Implementation of a Behaviour Based System for the Control of Mobile Robots

Daniel Venkitachalam

Abstract

The traditional approach of intelligent controller design for robots has been to follow a sense-plan-act organisation. Data is processed by a planning layer that produces output signals to drive the actuators of the system. The behaviour based robotics paradigm uses a different organisation, dividing the control task into smaller independent modules that perform their own processing and determine a local set of desired actions. Coordination between these modules is required for the robot to achieve an overall task.

Behaviour based robotics addresses some of the key issues that have limited the success of purely deliberative controllers. The complexity of the logic required by each simple behavioural module is considerably easier to understand than a large monolithic program. Consequently, the computational requirements of a purely behavioural system are relatively small, increasing the performance of the robot.

Unfortunately, the design of behaviour based systems is complicated by the difficulty of analysing the interactions between concurrently running behaviours. For large tasks consisting of hundreds of concurrent behaviours, the complexity may be greater than can be feasibly designed by a human. An alternative to manual design is to use an adaptive controller that learns a correct response to inputs from specified fitness criteria. Instead of completely specifying a task controller, we define desired attributes of the system and let the system develop a way to maximise them.

In this project we have designed a system for execution of behaviours on the Eyebot. We implemented reactive behaviours to achieve a ball finding task and coupled their activations to a neural network controller. The controller was evolved over several iterations with a genetic algorithm in an attempt to create a ball finding program from specification of the task criteria only.

Acknowledgements

Thomas Bräunl was a wonderful project supervisor for the whole year. He let me make my own mistakes but was always available to correct them, and I am thankful to him for these reasons, but also for providing his students a surrogate home in the robot lab.

Adrian Boeing was a fellow who also used the robot lab quite a lot. There he thought of some good ideas and lousy jokes which encouraged others to do the same, so I must thank him too.

Estelle Winterflood was a girl who spent many nights in the lab. She illuminated the lab when it was dark outside yet found time to care for a homeless little puppy and so the dog and I are both grateful.

Jason Foo was another lab rat who was often grouchy when someone needed to be so he will be acknowledged and thanked sincerely this one time only.

Carly Taylor saved my life a long time ago and for that I will always be grateful.

Finally, my family have looked after me and still do so I thank them as a whole. I will single out my father for teaching me things since I was young and now look what happened.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Control of mobile robots produces tangible actions from perceptual inputs. A controller for a robot receives input from its sensors, processes the data using relevant logic and sends appropriate signals to the actuators to perform actions. For most large tasks, the ideal mapping from input to action is not clearly specified nor readily apparent. Such tasks require a control program that must be carefully designed and tested in the robot's operational environment. The creation of these control programs is an ongoing concern in robotics as the range of viable application domains expand, increasing the complexity of tasks expected of autonomous robots.

Traditional attempts at formulating complex control programs have been rooted in artificial intelligence theory. The dominant paradigm of this approach has been the sense-plan-act (SPA) organisation: a mapping from perception, through construction of an internal world model, planning a course of action based upon this model and finally execution of the plan in the real-world environment. Aspects of this method of robot control have been criticized, notably the emphasis placed on construction of a world model and planning actions based on this model [1, 10]. The computational time required to construct a symbolic model is significant, impacting the performance of the robot. Furthermore, disparity between the planning model and the actual environment may result in actions of the robot not producing the intended effect.

An alternative to this approach is described by behaviour-based robotics. Reactive systems that do not use symbolic representation of are demonstrably capable of producing reasonably complex behaviour [9]. Behaviour-based robotic schemes extended the concept of simple reactive systems to combining simple concurrent behaviours working together.

An overall controller may be employed to coordinate activations of these behaviours at appropriate times to achieve a desired objective. Unfortunately, for non-trivial tasks the complexity of interactions between behaviours makes the manual design of such controllers difficult. This complexity further increases with tasks of increasing scope and larger numbers of interacting tasks.

These shortcomings in scalability may be offset by combining the simple units of a purely reactive behaviour system with an adaptive controller. Such a controller uses machine learning techniques to develop the correct selection response from the specification of desired outcomes. The controller is the "intelligence" behind the system, deciding from sensory and state input which behaviours to activate at any particular time. The combination of a reactive and planning (adaptive controller) component produces a hybrid system.

Hybrid systems combine elements of deliberative and reactive architectures. Various hybrid schemes have been employed to mediate between sensors and motor outputs to achieve a task. Perhaps the most appealing aspect of combining an adaptive controller with a hybrid architecture is that the system learns to perform the task from only the definition of criteria favouring task completion. This shifts the design process from specifying the system itself to defining outcomes of a working system. Assuming that the criteria for successful task completion is easier to specify than a complete system specification, this would significantly reduce the work required of the system designer.

The learning component of adaptive controllers may in turn be classified as online or offline, based on whether the system evolves *in situ* or outside of execution environment respectively. There are potential drawbacks to the online learning approach that could be addressed by offline learning. These include:

- Generation of ideal behaviour - there is the possibility of the system adapting to a state that fulfills some but not all of the task's fitness criteria. This typically happens when the method of learning relies on gradient descent and becomes stuck in a local fitness maxima [38]. Offline evolution allows the application of more complex (and hence processor intensive) optimization techniques to avoid this situation. The computational requirements of these may more than

- Time to convergence - the time a robot takes to converge to an appropriate controller state reduces the robot's effective working time. By evolving suitable parameters offline, the robot is in a suitable working state at runtime.

- Testing of system - evolution of behaviour in a real environment limits our ability to test the controller's suitability for a task. Offline evolution enables extensive testing in of the system in simulation before actual use.

- Physical damage to system - while the controller is evolving, its response may cause damage to the physical robot until it learns to perform a task safely. Evolving the controller in simulation allows such responses to be modified before real harm can be done to expensive hardware.

The focus of this project has been to produce a hybrid behavioural framework combining a low-level reactive system and a higher offline learning layer. Above a behavioural framework, we constructed an evolutionary artificial neural network controller to arbitrate between individual higher-level behaviours. The neural net controller structure is evaluated within a simulated environment and incrementally evolved offline by the genetic algorithm until a suitable controller for the specified task is produced. After several iterations an optimal controller configuration to maximise the fitness criteria and subsequently fulfill our set task is attained.

1.2 Project outline

The project required the creation of four components:

1. Behavioural framework
2. Simulator for execution of controlled trials
3. Neural net controller
4. Genetic algorithm class libraries

The behavioural framework is the lowest software level of the system; it enables the programming and execution of concurrently-modelled primitive behaviours on the robot platform. Above the framework is the simulator level where trials of the neural net controller are run. Results of each timed trial are evaluated and recorded by a fitness function. Over multiple iterations of the simulator, the parameters describing the controller configuration are evolved by a genetic algorithm according to their fitness values. The complete cyclic process is illustrated in Figure 1.1.

Using a genetic algorithm to evolve a neural net configuration, we incrementally developed a suitable controller for a ball searching task inside an accurately simulated environment. The resulting controller may then be immediately deployed onto real hardware in a known working state, without requiring further online training to execute the task.

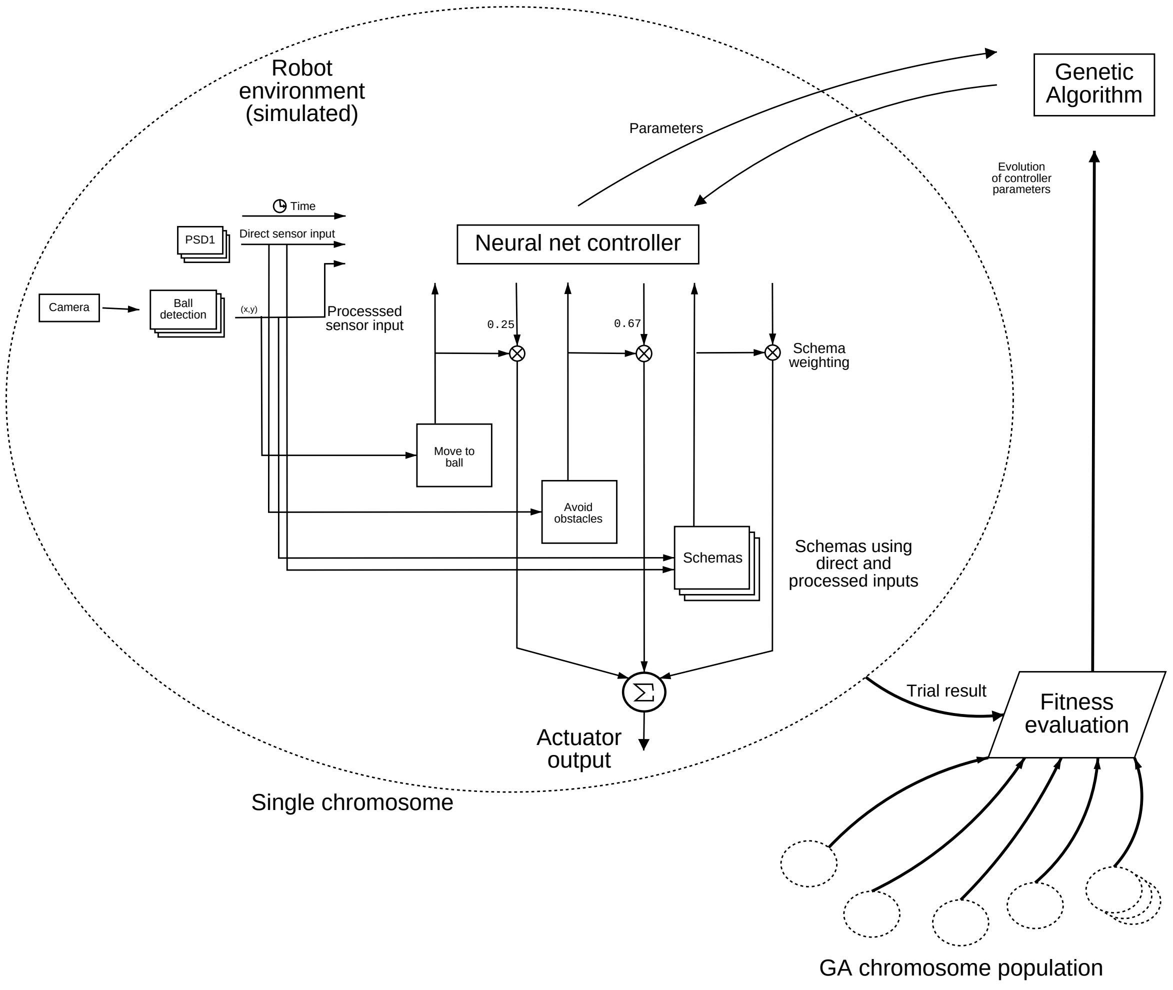


Figure 1.1: Overview of combined system

1.3 Outline of dissertation

Theory

Covers the background of systems used in the project, outlining the theory and general concepts associated with behaviour-based robotics, neural networks and genetic algorithms.

Literature review

Reviews approaches taken by others to achieve the task of robot control with behaviours. A survey of behavioural architectures and adaptive controller techniques researched in the field is presented.

Behavioural framework

Describes the design and implementation of a behavioural framework for the control of mobile robots. Programs produced with this framework may then be executed in the EyeSim simulator environment, or directly on EyeBot hardware.

Adaptive controller

Details the development of components used to produce an adaptive controller. The implentation of neural net and genetic algorithm class libraries is presented with an overview of the simulator system used. A brief description of preliminary tests of the genetic algorithm code within a different simulator is also provided.

Implementation of a ball-finding algorithm

Presents the use of the behavioural and adaptive controllers to implement a ball finding task within a simulated space. The initial investigation, design and implementation are described in full and accompanied by a set of analysed results from the evolved controller.

Chapter 2

Theory

2.1 Behaviour based robotics

The term behaviour based robotics is broadly applicable to a range of control approaches. Concepts taken from the original subsumption design [10] have been adapted and modified by commercial and academic research groups, to the point that the nomenclature has become generic. Some of the most frequently identified traits of behaviour-based architectures are [3]:

- Tight coupling of sensing and action. At some level, all behavioural robots are reactive to stimuli with actions that do not rely upon deliberative planning. Deliberative planning is eschewed in favour of computationally simple modules that perform a simple mapping from input to action, facilitating a rapid response. Brooks succintly expressed this philosophy with the observation that "Planning is just a way of avoiding figuring out what to do next." [10].

- Avoiding symbolic representation of knowledge. Rather than construct an internal model of the environment to perform planning tasks, the world is used as its own best model. The robot determines future behaviour directly from observations of its environment, instead of trying to produce an abstract representation of the world that can be internally manipulated and used as a basis for planning future actions.

