INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

AN OBJECT-ORIENTED DESIGN OF A SUBSUMPTION ARCHITECTURE

Nadia Andrea Gantchev

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

September 2001
© Nadia Andrea Gantchev, 2001

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-68515-2

Canada

Abstract

An Object-Oriented Design of a Subsumption Architecture

Nadia Andrea Gantchev

The hardware subsumption architecture for robots as developed by Rodney Brooks is implemented in software in an object-oriented way and used for strategies of trucks in the Truckin' simulation game.

The subsumption architecture is a layered mediator invented by Rodney Brooks for behaviour-based control of robots. The layers are minimally dependent and use minimal communication. We develop an object-oriented software design for the subsumption architecture, and demonstrate that each layer can be used as a slot for a set of plug-and-play components that implement different micro-strategies for achieving a particular goal. Guidelines for the development of specific layers and components of a subsumption architecture are also presented.

Acknowledgements

I would like to thank my supervisor, Dr Gregory Butler, for his patience and valuable guidance.

Helpful discussions were had with other members of the Truckin' project, especially with Dr Peter Grogono, Debbie Papoulis and Jeff Edelstein, and with Jeff Allen from University of Maine.

This work was supported in part by a Seagram Innovative Research grant from Concordia University.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Subsumption architecture was originally proposed by Rodney Brooks in 1986, providing a new architecture for the control of mobile robots. Since then, applications of the architecture has been primarily to mobile robots or simulations of mobile robots.

Many intelligent systems such as agents, knowledge-based systems, planners, and adaptive software need to be able to integrate and reuse a variety of decision-making components. Today most of these systems are developed in software, although there is still strong interest in autonomous robots that have such capabilities.

The subsumption architecture is a layered mediator invented by Rodney Brooks for behaviour-based control of robots. Behavior-based systems consist of task-oriented modules implementing domain-specific solutions with representations de-emphasized, and control decentralized. In subsumption, each layer of behavior includes as a subset the behaviour of the lower layers. There is a fixed priority arbitration scheme to handle conflicts between layers, and a layer may suppress or inhibit lower layers. The layers are minimally dependent and use minimal communication. We develop an object-oriented software design for the subsumption architecture, and demonstrate that each layer can be used as a slot for a set of plug-and-play components that implement different micro-strategies for achieving a particular goal. Furthermore, a set of guidelines for the development of specific layers and components of a subsumption architecture is also presented.

This thesis is an application of subsumption architecture in a new context: genetic programming. Specifically, we will present an object oriented design of a player in the game called Truckin' modeled on Brook's subsumption architecture.

1.1 Our Context

The software architecture, and the reuse of micro-strategy components, is validated by developing truck agents within the Truckin' simulation game. The game is played in a simulated country. In this country, there are three kinds of dealer: producers, retailers, and consumers who trade in a single commodity. The producers, retailers, and consumers have fixed locations and cannot trade directly. They rely on trucks to ship goods from one place to another. Trucks can buy, sell, move, or make phone calls to gather information. Trucks can move around the country, using gas. The country contains several gas stations at which trucks can buy gas. A truck that does not trade will eventually not have enough money to buy gas. A truck without gas is stuck and can do nothing. The objective of the game is for retailers and trucks to trade in such a way that there is a steady flow of goods from producers to consumers. The winning truck is the one with the most capital at the end of the game. (The original Truckin' game was invented by Mark Stefik and others at Xerox PARC as part of their research into expert systems.)

The development of truck agents for Truckin' demonstrates that the subsumption architecture provides a well-designed structure for strategies, and that micro-strategy components can be reused as plug-and-play components within the subsumption architecture.

The design characteristics drawn from the context are :

1. The experimental objective of OOT requires a design that is reuseable and extendable.

2. Genetic programming requires a design that manages interchangeable components.

3. The analysis of genes is through the analysis of behavior; a design should capture the relationship between genes and behavior.

1.1.1 The Truckin' Game

The Truckin' Project is part of a research program coordinated by Peter Grogono and Greg Butler. The goal of the research is to explore ways in which object oriented software can dynamically evolve by means of genetic programming. Object Oriented

Truckin' was developed to serve as a framework for developing programs that adapt to their environment. It is loosely based on a game invented at the Xerox Palo Alto Research Center by Mark Stefik and others during the eighties [3]. Object Oriented Truckin' models a country in which commodities are distributed by trucks. Trucks negotiate with dealers to buy and sell commodities. The simulation can be viewed as a competition between trucks, where the winner at end of a simulation is the truck that was most successful at buying and selling commodities. The objective of running the simulation is to evolve trucks that are increasingly successful at trading.

The game is played in a simulated country. In this country, there are three kinds of dealer: producers, retailers, and consumers who trade in a single commodity. It doesn't really matter what the commodity is, but it should be something for which there is a steady demand, such as BMWs or spinach.

A producer produces the commodity in large quantities, called crates, and will sell any amount in exchange for payment. Retailers buy crates, split them up into smaller quantities called units, and sell the units to consumers. The incentive for a retailer to trade is a price difference: for example, producers might sell crates containing 100 units at $100/crate, and consumers might buy units at $2/unit. Producers and consumers do not need an incentive: they simply produce and consume as requested.

The producers, retailers, and consumers have fixed locations and cannot trade directly. They rely on trucks to ship goods from one place to another. A truck can buy crates from a producer and sell them to a retailer, or it can buy units from a retailer and sell them to a consumer, or it can do both.

Trucks can move around the country, using gas. The country contains several gas stations at which trucks can buy gas. A truck that does not trade will eventually not have enough money to buy gas. A truck without gas is stuck and can do nothing.

The objective of the game is for retailers and trucks to trade in such a way that there is a steady flow of goods from producers to consumers. Since the game is simulated by computer, the actual objective is to write code for a retailer or truck. The retailer or truck that has the best performance in a simulation is the winner of the game.

1.1.2 Evolutionary Programming

Darwin's theory of evolution through natural selection is the underlying rationale for genetic programming. Darwin argued that a population of a given species includes individuals of varying characteristics. The population of the next generation will contain a higher frequency of those types that most successfully survive and reproduce under the existing environmental conditions. Thus the frequencies of various types within a population will change over time [4]. Variety among individuals in a population is key, otherwise no amount of selective reproduction will affect the composition of the population. The mechanism that provides variation is the gene. Genes control the development of an individual, and variations in the genes cause variation in the individuals of a population. Therefore it is important that we devise a genetic description of a truck that allows for variety in the expression of genes, and permits sensible genetic recombination: the product of recombination should be a functioning truck.

There is a very close link between genes and behavior. Often when we describe a gene we are in fact describing a trait or behavior. This is because the trait or behavior is the observable effect of the interaction of the gene with the environment, the phenotype of the organism's genotype. If we know which genes are responsible for a specific behavior than the effect of varying these genes are apparent through changes in the behavior. In Truckin' the evolution of trucks is based on profitability, the outward effect of trading.

1.1.3 Brooks' Subsumption Architecture

In the mid-eighties researchers in artificial intelligence began developing a new approach to designing mobile robots. Robots based on traditional AI approach were not able to operate in real time in a real world. They relied on a perfect internal representation of the world which can not be achieved in a dynamic unpredictable environment, and the focus on "depth" search to provide solutions was not timely enough in an environment where quick reaction is critical. Rodney Brooks argued that internal world models that are complete representations of the external world were unnecessary for robots to act in a competent manner. According to Brooks the world is its own best model [6]. Inspired by ethology and biology, he observed that actions of a robot are separable and that coherent intelligence could emerge from the

interaction of independent reactive sub components with the environment [7]. Brooks aimed to build robots with intelligence on the scale of insects as the first step towards building robots with higher intelligence [4]. Brooks' approach to building insect-like robots is in many ways orthogonal to the approach taken earlier. Instead of a top-down, centrally controlled system built around an internal world model, Brooks' system is built bottom-up, has distributed control, with the components interfacing directly with the world (see Figure 1). This approach is referred to as behavior-based robotics.

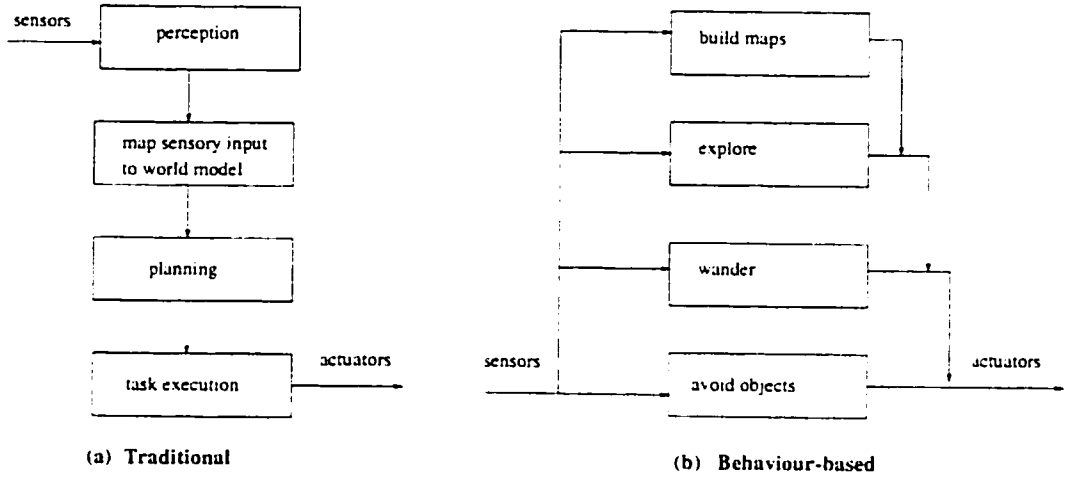

Figure 1: Traditional Approach vs Behaviour-based Approach

Brooks described the subsumption architecture as an instance of behavior-based robotics used to build robots that operate in the real world [11]. It is a framework from which to build behavior-based robots. Maes [20] argues that the behavior-based approach is appropriate for the class of problems that require a system to autonomously fulfill several goals in a dynamic, unpredictable environment. She gives examples of applications, such as virtual actors, process scheduling, interface agents to name a few, where the behavior-based approach could be applied. While Maes argues for the generality of the behavior-based approach, Bryson [13] suggests that subsumption architecture can serve as a general framework to develop behavior-based systems.

