UNIVERSITY *of* LIMERICK

"A novel behaviour-based robot architecture and its application to building an autonomous robot sentry"

by
Paul Fitzpatrick

A thesis submitted for the degree of
Master of Engineering

based on work done at
The Department of Electronic and Computer Engineering
College of Informatics and Electronics
University of Limerick, Ireland

Declaration

I hereby declare that this thesis is entirely my own work and has not been submitted as an exercise to any other university.

Paul Fitzpatrick

Abstract

Industrially deployed robots are currently very dependent on human guidance and support. They rely on operator intervention to maintain a stable, known environment within which they perform specific well-defined tasks. Such constraints limit the extent to which robots can be employed. Many useful applications await the development of self-reliant, autonomous robots-applications ranging from exploring Mars to cleaning factory floors. New design techniques are evolving to meet this challenge. One technique that has met with considerable success is "behaviour-based" control. In this form of control, the modules or "behaviours" within the robot's control system take their cues primarily from the environment, rather than from each other as in more traditional schemes. Robots constructed in this way need to make fewer assumptions about their environment, and so are more robust. However, it can be difficult to control the interaction of such behaviours. In this thesis, a novel scheme for orchestrating the actions of collections of behaviours is presented, and implemented on a physical robot.

Another major challenge for autonomous robots is how to learn about their surroundings so they can navigate from place to place efficiently. Much of the literature addressing how robots can construct maps of their environment is oriented towards robots with vision sensors, and relies on computationally expensive image processing algorithms. However, there is currently considerable research interest in investigating the potential applications of groups of small, cheap, miniature robots. Such robots are typically limited in their sensing ability and processing power. This thesis examines how a miniature robot can build and maintain a useful map of its environment with short-range proximity sensors only, and do so in real-time even with the restricted processing power of such robots. This is a desirable ability because it could allow robots to be used in a range of niches where sophisticated sensing and image processing would be impractical. Again, the technique described is implemented on a physical robot that behaves as a sentry.

Acknowledgements

I am sincerely grateful to my supervisor, Colin Flanagan, for his support in this work. His feedback was invariably enlightening, and was particularly invaluable in the drafting of this thesis. I am indebted to him for all his help. My gratitude also to Prof. Tom Coffey for his input.

Heartfelt thanks to the group of people who acted as cheerleaders along the way, and whom it gives me great pleasure to acknowledge in print. To my mother, Ann, and all of my family, for support that is the foundation of all I do. Thanks to my sister, Deirdre, for introducing me to the Zen of tea. My cousin David, for all the information about seaweed and Finland an engineer could ever want, and for many Interesting Times. Willemien, to whom I owe many, many sandwiches. Mary, for always having some refreshingly mindless work on the farm lined up for me when I came home. And everyone else for continually asking when I was going to be finished, and generally keeping me going despite myself.

Finally, I want to thank my robot, for being very well behaved, and never throwing a major tantrum. May your cogs always run free, and may your battery charge last forever.

Table of Contents

1. Introduction

The field of robotics is advancing along two fronts, both of which find expression in this thesis. Firstly, progress is being made through the sheer weight of experience gained by researchers as they attempt to apply robots with varied capabilities to widely different and novel tasks. Secondly, the field is advancing as the lessons learned in these specific applications are encoded in reusable structures called "robot architectures" which act as frameworks to simplify the development of future applications.

This thesis contains contributions to both fronts. It shows how a robot possessing only short range sensors can successfully perform "sentry duty" by exploring, mapping, and patrolling its environment- tasks which would normally be expected to require long range sensors, such as sonar or vision. In parallel with the development of this application, a new robot architecture called Lateral is evolved which makes behaviour co-ordination much simpler to implement than in other related architectures such as Subsumption [[7]]. The "sentry duty" application plays a dual role as a "proof of concept" of the new architecture, and as an innovative application in its own right.

This chapter outlines the nature of the two threads of the thesis, what problems they seek to solve, and interesting features of the proposed solutions. It describes how the two threads relate to each other, and how the thesis is structured. The "Novel Application" thread is discussed first, then the "Robot Architecture" thread, and finally a "road-map" for the thesis is given, capturing the structure of the thesis and how each part of it relates to the whole.

1.1 Novel application contribution

The novel application discussed in this thesis is to build a robot capable of performing "sentry duty" (prowling, patrolling, exploring, etc.) with only very short-range sensors. The idea is to take a second look at what is possible with minimal sensing equipment, because there is a range of potential niches for robots where sophisticated sensing and image processing would be impractical- particularly where the robot has to be small and inexpensive, as when large swarms of cheap "gnat" robots are to be used.