- Decomposition into contextually meaningful units. Behaviours act as situation-action pairs, being designed to respond to certain situations with a definite action.

- Time-varying activation of concurrent relevant behaviours. A control scheme is utilized to change the activation level of behaviours during run time to accommodate the task that is trying to be achieved.

2.2 Emergent functionality

The term emergent functionality is used to describe the manifestation of an overall behaviour from a combination of smaller behaviours that may not have been designed for the original task [26, 34, 33]. The appearance of this behaviour can be attributed to the complexity of interactions between simple tasks instead of the tasks themselves. Behaviour is generally classified as emergent if the response produced was outside the analysis of system design but proves to be beneficial to system operation.

Arkin argues that the coordination between simpler subunits does not explain emergence completely [3]. As coordination of a robot is achieved by a deterministic algorithm, a sufficiently sophisticated analysis should be able to perfectly predict the behaviour of a robot. Rather, the emergent phenomenon is attributed to the nondeterministic nature of real world environments. These cannot ever be modelled completely accurately, and so there is always a margin of uncertainty in system design that could cause unexpected behaviour to be exhibited.

2.3 Neural networks

Neural networks are based on a model of processing loosely derived from the action of the biological neuron [16]. A network is formed from the interconnection of individual perceptron units. Each perceptron takes a number of inputs which are weighted by coefficient values before being processed by the unit. The sum of all weighted arcs entering a perceptron is passed through a nonlinear activation function, then thresholded by a chosen function. Depending on the result of thresholding the output of the activation function, the perceptron produces an output value.

Layers of perceptrons with outputs of one perceptron layer feeding into the inputs of another form a complete neural network. Networks may be classified as feedforward or feedback depending on whether the output layer of the network is connected to the input layer in a learning configuration.

Through learning techniques such as backpropagation, neural networks may be trained to respond in a certain way when given a set of inputs and training set of desired outcomes. Neural networks are commonly used to approximate a response function through some training algorithm. The value of a neural network is its ability to learn to produce a response for which we cannot algorithmically generate a function, and adapt its response with changing input parameters.

2.4 Genetic algorithms

Genetic algorithms (GAs) are a family of optimization techniques based on biological concepts of genetic recombination and evolution [7]. The premise of the GA approach is to begin with a population of parameter sets that are evaluated by the function we wish to optimize. The population is modified (evolved) by a set of genetically inspired operators based on the fitness values recorded for each parameter set. The purpose of these operators is to preserve and recombine the highest performing members while removing the lowest performing sets. They effectively alter the parameters to cover the problem search space, preserving and combining the highest performing parameter sets. After numerous iterations of population evolution, the highest performing sets should converge towards one or more optimal solutions of the objective function if they exist.

Genetic algorithms borrow terminology from biology to describe the interacting components. Populations are collections of function parameter sets referred to as chromosomes. Chromosomes in turn are sequences of genes. A gene encodes some part of the complete set of parameters evaluated by the objective function. Operators are also named after molecular biology terms describing interactions between DNA sequences at the molecular level.

2.4.1 Operators

The operators used by GAs are named after their analogous biological counterparts. The two major operators are crossover and mutation. Other operators used include reproduction of chromosomes, reordering and inversion of gene subsequences.

2.4.1.1 Crossover

Crossover has been identified as the major contributor to the efficiency of genetic algorithm operation. In its simplest form, crossover acts on two chromosomes of equal length, swapping all the parameters between the two at a selected point (Figure 2.1). The crossover point is usually randomly selected.

Other crossover types include multiple point and uniform. Multiple point crossover is a variation of the single point operation. Crossover points define spliced sections which are exchanged between two chromosomes of equal length. Sections at the beginning and end of the gene can now be exchanged by considering the chromosome as loop joining the end and beginning of the chromosome [8].

Uniform crossover works by generating a random binary mask with one bit per gene. During crossover, each bit of the mask is examined; the binary value at that point determines the parent whose gene is copied into the new offspring gene.

Opinion of the relative efficiencies of crossover variations is mixed. While 2-point crossover has been suggested to produce an improvement over simple 1-point for large populations, higher order crossovers were not observed to produce noticeable improvements. The value of uniform crossover as an alternative to the 1- and 2-point crossover operations has also been questioned and defended by different studies [32, 35].

2.4.1.2 Mutation

The second most important operator used is mutation. This introduces variety into a population by randomly altering genes of a chromosome with a small probability. Mutation ensures that all parts of the search space always have a finite chance of being checked regardless of the population state. The importance of mutatation increases as the population's fitness values converge [12]. Mutation then becomes the major source of variation between the similar chromosomes of a converged set.

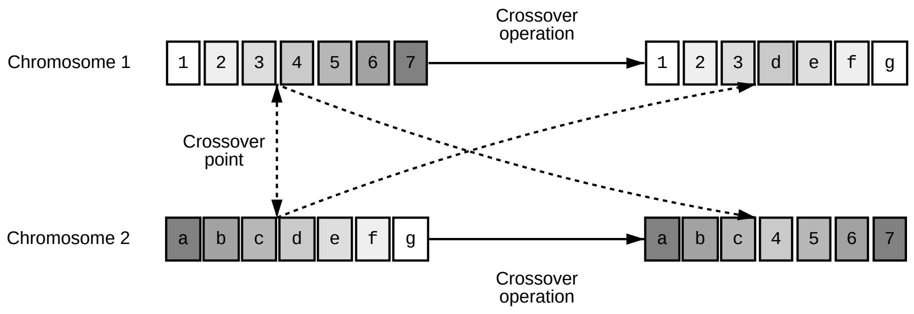


Figure 2.1: Single-point crossover operation

2.4.2 Encoding

The representation of parameters as genes is an important part of the implementation of a genetic algorithm. The mapping from parameter to gene is referred to as the encoding of a parameter. Different encodings affect the way operators alter existing genes, and hence the operation of the algorithm itself.

The traditional representation of genes is as directly encoded binary strings [18]. Other representations used include encoding with higher order cardinality alphabets, or operating directly on genes by their values instead of manipulating an encoding.

2.4.3 Behavioural memory

Behavioural memory works on the basis that an evolved population contains more information than just an optimal set of parameters [14]. Other characteristics may be represented intrinsically by the population; e.g. the diversity of high performing chromosomes may be an indicator of fitness function stability. These implicitly expressed properties may be utilized in the solution of a different problem set seeded with the current population.

Incrementally evolving a population set over a series of fitness functions requires preceding stages of evolution to have produced solutions that are viable for the subsequent stages. This process has been likened to a biological organism evolving in an environment changing over time, adapting to meet the requirements of the new conditions while retaining traits that enabled them to survive past challenges [31].

2.4.4 Application to behavioural control

The efficacy of a genetic algorithm in finding a solution is dependent on the problem domain and the existence of an optimal solution to the problem at hand. Applying GAs to problems that may be solved algorithmically is decidedly inefficient. They are best used for solving NP-hard problems. NP-hard problems are characterised by a difficulty of finding a solution due to a large solution search space, but being easy to verify once a candidate solution has been obtained. At least one configuration of behavioural robotic control has been demonstrated to be amenable to resolution by using a GA [29].

Chapter 3

Literature review

3.1 Behaviour classification

A general classification of robot behaviours has been outlined by Arkin [3]. A brief description and examples of each identified type is given below:

Exploration/directional

Exploration/directional behaviours involve movement of the robot in a definite direction.

- Specific direction - e.g. a heading relative to the current robot position
- Random - e.g. a wandering behaviour may exhibit random movements.

Goal oriented

Goal oriented behaviours involve movement towards an attractor. Attractors may be placed in a specific location, or refer to general area.

- e.g. an attractor object, such as a ball on a playing field (specific location)
- e.g. a home region (general area)

Aversive/protective

Aversive/protective behaviours avoid collisions between the robot and the objects present in its environment. This is clearly desirable to prevent damage occurring to the robot. Examples of these include object avoidance and aggression behaviours.

- e.g. avoid stationary objects, elude moving objects (dodge, escape)
- e.g. deliberately moving towards another [mobile] object to discourage it from colliding with the robot

Path following

Path following behaviours move the robot along a predetermined path. The methods for maintaining a navigational course along different types of path is dependent on the sensors used and the characteristics of the path. Examples of path following behaviour include:

- e.g. road following, hallway navigation, stripe following

Postural

Postural behaviours affect robots with articulations that affect their stability. These include biped and multi-legged robots. By interpreting sensor feedback, postural behaviours can determine how to drive the robot actuators to produce a stable configuration.

- e.g. balancing, or maintaining stability when subject to a disturbing force

Social/co-operative

Social/co-operative behaviours describe interactions between groups of mobile robots. The behaviours are similar to those observed in colony creatures such as ants, or herding animals.

- e.g. foraging - searching outwardly in groups to obtain items that are returned to a home area
- e.g. flocking/herding - moving in formations that dynamically change form as conditions change and distance is covered

Teleautonomous

Teleautonomous behaviours describe coordination between behaviours and a human operator. The actions of the human controller interacting with the behaviour can alter how the robot operate in its environment.

- e.g. influence - a directive from the human is factored into the action taken by the robot, e.g. messaging a robot a suggested location to move to
- e.g. behavioural modification - the behaviour of the robot is manually adjusted by the human by some established protocol

Perceptual

Perceptual behaviours affect how the robot assimilates information from a visual source.

- e.g. saccades - rapid sudden movements of visual sensors
- e.g. visual search - systematic search of a perceived area for a distinguishing feature
- e.g. ocular reflexes - automatic reaction to visual stimuli aiding further investigation of a visual scene

Walking

Walking behaviours enable legged robots to move by locomotion of their limbs.

- e.g. gait control - a behaviour to control the type of walking (static/dynamic) and speed at which the movement is attempted

Manipulator specific, gripper/dextrous hand

Manipulator specific behaviours (including gripper/dextrous hand types) operate an actuator specific to the robot, that can interact with the environment around it.

- e.g. grasping - a robotic hand behaviour for holding objects
- e.g. enveloping - a behaviour to cover another object with the manipulated appendage

3.2 Behavioural architectures

3.2.1 Subsumption

The subsumption architecture [10] as proposed by Brooks is widely acknowledged as the original behaviour-based system. Purely reactive systems predate subsumption [9], but did not attempt to provide a holistic design methodology. A significant departure from prevalent models of the time, subsumption introduced and rigidly adhered to tenets later refined and adapted by subsequent robot architectures.

Subsumption distinguishes between functional and behavioural decomposition of a task. Traditional control of mobile robotic systems as related by Brooks adhered to a horizontal organisation . An overall task is divided by functionality into sequential perception, planning and acting layers. Perceptual inputs enter the first layer and are are processed through the modules in order until they produce actuator output (Figure 3.1).

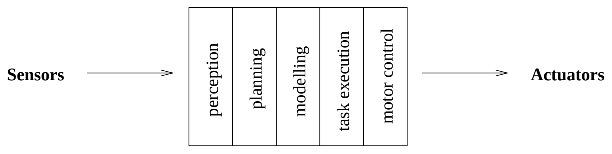


Figure 3.1: Functional organisation

The alternative structure proposed by subsumption is to divide the problem into independent behavioural modules, each running concurrently and arbitrated between in some way to produce actuator output (Figure 3.2). Each behaviour produces its own outputs from processing the same inputs at the same time as other behaviours in the system.

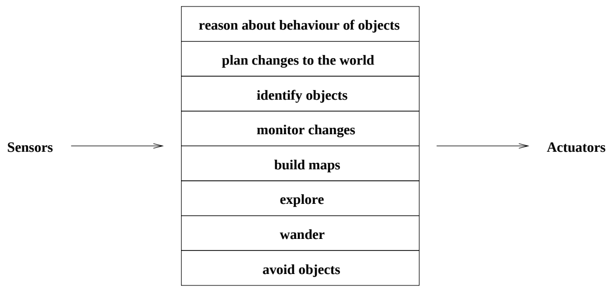


Figure 3.2: Behavioural organisation

Four key goals identified by subsumption are:

1. Achievement of multiple goals
2. Utilisation of multiple sensors
3. Robustness - adapting to sensor failure, environment change etc.
4. Additivity (scalability) - increasing of processing power as more sensors and capabilities are added to the system

Subsumption is a purely reactive system with a fixed hierarchy of control between behaviours. A behaviour is implemented as an augmented finite state machine (AFSM) (Figure 3.3). Individually, these transform an input into a behavioural output. Each behavioural module consists of a sensing behaviour, inhibitory input line and suppressing output line. Modules are completely independent of one another with no shared bus, clock or memory. They are arranged in a hierarchy arranged by levels of priority over one another. Higher levels may override the activation of lower level modules by sending appropriate signals to either the inhibitory or suppressing lines.