Rodney Brooks points out that traditional AI had difficulty with the integration of multiple sensor devices, achievement of multiple goals, robustness, and extensibility

when it came to systems for control of autonomous mobile robots [7]. The subsumption architecture was developed to address those difficulties. The fundamental ideas of the subsumption architecture are a decomposition into layers of task-achieving behaviors, followed by an incremental composition through debugging in the real world [6].

Brooks proposed a decomposition of an autonomous intelligent system based on desired external manifestations of the system. The decomposition resulted in a collection of simpler independent behaviors which when composed produced more complex behavior. Each behavior should achieve a task that is in some way observable. A set of behaviors together provide the robot with some level of competence. The behaviors should be designed so that as new behaviors are added to the system the level of competence of the system increases. A set of behaviors that produces a level of competence is referred to as a layer, and the process of increasing the level of competence by adding new behaviors to existing sets of behaviors is called layering [4]. Each layer connects its own sensing to action and is not dependent on any other layer to decide what it should do. The layers operate in parallel with minimal communication.

The overall system is robust and extensible. Multiple distributed layers of behavior mean there is less chance that the system will collapse given some drastic change in the world. Each layer has its own sensors to monitor the world, by sensing the environment often enough, it is able to decide on the appropriate goal to pursue in light of the current environment. The layers are able to make timely adjustments to their goals in response to changes in the world. New layers are added to the system without changing the original system, all that is required is to interface the new layer to the existing system.

Brooks' computational model is organized as an asynchronous network of augmented finite state machines, with a fixed topology of unidirectional connections. Each layer of control is a finite state machine with some instance variables, and input/output lines that can send and receive typed messages. The connections between layers are predefined wires which allow higher layers to suppress and replace input, or inhibit the output of lower layers. The messages sent over connections are small numbers. The meanings of the numbers are determined by the designers of the sender and receiver, and are dependent on the state of both the sender and receiver. The layers operate asynchronously, each layer outputs actuators in response to its own

sensory information. There is a fixed priority arbitration scheme to handle conflicts that occur when more than one layer produces actuators at the same time. Under this scheme only one layer has control of the robot's effectors at a time. All data is distributed over many computational elements, and there is no central locus of control.

The three key ideas [11] introduced in the subsumption approach above are

1. Improvements in performance come about by incrementally adding more situation specific circuitry while leaving old circuitry in place, able to operate when new circuitry fails to operate. Each additional collection of circuitry is referred to as a new layer, and each new layer produces some observable behavior in the system interacting in the environment.

2. Keep each added layer as a short connection between perception and actuation.

3. Minimize the interaction between layers.

1.2 Contribution of the Thesis

We develop an object-oriented software design for the subsumption architecture, and demonstrate that each layer can be used as a slot for a set of plug-and-play components that implement different micro-strategies for achieving a particular goal. The software architecture uses a communication backplane that acts as a short-term memory, and as a communication channel, to provide the feedback from the "real world" to the layers.

The design for the subsumption architecture is implemented in C++. It is incorporated into a C++ implementation of the Truckin' simulation game.

The software architecture, and the reuse of micro-strategy components, is validated by developing truck agents within the Truckin' simulation game.

Furthermore, a set of guidelines for the development of specific layers and components of a subsumption architecture is presented.

This work has led to a conference paper [14] and a journal paper [15].

1.3 Organization of the Thesis

Chapter 2 presents the background on the Truckin' simulation game, evolutionary programming, and the subsumption architecture. Chapter 3 presents the object-oriented design for the subsumption architecture. Chapter 4 concludes with a discussion of the validation of the design by developing truck agents within the Truckin' simulation game, and with a presentation of a set of guidelines for the development of specific layers and components of a subsumption architecture.

We assume the reader is familiar with object-oriented modeling as in OMT [26], or in the *Unified Modeling Language* (UML) [3], and with the features of the draft ANSI standard C++ programming language [28].

Chapter 2

Background

2.1 The Truckin' Experimental Workbench

Object Oriented Truckin' (OOT) is a framework for developing programs that adapt to their environment. It is loosely based on a game invented at the Xerox Palo Alto Research Center by Mark Stefik and others during the eighties. The goal of the design of OOT is to provide an environment that is sufficiently complex to provide interesting behaviour and yet simple enough to achieve such behaviour with modest programming effort.

OOT models a country (OOTland) in which commodities are distributed by trucks. Trucks negotiate with dealers to buy and sell commodities. The game can be seen as a competition between trucks and dealers to maximize their profits.

OOT is implemented in C++. The simulation code contains a base class TRUCK from which all other truck classes must be derived. The objective is to build an adaptive truck. The evolution of truck strategies is the primary rationale of the simulation. Running the simulation for an extended period of time should yield trucks of increasing sophistication.

2.1.1 Features of Object Oriented Truckin'

2.1.1.1 Topography

OOTland is a square country with a grid of highways. *Avenues* run north-south and *streets* run east-west. All highways allow traffic to travel in both directions. All events take place at intersections of the grid. In the current version of the simulation, there are 10 avenues, 10 streets, and 100 intersections.

A *place* is determined by two coordinates: an avenue number and a street number.

A *step* is a move between adjacent intersections.

2.1.1.2 Dealers

At each highway intersection, there is a *dealer* who trades in a particular commodity. The commodities that are traded are: NONE, CRATES, ITEMS, and GAS.

There are five types of dealers:

- A dealer who trades in NONE will not buy or sell anything.
- A **Producer**: a dealer that produces and sells CRATES, a bulk commodity. Crates are produced at a fixed rate by the producer. The selling price is fixed, and it is the same for all producers.
- A **Consumer**: a dealer that buys and consumes ITEMS, small quantities of a bulk commodity. Items are consumed at a fixed rate by the consumer. The buying price is fixed, and it is the same for all consumers.
- A **Retailer**: a dealer who buys CRATES, converts crates to items, and sells ITEMS. The retailer has a limited storage capacity to store crates and items. Each retailer sets its own buying price for crates, and selling price for items.
- A **Gas-station**: a dealer who trades in GAS. A gas-station sells gas but does not buy it. Quantities are unlimited: gas-stations do not run out of gas.

Trucks transport bulk goods from producers to retailers and small quantities from retailers to consumers. The rate of flow of commodities through the system is controlled by consumers, who attempt to consume at a fixed rate (they may fail to receive all that they want, but they never buy in excess of their requirements). The retailer sets a buying price for crates, and a selling price for items. In most cases, retailers will set the unit buying price of crates less than the unit selling price of items to profit by trading. Prices vary across the country, however. Truckers and retailers are motivated by differentials between buying and selling prices.

Table 1 shows the commodities and flows. Crates are shipped by trucks from producers to retailers, who unpack the items from the crates. The individual items are shipped by trucks from retailers to consumers.

Commodity	Produced by	Consumed by
Crate	producer	retailer
Item	retailer	consumer
Gas	gas station	truck

Table 1: Commodities and Flows

2.1.1.3 Trucks

Trucks travel around the country trading with dealers. At the start of the simulation, each truck has a certain amount of money (its *capital*), and a certain amount of gas. The truck attempts to increase its capital by trading. At the end of the simulation, the winner is the truck with the most capital.

A truck can obtain information, travel along the highways, and trade. Each of these activities consumes resources: time, money, and gas.

2.1.2 Features of the Simulation

The simulation program models the country and its features, as described above, and includes a number of instances of classes derived from the base classes TRUCK and

DEALER. These instances are referred to as "trucks" and "dealers".

Trucks and dealers do not have direct access to the data structures representing OOTland. To prevent cheating, all of their actions are mediated by two other classes. Associated with every truck, there is an instance of class CONTROL (instances are "controllers"); trucks obtain information and perform actions by sending messages to their controller. Similarly, there is an instance of class MANAGER (instances are "managers") associated with each dealer, and dealers can send messages only to their managers. The controllers and managers ensure that all actions are consistent with the rules of the simulation.

2.1.2.1 Time

The simulation time increases in steps of 10 minutes. The 10-minute intervals are called *time slots* or simply *slots*.

2.1.2.2 The Referee

The *referee* is in charge of the simulation. The referee sends a message to each truck and each manager at the beginning of each time slot. The referee also informs each controller of the current simulation time.

