
```

After evolving in simulation, we transferred the above evolved controller to the Khepera robot. Fig.4 shows the typical behaviors of the simulated (left) and real (right) robots. This controller was tested many times on the real robot and each time it started from an arbitrary position and heading, in which it can sense the box. It always generated consistent behavior which is to turn to face the box and then to approach and push it. Although the robot did not contact and push the box exactly at the point of front center, it did achieve the goal reliably: continuously pushing the box forward.

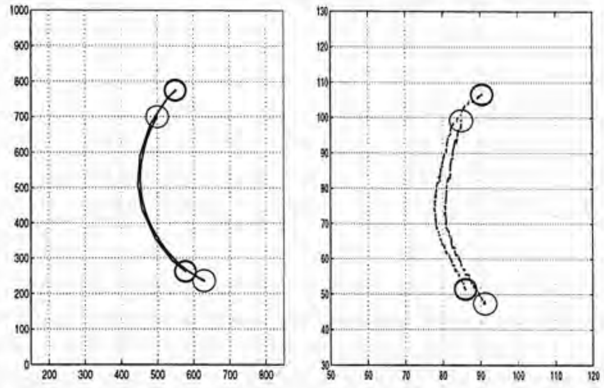

Fig. 4. The trajectories of simulated (left) and real (right) robots in pushing a box (the darker circles represent the robots). The figure for real robot is conducted by setting LEDs on the tops of the robot and box and using a video tracking system [9] to record their trajectories (©W.-P. Lee, J. Hallam, H. H. Lund, 1997).

B. Evolving Primitive Box-Side-Circling

Box-side-circling is to keep a robot moving forward and circling along the sides of a box. To prevent itself from touching the box too often, the robot is encouraged to keep a certain distance away from the box while circling the box. The evaluating criteria are described as to keep the robot's speed positive and the sensor IR*1* with a certain activation value. The fitness function is then defined as:

$$f(C) = \sum_{t=1}^{T} \alpha * abs(s(t) - k) + \beta * (1 - v(t))$$

where *abs* is a function which gives the absolute value of it argument; $s(t)$ is a normalized activation value of the specific sensor; k is a pre-defined constant indicating the distance between a robot and the box, in terms of the normalized sensor range; and v is the forward speed of a robot.

As in the *box-pushing* control work, this controller is a primitive and the output is interpreted exactly the same as the *box-pushing* primitive. In this experiment, two populations of 50 individuals were used. Each individual was evaluated 150 time steps and the number of generations was 50. The resulting controller is:

```
(PROG
(NOT (>= IR6 IR1))
(>= IR1 0.78)
(OR (>= IR5 IR1)(>= IR3 IR3))
(OR (>= 0.32 IR3)(>= IR5 IR4))
(>= IR4 IR3)
(OR (>= IR5 IR1)(>= IR6 IR3)))
```

Fig.5 shows the typical behaviors of the simulated and real robots. We tested this evolved controller several times by putting the real robot around the box with an arbitrary heading each time. In all the tests, the robot showed similar behavior: it performed turning until the specific sensor IR*1* faced the box and then moved along the side of the box. From the testing results, we can see that the robot has been able to achieve the specified goal and generate reliable behavior.

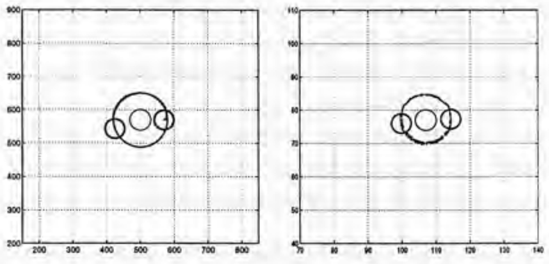

Fig. 5. The behaviors of simulated (left) and real (right) robots (©W.-P. Lee, J. Hallam, H. H. Lund, 1997).

C. Evolving the Behavior Arbitrator

Having evolved the first two controllers *box-push* and *box-side-circling*, we then evolve the third controller serving as the decision-making mechanism to manage the first two controllers to achieve the task of pushing a box toward a light source.

To achieve this goal, the robot needs both infra-red sensors and ambient light sensors to detect the box and the light, and the latter must be higher than the box to be able to receive the light in the case that the box is between the robot and the light. However, in the original design of the Khepera robot, there is no physical distinction between infra-red and ambient light sensors: the eight sensors equipped on the robot serve as both. We are currently duplicating the eight sensors on the top of our Khepera robot to test the evolved arbitrator. The results presented here are from the simulation. In the simulation, the look-up table of the light response is constructed from the existing sensors and we assume that, in addition, there are equivalent sensors on top of the simulated robot to respond to the light.

In this experiment, the box was placed 22 cm away from the light, which was masked by wrapping a piece of paper around it and placed 11 cm above the area. The robot is expected to push the box as close as possible to the center of the area brightened by the light. The fitness function is defined as:

$$f(C) = \sum_{t=1}^{T} D_{b,l}(t)$$

in which $D_{b,l}(t)$ represents the distance between the box and the light source at each time step t.

This controller is a behavior arbitrator and its single output is used to activate one of the above evolved primitives at each time step, according to the environment conditions. For a single run, 4 populations of 40 individuals were used and each individual was trained to act 600 time steps. The evolved controller is:

```
(PROG
(OR (OR (>= 0.62 LDR5))(OR (OR (>= IR6 LDR3)(>= 0.62
LDR7)) (AND (>= LDR6 LDR7)(>= LDR3 LDR7)))) (AND
(>= LDR3 LDR7)(OR (OR (OR (>= LDR5 LDR3)(>= LDR3
0.62)) (>= IR6 LDR7))(AND (NOT (>= IR6 LDR4))(>= LDR3
LDR7)))))
```

Fig.6 illustrates the typical behavior of the simulated robot. The arbitrator first activated the primitive *box-side-circling* to move along the side of a box to an appropriate position in which the box was between the light and the robot itself. It then switched to the other primitive, *box-pushing*, to push the box forward. The *box-side-circling* behavior was activated again if the robot's path was deviated. From Fig.6, we can see that the box was pushed to almost the center of the bright area and the robot continuously performed *box-side-circling* primitive to prevent pushing the box away from the light, after the box was pushed to the target position.

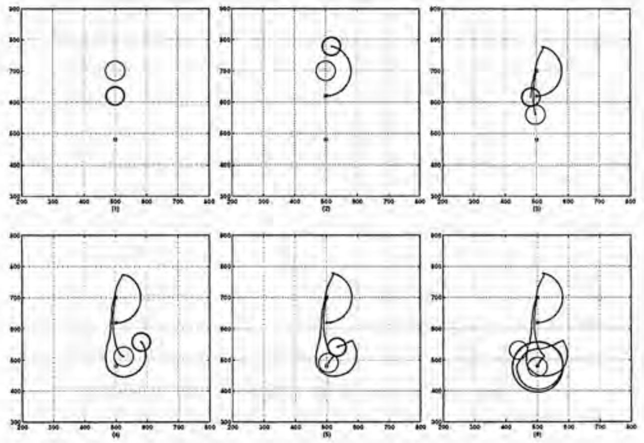

Fig. 6. The behavior sequence of the robot: (1) The initial positions of the robot, the box and the light; (2) the robot moved along the side of the box; (3) pushing the box forward; (4) circling again to an appropriate position; (5) pushing the box again to the goal position; (6) continuously circling after the box has been pushed to the goal position (©W.-P. Lee, J. Hallam, H. H. Lund, 1997).

V. Conclusion and Future Work

In this paper, we have presented our approach in applying evolutionary techniques to evolve the control system for a simulated robot and then transferred evolved controllers to a real robot. Instead of hand-coding all of the individual components needed in a general behavior system or evolving the whole control system for an overall task, we propose to take the control architecture of a behavior-based system and construct the behavior primitives and arbitrators by the use of an evolutionary approach. To assess the performance of the developed approach, we have evolved a control system to achieve an application task of box-pushing as an example. Experimental results show the promise and efficiency of the presented approach.

By employing the evolutionary approach, our work shows a way to aid the design of a behavior-based robot, including automatically programming all the individual behavior controllers and dealing with the selection problem among these behavior controllers. Since the control architecture in our work is arranged to be similar to that of behavior-based robots, the resulting control system therefore inherits the characteristic of the behavior-based robots: it delivers real-time performance.

As far as applying an evolutionary approach is concerned, our approach also has several potential advantages. First of all, owing to the separate design of behavior primitives and arbitrators, the goal of each task is relatively simple. This means that designing a fitness function for such a task is not as difficult as generally thought. Secondly, since each individual controller involves fewer inputs and outputs, one can apply a simple evolutionary system to evolve a robust controller without too much effort. For example, in the application task achieved in this paper, the evolutionary system can converge to a stable and sufficient solution within only 30 generations. Thirdly, our controllers only involve logical operators, such as AND OR NOT, that are very simple to evaluate. This means that our approach is computationally cheap; especially, in the cases that the first argument of a AND is false, or the first argument of a OR is true, there is no need to evaluate the other argument. Finally, because our controllers are constituted of very basic logic components, they can be easily compiled to custom hardware such as FPGAs to speed up the evaluation in controlling a robot.

Some further research work is currently in progress. The first is the construction of new sensors on the real robot to test the behavior arbitrator. Another is to investigate whether this approach can be applied to evolve control systems for even more complicated tasks.

ACKNOWLEDGEMENTS.

Henrik Hautop Lund is supported by EPSRC grant nr. GR/K 78942 and The Danish National Research Councils. Facilities provided by University of Edinburgh.

REFERENCES

[1] R. A. Brooks. A Robust Layered Control System for a Mobile Robot. In *IEEE Journal of Robots and Automation*, vol RA-2(1), pp.14-23, 1986.