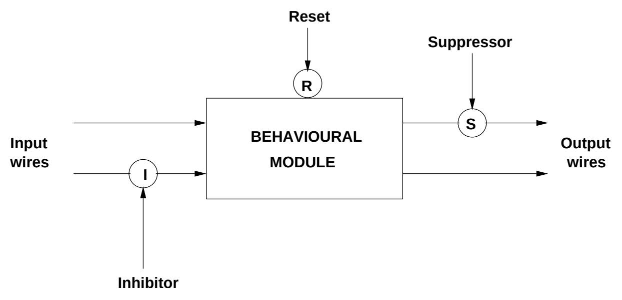


Figure 3.3: Augmented Finite State Machine (AFSM) used in subsumption

Coordination in subsumption is achieved by the specification of the fixed priority hierarchy. Simple low-level behaviours are subsumed by more complex ones, in an order specified by the hierarchy at design time. This structure is fixed during execution of the control program, thus systems built with subsumption must be carefully designed to operate within the parameters of the application task.

3.2.2 Autonomous Robot Architecture (AuRA)

3.2.2.1 Overview

AuRA (Autonomous Robot Architcture) is a modular robot architecture proposed by Arkin that combines deliberative and reactive components [2]. The deliberative component is divided into a descending hierarchy of mission planner, spatial reasoner and plan sequencer. This part is coupled with a schema manager that continuously controls and monitors the reactive behaviours while running.

The hierarchical (planning) level of the architecture relies on a controller component to produce plans consistent with the behaviour required of the robot. The actual implementation of the controller is not specified by the architecture; an A* planner and a finite state sequencer were originally used by the architecture's creators [4, 24], while other AuRA implementations have employed more experimental controllers.

Once the planner has decided upon a course of action, it attempts to select a set of behaviours that will be able to execute the plan. At this point control is switched from the deliberative component to the reactive component. The reactive component runs until it completes the planned sequence or detects a failure in the execution of the mission. In these cases, the deliberative controller is called upon again to determine a new course of action.

3.2.2.2 Schema-based control

Schema-based control is used in the reactive component of AuRA. A schema can be described as a simple behaviour. Each motor schema enables one fundamental behaviour for the robot [11]. The motor schemas asynchronously receive input from perceptual schemas to produce response vector outputs. The individual outputs are averaged by a process that sends the resulting signal to the low-level motor control system.

The concept of summed vectors combined to form an action has been postulated to play a role in the neural control of limb movements in the nervous systems of rats and humans [23, 27]. Research suggests that the firing of particular neurons activates a vector field that produces simple movement of limb in one particular direction. It is the numerous combination of neural firing patterns that enable a wide range of limb movements and articulations.

3.3 Adaptive controllers for behavioural selection

3.3.1 Neural networks

Neural networks have been successfully used to mediate directly between sensors and actuators to perform tasks. Past research has focussed on using neural net controllers to learn individual behaviours. Vershure et al developed a working set of behaviours by employing a neural net controller to drive a set of motors from collision detection, range finding and target detection sensors [36]. The online learning rule of the neural net was designed to emulate the action of Pavlovian classical conditioning [15]. The resulting controller associated actions beneficial to task performance with positive feedback.

Adaptive logic networks, a variation of neural networks that only uses boolean operations for computation, were successfully employed in simulation by Kube et al to perform simple co-operative group behaviours [22]. The advantage of the ALN representation is that it is easily mappable directly to hardware once the controller has reached a suitable working state.

3.3.2 Genetic algorithms

Genetic algorithms are described in 2.4. They have been used in a variety of different ways to produce or optimize existing behavioural controllers. Ram et al used a genetic algorithm to control the weightings and internal parameters of a simple reactive schema controller [29]. By tuning the parameters of the fitness function, robots optimized for the qualities of safety, speed and path efficiency were produced. This graphically demonstrates how behavioural outcomes may be easily altered by simple changes in a fitness function.

Harvey et al used a genetic algorithm to evolve a robot neural net controller to perform the tasks of wandering and maximising the enclosed polygonal area of a path within a closed space [17]. The controller used sensors as its inputs and was directly coupled to the driving mechanism of the robot. This approach is similar to the one taken in this project, but directly drives the robot as opposed to controlling a set of simple behaviours..

3.3.3 Genetic programming

Genetic programming uses genetic algorithm methods to evolves a working program from a fitness function [20]. It works by encoding the structure of a program and evolving this with GA operators. Nordin et al described a system for evolving controllers for Khepera robots with an optimized genetic programming implementation [28]. Rather than follow the common approach of evolving symbolic high level code structures, their technique works directly on the machine code that controls a robot. This resulted in significantly smaller memory and processing requirements, while producing learning results faster than a neural network in certain cases.

Another benefit of the genetic programming methodology is the resulting controller is in a purely symbolic form. It is easier to understand the operation of these than complex neural network structures, as are in a form familiar to designers of such systems.

Using this approach, small controllers for the simple tasks of Object Tracking and Object Avoidance were evolved online. The resulting programs evolved at a comparable rate to similar neural net controllers, and displayed robust characteristics in a noisy environment.

Chapter 4

Behavioural framework

4.1 Overview

The objective of the behavioural framework was to enable behaviourally based programs to be written and run on the Eyebot robots developed in our lab. At its foundation is a programming interface for consistently specifying behaviours.

We adapt the convention of referring to simple behaviours as schemas and extend the term to encompass any processing element of a control system. The specification of these schemas is made at an abstract level so that they may be generically manipulated by higher level logic and/or other schemas without specific knowledge of implementation details.

Schemas may be recursively combined either by programming or by generation from a user interface. Aggregating different schemas together enables more sophisticated behaviours to be produced. The mechanism of arbitration between grouped schemas is up to the system designer. When combined with coordination schemas to select between available behaviours, the outputs of the contributing modules can be directed to actuator schemas to produce actual robot actions. A commonly used technique is to use a weighted sum of all schemas that drive an actuator as the final control signal.

4.2 Architecture

4.2.1 Design

The framework architecture was inspired by AuRA's reactive component 3.2.2, and takes implementation cues from the TeamBots environment realisation [6, 5].

The basic unit of the framework is a schema, which may be perceptual (e.g. a sensor reading) or behavioural (e.g. move to a location). A schema is defined as a unit that produces an output of a predefined type. In our implementation, the simplest types emitted by schemas are integer, floating point and boolean scalar values. More complex types that have been implemented are the two-dimensional floating-point vector and image types. The floating point vector may be used to encode any two dimensional quantity commonly used by robot schemas, such as velocities and positions. The image type corresponds to the image structure used by the Eyebot's image processing routines.

Schemas may optionally embed other schemas for use as inputs. Data of the predefined primitive types is exchanged between schemas. In this way behaviours may be recursively combined to produce more complex behaviours. From the higher level schemas' point of view, the inputs from lower level schemas are indistinguishable by schema complexity.

In a robot control program, schemas organization is represented by a processing tree. Sensors form the leaf nodes, implemented as embeddable schemas. The complexity of the behaviours that embed sensors varies, from simple movement in a fixed direction to ball detection using an image processing algorithm. The output of the tree's root node is used every processing cycle to determine the robot's next action. Usually the root node corresponds to an actuator output value. In this case output from the root node directly produces robot action. This is illustrated in Figure 4.4.

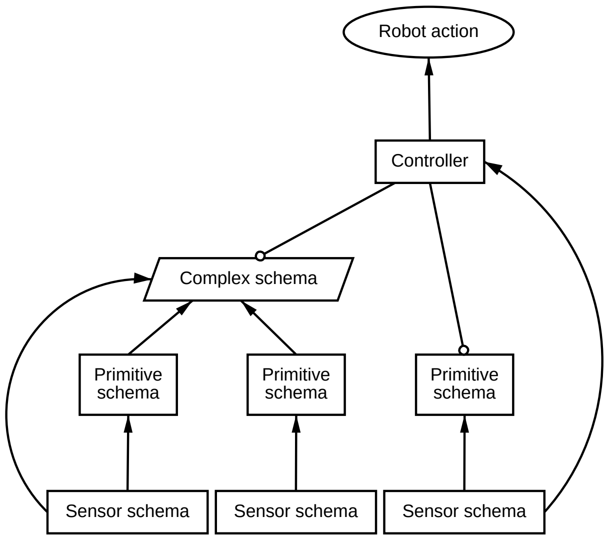


Figure 4.1: Tree organisation of schemas and controller

4.2.2 Implementation

The behavioural framework has been implemented in C++, using the ROBIOS API to interface with the Eyebot. These same functions are simulated and available in EyeSim, enabling programs created with the framework to be used on both the real and simulated platforms.

The framework has been implemented with an object-oriented methodology. There is a parent "Node" class that is directly inherited by type-emitting schema classes for each predefined type. For example, the NodeInt class represents a node that emits an integer output. Every schema inherits from a node child class, and is thus a type of node itself.

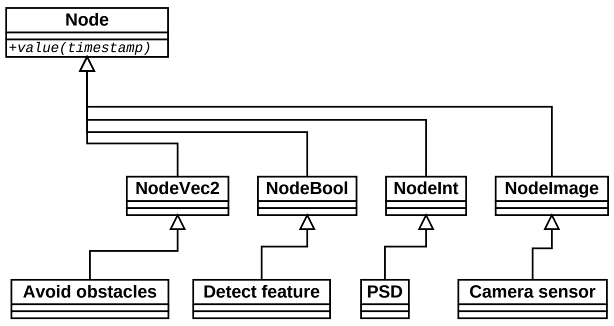


Figure 4.2: C++ inheritance of Node and sample child schemas

All schema classes define a value(t) function that returns a primitive type value at a given time *t*. The return type of this function is dependent on the class - for example, schemas deriving from NodeInt returns an integer type. Embedding of schemas is by a recursive calling structure through the schema tree. Each schema class that can embed nodes keeps an array of pointers to the embedded instances. When a schema requires an embedded node value, it iterates through the array and calls each embedded schema's respective value(t) function. This organisation allows invalid connections between schemas to be detected at compile time: when a schema embedding a node of an invalid type tries to call the value function, the returned value will not be of the required type. The compiler checks the types from connected emitting and embedding nodes are the same at compilation time and will flag any mismatch to the programmer.

The hierarchy of schema connections forms a tree, with actuators and sensors mediated by various schemas and schema aggregations. Time has been discretised into units, Schemas in the tree are evaluated from the lowest level (sensors) to the highest from a master clock value generated by the running program.

4.3 Implemented schemas

A small working set of schemas using the framework was created for use in the neural net controller design task. The set of schemas with a short description of each is listed in Table 4.1. The schemas shown are either perceptual (e.g. Camera, PSD), behavioural (e.g. Avoid) or generic (e.g. Fixed vector). Perceptual schemas only emit a value of some type that is used by the behavioural schemas. Behavioural schemas transform their input into an egocentric vector output that would fulfill its goal.

Schema	Description	Emits
Camera	Camera perceptual schema	image
PSD	PSD sensor perceptual schema	integer
Avoid	Avoid obstacles based on PSD reading	2D vector
Detect ball	Detects ball position in an image from its hue	2D vector
Fixed vector	Fixed vector representation	2D vector
Linear movement	Moves linearly from current position to another point	2D vector
Random	Randomly directed vector of specified size	2D vector

Table 4.1: Implemented schemas

4.4 Graphical user interface

A front-end program has been created to allow point-and-click assemblage of new schema from preprogrammed modules. The representation of the control program as a tree of schemas maps directly to the interface presented to the user (Figure 4.3).

For a schema to be recognised by the user interface, the programmer must “tag” the header file with a description of the module. A sample description block is shown in Figure 4.4. The graphical user interface then parses the header files of a schema source directory to determine how to present the modules to the user.

The header block specifies how a particular schema interconnects with other schemas. It includes a description of typed initialisation parameters for the module, a list of ports that can be used for emitting or embedding other modules, and various meta-information.

From the interconnection of the visual modules, the user interface generates appropriate code to represent the tree specified by the user. The structure of the program is determined by analysing the behaviour tree and translating this into a series of instantiation and embedding calls. The uniform nature of the behavioural API facilitates a simple code generation algorithm.