2.1.2.3 The Map

The unique object of class MAP represents the highway system. At each highway intersection, there is a manager and a dealer. Many of the dealers are "default" dealers who trade only NONE. Trucks cannot access the map directly but can obtain information about it from their controllers. Similarly, dealers can obtain information from their managers.

2.1.2.4 Controllers

The referee passes the simulation time to each controller at the beginning of each time slot. All other messages to a controller come from its truck. The controller screens all of the truck's actions, verifying that the truck has sufficient resources to pay for the action before either executing the action or forwarding it to the appropriate object.

2.1.2.5 Trucks

At the beginning of each time slot, each controller sends the message PLAY() to its truck. During the time slot, the truck can perform any of the actions listed below.

- A truck can obtain the time since the simulation started, the time remaining in the current slot, and the simulation time remaining. It can also obtain its current position, its current capital, its current stock of each commodity, and information about the dealer at the current position. All of this information is provided without cost to the truck.

- A truck can make a telephone call to another intersection to obtain the commodity, buying price, and selling price of the dealer there. A telephone call *fails* if the commodity traded at the intersection is NONE and *succeeds* otherwise. A successful telephone enquiry lasts 3 minutes and cost $3; an unsuccessful enquiry lasts 1 minute and costs $1.

- A truck can move any number of steps in a single direction. Moving one step consumes 6 minutes and 1 litre of gas.

- A truck can attempt to buy or sell from the dealer at its current position. The dealer must honour its advertised buying and selling prices but is not obliged to exchange the quantity requested.

If the actions require more than 10 minutes, the truck loses some of the next time slot. For example, a truck may choose to travel two steps, which requires 12 minutes. The journey would require all of the current time slot and 2 minutes from the next time slot.

The controller is the truck's interface to OOTland. All of the truck's request are sent to the controller. The controller ignores requests from a truck that does not have the resources required. For example, a truck that has exhausted its capital is not allowed to spend money; a truck that has run out of gas cannot travel; and a truck that has used up its time slot cannot do anything that consumes time. It is the responsibility of the truck to ensure that it has the resources required to perform a task and to check that its request achieved the desired effect.

2.1.2.6 Managers

Each manager monitors the action of a dealer. Managers receive messages from controllers. Managers ensure that dealers trade honestly and maintain positive capital and stock.

2.1.2.7 Dealers

A dealer is given an initial buying price, selling price, and stock. A dealer can change the buying and selling prices during the simulation. For example, a dealer with excessive stock might raise its buying price and lower its selling price.

2.1.2.8 Rules and Scoring

OOTland has 10 avenues and 10 streets with intersections 10 kilometres apart. Trucks drive at 100 km/h and consume gas at the rate of 10 litres per 100 kilometres. They therefore require 6 minutes and 1 litre of gas to travel between intersections. The truck starts with a full tank of gas containing 50 litres.

Money is measured in cents. Each truck starts with 50,000 cents ($500.00). Gas-stations may set their own prices, but a typical price for gas would be $1/litre.

The actions of trucks and dealers are restricted by the interfaces of controllers and managers. The simulation runs for a certain time that is announced at the beginning of the run and can be obtained by a truck. When this time has elapsed, the simulation

is stopped and the assets of each truck and dealer are recorded. The only asset of a truck is its capital. Commodities on the truck, including gas, have a zero value.

2.2 Evolutionary Programming

Evolutionary programming addresses the problem of how to generate computer programs automatically. It provides a methodology to design systems that generate computer programs that improve as they experience the data on which they are trained. It is part of a larger area of research called machine learning, the study of computer algorithms that improve automatically through experience [2]. Evolutionary programming models Darwin's theory of evolution through natural selection to evolve computer programs.

Darwin argued that a population of a given species includes individuals of varying characteristics. He considered evolution to be a process that creates a match between the species and the environment. Organisms with traits better suited to the environment have a greater chance of surviving and will leave more offspring, as result the species becomes better adapted to the environment over time. The relative probability of survival and rate of reproduction is called Darwinian fitness.

Genetic variation drives the evolutionary process, otherwise no amount of selective reproduction will affect the composition of the population. There are two main sources of genetic variation : mutation, and recombination. Mutation is the source of genetic variation, but due to a relatively low rate of mutation in nature, it does not drive evolution. Genetic recombination, which creates variation much faster than mutation, pushes evolution forward.

Evolutionary programming mimics aspects of natural evolution, natural selection, and reproduction. In evolutionary programming a quality criterion is defined and then used to measure and compare candidate solutions in an stepwise refinement of a set of data structures. A near optimal individual is located after a number iterations. Solutions are represented by genotypes, genomes, or chromosomes. The quality criterion is often referred to as fitness and it is with this standard that individuals are

selected for reproduction.

Variation in evolutionary programming comes from two operators: mutation and recombination. The mutation operator generates a random change to a parameter in the solution. If applied to every parameter, the result is a completely new solution. This operator ensures that different aspects of the problem are tried. The recombination operator preserves parameter values of discovered solutions, as well as introducing variation. It recombines the parameter values of two solutions forming a different solution. The variation operators are normally applied to the most fit solutions at each iteration. The driving force of simulated evolution is referred to as fitness-based selection.

The steps that make a complete run of evolutionary programming :

1. Create a random population of programs.
2. Evaluate each program assigning a fitness value according to a pre-specified fitness function which measures the ability of the program to solve the problem.
3. Using a predefined reproduction technique copy existing programs into the new generation.
4. Genetically recombine the new population with the crossover function from a randomly chosen set of parents.
5. Repeat steps 2 onwards for the new population until a pre specified termination criterion has been satisfied or a fixed number of generations has been completed.
6. The solution to the problem is the program with the best fitness within all generations.

2.2.1 Genetic Programming

2.2.1.1 Tree-Based Genetic Programming

Genetic programming, a variant of evolutionary programming, was established as a method for automatic programming by J. Koza. He used a tree structure as the program's representation in a genome, a crossover operator for genetic recombination,

and fitness-proportional selection to solve a variety of problems [18].

The structures that undergo evolution are hierarchical computer programs based on LISP-like symbolic expressions. The size, shape, and structure of the solution is left unspecified and is found by the genetic programming operators. Solving a problem is a search through all possible combinations of symbolic expressions defined by the programmer.

The creation of a program is a combination of the domain dependent symbolic expressions. The symbolic expressions are divided into two sets, a terminal set and a function set. The terminal set includes all zero argument functions and constants, while the function set contains all functions with one or more arguments. Each expression must be evaluated without error for all possible arrangements of expressions, and some combination of expressions must be sufficient to solve the problem.

Consider the tree representation of the LISP expression (+ (* 5 (- 6 1)) (* 5 5)). The terminal set in this example is 1, 5, 6, the function set is +, -, *, and the search space is all programs that can be composed recursively using the elements of the two sets. The standard convention for execution of this type of structure is to repeatedly evaluate the leftmost node for which all inputs are available, referred to as postfix order.

The two primary operators for modifying programs undergoing adaptation are fitness-based reproduction and crossover. Koza did not introduce mutation. Reproduction is an asexual operator, which takes one parent program and produces one offspring. The result is a copy of the parent. The crossover operator combines the traits of two parents and produces at least one offspring. The parents are chosen from the population based their relative fitness.

The crossover operation begins by randomly and independently choosing a node in each parent. The entire sub-trees rooted at the chosen nodes are than swapped. In this example two offspring result from the crossover.

2.2.1.2 Features of Genetic Programming

Since Koza introduced tree-based genetic programming, many other genetic programming systems have been developed. The systems vary in genome representation, and in the rates of crossover and mutation. Despite the variety of systems developed, they features in common.

Features of genetic programming [2]:

- Assembles a population of variable length program structures from basic units. The actual assembly of the programs from basic units occurs at the beginning of a run when the population is initialized.
- Uses genetic operators to transform programs in the population. Crossover between two individual programs is a principal genetic operator. Other operators include mutation, and reproduction.
- Simulates evolution by means of fitness-based selection. Fitness-based selection determines which programs are selected for further improvements.
- Uses pseudo-random numbers to mimic the randomness of natural evolution.

2.3 Subsumption: An Instance of Behavior-Based Robotics

In the mid-eighties researchers in artificial intelligence began developing a new approach to designing mobile robots. Robots based on traditional AI approach were not able to operate in real time in a real world. They relied on a perfect internal representation of the world which can not be achieved in a dynamic unpredictable environment, and the focus on 'depth' search to provide solutions was not timely enough in an environment where quick reaction is critical. Rodney Brooks' argued that internal world models that are complete representations of the external world were unnecessary for robots to act in a competent manner. According to Brooks the world is its own best model [6]. Inspired by ethology and biology, he observed that actions of a robot are separable and that coherent intelligence could emerge from the

interaction of independent reactive sub components with the environment [7]. Brooks aimed to build robots with intelligence on the scale of insects as the first step towards building robots with higher intelligence [4]. Brooks' approach to building insect-like robots is in many ways orthogonal to the approach taken earlier. Instead of a top-down, centrally controlled system built around an internal world model, Brooks' system is built bottom-up, has distributed control, with the components interfacing directly with the world. This approach is referred to as behavior-based robotics.