[2] D. Cliff, I. Harvey, and P. Husbands. Explorations in Evolutionary Robotics. *Adaptive Behavior*, 2(1):73-110, 1993.

[3] R. J. Firby. Task Networks for Controlling Continuous Processes. In *Proceedings of the Second International Conference on AI Planning Systems*, 1994.

[4] D. Floreano and F. Mondada. Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural-Network Driven Robot. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pp.421-430. 1994.

[5] T. Gomi, A. Griffith. Evolutionary Robotics – An Overview. In *Proceedings of IEEE International Conference on Evolutionary Computation*, 1996.

[6] J. R. Koza. *Genetic Programming: on the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

[7] J. R. Koza, J. P. Rice. Automatic Programming of Robots using Genetic Programming. In *Proceedings of AAAI-92*, pp.194-201, 1992.

[8] W.-P. Lee, J. Hallam, H. H. Lund. A Hybrid GP/GA Approach for Co-evolving Controllers and Robot Bodies to Achieve Fitness-Specified Tasks. In *Proceedings of IEEE International Conference on Evolutionary Computation*, 1996.

[9] H. H. Lund, E. V. Cuenca, J. Hallam. A Simple Real-Time Mobile Robot Tracking System. Research Paper no.41, Department of Artificial Intelligence, University of Edinburgh, 1996.

[10] H. H. Lund and J. Hallam. Evolving Sufficient Robot Controllers. In this volumn.

[11] S. Mahadevan, J. Connell. Automatic Programming of Behavior Based Robots Using Reinforcement Learning. In *Proceedings of AAAI-91*, 1991.

[12] M. Mataric, D. Cliff. Challenges in Evolving Controllers for Physical Robots. In *Robotics and Autonomous Systems*, 19(1): 67-83, 1996.

[13] O. Miglino, H. H. Lund, S. Nolfi. Evolving Mobile Robots in Simulated and Real Environments. In *Artificial Life*, 2(4), 1996.

[14] F. Mondada, E. Franzi, P. Ienne. Mobile Robot Miniaturation: A Tool for Investigation in Control Algorithms. In *Proceedings of the Third International Symposium on Experimental Robotics*, 1993.

[15] S. Nolfi, D. Floreano, O. Miglino and F. Mondada. How to Evolve Autonomous Robots: Different Approaches in Evolutionary Robotics. In *Proceedings of Artificial Life IV*, pp.190-197. 1994.

[16] L. Steels. Building Agents out of Autonomous Behavior Systems. In *The Artificial Life Route to Artificial Intelligence*. L. Steels and R. Brooks (eds), 1993.

[17] R. Tanese, Distributed Genetic Algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pp.434-439. 1989.

Evolving Robot Morphology

Henrik Hautop Lund **John Hallam** **Wei-Po Lee**

Department of Artificial Intelligence
University of Edinburgh, 5 Forrest Hill, Edinburgh EH1 2QL, Scotland, UK
henrikl@dai.ed.ac.uk john@dai.ed.ac.uk weipol@dai.ed.ac.uk
http://www.dai.ed.ac.uk/staff/Henrik_Lund.html

Abstract—**True evolvable hardware should evolve whole hardware structures. In robotics, it is not enough only to evolve the control circuit — the performance of the control circuit is dependent on other hardware parameters, the *robot body plan*, which might include body size, wheel radius, motor time constant, sensors, etc. Both control circuit and body plan co-evolve in true evolvable hardware. By including the robot body plan in the genotype as a kind of *Hox* gene, we co-evolve task-fulfilling behaviors and body plans, and we study the distribution of body parameters in the morphological space. Further, we have developed a new hardware module for the Khepera robot, namely ears with programmable amplifiers, synthesizers, and mixers, that allow us to study true evolvable hardware by modelling the evolution of auditory sensor morphology.**

I. True Evolvable Hardware.

The concept of evolvable hardware (EHW) or evolware has mainly been interpreted as reconfiguring Field Programmable Gate Arrays (FPGA) by using genetic learning to adapt the circuit architecture to new unknown environments [9; 10; 18; 23; 25]. However, we will argue that *true evolvable hardware* should be interpreted as hardware, where not only the primitive gates (e.g. AND, OR gates) or high-level functions (e.g. adder, subtracter, sine generator) of a FPGA are evolvable, but the whole physical morphology is evolvable.
Generally, EHW should refer to hardware that can change its architecture and behavior dynamically and autonomously with its environment: "EHW should be regarded as an evolutionary approach to behavior design rather than hardware design" [27]. Therefore, EHW should include the whole architecture of an integrated system and not only the circuit, especially when we are interested in the development of complete autonomous systems. The motivation for developing complete autonomous systems has been outlined by Malcolm, Smithers and Hallam [16], and we agree with Pfeifer [20] that complete autonomous systems should be autonomous, self-sufficient, embodied, and situated. Hence, the design of such a system must include self-organisation, and self-organisation means adaptation of *both* morphology and control architecture.
However, the EHW community has looked only at adaptation of control architectures, for example through the evolution of FPGAs. Let us, for instance, look at EHW for robot control, as suggested by Thompson [25]. In this case, traditionally, one would interpret the EHW as being the reconfigurable (evolvable) circuit, on which one can evolve a task-fulfilling robot controller for a specific task. Hence, one would evolve the architecture of the control circuit. As identified by Thompson, this might give some advantages over evolution of controllers on fixed hardware. On fixed hardware, the evolved control system will be implemented as a software simulation of a specific hardware configuration. Such a software simulation will likely slow down the processing speed. Secondly, by giving evolution control over the reconfigurable hardware, one might obtain controllers that exploit the reconfigurable circuit extensively by using designs that are traditionally prohibited by engineers, or by exploiting defects in the circuit. However, the circuit architecture is only a part of the hardware system, and ideally we would like to evolve the whole system. The hardware of a robot consists of both the circuit, on which the control system is implemented, and the sensors, motors, and physical structure of the robot.
True EHW should evolve the whole hardware system, since the evolution and performance of the electronic hardware is largely dependent on the other parts of the hardware that constitute the system. We call the latter part the *robot body plan.* A robot body plan is a specification of the body parameters. For a mobile robot, it might be types, number and position of sensors, body size, wheel radius, wheel base, and motor time constant. With one specific motor time constant, the ideal control circuit should evolve to a different control than with another motor time constant; different sensors demand different control mechanisms; and so forth. Further, the robot body plan should adapt to the task that we want the evolved robot to solve. An obstacle avoidance behavior might be obtained with a small body size, while a large body size might be advantageous in a box-pushing experiment; a small wheel base might be desirable for a fast-turning robot, while a large wheel base is preferable when we want to evolve a robot with a slow turning; and so forth. Hence, the performance of an evolved hardware circuit is decided by the other hardware parameters. When these parameters are fixed, the circuit is evolved to adapt to those

[0]Invited Paper. In *Proceedings of IEEE Fourth International Conference on Evolutionary Computation*, IEEE Press, NJ, 1997.

fixed parameters that, however, might be inappropriate for the given task. Therefore, in *true EHW*, all hardware parameters should co-evolve.

II. Biological Background.

A brain does not do much without a body, while a body cannot do much without a brain to control it. Brains and bodies have co-evolved and fit almost perfectly to each other. A human brain would not be much help to a parrot, and an elephant brain would not be appropriate to control a human body. The fact that the body largely determines the performance of the brain (or rather the control mechanism) has previously been ignored by the research communities studying evolutionary computation, artificial life, robotics, and adaptive behavior, where research has been on evolving/developing adaptive control systems for agents with a fixed structure. The normal practice has been that if one were to test hypotheses in a real agent in the real world, then one would buy a robot with a pre-defined structure (i.e. pre-defined sensors, motors, size, etc.), or, in some cases, build one's own robot, but then with a fixed non-reconfigurable body plan.
However, the biological facts about the evolution of body plans should appeal to these researchers. Nature tells us that body plans evolve, and the biological data suggest that body plans of all animals, from fruit flies to elephants, are controlled by the same kinds of genes, namely the *Hox* genes [2]. The first multicellular animals' body plans are believed to have been largely the work of a primitive set of *Hox* genes (Antp-like genes) and descendants of these genes have been sculpting the body plans of animals ever since. When changing *Hox* genes inside embryos, cells may change and, for instance, limbs might grow in the wrong place. For example, genetic experiments with homeotic genes in mice have demonstrated that *Hox* genes are in part responsible for the specification of segmental identity along the anterior-posterior axis, and it has been proposed that an axial *Hox* code determines the morphology of individual vertebrae [11], and *Hoxa-1* and *Hoxa-2* have been shown to play a critical role in head development in both mice and *Drosophila* [5].
Further, there exist both biological data and philosophical reflections that suggest that some biological forms are impossible in the morphological space [4; 21]. Different species are clustered together in the morphological space with a long distance e.g. from insects to mammals, and we agree with Emmeche [4] that there is no direct jump from one cluster to another (see Fig. 1). The displacement in the morphological space is a long evolutionary process that is largely based on exaptations and pre-adaptations [7; 8; 15].

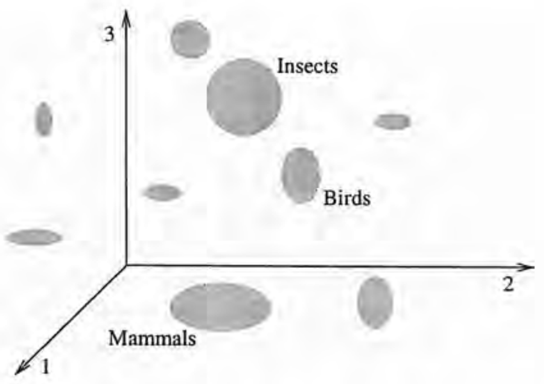

Fig. 1. The morphological space in which natural forms are clustered together. The three axes are morphological parameters. Modified from [4].