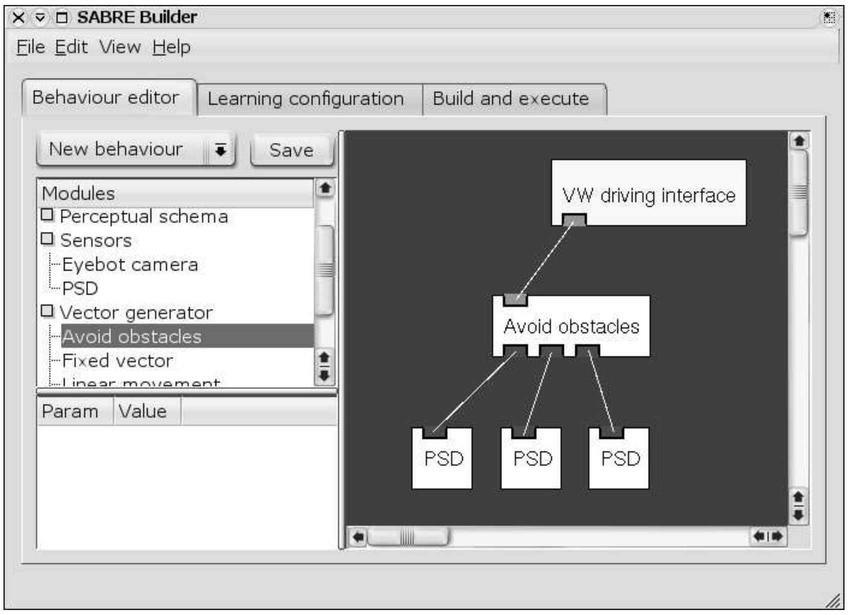
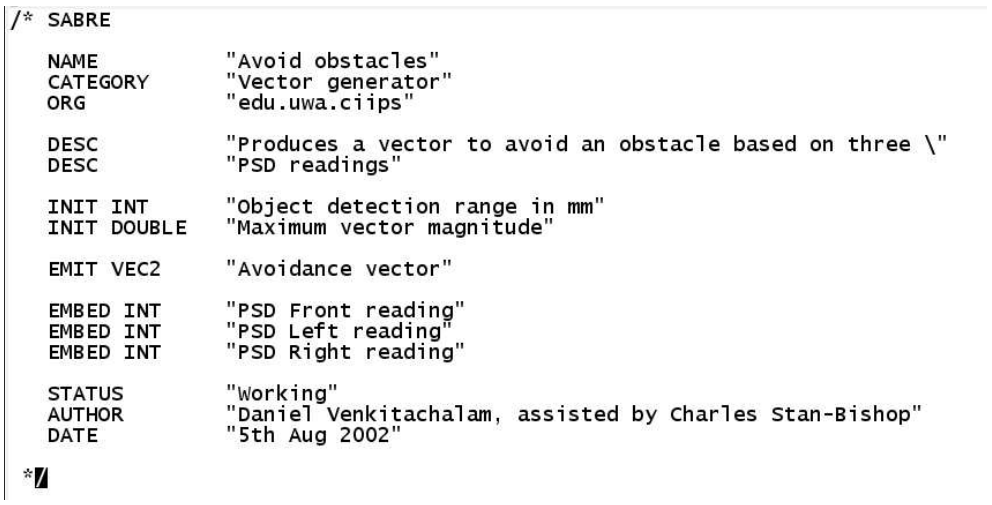


Figure 4.3: GUI for assembling schemas



```
/* SABRE

   NAME          "Avoid obstacles"
   CATEGORY      "Vector generator"
   ORG           "edu.uwa.ciips"

   DESC          "Produces a vector to avoid an obstacle based on three \"
   DESC          "PSD readings"

   INIT INT      "Object detection range in mm"
   INIT DOUBLE   "Maximum vector magnitude"

   EMIT VEC2     "Avoidance vector"

   EMBED INT     "PSD Front reading"
   EMBED INT     "PSD Left reading"
   EMBED INT     "PSD Right reading"

   STATUS        "Working"
   AUTHOR        "Daniel Venkitachalam, assisted by Charles Stan-Bishop"
   DATE          "5th Aug 2002"

 */
```

Figure 4.4: Schema header descriptor block

Chapter 5

Adaptive controller

5.1 Overview

The adaptive controller system used in this project consists of two parts: a neural net controller and a genetic algorithm learning system. The role of the neural net controller is to transform inputs to control signals that activate the behaviours of the robot at the appropriate time. The structure of the neural net determines the functional transformation from input to output. As a consequence of the neural network topology, this is effectively determined by changing the weights of all the network arcs. Evolution of the structure to achieve an objective task is performed by the genetic algorithm. The set of parameters describing the neural network arc weights is optimized to produce a controller capable of performing the task.

5.2 Neural net

The neural net component of the controller implements a fully-connected feed-forward structure. Perceptrons in the input and hidden layers are each joined to every perceptron of the next layer by a weighted arc. Figure 5.1 shows the topology of a typical 4 layer neural network, with 3 inputs, 2 hidden layers of 4 perceptrons each, and 2 perceptrons in the output layer. The number of hidden layers is definable by the system designer.

Input values enter the network in the first (input) layer and propagate via weighted arcs and intermediate hidden layers through to the output layer. The sum of input-weight products entering each neuron is evaluated by an activation function to determine the output of that neuron.

The network implementation is specified in such a way that each neuron is evaluated once per cycle. This eliminates redundant processing and any ambiguities that could be introduced from evaluation of directly or indirectly dependent neurons at different times.

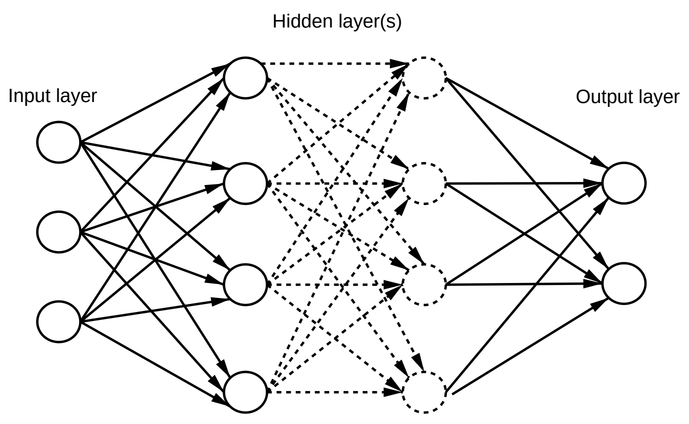


Figure 5.1: Example of a fully connected feed-forward neural network

Each neuron in the net utilises a standard non-linear sigmoidal activation function with parameter ρ (Equation 5.1). This function approximates the Heaviside step function, with parameter ρ controlling the graph slope. For our controller networks we set ρ to 1 (Figure 5.2).

$$a(x) = \frac{1}{1 + \exp(-\rho x)} \tag{5.1}$$

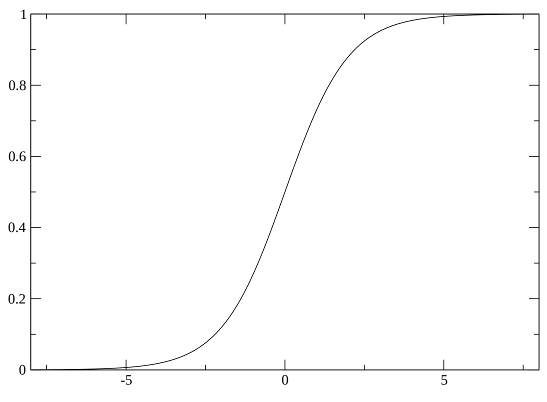


Figure 5.2: Sigmoidal activation function, $\rho = 1$

The result of the activation function is thresholded before propagating through the network. Thresholding of the activation function performs basic clipping for values above a preset threshold τ (Equation 5.2). Any value below the threshold is set to zero. The threshold of each neuron may optionally be made an evolvable parameter of the genetic algorithm.

$$t(a(x)) = a(x) - \tau \tag{5.2}$$

5.3 Genetic algorithm

Our implementation of a genetic algorithm uses a direct binary encoding scheme to encode the numeric weightings and optionally the thresholds of the controller's neural net. A single complete neural net controller configuration is encoded into a chromosome. The chromosome itself is a concatenation of individual floating point genes. Each gene encodes a single weights of the neural network. The population consists of a number of chromosomes, initially evaluated for fitness and then evolved through numerous iterations. Population evolution is achieved by a single-point crossover operation at gene boundaries on a set percentage of the population. This is supplemented by mutation operations (random bitwise inversion) on a small percentage of the population, set as a user parameter. The top performing members of a population are preserved between iterations of the algorithm (elitism). The lowest performing are removed and replaced by copies of the highest performing chromosomes. In our trials we have used population sizes of between 100 and 250 chromosomes.

5.4 Simulator

The simulator environment was built around an early version of EyeSim 5, developed within the Mobile Robots research group [37]. EyeSim is a sophisticated multi-agent simulation of the Eyebot hardware platform set in a virtual 3D environment. As well as simulating standard motor and hardware sensors, the environment model allows realistic simulation of image capture by mounted camera hardware (Figure 5.3). This allows for complete testing and debugging of programs and behaviours using image processing routines, an ongoing focus of work in our group.

At the time of the project's commencement, EyeSim was in a premature state with respect to stability and features required for interaction with the learning system. Preliminary work on the project required debugging of the simulator and adding features to enable integration with the learning system.

A logging API was added to the simulator to allow programs to log robot and ball

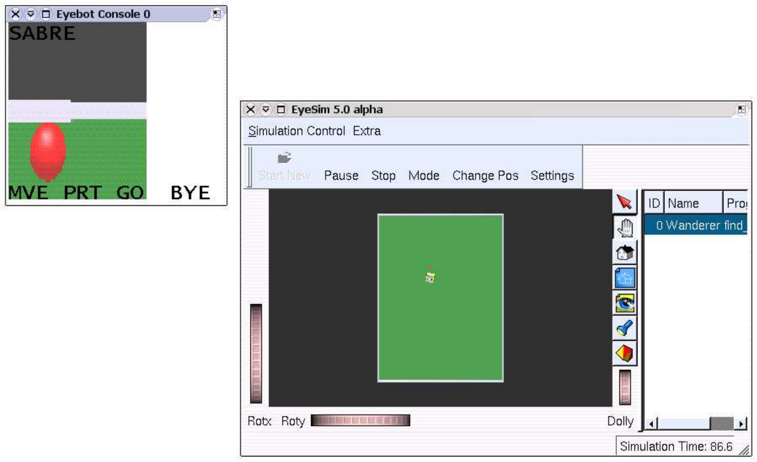


Figure 5.3: EyeSim 5 simulator screen shot

positions. Because we run our programs in a simulated environment, we can obtain records of the positions and orientations of all objects in the environment with perfect accuracy. The logging calls determine positions during execution from the simulator's internal world model. The control program of the robot calls these functions after it completes execution and writes them in a suitable file format for reading by the separate evolutionary algorithm. The results are not used by the robot to enhance its performance while running. The final output of the program to the logfile is analysed after termination to determine how well the robot performed its task.

5.5 Preliminary testing of genetic algorithm

5.5.1 Background

During preparation of the simulator environment and supporting class libraries, research into the learning components of the project was postponed as these were not yet running stably enough for testing purposes. For this reason, the genetic algorithm component was originally written and tested in Java under the TeamBots simulation environment [6]. TeamBots is a simulation and programming environment that adheres to AuRA's use of reactive schemas in the behavioural layer (3.2.2.2). The TeamBots software distribution provides a simulated set of motor and perceptual schemas that may be combined and arbitrated between with a user programmed coordination module. An implementation of robot foraging and gathering using a finite-state machine controller to sequence behaviours is included with the package. An adapted version of this program incorporating the genetic algorithm code formed the basis of our initial tests.

5.5.2 Implementation

In TeamBots, the foraging task is implemented as a state machine selecting between Wander, Acquire and Deliver behaviours (Figure 5.4). Robots wander around the environment until they encounter an object that may be picked up. The perceptual schema that detects this object triggers the state machine controller to move into the acquire state, whereupon the robot attempts to obtain the item. When this is successfully achieved, the robot moves into the Deliver behaviour that compels it to return to its home base and drop off the object. On completion of delivery, the robot moves back into the wander state until either all objects have been delivered or the simulation terminates.