Brooks described the subsumption architecture is an instance of behavior-based robotics used to build robots that operate in the real world [11]. It is a framework from which to build behavior-based robots. Maes [20] argues that the behavior-based approach is appropriate for the class of problems that require a system to autonomously fulfill several goals in a dynamic, unpredictable environment. She gives examples of applications, such as virtual actors, process scheduling, interface agents to name a few, where the behavior-based approach could be applied. While Maes argues for the generality of the behavior-based approach, Bryson [13] suggests that subsumption architecture can serve as a general framework to develop behavior-based systems.

2.3.1 The Behavior-Based Approach

In traditional AI , the system is decomposed along functional modules such as perception, execution, planner, inference engine. The modules are developed independently and rely on some central representation as their means to interface. The modules are modeled to be as domain independent as possible to facilitate module reuse across domains. The control structure is sequential with each module taking its turn to process the internal representations. Normally the perception module updates the internal model, then planning produces a plan, finally an execution module executes the plan. The behavior of the systems emerges from the interaction of the functional components.

In contrast, the new behavior-based approach, decomposes the system into task-achieving behaviors. The modules communicate directly with very simple messages, or indirectly through the environment; by changing some aspect of environment a

module may trigger another module. The modules are specifically designed for a task in a given environment. Each module is responsible for doing all the representation, computation, execution necessary to carry out its task, and is free to employ completely different techniques and representations. The modules operate in parallel, each one independently produces commands in response to its particular view of the world. There is a simple arbitration method to select or fuse the commands produced by the modules. Functionality, like planning, emerges from the interaction among behavior modules and the environment.

The behavior-based approach concentrates on modeling systems that are situated in both space and time, reducing the need to build an internal world model. The space or environment can be used as an external memory for reminding the system what has been done or what it must do, or it has particular characteristics that the system can exploit. Situated in time means that the system must react in a timely fashion and deal with interrupts, but it also allows for the construction of an iterative, incremental solution to the problem.

Behavior-based AI takes advantage of the interaction dynamics between the system and the environment, and between different components within the system. It is often possible, based on the properties of the environment, to find an interaction loop, a set of feedback or reflex mechanisms that will produce the desired behavior. As a consequence, a relatively simple system can operate in a complex environment. Simple interaction dynamics of components within the system can also lead to emergent behavior, as a result inter-module communication and dependence can be minimized and control can be distributed among the modules.

Behavior-based systems consist of task-oriented modules implementing domain-specific solutions with representations de-emphasized, and decentralized control. The systems tend to react quickly because they have fewer layers of information to process, are distributed, non-synchronized, and require less computation. They are able to deal with unforeseen situations since they rely on the environment for information and as a determiner of what to do, rather than on a possibly outdated or incomplete internal model. They are robust because all modules are equally critical and they incorporate

redundant methods.

2.3.2 Behavior-Based Control Systems

An architecture provides a set of principles for organizing control systems and a set of constraints on the way control problems can be solved. There are four basic approaches to autonomous agent control: planner-based, reactive, hybrid, and behavior-based.

Traditional AI systems are planner-based systems. They rely on an internal central world model and a reasoning engine to generate a sequence of actions. Sensory information is fused into the internal model, the planner then works on this model to determine which goals should be fulfilled, which are than translated into a sequence of actions. The plans and goals of the system are explicitly defined. The behaviors emerge from the interplay of the planner, goals of the system, and the world model. The internal model is necessarily as complex as the external world making the task of modeling very challenging for complex environments. An environment that changes often means that the system must frequently replan slowing reaction time to the extent that the agent does not respond to all changes in the world. Planner-base systems are not workable for complex dynamic environments.

Reactive systems consist of a collection of preprogrammed condition-action pairs with minimal internal state. They do not have internal models and do not perform searches. They apply a simple mapping of stimuli and appropriate responses. The system links sensing directly to action, and rely on fast feedback from the environment to ensure that the appropriate action is executed. The behaviors are the emergent property of this kind of system. The system encodes an action for every input state, the designer is responsible to account for all possible input states, as a consequence these systems do not scale well to complex environments.

Hybrid systems combine reactive and planner-based approaches in one system. Usually a reactive system is built to handle low-level real-time control issues, and a planner is built to handle higher-level decision making. The result is a control

system that is composed of two communicating independent parts: a reactive process to take care of survival and a planner process to select action sequences. The reactive component's actions can change the world or the state of the agent, as a consequence the planning component must be able to replan and recover from interrupts.

Behavior-based systems are an extension of reactive architectures and often fall between reactive and planner-based extremes [24]. Behavior-based systems use various forms of distributed representations and distributed computations, unlike planner-based systems which have centralized representation and computation, and reactive systems which are limited to lookup and execution of simple functional mappings. In these systems the behaviors are explicitly defined and higher-level activities such as planning and goal setting emerge from the interaction of the behaviors in the environment.

The general constraints imposed on behavior-based systems are a decentralized control structure of behavior producing modules that interact primarily through the world and not internally. The behaviors should be relatively simple but with more time-extended capability than a reactive rule, and they should be designed to be incrementally added to the system.

The organization methodology of behavior-based systems concerns the coordination of many behaviors functioning in parallel, making behavior-arbitration the most challenging part of the design [24]. The problem of behavior-arbitration is, given a set of behaviors outputting some actions, which ones of those should be given priority, or how should their outputs be combined. In the subsumption architecture, for example, behavior-arbitration is based on a built-in, fixed control hierarchy imposing a priority ordering on behaviors. Other methods involve selecting a behavior or a set of behaviors to activate based on voting schemes or spreading of activation.

2.3.3 Brooks' Subsumption Architecture for Robots

Rodney Brooks points out that traditional AI had difficulty with the integration of multiple sensor devices, achievement of multiple goals, robustness, and extensibility

when it came to systems for control of autonomous mobile robots [7]. The subsumption architecture was developed to address those difficulties. The fundamental ideas of the subsumption architecture are a decomposition into layers of task-achieving behaviors, followed by an incremental composition through debugging in the real world [6].

Brooks proposed a decomposition of an autonomous intelligent system based on desired external manifestations of the system. The decomposition resulted in a collection of simpler independent behaviors which when composed produced more complex behavior. The organization of the decomposition followed two principles [4]: One concerned individual behaviors, the other with sets of behaviors. Each behavior should achieve a task that is in some way observable. A set of behaviors together provide the robot with some level of competence. The behaviors should be designed so that as new behaviors are added to the system the level of competence of the system increases. A set of behaviors that produces a level of competence is referred to as a layer, and the process of increasing the level of competence by adding new behaviors to existing sets of behaviors is called layering [4]. Each layer connects its own sensing to action and is not dependent on any other layer to decide what it should do. The layers operate in parallel with minimal communication.

The overall system is robust and extensible. Multiple distributed layers of behavior mean there is less chance that the system will collapse given some drastic change in the world. Each layer has its own sensors to monitor the world, by sensing the environment often enough, it is able to decide on the appropriate goal to pursue in light of the current environment. The layers are able to make timely adjustments to their goals in response to changes in the world. New layers are added to the system without changing the original system, all that is required is to interface the new layer to the existing system.

Brooks' computational model is organized as an asynchronous network of augmented finite state machines, with a fixed topology of unidirectional connections. Messages sent over connections are small numbers whose meanings are dependent on the dynamics designed into both the sender and receiver. Each layer of control is a finite state machine with some instance variables, and input/output lines that

can send and receive typed messages. The layers are connected by predefined wires which allow higher layers to suppress and replace input, or inhibit the output of lower layers. The layers operate asynchronously, each layer outputs actuators in responds to its own sensory information. There is a fixed priority arbitration scheme to handle conflicts that occur when more than one layer produces actuators at the same time. Under this scheme only one layer has control of the robot's effectors at a time. All data is distributed over many computational elements, and there is no central locus of control.

The first three layers of behavior defined by Brooks for the robot Allen [5]

1. Avoid obstacles.
2. Wander aimlessly around without hitting things.
3. Explore the world by seeing places in the distance that look reachable and heading for them.

Each layer of behavior includes as a subset the earlier layers of behavior. Exploring includes the ability to wander without hitting things, wandering without hitting things includes the ability to avoid contact with objects. This permits layers to be built incrementally beginning with lowest layer on up. The design of a layer can rely on the presence of successful operational earlier layers. The layers do not call on one another explicitly, instead their reliance is implicit. The Wander layer does not have to worry about avoiding obstacles because there is an operational Avoid Obstacles layer that successfully ensures that obstacles are avoided.

The lowest level layer of control (avoid obstacles) was implemented and completely debugged before adding a higher layer. This layer results in a robot that avoids collision with objects. The robot moves away from approaching objects, and halts before colliding with stationary objects. When the next level of control (wander) is added the robot moves in a random direction every few seconds. The Wander layer suppresses the heading produced by the Runaway module of the Avoid Obstacles layer. In fact the Avoid module combines the two headings resulting in a heading that points in the direction specified by the Wander module, but avoids any obstacles.

The Wander layer subsumes the Avoid Obstacles layer when it suppresses the output of the Runaway module.