III. Evolving Robot Body Plans.

Impossible regions in the morphological space are not considered in Sims work [22], where he co-evolves morphologies in simulation. Hence, that approach might fail in real robots, since also the possibilities in the robot morphological space are limited. A more fruitful approach should combine our knowledge about robots with genetics.
The biological facts about *Hox* genes suggest that the evolution of body plans can be modelled by having parts of a genetic string to express growth of different body parts. The idea of a developmental morphogenetic model has been investigated in simulation, e.g. [1; 3], but only in order to develop the neuronal structure and not the physical structure of an agent. Spirov [24] has investigated the pattern-form interplay in a model of the early development of sea urchin. However, this simulation work does not give us indications of how to evolve real hardware structures.
Our first approach toward evolving robot body plans uses a simple direct encoding from gene to physical expression, since our immediate goal is to show the validity of evolving hardware structures rather than the validity of a developmental model. The genotype of a robot expresses a tree structure that is used as controller (and determines the number of sensors and their position) and a list of real numbers that determines the robot body plan (i.e. body size, wheel base, wheel radius, motor time constant). Hence, we co-evolve robot controllers (the traditional EHW) *and* robot body plans.
By using an evolutionary algorithm (in our case, we use genetic programming to evolve the controller and a genetic algorithm to evolve the body parameters), we can evolve robots that adapt to specific tasks, as shown in [12]. This is done in a carefully made robot simulator, and as shown in [14; 17], simple robot behaviors can be transferred from a carefully made robot simulator to reality with little difficulty. With this technique, we are currently building a simulator for LEGO robots that will

allow us to co-evolve controllers and body plans in simulation before constructing and re-building the LEGO robots according to the evolved body plans.
When co-evolving robot controller and body, we can interpret the dimensions of the morphological space as being sensor number and position, body size, wheel base, wheel radius, and motor time constant. For a specific task, we can then analyze how the evolved robots cluster in specific regions of the morphological space. For practical reasons, we map only two dimensions here.
Figure 2 shows the distribution of body size and wheel

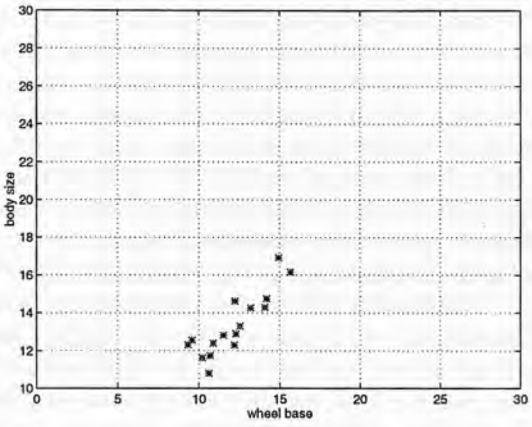

Fig. 2. Two dimensions of the robot morphological space. The plots are of the best robots from a number of evolutionary runs. There is an almost linear relationship between robot body size and wheel base in the robots that are evolved to perform obstacle avoidance tasks.

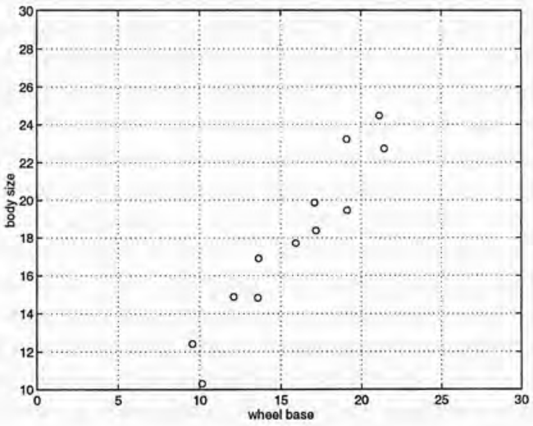

Fig. 3. The relationship between robot body size and wheel base for robots with double sensor range. The relationship is still linear, but the body size and wheel base can become larger, since the robots sense an obstacle further away and have more time steps to react.

base in the morphological space of a number of robots evolved to perform an obstacle avoidance task. We observe that relative small body sizes seem to be the most fit for this task and that there is an almost linear relationship between body size and wheel base (the correlation coefficient is 0.8694). These results explain that a small size robot is better at performing obstacle avoidance, and the body-size wheel base reduces the turning rate in order to keep the robot moving stably. Further,when increasing the sensor range, as in Fig. 3, the correlation between the wheel base and body size maintains approximately linearity, but the range of coverage of the evolved wheel base and body size becomes wider. This is due to the fact that the robots can sense obstacles further away, and therefore have more time to react (e.g., to turn). Hence, a larger body and a larger wheel base that means more time steps to turn the robot can evolve. In other words, the upper limit of the size and base of a robot is constrained by the sensor range.

IV. Evolving Auditory Sensor Morphology.

We have developed a new piece of hardware for the Khepera roboti [19], namely ears (see Fig. 4). This hardware is reconfigurable, and will allow us to study the co-evolution of controller and ears morphology.
As an example consider the cricket. The male cricket

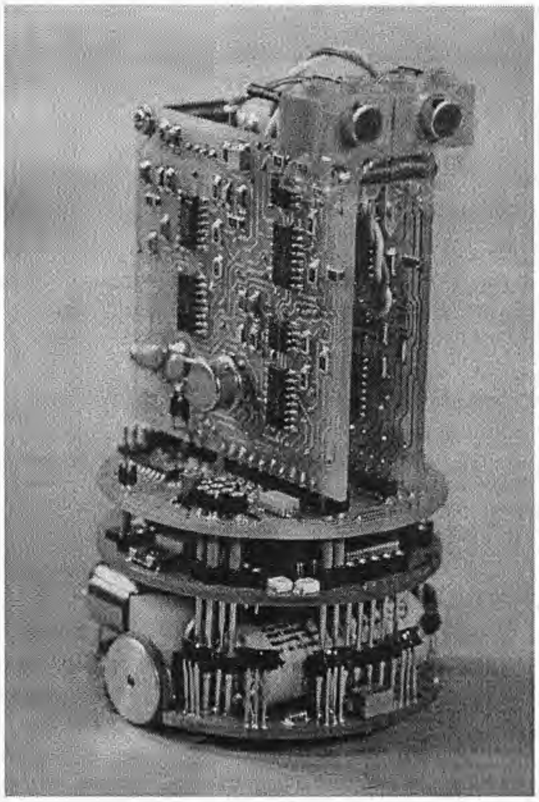

Fig. 4. The Khepera robot with ears. The ears have programmable amplifiers, synthesizers, and mixers.

produces a species-specific song by rubbing his wings together, and the female is able to locate males generating her conspecific song by phonotaxis. The morphology of the cricket's auditory apparatus is crucial to the successful performance of this skill.
The female cricket has four auditory openings: an ear (tympanum) located on each upper foreleg, and an auditory spiricle (or hole) on each side of the frontal section

of her body. The four are linked internally by means of tracheal tubes. Sound reaches the tympani directly through the air and, after propagation through the internal tubes, from the other auditory openings. The sound transduced from each tympanum by the cricket's auditory receptors is thus a combination of delayed and filtered signals from the other tympanum and the spiricles arriving at the back of the tympanum with the direct sound arriving at its outer face.

The delays and filtering performed by the auditory morphology improve the cricket's ability to discriminate the arrival direction of the conspecific song since the phased combination of sounds from the different sources induces a strong directional sensitivity into the response of each tympanum. Essentially, sounds arriving from the same side as the tympanum are delayed by the internal structures to arrive in anti-phase with respect to the direct path at the ipsilateral ear and in phase at the contralateral ear. Since the sounds arriving by the two paths are subtracted (being on opposite sides of the tympanum), the stimulus intensity at the ipsilateral ear is enhanced while at the contralateral ear it is diminished.

In the cricket, the delays and filter characteristics of the internal auditory structures are species-specific. To model the auditory morphology of the cricket, we have built an electronic emulation of some of these characteristics (see Fig. 5). Sound is collected by two or four microphones whose spacing is carefully controlled. After amplification and initial filtering three delayed copies of the sound are generated with programmable relative delays, which are then scaled and added together to construct a tympanal response. The intensity of the resultant signal is transduced using an analogue-to-digital conversion system for use by the control program. This hardware allows us to approximate the auditory morphology of various crickets by adjusting the programmable delays and the summing gains. It is not a perfect emulation of the insect, however: two programmed delays allow us to sum signals from each tympanum and both spiricles, but not from all auditory openings; and the summation system allows us to program relative gains, but not frequency dependent gains.

Nevertheless, the emulation circuitry is able to model a variety of specialised morphology auditory systems, and allows us to investigate the relationship between the auditory morphology, the conspecific song, and the internal control system that generates the phonotaxis behaviour shown by the female cricket in response to the call of a mate. One possible investigation is then to co-evolve controller and auditory morphology to give good phonotaxis to a specific song while giving good discrimination between different kinds of songs.

As a test of the reconfigurable hardware, we look at fixed amplifications, delays, scaling, and adding (i.e. we model a specific morphology (of *Gryllus bimaculatus*)). With these fixed parameters, we can verify that the

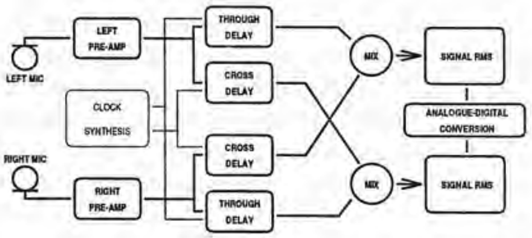

Fig. 5. Simplified diagram of the ears circuit. When sound arrives at each microphone (the analogue of the cricket's tympanum), the received signal is pre-amplified. The signal is then sent with a 'through delay' to the mixer at the same side, and with a 'cross delay' to the mixer at the opposite side. The mixed signal is sent through an RMS and an A/D converter to one of the Khepera's input channels. The same happens on the opposite side.