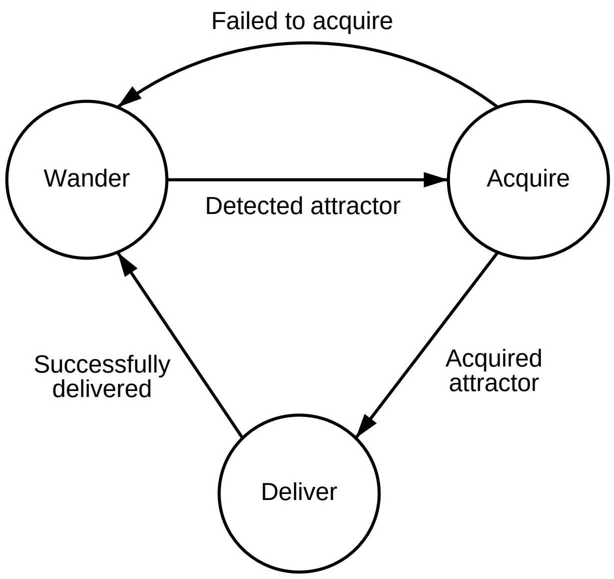


Figure 5.4: State machine control of foraging task

Each of the Wander, Acquire and Deliver behaviours is itself composed of more primitive schema. The weightings of the schemas that comprise these is configured by the programmer at compilation time. The relative weightings determine the magnitude of effect individual motor schema will contribute towards the combined behaviour. By adjusting the weightings of the motor schemas we are changing the functional composition of the respective higher level behaviour.

The original program was modified to conduct a timed trial of robot performance over 20 simulated minutes. There were a maximum of 16 targets that could be retrieved by the robot; 8 static and 8 moving in randomly seeded directions. The environment also contained obstacles that had to be avoided by the robot, the degree of avoidance being determined by a schema weighting. The fitness metric for the task was simply defined as the number of items retrieved and returned to the home base within the allotted time.

5.5.3 Results

Beginning with a random population of weightings, the first implementation of the genetic algorithm was used to evolve an optimal set of schema coefficients over several trials. From a testing selection of the top performing chromosomes in the final population, we observed a significant increase in foraging task performance. Convergence was achieved in approximately 70 iterations of the algorithm, each taking an average of 20 minutes to run. The assemblage weightings determined by the genetic algorithm displayed an improvement of $\sim$20% over the default programmed weightings.

Chapter 6

Implementation of a ball-finding task

6.1 Overview

The evolved controller task implemented in this project is to search an enclosed space to find a coloured ball. We began by identifying the primitive schema that could be combined to perform the task. These are selected by the evolved controller during program execution to perform the overall task. A suitable initial fitness function for the task was constructed and then an initial random population generated for refinement by the genetic algorithm.

6.2 Primitive schemas

We identified the low level motor schema that could conceivably perform this task when combined together. Each schema produces a single normalised 2-dimensional vector output, described in Table 6.1.

Behaviour	Normalised vector output
Move straight ahead	In the direction the robot is facing
Turn left	Directed left of the current direction
Turn right	Directed right of the current direction
Avoid detected obstacles	Directed away from detected obstacles
Move towards a detected ball	Tangential to facing direction, towards a detected ball

Table 6.1: Primitive schemas (behaviours)

The basic schema “Move straight ahead” can be combined with the “Turn left/right” schemas to constantly provide new destinations for the robot. The relative weightings of these determine the direction the robot will want to move of its own volition.

The "Avoid detected obstacles" schema embeds PSD sensor schemas as inputs, mounted on the front, left and right of the robot (Figure 4.3). These readings are used to determine a vector away from any proximal objects (Figure 6.1). Activation of the "Avoid detected obstacles" schema prevents collisions with walls or other objects, and getting stuck in areas with a clear exit.

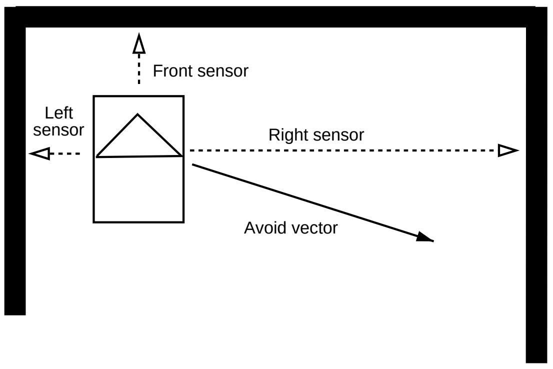


Figure 6.1: Avoidance schema

Ball detection is achieved by a hue recognition algorithm that processes images captured from the Eyebot camera (Figure 5.3). When the ball is recognised in the image, the schema produces a vector that will rotate the robot towards the direction of the ball. The position of the detected ball's x-axis centroid in the image is used to produce this vector. There is assumed to be a central "dead-zone" band in the image of definable width. If the centroid lies within this region the detection schema does not produce output. Outside of this band, the magnitude of the movement vector increases linearly with distance from the image centre, producing a turning vector in the left or right direction. The turning vector by itself produces only rotation in the direction of the ball - it must be combined with the movement straight ahead to actually reach the ball.

When the ball is not visible to the camera (either due to the orientation of the robot or the camera's angle of inclination), the "Move towards a detected ball" schema returns a 0 vector, effectively disabling its contribution to the final vector.

6.3 Neural network controller

The role of the neural network controller is to combine the individual vector outputs of the primitive schemas each processing cycle to produce a single new destination vector. In principle, the neural network receives information from all sensor inputs and a clock value to determine the activity of each of the schemas. Inputs may be in the form of raw sensor readings or processed sensor results such as distances, positions and preprocessed image data. Information is processed through a number of hidden layers and fed into the output layer. Our design sets the number of hidden neurons in each hidden layer to be equal to the number of input neurons. Given i inputs (including time) and o behaviours to arbitrate between, the input layer will hold i neurons, each hidden layer will each contain i neurons and the output layer o neurons.

The controller's output neurons are responsible for producing the weights of each schema per cycle. Each schema's output is multiplied by its weight, and then all weighted outputs are summed to produce a single actuator control signal. In this application, the single actuator output required is a vector representing a target destination for the current timeslice. This is translated into a motor driving command and executed before the next processing cycle.

A controller is set to run for a fixed number of cycles. Either at the end of this time or the time of ball approach, the results of the trial (Table 6.3) are evaluated and recorded by a fitness function. Using these fitness results, the parameters of the neural network controller are evolved by the genetic algorithm as described in Section 5.3.

6.4 Preliminary investigation

Configurations of the neural network and genetic algorithm that we have experimented with are outlined in Table 6.2.

Configuration	Parameters
Hidden layers	1, 2
Fixed threshold	On, off (evolved)
Ball placement	Single random point, preset positions, averaged random positions
Movement schema	Random and straight, directed (left/right) and straight
Population size	100, 150, 200

Table 6.2: Controller evolution configurations

The observed effects of these parameters were as follows:

- A single hidden layer was found to produce controllers with comparable fitnesses to a two-layer neural net for the task, with the advantage of taking less time to converge to a final state.

- Making the threshold an evolvable parameter of the genetic algorithm did not improve nor degrade the fitnesses obtained by the GA. However, the time taken for a the population to converge did increase.
- Placing the ball in a set of predefined positions and taking the weighting average of these produced a neural network controller capable of seeking to the specific positions of the staging. When this controller was tested in a random environment it was unable to maintain similarly high fitness values, performing worse than controllers evolved from averaged random positions.
- The use of only random and straight movement schemas resulted in controllers that could not consistently achieve high fitness values. This is likely because the controller is unable to learn a consistently performing selection of behaviours incorporating the random behaviour. Using directed and straight motor schemas only did not have this problem.
- No convergence was observed using single random point trials (where a random maze configuration is set up and a single fitness value is taken). The uncertainty of the setup means that chance plays a large role in determining whether a fit chromosome is actually awarded a higher fitness value than other chromosomes which may fulfill the fitness criteria accidently. Similarly, the presence of random direction schema By averaging a series of random trials the confounding effect of this uncertainty can be reduced; we found weighted random trials sufficient to produce convergence of the top 50% of the population in the vast majority of trials.
- Population sizes large than 100 took longer to converge but did not produce noticeably better fitness results.

6.5 Staged evolution of a controller

The neural network controller was evolved in incremental stages, following the principle of behavioural memory (2.4.3). In each phase, every chromosome of the population was evaluated for fitness. Each chromosome was decoded into a neural network structure and tested over 5 trials in a newly generated environment. For each trial, the initial position and orientation of the robot with respect to the ball was randomly set by the maze generator. The results of these were then averaged to determine a final fitness value for the chromosome. These averaged fitness values were recorded and evolved by the genetic algorithm to produce a new population.

We configured the genetic algorithm to use a population of 100 chromosomes, initially set to random values. The crossover rate of the algorithm was set at 60% and the mutation rate to 5%. At the end of each entire population iteration, 36% of the population was removed and replaced with the top performing chromosomes of the population, with a probability related to their comparative fitnesses. A proportion of the highest performing chromosomes was preserved between populations to increase their chance of propagation through the genome.

Parameter	Description
s_i	Initial distance to the ball
s_f	Final distance to the ball
ϕ_f	Final angle to the ball
t_f	Time taken to reach the ball
T	Total time trial is run for

Table 6.3: Fitness measurements taken after trial completion

First stage

The goal of the first stage of controller evolution was to produce a set of parameter configurations that would place the robot close to the object. We constructed a fitness function (Equation 6.1) to reward controllers that ended up close to the ball at the end of the allotted time, or reached the ball before the total time had elapsed. The parameter β may be used to bias towards the selection of controllers that complete in less time. In our trials we set $\beta = 4$.

Equation 6.1 only checks the robot's final distance and time in its evaluation, and so may reward controllers that do not perform the actual desired behaviour. For example, robots that end the trial anywhere in the vicinity of the ball will be rewarded a positive fitness value depending on the final distance, regardless of whether it was heading towards the target or not.

$$f(x) = \left(\frac{s_i - s_f}{s_i}\right) + \beta\left(1 - \frac{t_f}{T}\right) \tag{6.1}$$

The population took around 16 iterations to converge. The time taken to test and evolve an entire population was initially ~24 minutes, but reduced to ~10 minutes due to the increased fitness (and hence early completion) of the population. The total evolution time on an 1200Mhz Athlon took over three hours. A plot of the evolving fitness is given in Figure 6.2.

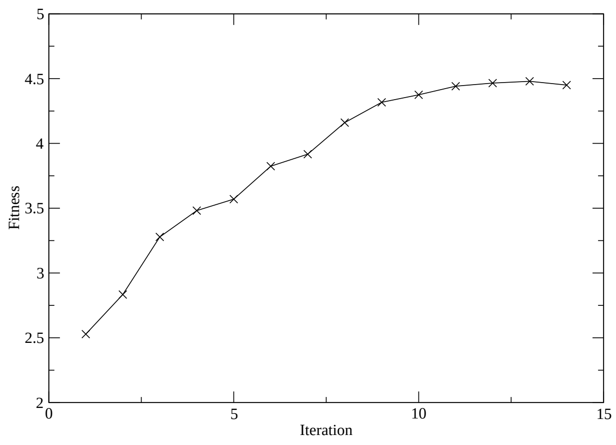


Figure 6.2: Fitness of stage 1 population's top 25% chromosome

Second stage

In the second stage of evolution, we imposed a condition that the robot must have reached the ball to be awarded a fitness value. If it has not reached the ball, the fitness value is automatically set to 0. The original fitness function (Equation 6.1) now only needs to reward short completion times, as the first term (measuring final distance to the ball) will always be zero upon successful termination. The fitness function of the second stage is given in Equation 6.2.

$$f(x) = \beta\left(1 - \frac{t_f}{T}\right) \tag{6.2}$$

Convergence of the population was achieved in 16 iterations, with each iteration taking an average of 10 minutes. The total time to convergence for this stage was approximately 2.5 hours.

Third stage

In the third stage of evolution, we change the time-based reward component. Instead of being calculated from the proportion of completion time to allotted time, it is now based on the proportion of completion time to optimal completion time. The optimal completion time is derived from the original distance to the ball divided by the distance that the robot covers per time period. This gives us the minimal time it would take a robot facing the ball to move to it taking a straight path directly towards the ball. However, in the case that the robot is not directly facing the ball the turning time is not taken into account.

In addition to the change in time reward allocation, we introduce a fitness restriction on facing angle similar to the distance restriction from the second stage. A trial can only be successfully completed (and hence a fitness value awarded) if the robot is facing the ball and has approached it. The definition of facing the ball is that the angle between the robot's medial axis and the ball is within a range $\pm\theta$ of the angle between the robot's centre and the ball, whereθ is a parameter of fitness function set by the designer.

We set a maximum find ball time limit of $\frac{1}{20}$ the optimal time, and calculated the relative weighting of distance based on this proportion. Any controller that takes longer than this is given a fitness value of 0. The minimum facing angle θ was set to $\frac{\pi}{4}$.

Convergence of the population was achieved in 20 iterations, taking an average of 8 minutes each to complete. The complete evolution of this stage again took approximately 2.5 hours.

Fourth stage

In the fourth stage, we tune the parameters introduced in stage 3. The facing angle was made smaller ($\theta = \frac{\pi}{6}$) and the time limit to be awarded a fitness set to $\frac{1}{5}$ of the optimal completion time.