The Explore layer looks for corridors of free space then moves the robot towards the free space. The Whenlook module of the Explore layer looks for a corridor of open space whenever it detects that the robot has been idle for a few seconds. It inhibits the Wander layer so it can take some pictures and process them without wandering away. The Avoid Obstacles layer continues to operate, ensuring that no objects collide with the robot. Once a free corridor is found a heading is sent to the Avoid module suppressing any heading that may have been produced by the Wander module. The Wander layer in turn suppresses the Runaway module of the Avoid Obstacles layer. The Explore layer subsumes the Wander layer whenever it inhibits or suppresses the Wander layer.

The three key ideas introduced in the subsumption approach above are [11]

- Improvements in performance come about by incrementally adding more situation specific circuitry while leaving old circuitry in place, able to operate when new circuitry fails to operate. Each additional collection of circuitry is referred to as a new layer, and each new layer produces some observable behavior in the system interacting in the environment.
- Keep each added layer as a short connection between perception and actuation.
- Minimize the interaction between layers.

2.3.4 Implementations using the Subsumption Architecture

The subsumption architecture resulted in the first mobile robot capable of a navigating in a dynamic world. Since then various implementations of mobile robots using this architecture have shown that the subsumption approach can incorporate non-reactive competencies, planning and goal setting, and learning about representations in the world. Different methods of behavior activation have also been tried, demonstrating that the approach works as more behaviors are added, indicating that the approach scales to more complex systems. Since the introduction of this architecture,

it has been used primarily in robotics whether as purely subsumption systems or hybrid systems consisting of a reactive subsumption component coupled with a symbolic planner. More recently it has been used as a control structure for applications outside of robotics. Software applications operate under different technological constraints than those of mobile robots, as a consequence the computational models for software agents can be different from that used by Brooks for mobile robots. One software application incorporates a knowledge base as part of a layer in the architecture. What is important is that the computational model used results in layers that react in a timely fashion to changes in the environment.

2.3.4.1 Allen

The first implementation of the subsumption architecture was the robot Allen [5] developed by Rodney Brooks at the MIT Artificial Intelligence Laboratory. This robot, described in 2.3.3, is almost entirely reactive. The lowest layer uses sonar readings to keep away from moving obstacles while not colliding with stationary obstacles. The highest layer is non-reactive since it selects a goal to head towards, and then move towards the goal while the reactive layer avoids obstacles. The robot demonstrates that the subsumption architecture can combine reactive and non-reactive capabilities using the same sort of computational mechanism for both.

2.3.4.2 Herbert

The second robot, Herbert [16], illustrates that the external world can serve as the only medium for inter-module communication and integration. Herbert wanders around looking for soda cans, picks one up, and brings it back to where it started from. Herbert uses a laser scanner to find soda cans, proximity sensors to navigate by following walls and going through doorways, a magnetic compass for global position, and arm sensors to pick up cans. Herbert does not maintain internal state longer than three seconds and there is no communication between behavior generating modules. Each module is connected to sensors on the input side, and a fixed priority arbitration on the output side of the modules determines the actuators to pass on to the robot's

effectors. The integration of behaviors and communication between modules is carried out through the environment. For example, the soda can object finder moves the robot so that its arm is lined up with a soda can, the arm behavior which monitors the wheels, notices that the robot is not moving which triggers its activation. When the arm locates the soda can, it moves the hand so that the two fingers line up on either side of the can, breaking an infrared beam between the fingers. The grasp reflex is triggered whenever this beam is broken causing the hand to grasp the can. Modules never explicitly pass any information to, or call upon any other module, integration of behavior and communication results from a module changing the environment, which in turn causes another module to react.

2.3.4.3 Toto

Toto [21] demonstrates that subsumption systems can make plans, and have goals without central representations or symbolic representations. Toto explores its environment, builds a map as it explores, and carries out path planning. The map is an active decentralised structure which does the computations necessary for path planning.

Toto wanders around office environments building a map based on landmarks. The primitive layers of control let Toto wander around following boundaries. A layer that detects landmarks runs in parallel. This layer informs the Map layer whenever it detects a landmark. The map itself is a topological network of processes corresponding to landmarks in the environment. Each process has its own rules and state. A landmark is described by a type, a compass heading, and its position relative to other landmarks. When a landmark is detected, it is broadcast to all the processes in the map. If none recognizes it, it is added to the map, otherwise the process that recognized the landmark becomes the robot's current position on the map. Planning is accomplished by distributing the goal through the map network. The user selects a goal location whose associated landmark becomes activated. Activation is spread to its neighbors, and propagated throughout the graph in the form of a spreading wave front [21], estimating total path length. The activation eventually arrives at the current position on the map with a recommendation of the direction to travel to

follow the shortest path to the goal. This scheme can have multiple active goals, the robot will head towards the nearest one.

2.3.4.4 Attila

A complex behavior repertoire requires some mechanism which integrates many competing behaviors. This mechanism should be decentralised, non-manipulable, and not have central control or representation to meet the criteria of the subsumption architecture. Brooks introduces such a mechanism in [8], where a hormonal activation scheme is used to integrate multiple behaviors of the mobile robot Attila. The hormone system can be viewed as low band width global communication system, where the release of hormones and activation of behaviors are local. It provides a global repository of state which predisposes appropriate behaviors to be active, and a way to switch behaviors on or off as the global state changes.

Attila is a six legged planetary explorer robot intended to operate for many days without any external commands. Its task is to manage its internal needs such as recharging its batteries from solar power, shut itself down in a low energy overnight mode, wander around in local exploratory mode, recovers from falls, do long traverses to get to new areas, carry out measurements with scientific instruments, and choose interesting views to digitize and radio back to a relay station. Behaviors which process sensory information are able to excite a condition in the hormone system which contributes to the level of a hormone. The hormone level at any point in time is a function of the current levels of excitement of some conditions. The excitation level of a condition decays by a programmer defined rate. Each behavior has an activation level which is a function of hormone levels, and which causes the behavior to become active when it passes some threshold. The hormone levels together reflect the global state of the robot, as conditions in the environment change, so do the hormone levels, causing a different set of behaviors to become active.

2.3.4.5 Sumpy

Sumpy is a non-robotics implementation of the subsumption architecture. It is a software agent that lives in and helps to maintain a UNIX file system for better disk space utilization [27]. One of Sumpy's layers utilizes a fuzzy controller employing a knowledge base consisting of fuzzy inference rules, and an inference engine. The layer supplies the fuzzy controller with input values, and receives a Boolean solution from the controller.

The Sumpy senses its environment by issuing UNIX commands and noting the responses. Each layer issues sensory commands for its own purposes, and all layers can note the response. Sumpy has four layers: wander around the file system, compress files as needed, backup files as needed, and put Sumpy to sleep. The Wanderer layer moves Sumpy randomly through the file system. It issues a "pwd" command to know where it is, a "ls" command to find a directory to go to, and "cd" command to change to that directory. The Compressor layer notes the result of the "pwd" command issued by the Wanderer layer, when Sumpy is in a new directory the Compressor layer inhibits the Wanderer layer, and looks for files that need to be compressed. The Compressor layer makes use of a built-in fuzzy controller which determines whether to compress a file or not. The Compressor selects a file and gathers information for the fuzzy controller. The controller returns a signal to the Compressor, telling it whether or not to compress the file. When all files in the directory have been checked, the compressor uninhibits the Wanderer. The Backup layer also notices when Sumpy is in a new directory causing it to compare the new directory with its list of recently serviced directories. If the new directory is not on the list, Backup suppresses Compressor and inhibits Wanderer, then proceeds to backup files in the directory. When completed the Backup layer unsuppresses Compressor and uninhibits Wanderer. The highest layer, Sleepy, checks the CPU load regularly. When the load passes a certain threshold, it puts Sumpy to sleep by suppressing all three lower layers. When Sleepy notices that the CPU load has fallen, it unsuppresses the lower layers.

2.3.4.6 The Reactive Accompanist

The Reactive Accompanist is another non-robotics example of a subsumption software. It is a software "folk musician" developed by Joanna Bryson. The software accompanies unfamiliar melodies in real time without knowledge of music theory or any form of rule base [13]. The goal of the software is to derive chord structure from a melody in real time.

Bryson proposed the following levels of competencies for her system:

1. pitch recognition — transforms notes to pitches.
2. chord recognition — transforms pitches to chords.
3. time recognition — transforms chords to timed sequences of chords.
4. structure recognition (not implemented)
5. song recognition (not implemented)

The Note module is a neural network that transforms the input melody to a weighted array representing pitches. The output units correspond to the pitches that the net was trained on. The Chord module uses a neural network of predefined chords to transform the input pitches to chords. All the other modules are modeled on finite state machines.

Chapter 3

Object-Oriented Design of the Subsumption Architecture

This chapter explains the design of the software architecture for subsumption and presents the validation of the reusability of micro-strategies within the Truckin' simulation game. The design follows the OMT notation [26] and was implemented in C++[28].

3.1 Overview

The software version of the subsumption architecture preserves the essence of subsumption:

- a layered architecture,
- a fixed priority scheme between layers,
- the ability of a layer to inhibit lower layers, and
- the use of the "real world" as the source of feedback on the consequences of actions.

The major differences between the software architecture and the hardware architecture for subsumption are the sequential nature of the trucks and the lack of "sensors" to the "real world" since a truck is embedded in an existing Truckin' simulation framework.