hardware works as intended. We do so by designing a control system that models the female cricket's control mechanism, and by emitting recordings of male cricket song from a loud speaker. We would then expect the robot to navigate toward the loud speaker.

Figure 6 shows the result of five such runs where we

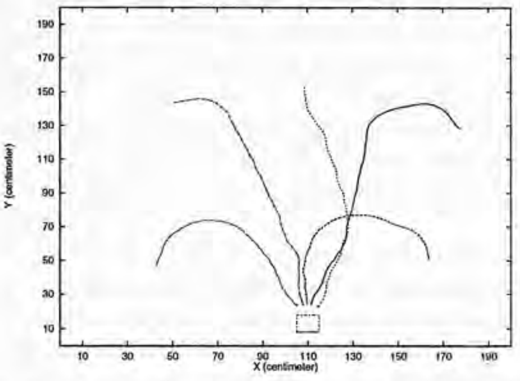

Fig. 6. The trajectories of the Khepera robot with ears parameters set as in *Gryllus bimaculatus*, and with male song from the same species being emitted from the loud speaker at the bottom. Data are collected by putting a LED on the robot and then using a video-tracking system [13].The robot hits the speaker, but the shown trajectories stop before, because the LED is placed in the center of the robot.

have modelled the control mechanism and the ears morphology of the female cricket *Gryllus bimaculatus*, and used recordings of the male *Gryllus bimaculatus*. In this case, the robot navigates toward the male song emitted from the loud speaker. Initially, there is no sound present, and the robot moves forward in a straight line. When we start to emit the cricket male song from the loud speaker, the robot will turn and move toward the loud speaker. The result is significantly different from our previous results with a LEGO robot prototype in that we are now able to use real cricket songs where we previously had to use computerized songs with syllables much longer than in the real cricket song, though we expect to be able to demonstrate the same statistical properties of the robot behavior relative to the cricket as we have done for the LEGO robot version [26].

V. Conclusion and Future Work.

We have outlined here the framework under which one can investigate the co-evolution of reconfigurable control systems and reconfigurable body plans. It is our view, that, at least in robotics, the concept of evolvable hardware should be extended to include the robot body plan, since the evolution of the circuit architecture is highly dependent on the specific body plan. Analyses of the robot morphological space tell us that some body plans are impossible or impracticable, so evolved control circuits might fail if this is not taken into consideration. It might be difficult to imagine how to obtain reconfigurable robot body plans, yet LEGO robots are one possibility. We are currently designing a LEGO robot simulator that allows us to co-evolve LEGO robot brains and bodies before assembling a LEGO robot accordingly and down-loading the control system. Fukuda [6] has also suggested a Cellular Robotic System that consists of many robotics units that can be reconfigured depending on given tasks and environments. Another possibility is to use devices similar to the ears that we have developed for the Khepera robot. These do indeed allow on-line reconfiguration, and we can use them to study the co-evolution of controller, song, and morphology.

Acknowledgements.

Henrik Hautop Lund is supported by EPSRC grant nr. GR/K 78942 and The Danish National Research Councils. Robotic experiments in cricket phonotaxis is a collaboration with Barbara Webb, and the development of the ears circuit has been done together with Andrew Haston.

References

[1] A. Cangelosi, D. Parisi, and S. Nolfi. Cell division and migration in a 'genotype' for neural networks. *Network*, 5:497–515, 1994.

[2] S. Day. Invasion of the shapechangers. *New Scientist*, (2001):30–35, 1995.

[3] F. Dellaert and R. D. Beer. Toward an evolvable model of development for autonomous agent synthesis. In R. Brooks and P. Maes, editors, *Proceedings of ALIFE IV*, Cambridge, MA, 1994. MIT Press.

[4] C. Emmeche. Mine molekyler vil frikende mig. Unpublished manuscript, Niels Bohr Institute, 1996. http://connect.nbi.dk/~emmeche/emmeche.html.

[5] M. Frasch, X. Chen, and T. Lufkin. Evolutionary-conserved enhancers direct region-specific expression of the murine *Hoxa-1* and *Hoxa-2* loci in both mice and *Drosophila*. *Development*, 121:957–974, 1995.

[6] T. Fukuda. Structure decision method for self organizing robots based on cell structure-CEBOT. In *Proceedings of International Conference on Robotics and Automation '89*, 1989.

[7] S. J. Gould. Exaptation: A crucial tool for an evolutionary psychology. *Journal of Social Issues*, 3:43–65, 1991.

[8] S. J. Gould and E. S. Vrba. Exaptation - a missing term in science of form. *Paleobiology*, 8:4–15, 1986.

[9] H. Hemmi, J. Mizoguchi, and K. Shimohara. Development and evolution of hardware behaviours. In R. Brooks and P. Maes, editors, *Proceedings of ALIFE IV*, Cambridge, MA, 1994. MIT Press.

[10] T. Higuchi, T. Niwa, T. Tanaka, H. Iba, H. de Garis, and T. Furuya. Evolving hardware with genetic learning: A first step towards building a Darwin machine. In J. Meyer, H. L. Roitblat, and S. W. Wilson, editors, *From Animals to Animats II: Proceedings of the Second International Conference on Simulation of Adaptive Behavior (SAB92)*, Cambridge, MA, 1992. MIT Press-Bradford Books.

[11] M. Kessel and P. Gruss. Murine developmental control gene. *Science*, 249:347–379, 1990.

[12] W.-P. Lee, J. Hallam, and H. H. Lund. A Hybrid GP/GA Approach for Co-evolving Controllers and Robot Bodies to Achieve Fitness-Specified Tasks. In *Proceedings of IEEE Third International Conference on Evolutionary Computation*, NJ, 1996. IEEE Press.

[13] H. H. Lund, E. d. V. Cuenca, and J. Hallam. A Simple Real-Time Mobile Robot Tracking System. Technical Paper 41, Department of Artificial Intelligence, University of Edinburgh, 1996.

[14] H. H. Lund and O. Miglino. From Simulated to Real Robots. In *Proceedings of IEEE Third International Conference on Evolutionary Computation*, NJ, 1996. IEEE Press.

[15] H. H. Lund and D. Parisi. Pre-adaptation in populations of neural networks evolving in a changing environment. *Artificial Life*, 2(2):179–197, 1996.

[16] C. Malcolm, T. Smithers, and J. Hallam. An Emerging Paradigm in Robot Architecture. In *Proceedings of Intelligent Autonomous Systems 2*, Amsterdam, 1989.

[17] O. Miglino, H. H. Lund, and S. Nolfi. Evolving Mobile Robots in Simulated and Real Environments. *Artificial Life*, 2(4):417–434, 1996.

[18] J. Mizoguchi, H. Hemmi, and K. Shimohara. Production genetic algorithms for automated hardware design through an evolutionary process. In *Proceedings of First IEEE International Conference on Evolutionary Computation*, NY, 1994. IEEE Press.

[19] F. Mondada, E. Franzi, and P. Ienne. Mobile robot miniaturisation: A tool for investigation in control algorithms. In *Experimental Robotics III. Lecture Notes in Control and Information Sciences 200*,

pages 501–513, Heidelberg, 1994. Springer-Verlag.

[20] R. Pfeifer. Building "Fungus Eaters": Design Principles of Autonomous Agents. In P. Maes, M. J. Mataric, J. Meyer, J. Pollack, and S. W. Wilson, editors, *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, Cambridge, MA, 1996. MIT Press.

[21] R. Riedl. *Order in Living Organisms. A System Analysis of Evolution.* John Wiley & Sons, Chichester, 1978.

[22] K. Sims. Evolving 3D Morphology and Behavior by Competition. In R. Brooks and P. Maes, editors, *Proceedings of ALIFE IV*, Cambridge, MA, 1994. MIT Press.

[23] M. Sipper. Designing Evolware by Cellular Programming. In *Proceedings of First International Conference on Evolvable Systems: from Biology to Hardware*, Heidelberg, 1996. Springer-Verlag.

[24] A. V. Spirov. Changes of initial symmetry in the pattern-form interaction model of sea urchin early development. *Journal of Theoretical Biology*, 161:491–504, 1993.

[25] A. Thompson. Evolving electronic robot controllers that exploit hardware resources. In F. Moran, A. Moreno, J.J. Merelo, and P. Charon, editors, *Advances in Artificial Life: Proceedings of 3.rd European Conference on Artificial Life*, Heidelberg, 1995. Springer-Verlag.

[26] B. Webb. Using robots to model animals: a cricket test. *Robotics and Autonomous Systems*, 16:117–134, 1995.

[27] X. Yao and T. Higuchi. Promises and Challenges of Evolvable Hardware. In *Proceedings of First International Conference on Evolvable Systems: from Biology to Hardware*, Heidelberg, 1996. Springer-Verlag.

Learning Complex Robot Behaviours by an Evolutionary Approach

Wei-Po Lee, John Hallam, Henrik Hautop Lund

Department of Artificial Intelligence,
University of Edinburgh,
Edinburgh EH1 2QL, Scotland, UK
email:{weipol,john,henrikl}@dai.ed.ac.uk

Abstract. Building robots can be a tough job because the designer has to predict the interactions between the robot and the environment as well as to deal with them. During the last few years, some researchers have shown the advantages of using evolutionary algorithms to automate the design of robots. However, the tasks achieved so far are fairly simple. In this work, we analyse the difficulties of applying evolutionary approaches to learn complex behaviours for mobile robots. Instead of evolving the controller as a whole, we propose to take the control architecture of a behavior-based system and to learn the separate behaviours and the arbitration by the use of an evolutionary approach. By using the technique of task decomposition, the job of defining fitness functions becomes more straightforward and the tasks become easier to achieve. To assess the performance of the developed approach, we have evolved a control system to achieve an application task of box-pushing as an example. Experimental results show the promise and efficiency of the presented approach.