This stage took 25 iterations to converge, each taking an average of 7 minutes to complete to a total of just under 3 hours.

Fifth stage

In the fifth stage, a chromosome's fitness must be greater than one in each of the five trials that are averaged to a final fitness value. If a single trial returns a value of less than one, the chromosome is assigned a final fitness of zero. This is to ensure that the controller is able to find the ball in all situations.

The other change made from the fourth stage was a reduction in the allotted simulation time of the robot. Combined together this makes it difficult for unsuitable controllers to persist over several iterations.

After two days of running this test a complete convergence had not yet been achieved. A controller taken from the most fit population to date was used in the evaluation of the controller.

6.6 Evaluation of controller

The highest performing chromosome of the fifth stage population was tested in a number of observed randomised trials. We found that the controller was capable of finding the ball, frequently with a close to optimal trajectory. There were times when the path taken was sub-optimal but still reasonably direct. However, there remain instances where despite reaching the target in reasonable time, the path taken by the robot is erratic and clearly not directed towards the ball. A set of scenarios that illustrate the observed behaviour of the controller are described below.

Each scenario plot includes the behaviour of a reference implementation of the ball finding problem performing in the same environment. The reference ball finding program is a simple monolithic search program that does not use behaviours or complex control. It rotates the robot about its current position until aligned with a detected ball. The robot is then driven in a straight path towards the ball, adjusting its course by reorienting during driving as necessary. This method results in the robot always taking the shortest (straight line) path from its starting location to the ball. However, this is not necessarily the fastest route to the ball; the time taken rotating in a fixed position could be used to move in an unknown direction while looking for the ball. This would likely move the robot towards the ball in an open space, while simultaneously performing the search task. Such a course of action would resemble the beginning of a spiral until the ball was found, the radius being dependent on the speed of the robot.

The different parameters and logging intervals of each program mean that the times to complete the task displayed on the plots are not directly comparable. They are used to get an indication of the relative time spent seeking, and the trend of distance gains to the ball over time.

6.6.1 Scenario A

Scenario A is an example of a path taken by the robot that is optimal for the situation. In this scenario the robot begins with the ball to the right of the robot, and within the camera's field of vision (Figure 6.3).

A trace of the path taken is shown in Figure 6.4. It can be seen that the path taken is close to the optimal path; i.e. a straight line towards the ball.

Figure 6.5 shows plots of the robots' facing angle and distance from the ball over time. It can be seen that both robots orient themselves to the ball then move in a constant direction towards it, readjusting themselves near the end of the trajectory. The plot of

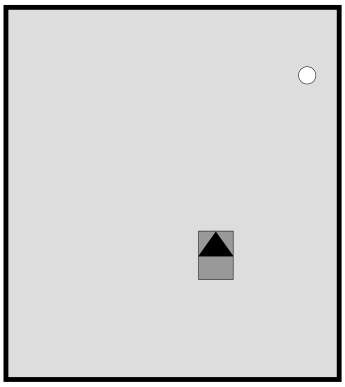


Figure 6.3: Scenario A: initial position and orientation

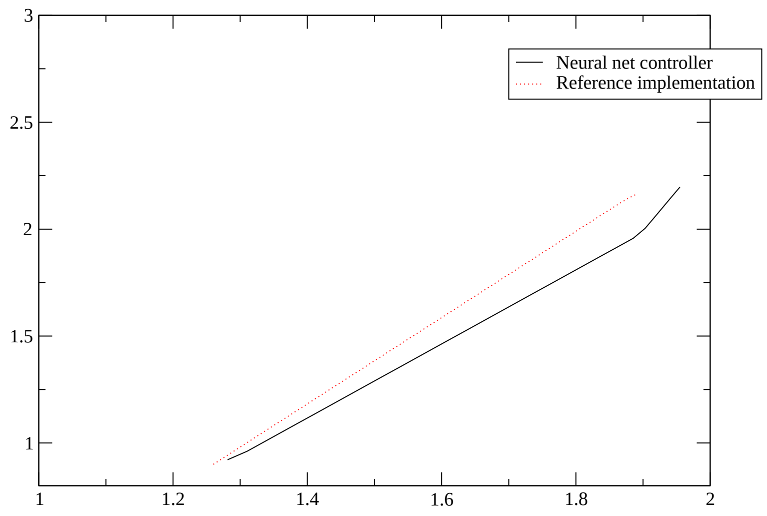


Figure 6.4: Scenario A: trace of path taken by robot

distance to ball shows the robot moving towards the ball from the beginning of the trial; unlike the reference implementation, it does not wait to orient itself completely before moving. As a result, it should arrived at the ball faster than the manually designed program with all other parameters equal.

6.6.2 Scenario B

Scenario B shows the controller exhibiting behaviour that deviates from the optimal but completes the task within a reasonable time interval of the preprogrammed solution. In this particular scenario, the paths taken and time to rotate the robot suggests the manual controller should complete the task earlier than the evolved one. The initial configuration

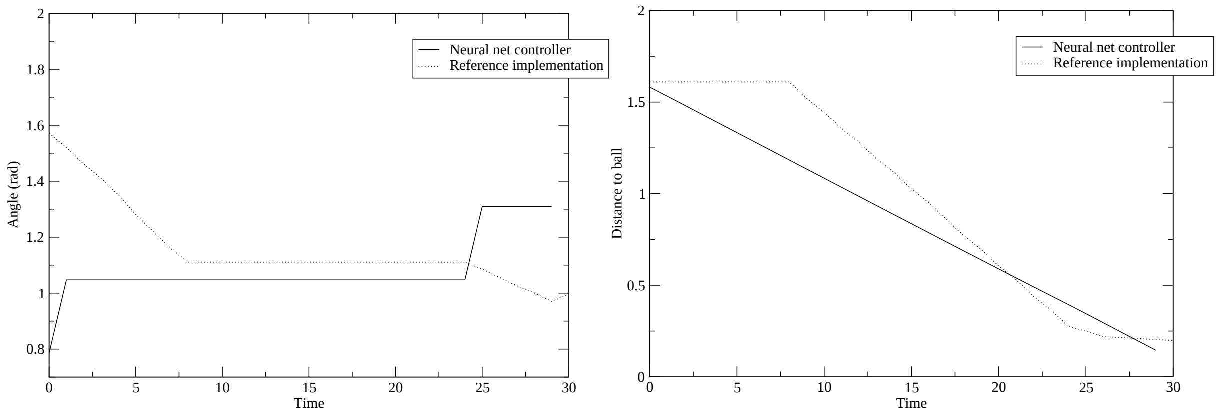


Figure 6.5: Scenario A: plots of robot facing angle and distance to ball against time

is shown in Figure 6.6, with the robot starting in a position that faces away from the ball and hence not being able to detect it at the beginning of the trial.

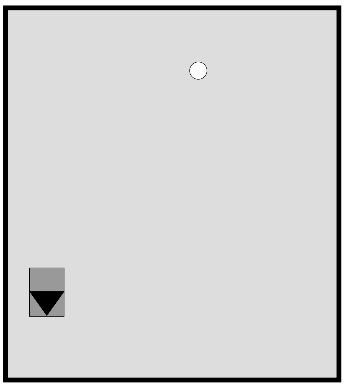


Figure 6.6: Scenario B: initial position and orientation

A trace of the path taken by the robot is shown in Figure 6.7. It is clearly visible that the path is longer than the optimal line from robot to ball taken by the reference implementation.

Plots of the robot's facing angle and distance to ball over time are given in Figure 6.8. The angle plot shows the robot sharply turning until it can see the ball at the beginning of the trial. There is some orientation of the robot, turning slightly left and right until it begins moving in a straight line towards the ball. The path taken now closely follows the optimal path shown but then deviates slightly as the robot drifts right. This deviation causes the robot to stray progressively further away from the optimal path before reorienting itself. Notably, the time of reorientation is shortly after the distance to

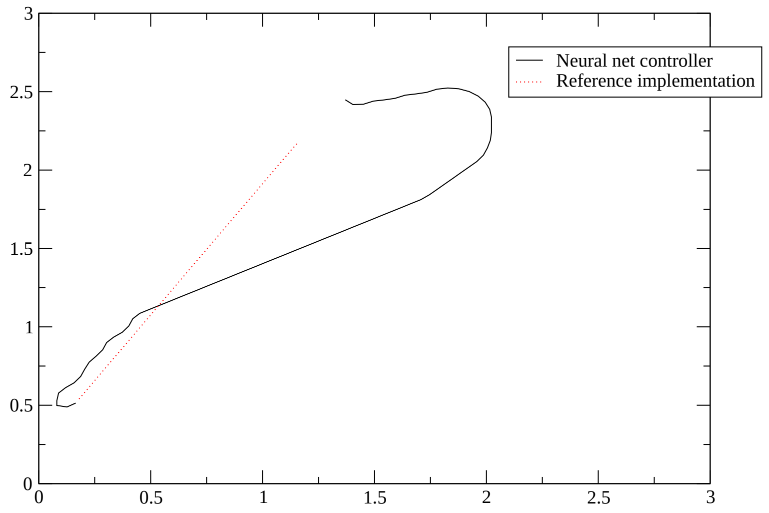


Figure 6.7: Scenario B: trace of path taken by robot

the ball begins to increase.

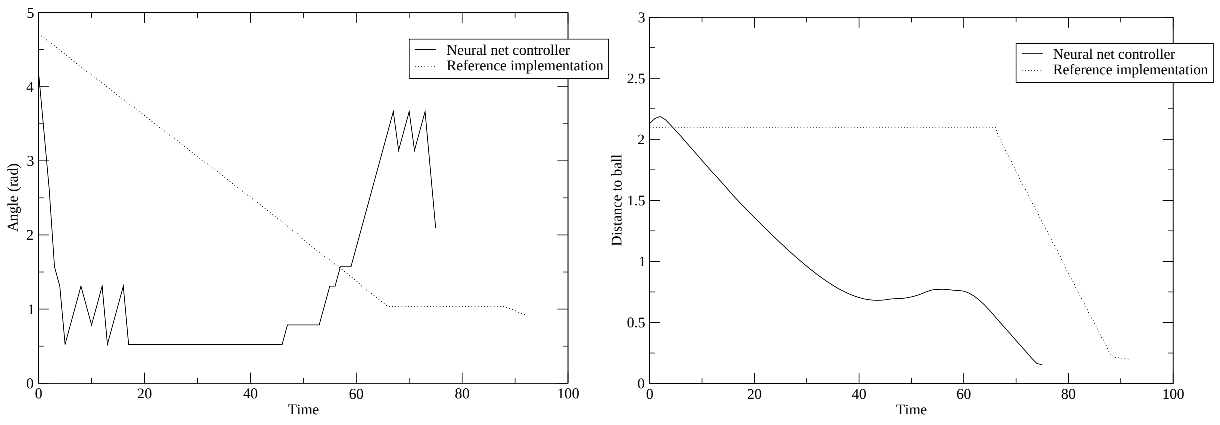


Figure 6.8: Scenario B: plots of robot facing angle and distance to ball against time

The behaviours exhibited in this scenario shows that the neural net has not evolved a recognisable behaviour as we would expect from Scenario A. The robot begins taking the optimal path but shortly thereafter strays from it; the network does not maintain a state of moving towards the goal as we expect it to. Achievement towards the final objective is compensated by its subsequent recovery action of turning and moving to the ball at the end of the trajectory.

6.6.3 Scenario C

Scenario C displays characteristics similar to Scenario B, but from a different initial configuration. Again, the ball is not within the camera's field of vision at the beginning of the trial, but this time the ball is situated to the left of the robot (Figure 6.9).

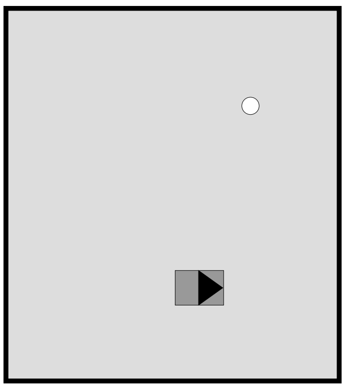


Figure 6.9: Scenario C: initial position and orientation

A trace of the robot's trajectory is shown in Figure 6.10.