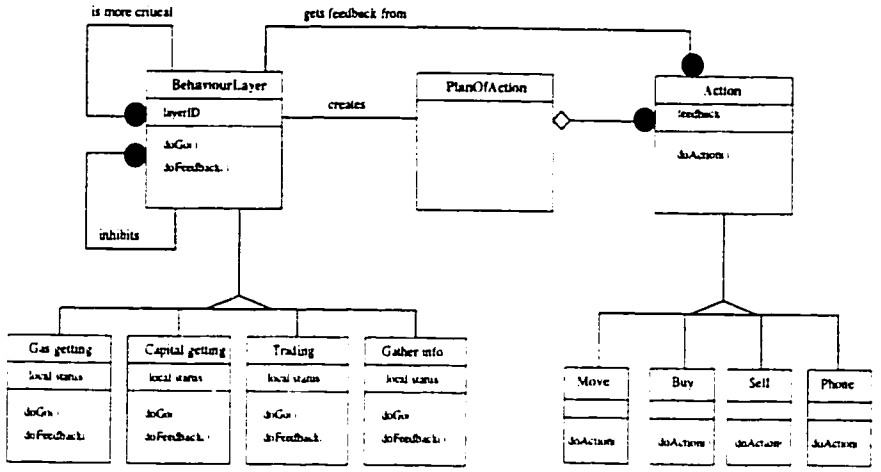

Figure 2: Conceptual View

The major concepts, see Figure 2, are the following:

- A BEHAVIOURLAYER is concerned with one particular set of behaviours and subgoals. The BEHAVIOURLAYERs are linearly ordered in a fixed priority, where a BEHAVIOURLAYER subsumes the behaviour of the BEHAVIOURLAYER below it, and may inhibit all BEHAVIOURLAYERs below it.

- A PLANOFACTION is generated by a layer in order to make progress towards its subgoals.

- An ACTION is one of the primitive steps available to the agent, and are the components of a PLANOFACTION.

In the Truckin simulation, trucks can perform a combination of four actions, namely move, buy, sell, or phone for information. Each action has associated costs in terms of money, time, capital, and/or gas. The layers are associated particular subgoals, namely

1. Gas Getting, in order to retain the ability to move;

2. Capital Maintenance, in order to make a profit from the current trade with the local dealer;

3. Trading, in order to maximize overall profit from trades over the entire game; and

4. Gather information, in order to have knowledge of distant positions in the simulation world.

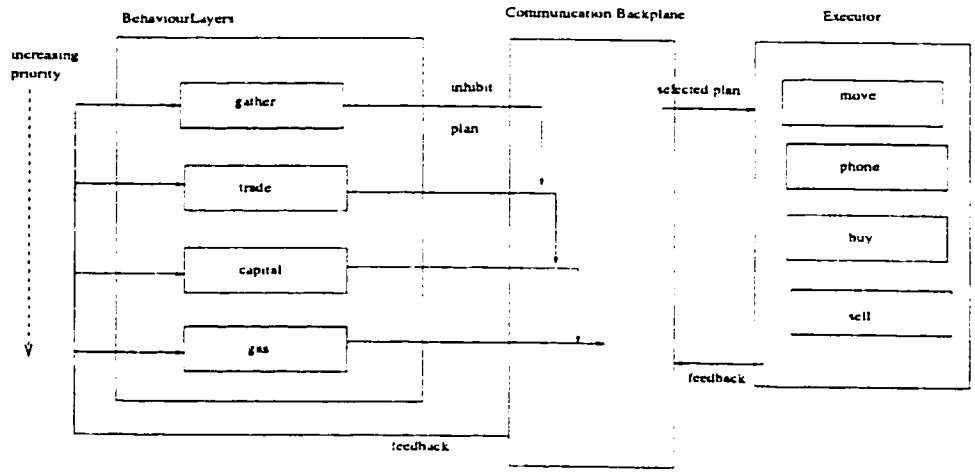

Figure 3: Overview of Architecture

The major architectural design decision is the use of a communication backplane. The backplane acts as a short-term memory, and as a communication channel, to provide the feedback from the "real world" to the layers. As short-term memory, the backplane stores the proposed plans of action and the inhibition information. With the fixed priority scheme, this memory allows the mediated selection of the current course of action. This selection is also recorded by the backplane and allows each layer to acquire the necessary feedback about the state of the world following the execution of the selected actions. Once the feedback has been acquired, the short-term memory is cleared.

The use of a communication backplane leads to a two phase behaviour of the layers: first obtain feedback, then propose a plan of action (and optionally inhibit lower layers).

3.2 Major Classes

Besides the classes representing the main concepts, BEHAVIOURLAYER, ACTION, and PLANOFACTION, there are classes for the control and feedback mechanisms, namely SELECTOR, EXECUTOR, and COMMUNICATIONBACKPLANE. These mechanisms utilise the CONTROLDATA class.

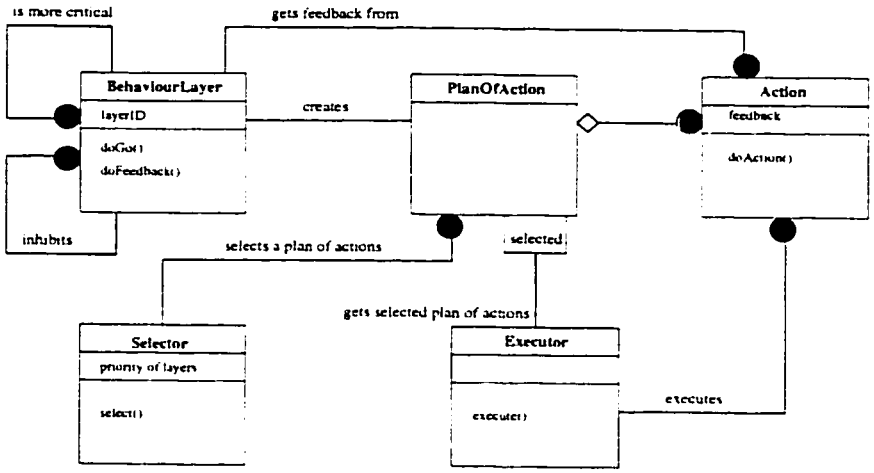

Figure 4: Object Model of the Design

A BEHAVIOURLAYER is concerned with one particular set of behaviours and subgoals. Associated with a BEHAVIOURLAYER is a *layerID* that determines its rank in the linear order of priorities, and information about a BEHAVIOURLAYER is usually indexed by its *layerID*. The behaviour of a BEHAVIOURLAYER is done in two phases: first to gather feedback from the "real world" using *doFeedback()*, and second to determine a plan of action using *doGo()*. The method *doGo()* may inhibit all BEHAVIOURLAYERs below it. (These are the ones with higher priority.) A BEHAVIOURLAYER subsumes the behaviour of the BEHAVIOURLAYER below it.

A PLANOFACTION is generated by a layer in order to make progress towards its subgoals. In the detailed design, this class does not exist; rather, it is a collection of actions stored within a CONTROLDATA object.

An ACTION is one of the primitive steps available to the agent. In the Truckin' simulation, trucks can perform a combination of four actions, namely move, buy, sell, or phone for information. Each action has associated costs in terms of money, time, capital, and/or gas. An ACTION is given effect by calling the *doAction()* method.

The COMMUNICATIONBACKPLANE implements the associations amongst layers themselves, and the associations amongst layers and the SELECTOR and the EXECUTOR. The COMMUNICATIONBACKPLANE acts as a short-term memory, and as a communication channel, to provide the feedback from the "real world" to the layers. As short-term memory, the backplane stores the proposed plans of action and the inhibition information: for each layer this is stored in a CONTROLDATA object. The

SELECTOR uses the fixed priority scheme and the inhibition information to select the current course of action, and records this selection in the COMMUNICATIONBACKPLANE. This allows each layer to acquire the necessary feedback about the state of the world following the execution of the selected actions. Once the feedback has been acquired, the short-term memory is cleared.

The SELECTOR is responsible for interpreting the fixed priority scheme, as defined by the *layerID* of the layers, and for interpreting the inhibition information in the COMMUNICATIONBACKPLANE in order to select a plan of action. The SELECTOR also controls the deliberations of the subsumption layers. The method *select()* controls the two phase behaviour of layers by (1) calling *doFeedback()* for each layer, (2) clearing the short-term memory of the COMMUNICATIONBACKPLANE, (3) calling *doGo()* for each layer, and (4) selecting the plan of action.

The EXECUTOR is responsible for executing the selected plan of action. Its *execute()* iterates over the actions in the plan, and calls the *doAction()* method of the action.

The CONTROLDATA class is responsible for storing a plan of action and whether that plan has been inhibited or not. A CONTROLDATA object is associated with each layer and is internal to the COMMUNICATIONBACKPLANE.

3.3 Behavioural Description

We describe the behaviour of the subsumption architecture in a top-down fashion, first explaining how the behaviour of the Truckin' framework leads to calls to the *play()* method of TRUCK, and then how *play()* utilises the subsumption architecture. Last we describe the internal behaviour of an individual BEHAVIOURLAYER.

The outcome of a Truckin' simulation is determined by a truck's performance in a *competition* with other trucks. Each competition consists of a series of *games*. Games vary in the starting position of trucks, and in the position (and kind) of dealers in the simulated country, but all games have the same participating trucks. A game is played for a certain number of *rounds* where first the dealers play and then the trucks play. The order amongst dealers and amongst trucks is random from one round to the next, however all dealers play before any truck plays. When it is a truck's turn to play, the Truckin' framework calls the *play()* method of the truck's CONTROLLER.

which in turn calls the *play()* method of the truck.