1 Introduction

In recent years, building reactive control systems for robots has become a major alternative to traditional robot design. This approach has been proven to be able to achieve real-time performance for robots by creating tight coupling between perceptions and actions. However, predicting and dealing with unforeseen situations and circumstances in the environment make the design still difficult. Consequently, the idea of getting the robot to learn to achieve the tasks, through the interaction between the robot itself and the environment, is advocated.

Evolutionary Robotics is a kind of approach which enables the robot to learn behaviours by the use of evolutionary techniques. This approach differs from other learning skills in that it operates a population of agents rather than a single one. This kind of approach has recently attracted much attention; a lot of work has been conducted to evolve robot controllers and their preliminary results have shown the promise of this approach (e.g.,[3][6][12]). Yet, the tasks achieved so far, such as obstacle avoidance or light seeking, are relatively simple. To help the human designer to develop control systems which are difficult to handcode, we have to resolve the problem of how to scale up the evolution-based approach in controlling robots. In this work we discuss, from different points of view, some of the difficulties one will encounter in evolving controllers to accomplish complex tasks. We also suggest that task decomposition is an efficient technique in supporting the evolutionary approach to evolve controllers for complex tasks. To demonstrate this, we undertake an application task, in which a robot is required to push a box toward a specific position indicating by a light source. This task is fairly complex compared to the ones achieved in previous Evolutionary Robotics experiments and the results show that the robot can achieve the specified task reliably.

2 Evolving Controllers to Achieve Complex Tasks

2.1 The Difficulties

Generally speaking, the evolutionary approach is a kind of search-based approach in which genetic operators, such as reproduction, crossover, and mutation, are used and expected to find a satisfactory solution from a hyper-space; and the dimension of this space is determined by the length of the chromosome. For instance, in a binary encoding scheme, a chromosome with length n indicates that the evolution techniques are expecting to find the appropriate solution from a space with 2^n candidates. Thus, when the length of the chromosome is reasonably increased in respect to the increment of task complexity, the solution space will grow exponentially and lead the search to be more and more difficult. This is particularly apparent in the work that uses recurrent neural networks as control systems since the characteristic of recurrence leads the length of chromosome to increase quadratically and thus enlarges the search space even faster.

The increasing task complexity also introduces difficulty in defining fitness functions to guide the search direction during the evolution. In general, the increment of task complexity implies a higher-level goal to achieve, which almost always involves the interaction of multiple subgoals. For a complex task, directly defining a fitness function at the higher-level is relatively simple but it makes the task difficult to be achieved. On the contrary, defining fitness functions at lower-level is more difficult while it makes the task more achievable. For example, in the work [13], the authors have shown that in their grasping task, if the fitness function is simply defined as the number of objects grasped and deposited correctly, then the desirable behaviour cannot be evolved successfully. This is due to the fact that during the earlier generations none of the individuals can achieve the complete task; it results in the equally bad fitness for all the populations (all scored zero) and makes all the control systems indistinguishable in performance. On the other hand, if lower-level subgoals are introduced to the fitness function, such as rewarding the behaviours of recognizing objects and picking objects up, the performance of controllers becomes more distinguishable and then the target is achieved. Manipulating fitness at lower levels can assist the evolutionary system to converge; however defining an appropriate fitness function at a lower level is never easy because it has to deal with the multiple subgoals simultaneously. Further, this kind of difficulty occurs in consequence of the increment of the task complexity.

On the other hand, from the point of view of controlling a robot, one may want the evolved control systems to be distributed for their corresponding advantages. In a distributed architecture, the perceptual processing is distributed across multiple independent modules, and every module only deals with the sensory information directly related to its particular need. This not only reduces the sensory bottleneck but also allows each control module to be developed with the most suitable representation and approach with least restriction. Owing to the modular and distributed characteristics, the performance of the overall system will degrade gradually, even if some of the devices or control strategies do not function properly. Further, with an explicitly distributed architecture, an overall system will be easily integrated from different subsystems which could be designed independently; it can also be easily maintained. Therefore, from the point of view of developing systems for robot control, distributed control architectures are preferred.

2.2 Task Decomposition

In order to reduce the search space to make the search easier, to simplify the job of defining fitness functions, and to obtain a distributed control system, a promising way is to adopt the divide-and-conquer problem-solving methodology. In this kind of approach, the designers break tasks from complex (higher-level) down to simple (lower-level) and then achieve the tasks in the reverse sequence. How to decompose a task of course depends on the designers' experiences, but human designers are normally quite skillful in doing it. The tasks are arranged to be achieved in the sequence of increasing complexity and, at each level, the control systems are evolved on top of the ones evolved at lower-levels. Hence, fitness functions will become easier to define and the tasks will be easier to achieve (the fitness function of a certain level task can be defined simply as the goal at this level, to reduce the difficulty in embedding the lower-level subgoals into it; and evolving control systems on top of other lower-level controllers can exploit their corresponding control skills to achieve the current goal). In addition, each subtask only needs to deal with the perceptual information directly related to it, which also makes the tasks easier to achieve. In the robot learning domain, some work has shown that the decomposition technique helps in achieving more complex tasks [4][11].

Actually, the concept of this kind of approach is much like behaviour-based control, which has been successfully and widely used in the robot community for building autonomous robots (e.g.,[14][17]); while the main difference is that the approach here employs evolutionary techniques to *evolve* new behaviours and behaviour coordinators rather than to *handcode* them. With evolutionary techniques, the human designer can concentrate on the system level design and let the evolutionary system take care of the implementation details. In addition, since in this approach the tasks are decomposed along the horizontal way as proposed in [2], the corresponding control architectures will be explicitly distributed and can fully exploit all the advantages of distributed architecture as analysed in the section above.

We are particularly interested in investigating ways to reduce the load of robot programmers and in evolving distributed architectures for complex tasks. Because, at the present stage, the task-decomposition technique seems to be the most direct way to achieve these, we will concentrate on investigating how to use this approach, with our genetic programming (GP) system [8][9], to evolve control modules and coordinators to achieve complex tasks.

3 Evolving Hierarchical Task-achieving Controllers

As described above, in order to evolve distributed control systems to achieve complex tasks, we intend to use the technique of task decomposition to break the overall tasks and use the GP techniques to evolve separate behaviour controllers and coordinators for integration. In this section, we will explain the aspect of the control architecture corresponding to the task decomposition, and then describe the genetic representation of the control module to be evolved.

3.1 Control Architecture

Since we will decompose tasks in a hierarchical way, the corresponding control system is organized as multiple layers. After the decomposition, the control system includes a set of *behaviour primitives* and *behaviour arbitrators*. Here, a behaviour primitive is a reactive controller with the representation described in the below section; it involves the lowest level sensory-motor control. Unlike the priority network in the subsumption architecture [2], a behaviour arbitrator here is not hardwired in advance; it is also treated as a reactive controller. The behaviour arbitrator has the same structure and representation as the primitives; the only difference between them is that the output of a primitive is used to control the motors while the output of an arbitrator is used to activate one of the involved sub-controllers. Thus, similar to a reactive planner [5] or a conditional sequencer [7], an arbitrator here allows the binding between environment conditions and activations of lower level behaviours to take place at the run time. This provides adaptiveness not only at the lower level sensory-motor control but also at the behaviour level.

Because our system does not support parallel computation, all the control modules are passive and the control flow is thus from top to down. At each time step, the highest level arbitrator evokes one of the involved sub-controllers to be in charge the control, according to certain sensory information. If the evoked sub-controller includes an arbitrator, this arbitrator will be evaluated first and its output can then be used to activate another controller. This process continues until a control primitive at the lowest level takes control and drives the actuators. Figure 1 illustrates the general architecture of our control systems and the implementation of an arbitrator.

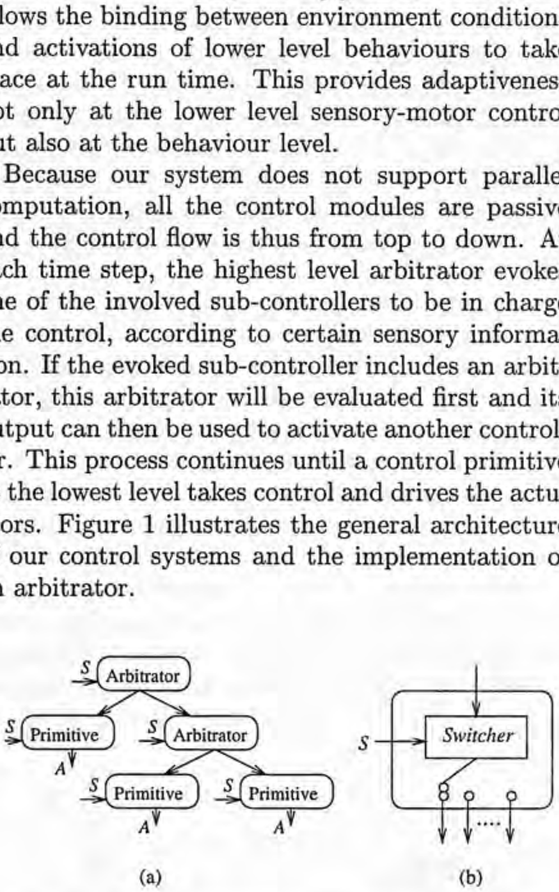

Figure 1: (a) The general architecture of a control system. S and A represent the sensors and actuators related to a certain control work; (b) the implementation of an arbitrator.