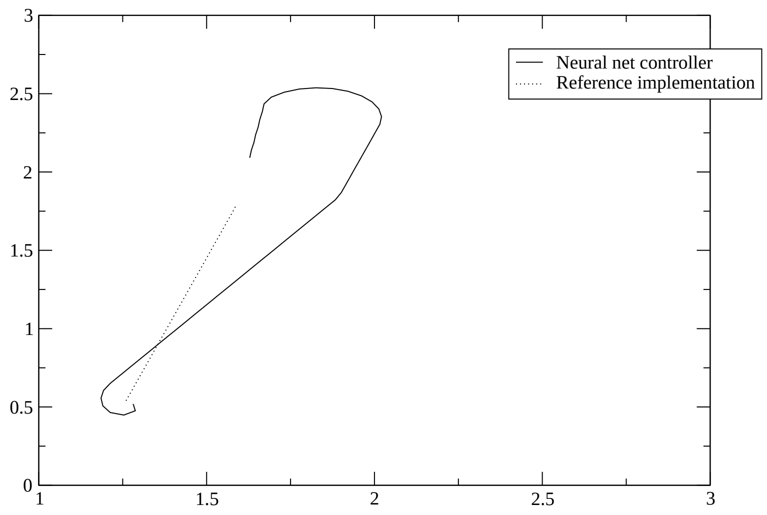


Figure 6.10: Scenario C: trace of path taken by robot

From the angle plot (Figure 6.11), we see the robot begins with a sharp turn until it is almost aligned with the ball, then moves straight for a time. As this path is oriented slightly away from the ball, after some time the distance to the ball begins to increase. Again, a shortly after the distance to the ball has begun to increase the robot turns again until it finds the ball and holds this path to completion.

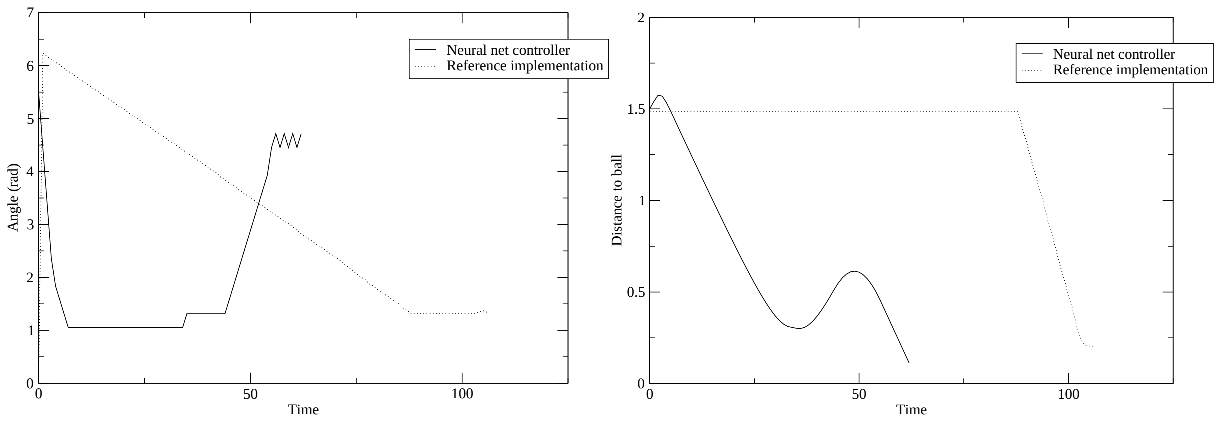


Figure 6.11: Scenario C: plots of robot facing angle and distance to ball against time

6.6.4 Scenario D

Scenario D is a similar situation to Scenario C, but this time the path taken by the robot is near-optimal. The robot begins with the ball ahead of it, but outside the camera's field of vision 6.12.

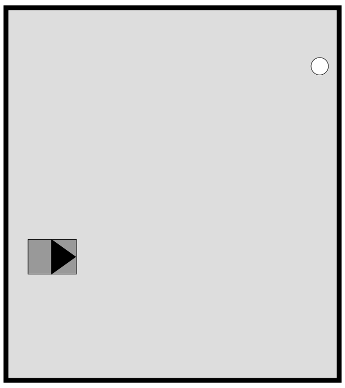


Figure 6.12: Scenario D: initial position and orientation

A trace of the robot's path is shown in Figure 6.13.

Plots of the robot's facing angle and distance to the ball over time are given in Figure 6.14. They show that the robot initially turns around until it has oriented itself to be approximately facing the ball then begins to move towards it. It holds this course until near the end, where it reorients again to move in the direction of the ball. Although the path varies slightly from the shortest path that could be taken, it is close enough that the difference in fitness observed by the GA using our fitness weightings would be minimal.

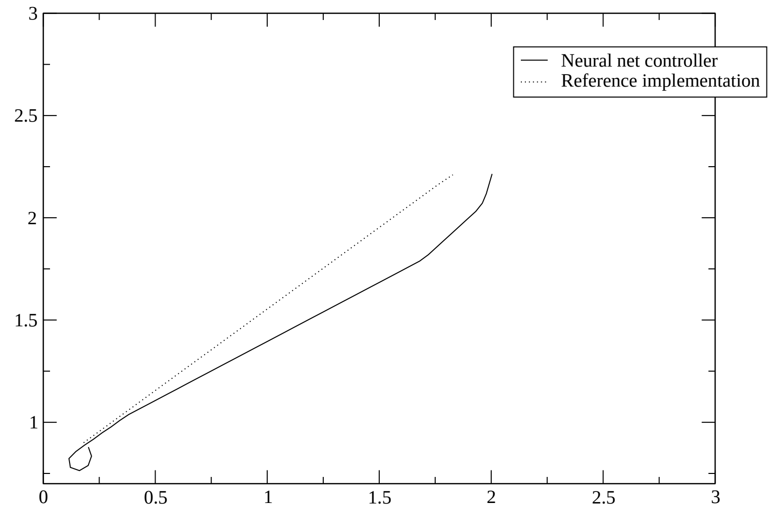


Figure 6.13: Scenario D: trace of path taken by robot

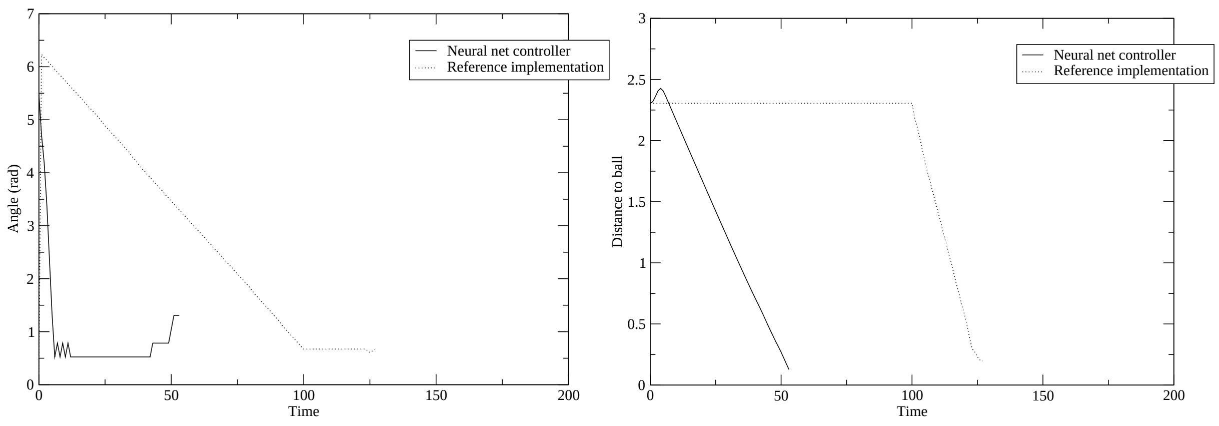


Figure 6.14: Scenario D: plots of robot facing angle and distance to ball against time

6.6.5 Scenario E

Scenario E illustrates an instance of erratic behaviour from the neural network controller with minimal disruption to the fitness value obtained. The robot begins as shown in Figure 6.15, with the ball to the right of it but outside the visible range of the camera.

The robot's path is traced in Figure 6.16. The trace shows the robot circling and then moving in the general direction of the ball. As in scenarios B and C, the actual path taken by the robot varies from the ideal linear path. We see the robot makes a little turn towards the ball. At this point, instead of moving towards the target the robot performs a complete revolution and begins on an incorrect trajectory, before turning around again and reaching the ball. From the angle and distance plots (Figure 6.17) we can see the

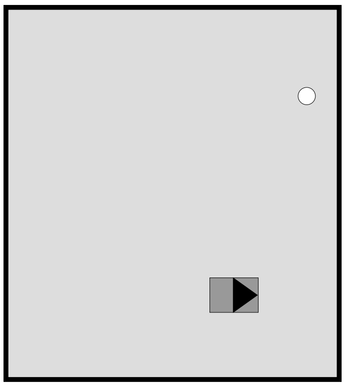


Figure 6.15: Scenario E: initial position and orientation

time taken for this manoeuver is relatively small and so does not have a great effect on the fitness value returned; however the behaviour is undesirable because it reaches the ball indirectly when it had two clear opportunities to get there directly.

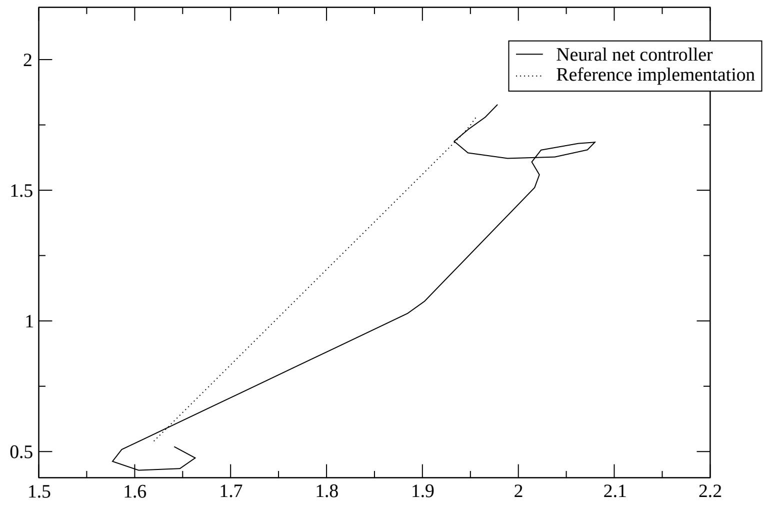


Figure 6.16: Scenario E: trace of path taken by robot

6.6.6 Scenario F

Scenario F is an example of atypical behaviour that has not been sufficiently penalised by the genetic algorithm to be removed from the population. The robot begins with the ball almost directly to its left (Figure 6.18).

Figure 6.19 shows a trace of the robot's convoluted path to the ball, and plots of the robot facing angle and distance to ball over time are given in Figure 6.20.. From it's

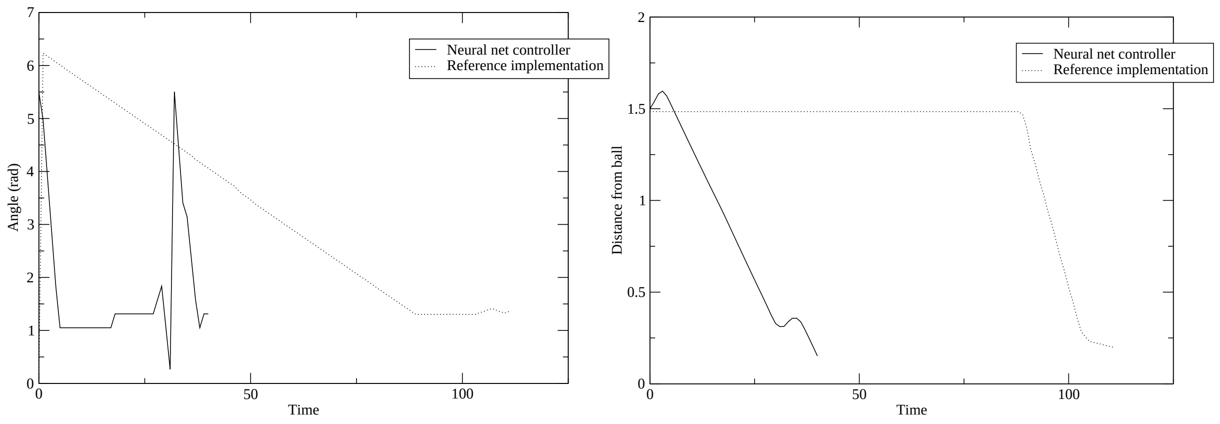


Figure 6.17: Scenario E: plots of robot facing angle and distance to ball against time

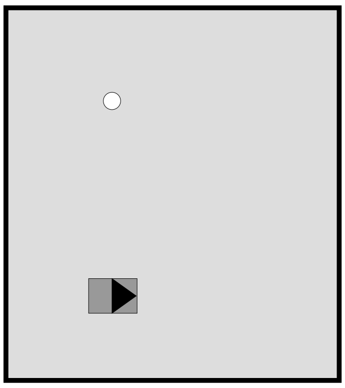


Figure 6.18: Scenario F: initial position and orientation

initial position it begins to move towards the ball, then enters a circling motion moving sideways across the field. It leaves this motion to move straight for a while in a direction away from the ball, then makes a 90 degree turn after some time. This is continued until it makes another turn and begins to move in the direction of the ball, before beginning another along circling pattern. This continues for some time until it finally stops turning and moves directly to the ball.