All interactions between a truck and the Truckin' framework go through the truck's CONTROLLER, which guarantees that the truck obeys the rules of the game, and correctly calculates usage of resources.

A truck plays a turn within a given time slot. A truck can do several actions within that time. Using the subsumption architecture, a truck repeatedly calls the *select()* method of the SELECTOR followed by the *execute()* method of the EXECUTOR until the time slot expires. Figure 5 shows a sequence diagram of this behaviour and Figure 6 an extended object diagram with pseudocode.

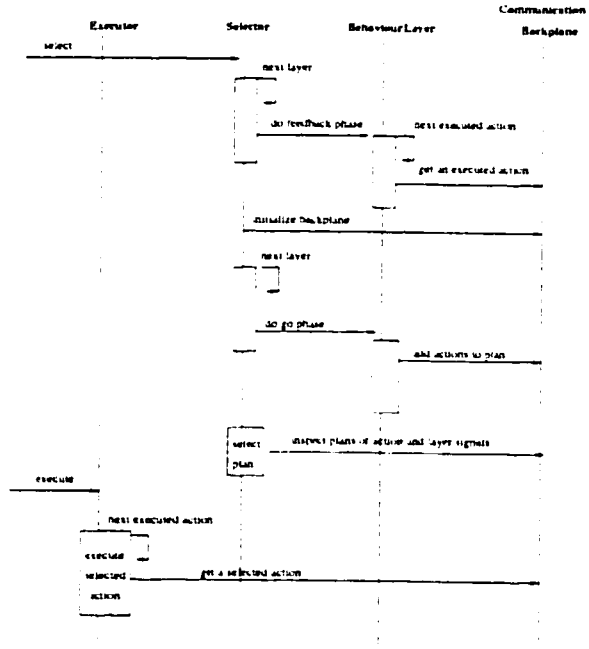

Figure 5: Dynamic Model of Architecture

The SELECTOR requests each layer to perform its feedback phase by calling *doFeedback()*. The SELECTOR then clears the short-term memory of the COMMUNICATIONBACKPLANE. Then the SELECTOR requests each layer to determine its plan of action by calling *doGo()*. Finally, the SELECTOR uses the priority scheme and the available inhibition information in the COMMUNICATIONBACKPLANE to flag one of the plans as the selected plan of action.

During *doFeedback()*, a layer requests from the COMMUNICATIONBACKPLANE the plan of action that was selected previously. During *doGo()*, a layer creates a plan of action and communicates that plan to the COMMUNICATIONBACKPLANE. Optionally,

a layer may also communicate inhibit signals to the COMMUNICATIONBACKPLANE.

The EXECUTOR fetches the selected plan of action from the COMMUNICATION-BACKPLANE, and requests that each action be executed by calling *doAction()*, which in turn calls the corresponding method of the truck's CONTROLLER.

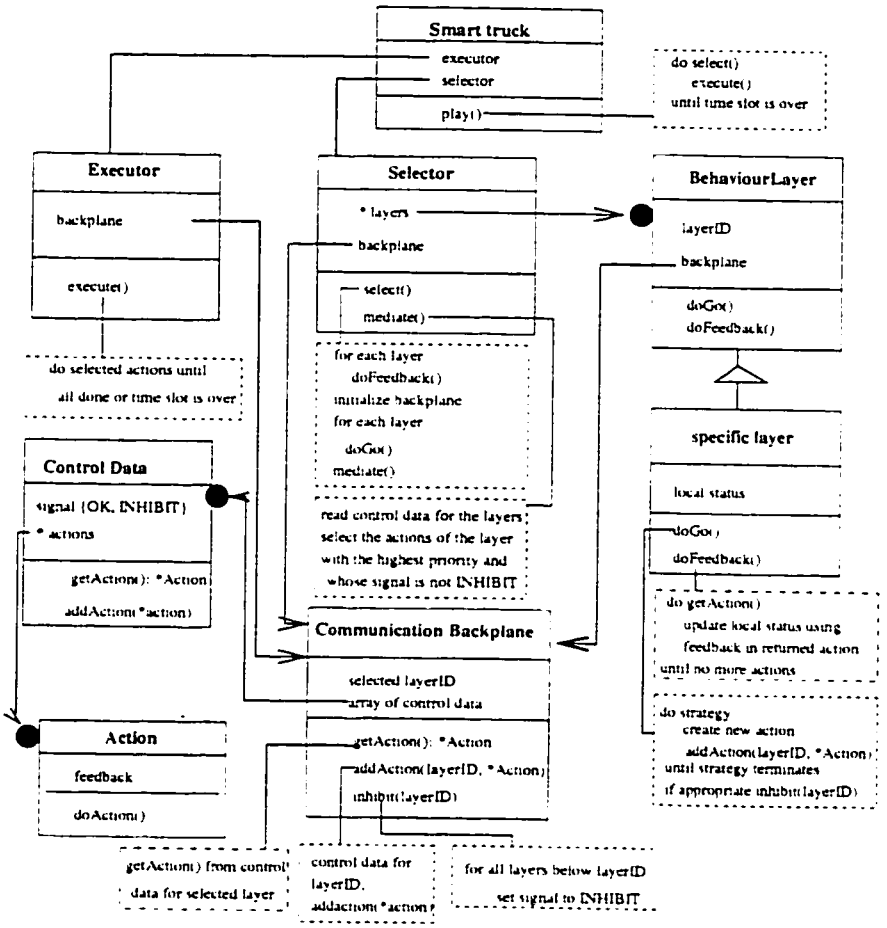

Figure 6: Behavior of Subsumption

Chapter 4

Conclusion

The subsumption architecture is a layered mediator for behavior-based control of robots. In subsumption, each layer of behavior includes as a subset the behaviour of the lower layers, there is a fixed priority scheme to handle conflicts between layers, and a layer may suppress or inhibit lower layers.

We develop an object-oriented software design for the subsumption architecture, and demonstrate that each layer can be used as a slot for a set of plug-and-play components that implement different micro-strategies for achieving a particular goal. The software architecture uses of a communication backplane that acts as a short-term memory, and as a communication channel, to provide the feedback from the "real world" to the layers. Furthermore, a set of guidelines for the development of specific layers and components of subsumption architecture is presented.

The software architecture, and the reuse of micro-strategy components, is validated by developing truck agents within the Truckin' simulation game.

4.1 Validation of Design using Truckin'

Within the Truckin' project we explore the use of genetic algorithms for developing robust, reliable components. In this context, a gene is represented by a layer, and the alleles (or specific manifestations of the gene) by the different instances of a layer. The genetic algorithm experiments consider many different combinations of alleles; each combination defines a truck. The natural selection amongst trucks is done through a Truckin' competition, and the "fittest" trucks are used to breed the next generation

of trucks.

For each layer we develop five components as instances of strategies. The strategies range from very basic to moderate in complexity. While most strategies are designed as state machines, they are not implemented as such. These are not strategies for the overall goal of winning the game, but strategies for achieving the subgoal(s) of one behaviour layer, so we call them *micro-strategies*.

A micro-strategy is a C++ subclass. It is derived from one virtual base class, the LAYER class. The LAYER base class contains the data and interface required by a behavior layer to function properly in the subsumption architecture. It is the wiring required by a micro-strategy to plug into the subsumption architecture. A subclass of the LAYER class is a specialized subsumption layer coded to execute a specific micro-strategy. It is completely dedicated to executing its strategy. All micro-strategies are subclasses of the LAYER class.

A truck's gene set is determined by the behavioral decomposition of the truck. We associate each layer of behavior with a gene. Our truck was decomposed into the following four layers of behavior: "Gas getting", "Capital getting", "Trading", and "Information gathering". The chromosome of a truck is a fixed length string. Each entry of the string is a gene in the gene set. In our case the chromosome is a string with four entries, the first entry represents the "Gas getting" gene, the second entry represents the "Capital getting" gene and so on. The value stored within each entry maps to a specific strategy from the population of strategies for the gene represented by that entry of the chromosome string. We produced a unique reference to each strategy within a specific gene population by enumerating the strategies in the population. Chromosome "0312", for example, is a truck that has "Gas getting" strategy numbered 0, "Capital getting" strategy numbered 3, "Trading" strategy numbered 1, and "Information gathering" strategy numbered 2.

The subsumption truck is a C++ class. The constructor method of this class is responsible for instantiating all the components of the truck. The truck's chromosome string is an argument of the constructor method. The constructor uses the chromosome string to instantiate the appropriate subclasses of the Layer base class. The relationship between the chromosome string and Layer subclasses is implemented by this method. The position of an entry on the chromosome string determines the specific behavior layer, and its value determines which subclass of that behavior

layer to instantiate. Given chromosome "0312", the constructor will instantiate "Gas getting" subclass numbered 0, "Capital getting" subclass numbered 3, "Trading" subclass numbered 1, and "Information gathering" subclass numbered 2.