3.2 Representation

When using evolutionary computation techniques to solve a problem, the first important step is to choose the proper representation for an individual. On one hand, a genetic representation must be able to express explicitly the features of the solution of the problem to be solved; on the other hand, it must be suitable to be manipulated by the genetic operators to obtain the solution. The following sections are about how we develop the genetic representation of the reactive controllers to be evolved in this work.

3.2.1 The Circuit Model of a Behaviour Controller

A promising choice to satisfy the above requirements is the *circuit network* which has been proven to provide a finer-grained view to represent a behaviour controller. In the *circuit approaches* [1][15][16], an agent (behaviour controller) exists in the form of digital hardware; and it is constructed by two types of components, pure functions and the delays, depending on what kind of tasks (reactive or sequential) it is achieving. Pure functions are logic gates, and delays correspond to flip-flops or registers. The output of one component may be input to one or more other components, thus forming a network. Signals propagate through the network and sensing is thus linked to action. As is well known, any finite state transduction can be carried out by such a network.

Since this work focuses on evolving reactive controllers, we will only discuss how to evolve controllers of this kind. The approach can be extended to evolve sequential ones with minor modification.

The genetic representation of our reactive controller is inspired by the logic representation in the circuit approaches. By duplicating and separating the components of which the outputs serve as inputs of multiple components and by introducing a dummy root node to connect the outputs of a circuit network together, we find it very straightforward to convert a circuit network to a circuit tree. Figure 2 shows an example. After structuring information from the environment and defining simple syntactic rules properly, we can use a GP system to evolve the circuit trees.

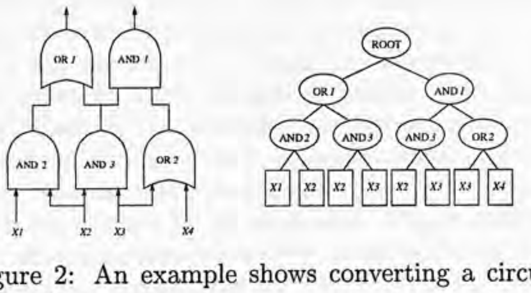

Figure 2: An example shows converting a circuit network to a tree.

3.2.2 Genetic Representation of Our Reactive Controllers

According to our design, the perception information is structured as *sensory conditionals* and connected to the inputs of a logic circuit. The structured sensor-conditionals involve comparing the responses of different sensors or comparing sensor response to numerical thresholds. For these purposes, both sensor responses and numerical thresholds are normalised to be between 0 and 1 inclusive. Thus, a sensor conditional has a constrained syntactic structure; it exists in the form of X >= Y, where X, Y can be any normalised sensor response or threshold which is determined genetically. Figure 3 shows the representation of our reactive controllers.

After organizing our genetic representation, we can classify the involved symbols into *terminals* and *non-terminals* for manipulating a GP system. The dummy root node, the logic components, and the comparator are defined as the non-terminals; while the normalized sensor responses and numerical thresholds, which constitute the sensory conditionals, are defined as terminals. In order to evolve the controllers with the above representation to solve different control tasks, we will need to define different sensor terminals depending on the requirements of the specific task. The experiment sections of this work will give the details of how to evolve such kind of controllers.

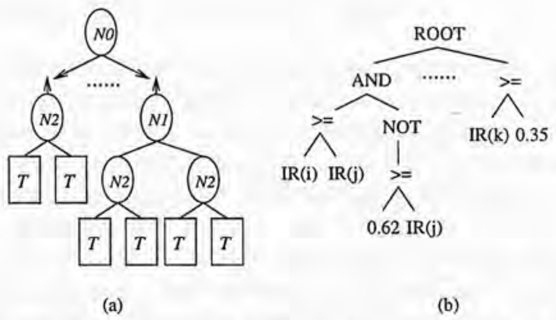

Figure 3: The aspect of a typical controller. In this figure, *N0* is a dummy root node, *N1* represent logic components, and *N2* is the comparator >=. *T* can be a normalized sensor response or a threshold between 0 and 1 inclusive. The outputs of the subtrees are used to drive the actuators or activate another control system.

4 Experimental Setup

4.1 Application Task

In the following experiments, we will follow the approach described above to evolve control systems for a moderately difficult box-pushing task. In this task, the robot is required to explore the given arena in order to find a box; once it detects the box, it then has to push the box toward a goal position indicated by a light source.

The task to be achieved is difficult for the following reasons. First of all, the robot is round, so that it only contacts the box at one point while pushing it, and the box tends to slide and rotate unpredictably when the pushing force exerted by the robot is not directed straight through the center of the box. Therefore, the robot has to adjust its own position occasionally in order to push the box forward as straight as possible. Furthermore, as there is no particular restriction on the initial relative positions of the robot, the box, and the ambient light, the robot can approach and detect the box at any position and orientation around the box; under such circumstance, the robot needs to move to a proper position deliberately in order to perform an efficient push to satisfy the final goal.

4.2 The Decomposition

To accomplish this task, we decompose it into two subtasks, *exploration* and *push-box-toward-light.* The former is to control the robot to explore the given arena in order to find the box without bumping into any wall; and the latter, to push the found box to a specific goal position. Again, the task *push-box-toward-light* is decomposed into two lower-level subtasks, *box-pushing* and *box-side-circling.* The goal of *box-pushing* is to keep the robot pushing a box forward, while the goal of *box-side-circling* is to keep the robot moving along the side of a box in order to provide the opportunity for the robot to move to suitable positions for pushing. Each of the subtasks, without being decomposed, is controlled by a separate behaviour primitive, and the different sub-controllers for the same task are merged by an arbitrator. Figure 4 shows the decomposition results and the aspect of the corresponding architecture for the target task. After the decomposition, the GP system is used to evolve both behaviour primitives and arbitrators.

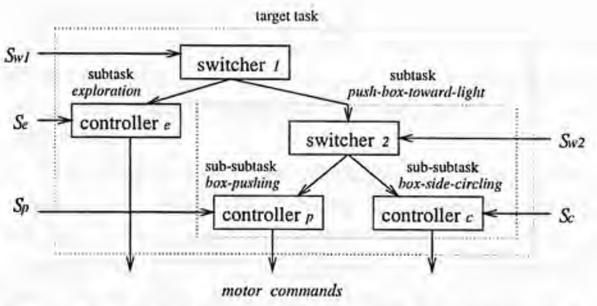

Figure 4: The decomposition and integration of the target task. S_i indicates the sensory information relevant to control work i.

4.3 The Hardware Limitation

In the experiments below, we will use the method of evolving controllers in simulation and then testing them on a real Khepera robot. The simulator is built by employing a look-up table approach which has been shown to provide a close match between simulation and reality [12].

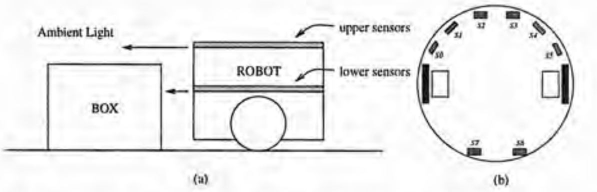

Figure 5: (a) A possible way to satisfy the sensor requirements for both arbitrators is to duplicate a set of eight sensors on the top of the robot; (b) the sensor arrangement – a sensor S_i can function as an infra-red or an ambient light sensor.

To accomplish the task *push-box-toward-light*, the arbitrator needs infra-red sensors and ambient light sensors to detect the box and the light respectively; and the ambient light sensors must be higher than the box to make sure they can detect the light even in the situation that the box is between the robot and the light. On the other hand, for the overall task, the other arbitrator will require certain perception information which can be organized as some sensory conditionals for the robot to recognize the box, to determine when to perform *exploration* and when to perform *push-box-toward-light.* For this purpose, we define a kind of virtual sensor *DR*, which can give the normalized reading difference between a pair of upper and lower infra-red sensors (The sensor pair here means two sensors pointing at the same direction but with different heights: one is higher and the other is lower than the box). A straightforward way to satisfy the requirements for both arbitrators is to duplicate the eight sensors of Khepera on its top (the sensors on the top are assumed to be higher than the box) so that the duplicated sensors can serve as ambient light sensors for the first arbitrator and as infra-red sensors for the second arbitrator (see Figure 5). However, the preliminary tests show that when the robot is within a certain area around the bulb, the infra-red sensors on the Khepera robot are seriously disturbed by the normal bulb light and thus cannot function properly. This will cause difficulty in verifying the simulation results on the real robot. Therefore, for the behaviour primitives which involved IRs only, we evolve them in simulation and test them on the real robot; but the two arbitrators are only evolved in simulation in which we assume that there are eight extra sensors on the top of the simulated robot as described above.

5 Experiments and Results

5.1 Evolving Primitives for Task *Box Pushing*

As mentioned above, the task of *box-pushing* is that the robot should keep pushing a box forward as straight as possible. To achieve such a task, the robot needs to use its IR sensors to acquire perception cues for the location of the box. Therefore, we define two kinds of terminals, IRs and numerical thresholds, for our GP system to evolve controllers capable of achieving this task.

The fitness functions in this work are defined as penalty functions. For this task, the fitness function is formulated through the quantitative description of the expected behaviour, which is to keep the activation value of its front IR sensor high, the robot moving forward, and the speed difference of two motors low. The pressure from keeping the front IR sensor with high activation value is to reinforce the robot to head toward a box, and the pressure from keeping robot moving forward with low speed difference is to encourage the robot move straight and to prevent it from getting stuck in front of a box. The combination of these will lead to a pushing-forward behaviour. Thus, the fitness function for evolving a behaviour controller of box-pushing can be defined as:

$$f = \sum_{t=1}^{T} \alpha * (1 - s(t)) + \beta * (1 - v(t)) + \gamma * w(t)$$

in which $s(t)$ is the average of normalized sensor activations of the front sensors IR*2* and IR*3*; $v(t)$ is the normalized forward speed; and $w(t)$ is the normalized speed difference of two motors at each time step t. During the evolution, each controller was evaluated in multiple trials and the average fitness value was used to measure the performance of this controller.