There are two contributing factors that could explain why this behaviour has not been removed from the population. Firstly, although the robot displays behaviour that is not helpful towards the goal of ball finding, the set task is eventually completed. It is consequently awarded a positive fitness value and not removed from the population in the fifth stage of evolution (**??**). Secondly, the behaviour exhibited here is the relatively infrequent case; in an averaged set of random trials, there is a higher probability that a better fitness value will be rewarded, and so offset the poorer fitness produced by this

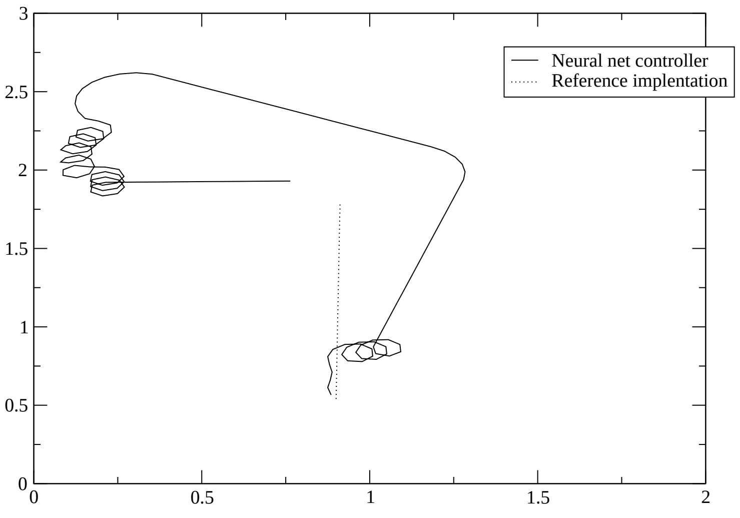


Figure 6.19: Scenario F: trace of path taken by robot

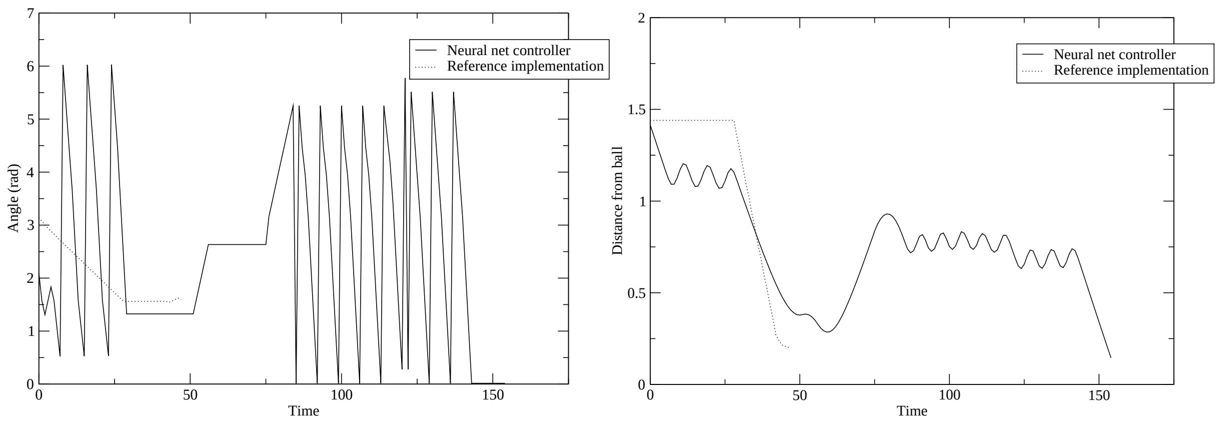


Figure 6.20: Scenario F: plots of robot facing angle and distance to ball against time

controller.

Chapter 7

Conclusion

Summary

From the initial system specification, we have developed a software platform that can be used with EyeSim or the Eyebot hardware environment to create behaviour-based robot control programs. A set of working behaviours was also implemented and have been tested to operate correctly for the situations they were designed for. A graphical utility to aid the composition has been written but needs to be developed further to produce complete self-contained programs, and for use with alternative behavioural controllers.

General neural network and genetic algorithm class libraries have been written that may be adapted for use in any program requiring these constructs. In our adaptive controller application we used these components to create a system that evolved the structure of a neural net behavioural controller. The controller took input from sensory schemas, the last behaviours (representative of state) of the robot and a time value. The outputs of the controller determined the activation levels of concurrently modelled behaviours. We set up a number of staged trials to produce a program capable of finding a ball inside a closed environment within a restricted time period. These trials were conducted on the Eyesim robot simulator, which required some debugging and development in preparation of the experiment.

The resulting controller was evolved over a series of stages, using the principle of behavioural memory. Fitness optimization of the controller was successful and produced a controller able to perfom the required task capably for most cases. However, there were observed cases where the robot deviated significantly from the behaviour expected of it. An analysis of results from the evolved controller suggests that a stricter fitness criteria needs to be applied to produce a more robust ball-finding implementation. The fitness functions currently used allow imperfect configurations to progress through genetic algorithm evolutions and be awarded high fitness values, at the possible expense of more suitable controllers.

Future directions

The existing code base may be used as the basis for other behavioural based projects in future. The current range of behaviours available is small and could benefit from the addition of new modules. Expanding the collection of simple behaviours would enable the construction of programs suitable for a variety of problem domains.

Presently only a neural network controller has been implemented for use with the behavioural framework. There is no consistent controller interface defined that is comparable to the schema API. Other controllers must be written specifically for a desired application and manually incorporated with the reactive schemas. A specific way to define controllers could be used to aid the creation of controllers by generator programs, and allow different high level controllers to be used with the same sets of reactive controllers. This would aid comparison between different controller types and facilitate creation of a completely code-free design environment.

The neural net controller for the ball finding task has considerable scope for improvement. The main concern is the erratic performance exhibited when it is placed in certain scenarios. Refinement of the fitness function that rewards high performing controllers appears to be the most straightforward way to deal with this. An alternative is to break down the program into smaller component behaviours that could produce the final behaviour required. These would then require some kind of sequencing mechanism between the independent behavioural controllers to control their activations at appropriate times.

Bibliography

[1] P. Agre, D. Chapman, "What are plans for?", *Robotics and Autonomous Systems*, vol. 6, No. 1-2, pp17-34, 1990

[2] R.C. Arkin, T. Balch, "AuRA: Principles and Practice in Review", *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, No. 2-3, pp175-189, 1997

[3] R.C. Arkin, *Behaviour Based Robotics*, Cambridge, MA: MIT Press, 1998

[4] R.C. Arkin, "Navigational Path Planning for a Vision-based Mobile Robot", *Robotica*, Vol. 7, pp49-63, 1989

[5] T. Balch, *Multiagent Reinforcement Learning*, PhD thesis, College of Computing, Georgia Institute of Technology, 1998

[6] T. Balch et al, TeamBots simulation environment available from http://www.teambots.org

[7] D. Beasley, D.R. Bull, R.R. Martin, "An Overview of Genetic Algorithms : Part 1, Fundamentals", *University Computing* vol. 15, No. 2, pp58-69

[8] D. Beasley, D.R. Bull, R.R. Martin, "An Overview of Genetic Algorithms : Part 2, Research Topics" *University Computing*, vol. 15, No. 4, pp170-181

[9] V. Braitenberg, *Vehicles, experiments in synthetic psychology*, Cambridge, MA: MIT Press, 1984

[10] R. Brooks, "A Robust Layered Control System For A Mobile Robot", *IEEE Journal of Robotics and Automation*, vol. 2, No.1, pp14-23

[11] R.J. Clark, R.C. Arkin, A. Ram, "Learning Momentum: On-line Performance Enhancement for Reactive Systems", in *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, May 1992, pp111-116

[12] L. Davis, *Handbook of Genetic Algorithms*, New York, NY: Van Nostrand Reinhold, 1991

[13] K. DeJong, *The Analysis and behaviour of a Class of Genetic Adaptive Systems*, PhD thesis, University of Michigan, 1975

[14] H. de Garis, "Genetic Programming : Building Nanobrains with Genetically Programmed Neural Network Modules", in *Proceedings of the International Joint Conference on Neural Networks*, July 1990

[15] H. Gleitman, *Psychology*, W.W. Norton & Co., 1991

[16] K. Gurney, *Neural Nets*, UCL Press Ltd, 2002

[17] I. Harvey, P. Husbands, D. Cliff, "Issues in Evolutionary Robotics", in *From animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior* (J.A. Meyer, S.Wilson Eds.), Cambridge MA: MIT Press, 1993

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

[19] L.P. Kaelbling, M. Littman, A.W. Moore, "Reinforcement Learning: A Survey", Journal of Artificial Intelligence 4, pp237-285

[20] J. Koza, *Genetic Programming*, Cambridge, MA: MIT Press, 1992

[21] C.R. Kube, H. Zhang, "Collective Robotics: From Social Insects to Robots",

[22] C.R. Kube, H. Zhang, X. Wang, "Controlling Collective Tasks With an ALN", IEEE/RSJ IROS pp289-293, 1993

[23] C.J. Lee, B.D. Reger, M.C. Tresch, J.E. Colgate, F.A. Mussa-Ivaldi, "Emulation of biological motor primitive in an artificial system: The generation of static force fields", in *Proceedings of the ASME Dynamics Systems and Control Division*, no.67, pp897-905, 1999

[24] D. MacKenzie, J. Cameron, R. Arkin, "Specification and Execution of Multiagent Missions", in *Proceedings of the International Conference on Intelligent Robotics and Systems (IROS '95)*, Pittsburgh, PA, pp51-58

[25] E. Martinson, A. Stoytchev, R. Arkin, *Robot Behavioral Selection Using Q-learning*,

[26] H. Moravec, *Mind Children: The Future of Robot and Human Intelligence*, Cambridge, MA: Harvard University Press, 1988

[27] F.A. Mussa-Ivaldi, S.F. Giszter, E. Bizzi, "Linear combinations of primitives in vertebrate motor control", in *Proceedings of the National Academy of Sciences*, no. 91, pp7534-7538, 1994

[28] P. Nordin, W. Banzhaf, "An On-Line Method to Evolve Behavior and to Control a Miniature Robot in Real Time with Genetic Programming", *Adaptive Behaviour*, Vol. 5 no. 2, pp107-140, 1997

[29] A. Ram, R.C. Arkin, G. Boone, M. Pearce, "Using Genetic Algorithms to Learn Reactive Control Parameteres for Autonomous Robotic Navigation", *Journal of Adaptive Behaviour*, vol. 2, No. 3, pp277-305

[30] L.M. Saksida, "Shaping Robot Behavior Using Principles from Instrumental Conditioning", Robotics and Autonomous Systems, 22(3/4) pp231

[31] M. Schoenauer, "Evolutionary Computation and Applications at Centre de Mathématique Appliquées de l'Ecole Polytechnique", [Online document], 1997 Mar., [2002 Oct 17], Available http://citeseer.nj.nec.com/schoenauer97evolutionary.html

[32] W.M. Spears and K. DeJong, "An analysis of multi-point crossover", in *Foundations of Genetic Algorithms* (G.J.E. Rawlins, Ed.). Morgan Kafumann, 1991

[33] L. Steels, R. Brooks, "Building Agents out of Autonomous Behaviour Systems", in *The Artificial Life Route to AI: building embodied, situated agents* (L. Steels, R. Brooks, Eds.), Hillsdale, NJ: Erlbaum Associates, 1995

[34] L. Steels, "Mathematical analysis of behavior systems", *Proceedings From Perception to Action Conference*, Lausanne: IEEE Computer Society Press, 1994

[35] G. Syswerda, "Uniform crossover in genetic algorithms", in *Proceedings of the Third International Conference on Genetic Algorithms*, (J.D. Schaffer, Ed.), chapter 21, pp332-349, Van Nostrand Reinhold, 1991

[36] P. Vershure, J. Wray, O. Sprons, G. Tononi, G. Edelman, "Multilevel Analysis of Classical Conditioning in a Behaving Real World Artifact", *Robotics and Autonomous Systems*, Vol. 16 1995, pp247-265

[37] A. Waggershauser, *Simulation of small mobile robots*, Master thesis, University of Western Australia

[38] X. Yao, *Evolutionary Artificial Neural Networks*, International Journal of Neural Systems Vol.4 No. 3 pp203-222, 1993