The Truckin simulation consists of several runs of the Truckin game; each run begins with a generation of trucks. The game executes for a pre-defined interval of time, this represents the truck's lifespan. At the end of the game the performance of the trucks are evaluated according to total profit made by each truck during the game. The top most profitable trucks are selected for genetic recombination. The genes of the selected trucks, their offspring, and some randomly generated gene combinations makeup the next generation of trucks.

Truck reproduction is applied to the chromosomes of the most profitable trucks at the end of a Truckin game. The chromosomes of these trucks makeup the mating pool. Chromosomes are randomly selected from this pool to form mating pairs. The crossover operation is applied to each pair of chromosomes resulting in two offspring per mating pair. A crossover point is a randomly generated number between 1 and the total number of genes on the chromosome. For each mating pair a crossover point is generated, the chromosome string of each parent is divided at this point producing two sub strings each. Two offspring are produced from these sub strings. One offspring chromosome is the string resulting from concatenating the left sub string from one parent with the right sub string from the other, the second from concatenating the remaining sub strings. For example, mating chromosomes "0312" and "1023", and crossover point 3 results in sub strings "031" and "2" from one parent, "102" and "3" from the other. The offspring are "0313" and "1022".

In our experiments 80% of the trucks in each generation were selected by truck reproduction. They included the top 40% most profitable trucks from the previous generation plus their offspring. The remaining 20% were produced randomly.

The subsumption development process requires extensive testing of the system in its environment. The Truckin simulation has the following parameters: number of games, number of trucks per game, and length of time per game. By varying the simulation parameters we are able to perform a variety of experiments. We ran Truckin games with 1, 5, 10, 15, and 20 trucks for 1000, 5000, 10000, and 15000 time intervals. Each combination of number of trucks and time interval was run for more than 1000 games. Extensive testing during the development process validates the

system decomposition and permits thorough debugging of the code.

4.2 Guidelines for Subsumption Systems

Subsumption systems model the interaction dynamics of the components of the system and the environment in order to produce the desired results. As a consequence, the designer must determine the reflex modules and how they should be combined for each task and environment. It's often not possible to transfer a solution for one class of problems to another, instead, the architecture provides a set of principles and a set of examples that might be useful to the designer.

Brooks provided a methodology to develop subsumption robots that manages the complexity of achieving emergent behaviors, and addresses the complexity associated with distributed control. A bottom-up decomposition, incremental design, testing and debugging in the real world, offers a controlled way to achieve the desired behaviors. Distributed control of many modules is simplified by the minimization of inter-module dependence and communication. The guidelines that follow, derived from [4, 5, 22], keep the development process within the boundaries of the subsumption methodology.

It must be emphasized that these guidelines are not a subsumption program recipe. They serve to help the designer apply the subsumption methodology. They act as signposts that direct the development process towards a subsumption program.

Guideline 1: Decompose the system into task-achieving behaviors beginning with the basic set of reflexes that provide survival in a dynamic unstructured world. A task-achieving behavior is observable in the world. Proceeding bottom-up, define task-specific modules which when combined with existing modules increase the capabilities of the system. See [22] for heuristics describing the process of using task-specific constraints to generate behaviors.

This guideline provides a qualitative way to decompose the system, resulting in layering of behaviors that can be incrementally designed and implemented. The overall behavior of the system provides top-down constraints on the bottom-up decomposition, as a consequence every behavior is geared towards the purpose of the whole system.

Guideline 2: A layer is a collection of task-achieving behavior modules that together produce a level of competence. The activity of a layer is stimulated by events in the environment, not instructions from another layer. Most information is obtained directly by the layer itself through sensing the environment. Perception should be tightly coupled to action within a layer. A layer is designed to take small incremental steps towards its subgoals relying on frequent sensing of the world for dynamic error correction. At each step a layer must react quickly enough to able to sense changes in the environment as they occur.

The effect of this guideline is autonomous reactive layers whose integration and communication medium is the environment.

Guideline 3: Layers may interact with lower layers via their input and output. A higher layer may suppress the input or inhibit the output of a lower layer. Suppression includes the ability to replace the input to a lower layer, whereas inhibition causes the output of the layer to be ignored.

This guideline contributes to extensibility. Direct inter-layer communication is minimized resulting in simple layer interfaces. Simple interfaces coupled with a hierarchical ordering of layers facilitates the process of layering: incrementally adding higher levels of competence to the system.

Guideline 4: There should be no simplified test environments. Subsumption applications must cope with unpredictability in the environment, and with imperfect sensory information.

The layers are designed to take advantage of the dynamics of their actions on and in some environment. Testing in a simplified environment could lead to a design of a layer that depends on some simplified property, which is not true in the real environment. This dependency will then propagate to the higher layers who rely on the lower layers.

Guideline 5: Each layer should be tested, and debugged extensively in the real world before adding another layer to the system.

This guideline helps to simplify the debugging process. When a new layer is added to the existing system any bugs are likely to be in the new layer, or in the interface

of the new layer with the system, and not in the existing system. Any bug fixing will be confined to the new layer.

Bibliography

[1] Balkenius, C., *Natural Intelligence for Autonomous Agents*. Lund University Cognitive Studies - LUCS 29 1994.

[2] Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D., **Genetic Programming, an Introduction: On the automatic evolution of computer programs and its application**. Morgan Kaufmann Publishers, 1998.

[3] Booch, G., Rumbaugh, J., Jacobson, I., **The Unified Modeling Language User Guide**. Addison-Wesley, 1999.

[4] Brooks, R.A., *Achieving Artificial Intelligence A Through Building Robots*. MIT A.I. Memo 899, May 1986.

[5] Brooks, R.A., *A robust layered control system for a mobile robot*. IEEE Journal of Robotics and Automation, RA-2, March 1986, 14–23.

[6] Brooks, R.A., *Intelligence without representation*. Artificial Intelligence Journal (47), 1991, pp. 139–159.

[7] Brooks, R.A., *Intelligence without reason*. Proceedings of 12th Int. Joint Conf. on Artificial Intelligence, Sydney, Australia, August 1991, pp. 569–595.

[8] Brooks, R.A., *Integrated systems based on behaviors*. SIGART Bulletin, Vol. 2, 1991, 46–50.

[9] Brooks, R.A., *New approaches to robotics*. Science 253, 1991, 1227–1232.

[10] Brooks, R.A., Stein, L., *Building brains for bodies*. Autonomous Robots, Vol. 1, No. 1, November 1994, pp. 7–25.

[11] Brooks, R.A., *From earwigs to humans*, Robotics and Autonomous Systems, Vol. 20, Nos. 2–4, June 1997, pp. 291–304.

[12] Bryson, J., *The Subsumption Strategy Development of a Music Modeling System*, Unpublished M. Sc. thesis, Department of Artificial Intelligence, U. of Edinburgh 1992.

[13] Bryson, J., *The reactive accompanist: Adaptation and behavior decomposition in a music system*, The Biology and Technology of Intelligent Autonomous Agents, Springer-Verlag, 1995.

[14] Butler, G., Gantchev, A., Grogono, P., *Reusable strategies for software agents via the subsumption architecture*, Proceedings of Asia-Pacific Software Engineering Conference, (Takamatsu, Japan, 7–10 December, 1999), IEEE Computer Society Press, Los Alamitos, CA, 1999, pp. 326–333.

[15] Butler, G., Gantchev, A., Grogono, P., *Object-oriented design of the subsumption architecture*, Software — Practice and Experience, **31** (2001) 911–923.

[16] Connell, J.H., *A Colony Architecture for an Artificial Creature*, MIT AI TR-1151, June 1989.

[17] Gamma, E., Helm, R., Johnson, R., Vlissides, J., **Design Patterns: Elements of Reusable Object-Oriented Software**, Addison-Wesley, Reading, Mass., 1994.

[18] Koza, J., *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems*, Stanford University, June 1990.

[19] Koza, J., **Genetic Programming: On the programming of computers by means of natural selection**, MIT Press, 1992.

[20] Maes, P., *Modeling adaptive autonomous agents*, Artificial Life 1(2), 1994, 135–162.

[21] Mataric, M., *Behavioral synergy without explicit integration*, SIGART Bulletin, Vol. 2, 1991, 85–88.

[22] Mataric, M., *Behavior-based control: Main properties and implications*, in Proceedings, IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems, Nice, France, May 1992, 46–54.

[23] Mataric, M., *Interaction and Intelligent Behavior*, Technical Report AI-TR-1495, MIT Artificial Intelligence Lab, 1994.

[24] Mataric, M., *Behavior-based control: Examples from navigation, learning, and group behavior*, Journal of Experimental and Theoretical Artificial Intelligence, Special Issue on Software Architectures for Physical Agents, Vol. 9, Nos. 2-3, Hexmor Horswill Kortenkamp eds., 1997.

[25] Michaud, F., Mataric, M., *Learning from history for behavior-based mobile robots in non-stationary conditions*, in Proceedings, Autonomous Agents, Minneapolis/St. Paul, May 1998.

[26] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorenson, W., **Object-Oriented Modelling and Design**, Prentice Hall, 1991.

[27] Song, H., Franklin, S., Negatu, A., *SUMPY: A fuzzy software agent*, in Proceedings of the ISCA Conference on Inteiligent Systems, Reno Nevada, June 1996, 124–129.

[28] Stroustrup B., **The C++ Programming Language**, 3rd edition, Addison-Wesley, Reading, Mass., 1997.