The typical box-pushing behaviour of the simulated robot, when performing the evolved controller, is illustrated in Figure 6(a). After evolution in simulation, this was transferred to a Khepera robot. Figure 6(b) shows the typical behaviour of the real robot. The figure for real robot was obtained by setting LEDs on the tops of the robot and box and using a video tracking system to record their trajectories [10]. This controller was tested on the real robot many times and each time it started from an arbitrary position and heading around the box. During the tests, the robot always generated consistent behaviour: it turned to face the box and then to approach and push it.

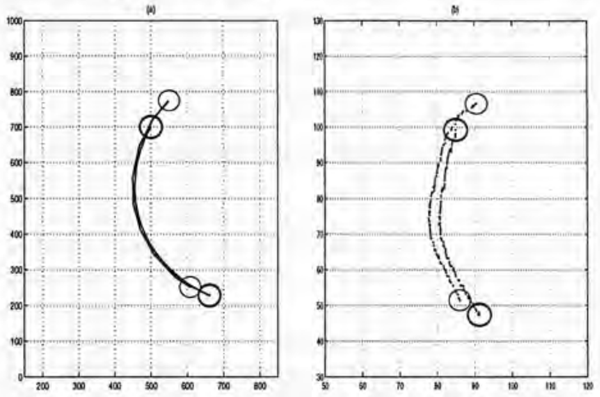

Figure 6: The trajectories of simulated (left) and real (right) robots when they are pushing a box (the darker circles represent the boxes; and the boxes are pushed from top to down).

5.2 Evolving Primitives for Task *Box-Side-Circling*

The task of *Box-side-circling* is that the robot needs to keep moving forward and circling along the sides of a box. As in the box-pushing task, the robot should use its own IRs to capture the location of the box. Thus, terminals for evolving a controller to achieve this task are the same as those in the box-pushing task: IRs and numerical thresholds.

Again, we should define a fitness function to guide the evolution, and it can be done through the quantitative description of the expected behaviour: to keep the side sensor IR*0* with a certain activation value and the speed positive. The former is to encourage the robot to keep a certain heading relative to the box and a certain distance away from the box; and the latter is to reinforce the robot moving forward. The combination of these will produce a box-side-circling behaviour. Thus the fitness function is defined as:

$$f = \sum_{t=1}^{T} \alpha * abs(s(t) - k) + \beta * (1 - v(t))$$

where *abs* is a function which gives the absolute value of it argument; $s(t)$ is a normalized activation value of the specific sensor IR*0*; k is a pre-defined constant indicating the distance between the robot and the box, in terms of the normalized sensor range; and v is the forward speed of a robot.

Figure 7(a) presents the evolved box-side-circling behaviour of the simulated robot, which shows that the task was achieved successfully in simulation. The evolved controller was then transferred to the real robot, and the typical behaviour of the real robot is demonstrated in Figure 7(b). We tested this evolved controller several times by putting the real robot around the box with an arbitrary heading each time. In all the tests, the robot showed similar behaviour: it performed turning to adjust its heading first and then moved along the side of the box. From the testing results, we can see that the robot is able to achieve the specified task reliably.

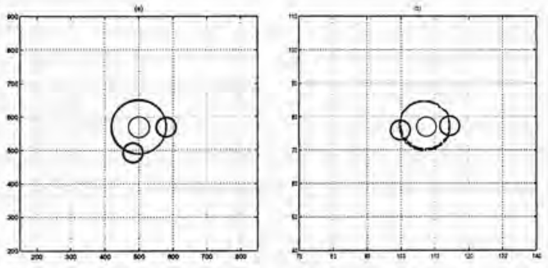

Figure 7: The box-side-circling behaviours of simulated (left) and real (right) robots.

5.3 Evolving Primitives for Task *Exploration*

This task is that the robot needs to wander safely in an enclosure and visit as much of the enclosed space as possible. It can be described quantitatively as the space being divided into some squares and the robot must visit as many squares as possible during a fixed period of time. In the experiment below, we intend to evolve a reactive controller to explore the space without using the location information.

Unlike the experiments presented above, the fitness measurement for this task is not to sum up the penalty of each time step but to give a fitness value after a complete trial. The main concern for the fitness here is to minimize the number of squares which have not been visited, while an extra pressure on the speed is added to encourage the robot to move forward in exploring. Thus, the fitness function is defined as:

$$f = \alpha * (1 - P) + \beta * (1 - Avg)$$

in which P is the percentage of the space visited, i.e., $\frac{visited-squares}{total-squares}$; Avg is the average speed of the robot during a complete trial; and α, β are the corresponding weights. The enclosure in the exploration experiment here is a square of 50 x 50 cm and each grid square is 5 x 5 cm.

As described above, the controller to be evolved is reactive and there is no location information provided here, so the robot must fully exploit its IR sensors to determine the turning angle carefully to achieve this task. Since IR sensors are the only mechanism for providing perception cues, the terminals for the exploration task are then defined to include IRs and numerical thresholds as in other tasks.

The exploration behaviour produced by the evolved controller is demonstrated in Figure 8(a), which shows that the robot is able to visit most of the specific arena during a fixed period of time. We should note that it is not important how the robot moves when it does not sense anything but the appropriate match of the turning angle (when the robot senses the wall) and the way it moves (when it does not sense anything) is nevertheless crucial for a reactive controller to perform exploration. As we can see in Figure 8(a), a successful match has been evolved and it enabled the robot to achieve the task.

Like the previous experiments, the evolved controller is downloaded to the real Khepera after the simulation. The behaviour observed from the real robot is presented in Figure 8(b). This behaviour is very similar to the one produced by the simulated robot in the simulated environment. Once again, it shows a successful example that we are able to evolve a behaviour controller by our GP system in simulation, and then transfer it to a real robot.

5.4 Evolving Arbitrators for Task *push-box-toward-light*

As mentioned above, an arbitrator is implemented as a reactive controller; its inputs are from the

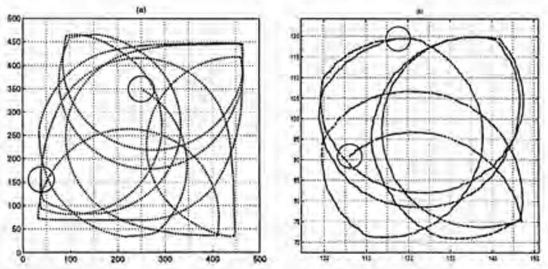
Figure 8: The exploration behaviours of simulated (left) and real (right) robots.

sensors and its outputs are used to trigger other controllers. For the arbitrator here, two kinds of sensors – infra-red and ambient light – are needed to detect the locations of the box and the light, so both kinds of sensors and numerical thresholds are defined as terminals to the GP system to constitute the structured sensory conditionals for the arbitrator. Since there are only two sub-controllers involved, the arbitrator is designated to have a single output to trigger them: if the output is 0, then the controller for subtask *box-pushing* dominates the control; otherwise the controller for subtask *box-side-circling* has the control. During the experiment, two subcontrollers are frozen and only the arbitrator is evolved.

In this task, the robot is expected to push the box as close as possible to the center of the area brightened by the light. Instead of measuring the distance between the goal position and the final position of the box at the end of a complete trial, we calculate the summation of the distance between the goal position and the box at each time step to reinforce the robot to push the box straight toward the light. Thus the fitness function is defined as:

$$f = \sum_{t=1}^{T} D_{b,l}(t)$$

in which $D_{b,l}(t)$ represents the distance between the box and the light source at each time step t.

Figure 9 illustrates, step by step, the typical behaviour of the simulated robot. As can be seen, the arbitrator first activated the primitive *box-side-circling* to move the robot along the side of a box. Once the robot reached an appropriate position in which the box was between the light and the robot itself, the control was then switched to the other primitive, *box-pushing*, to drive the robot to push the box forward. The *box-side-circling* and the *box-pushing* primitives were activated again in the same order if the box was not pushed directly toward the goal position. After the box was pushed to the goal position, the arbitrator continuously activated the primitive *box-side-circling* to circle the box in order to prevent pushing the box away from the box. From Figure 9, we can see that the box was successfully pushed to almost the center of the bright area.

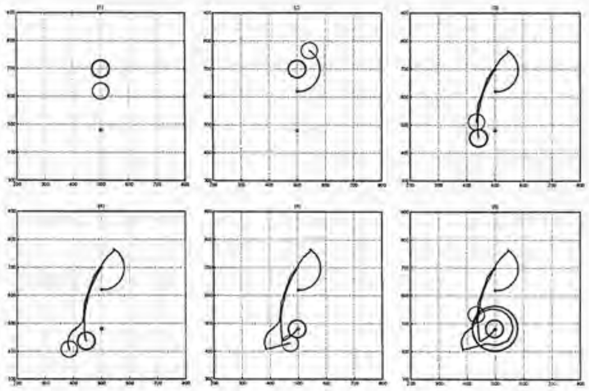
Figure 9: The behaviour sequence of the robot: (1) The initial positions of the box (dark circle), the light (smallest circle) and the robot; (2) the robot moved along the side of the box; (3) pushing the box forward; (4) circling again to an appropriate position; (5) pushing the box again to the goal position; (6) continuously circling after the box has been pushed to the goal position.

We can also examine whether the performed task decomposition has been exploited in achieving the higher-level goal by observing the output sequence of the arbitrator: if the sequence is separated as periods of consistent activation, then the performed decomposition is confirmed to be helpful; otherwise it is not. Figure 10 demonstrates the output sequence corresponding to the behaviour shown in Figure 9. According to this figure, the evolved arbitrator is able to generate periods of quite consistent activation, except the short oscillating period – the transition period between two different behaviours – which did not effect the global behaviour at all. This figure, in fact, indicates that the involved lower-level sub-controllers have been fully exploited.

5.5 Evolving Arbitrators for the Overall Task

After evolving an arbitrator to combine two pre-evolved lower-level primitives, we can regard the integrated control system, including one arbitrator and two primitives, as a building block, and then evolve a new arbitrator to combine this building block and the other evolved controller *exploration* to achieve the overall task. As described in the

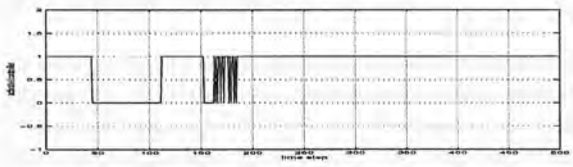

Figure 10: The output sequence corresponding to the behaviour in Figure 9. In this figure, the y-axis indicates the controller which was activated: 0 is for *box-pushing* and 1 is for *box-side-circling*.

above section, the arbitrator will need the perceptual information which can be used to recognize the appearance of the box, in order to generate proper output sequence to coordinate two involved control systems. Therefore, the virtual sensor *DRs* and numerical thresholds are defined as terminals for our GP system to evolve the desired arbitrator. As above, this arbitrator is designated to have one output: if the output is 0, the controller for *exploration* is activated; otherwise is the controller for *push-box-toward-light*. Again, the controllers to be combined are frozen and only the arbitrator is evolved.

The goal here is the same as the one above, to push the box as close as possible to the specified position, so the fitness function can be defined as above: to accumulate the distance between the box and the goal position at each time step. However, the criterion of simply measuring the fitness function for a fixed period time as before cannot give an objective evaluation here. This is because, in different trials, the robot could start from different positions and then take different numbers of time steps to find the box – it means that the lengths of the time of pushing are different. Therefore, in this experiment, the robot is given an extra period of time to find the box; and the fitness value is accumulated for a fixed period of time which starts from the moment the robot finds the box (or the end of the time period given for looking for the box). Thus, the fitness function is defined as:

$$f = \sum_{t=k+1}^{k+T} D_{b,l}(t)$$

in which $D_{b,l}(t)$ is the distance between the box and the goal position at time t; k is the time when the robot finds the box (or the end of the time period given for looking for the box); and T is the fixed period of time for fitness measurement.

The typical behaviour of the robot, when performing the whole control system, is shown in Figure 11. From these figures, we can see that the arbitrator firstly kept activating the controller *exploration* to drive the robot to explore the given environment and to avoid the walls. Once the robot found the box, the arbitrator began to activate the other controller, *push-box-toward-light*, according to the sensor stimuli. Since the arbitrator was able to activate this controller continuously after the robot found the box, the overall task was then achieved successfully. In fact, the performance of the arbitrator above can be observed from Figure 12, which shows the output sequence generated by this arbitrator during the test. It clearly indicates that this arbitrator can keep activating the controller *exploration* before the box was found, and afterward it can activate the other controller *push-box-to-light* consistently to achieve the target task.

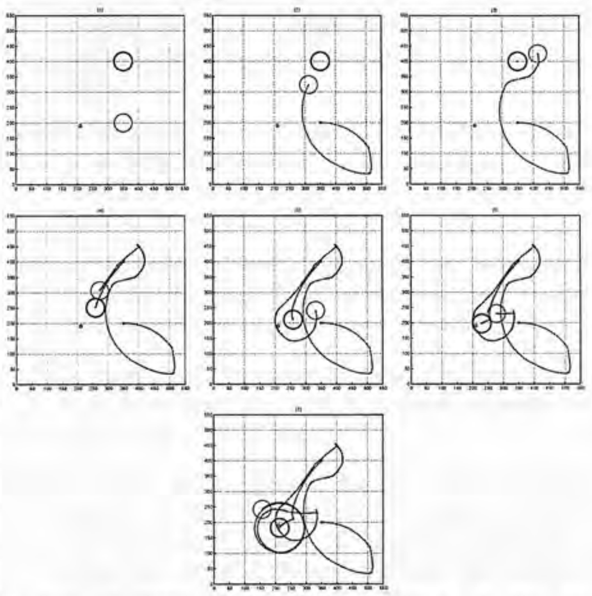

Figure 11: The behaviour sequence of the robot: (1) the initial conditions; (2) the robot wandered around the environment before found the box; (3)~(7) the robot continuously performed the building block controller *push-box-toward-light* to achieve the task.

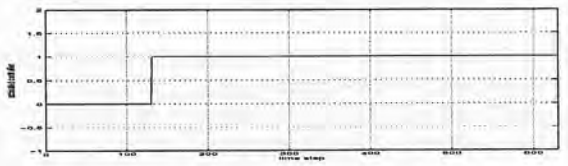

Figure 12: The output sequence corresponding to the behaviour in Figure 11. In this figure, the y-axis indicates the controller which was activated: 0 is for *exploration* and 1 is for *push-box-toward-light*.

6 Conclusion

In this work, we have analysed the difficulties of applying an evolutionary approach to learn complex behaviours for mobile robots. Instead of evolving the control system as a whole, we propose to take the control architecture of a behavior-based system and to learn the separate behaviours and arbitration by the use of an evolutionary approach. To assess the performance of the developed approach, we have evolved a control system to achieve an application task of box-pushing as an example. Experimental results show the promise and efficiency of the presented approach.

The proposed approach has some advantages. First of all, by using the technique of task decomposition, the job of defining fitness function becomes more straightforward and the tasks become easier to achieve. In fact, for all the evolution experiments, the GP system converged to stable and sufficient solutions within only 30 generations. In addition to that, our controllers only involve logical operators, such as AND OR NOT, that are very simple to evaluate. This means that our approach is computationally cheap; and the evolved controllers can be easily compiled to custom hardware such as FPGAs to speed up the evaluation in controlling a robot.

Some further research work is currently in progress. For example, we have been constructing new sensors on the real robot to test the behavior arbitrator. We have also been investigating whether this approach can be applied to evolve control systems for even more complicated tasks.

References

[1] P. Agre, D. Chapman. Pengi: an Implementation of a Theory of Activity. In *Proceedings of AAAI-87*, pp.268-272. Morgan Kaufmann, 1987.

[2] R. A. Brooks. A Robust Layered Control System for a Mobile Robot. In *IEEE Journal of Robots and Automation*, vol RA-2(1), pp.14-23, 1986.

[3] D. Cliff, I. Harvey, and P. Husbands. Explorations in Evolutionary Robotics. *Adaptive Behavior*, 2(1):73-110, 1993.

[4] M. Dorigo, M. Colombetti. Robot Shaping: Developing Autonomous Agents through Learning. *Artificial Intelligence*, 71(2):321-370, 1994.

[5] R. J. Firby. Task Networks for Controlling Continuous Processes. In *Proceedings of the Second International Conference on AI Planning Systems*, pp.49-54, 1994.

[6] D. Floreano and F. Mondada. Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural-Network Driven Robot. In *From Animals to Animats: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pp.421-430. MIT Press/Bradford Books, 1994.

[7] E. Gat. Robot Navigation by Conditional Sequencing. In *Proceedings of IEEE International Conference on Robitics and Automation*, pp.1293-1299, 1994.

[8] W.-P. Lee, J. Hallam, H. H. Lund. A Hybrid GP/GA Approach for Co-evolving Controllers and Robot Bodies to Achieve Fitness-Specified Tasks. In *Proceedings of IEEE International Conference on Evolutionary Computation*, 1996.

[9] W.-P. Lee, J. Hallam, H. H. Lund. Applying GP to Evolve Behaviour Primitives and Arbitrators for Mobile Robots. In *Proceedings of IEEE International Conference on Evolutionary Computation*, 1997.

[10] H. H. Lund, E. V. Cuenca, J. Hallam. A Simple Real Time Mobile Robot Tracking System. Technical Paper No.41, Department of Artificial Intelligence, University of Edinburgh, 1996.

[11] M. Mataric. Reward Functions for Accelerated Learning. In *Proceedings of International Conference on Machine Learning*, pp.181-189, 1994.

[12] O. Miglino, H. H. Lund, S. Nolfi. Evolving Mobile Robots in Simulated and Real Environments. *Artificial Life*, 2(4), 1996.

[13] S. Nolfi, and D. Parisi. Evolving Non-trivial Behaviors on Real Robots: An Autonomous Robot that Picks up Objects. In *Proceedings of the Fourth Congress of the Italian Association for Artificial Intelligence*. Spring-Verlag, 1995.

[14] R. Pfeifer, and C. Scheier. Sensory-Motor Coordination: the metaphor and beyond. In *Robotics and Autonomous Systems, special issue in Practice and Future of Autonomous Agents*, 1996.

[15] S. J. Rosenschein, L. P. Kaelbling. A Situated View of Representation and Control. In *Artificial Intelligence*, vol 73, pp.149-174, 1995.

[16] S. J. Rosenschein, L. P. Kaelbling. The Synthesis of Digital Machines with Provable Epistemic Properties. In *Proceedings of Conference on Theoretical Aspects of Reasoning about Knowledge*, pp.83-98. Morgan Kaufmann, 1986.

[17] L. Steels. Building Agents out of Autonomous Behavior Systems. In L. Steels and R. Brooks (eds), *The Artificial Life Route to Artificial Intelligence*. Lawrence Erlbaum Associates, 1993.