The application is implemented on both a physical and simulated robot. The robot used is the Khepera miniature robot (see Figure 1-1, and Section 7.2.1, page 207).

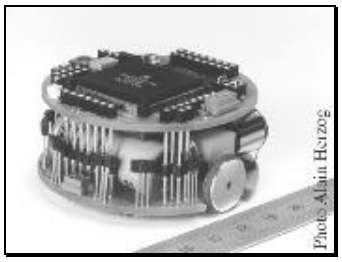


Figure 1-1: The Khepera Robot

For a discussion on the importance of making an "embodied" implementation of a robot system, see Section 2.4.7 (page 22). Serious reservations have been expressed on the usefulness of work that is entirely simulated [[10]], at least for the purposes of working towards truly autonomous robots. The fact that the application in this thesis, although quite complex, is still physically implemented was considered an important goal to achieve.

1.1.1 The problem

The major problems to be solved to build a sentry robot with no long range sensors are: how to detect landmarks with such restricted sensing equipment; and how to build, use, and maintain a useful map of the robot's environment in real-time.

1.1.2 The solution

The thesis presents a novel approach to detecting landmarks in a robot's environment using minimal sensing equipment. Landmarks are important for robust cartographic systems because they act as reference points from which the robot can correct its estimate of its position from time to time and avoid losing track of its location. When rich sensor data are available, plenty of cues are present from which landmarks can be recognised. Landmarks are difficult or impossible to extract from the sparse sensor data available from coarse short range proximity sensors. The solution proposed in this thesis is that, while landmarks may not be recognisable directly from any sensor signature, they can be identified by monitoring the robot itself when it interacts with them. For example, while it may not be possible to identify the presence and shape of a corner from sensor data, if the robot performs an algorithm to follow the curve of a boundary (a very simple task even with limited sensor data) then the characteristics of the corner become obvious by analysing how the robot itself moves around it. The landmark could

be said to be an emergent feature from the combination of the robot's behaviour with its environment. This is the key idea that allows the proposed cartographic system work with low bandwidth sensors.

How a map can actually be put to use effectively is equally as important as the mechanics of maintaining it. Any form of map will generally grow in size proportionately with the total area the robot has explored, so care is needed to ensure the map remains usable as it gets larger. This paper presents a method for maintaining continuously updated local "windows" onto the overall map at different scales, allowing the information needed for different tasks to be quickly constrained and filtered. This prevents the robot from being swamped by the mass of detail to be considered. This supports the use of the cartographic system on robots with less powerful processors.

The thesis shows how these ideas together give a useful cartographic system with less sensing and processing requirements than comparable systems, and describes the constraints it places on the behaviour of a robot using it.

1.1.3 Novel features

The main innovations of this section of the thesis are:-

- The cartographic system, capable of maintaining a useful map from proximity sensors alone.
- The set of behaviours of the robot, designed to achieve an overall "sentry"-like behaviour using the cartographic system and operating symbiotically with it to ensure it receives the data it needs to map correctly. The behaviours also allow general goal seeking ability.

These areas are explored in Chapters 4 and 5 (see Figure 1-2 at the end of this chapter for a thesis "road-map").

1.2 Robot architecture contribution

The novel contribution to robot architectures in this thesis is embodied in "Lateral", a new robot architecture. Lateral is a "behaviour-based" control technique where the desired functionality of the robot is implemented by building and combining a set of "behaviour" modules (see Chapter 3).

1.2.1 The problem

Behaviour-based control has many advantages over traditional techniques in terms of reaction speed, robustness, and ease of design, as exemplified by the best known of such control architectures, "Subsumption". The characteristic feature of Subsumption is that the robot's control system is implemented as a series of layers, each enhancing the layer below it (by selective suppression, inhibition, and replacement of dataflows) to give an incrementally more competent robot. If a layer becomes disabled, the robot can still operate by reverting to a lower level of competence. This simple organising principle is naturally robust, and has proven remarkably powerful. It is less helpful, however, for structuring control systems which cannot easily be made to fit into a simple layered hierarchy.

1.2.2 The solution

In this thesis a new behaviour-based robot control architecture, called "Lateral", is presented that supports structured behaviour combination rather than requiring the strict layering mandated by Subsumption. It allows behaviours to be organised in whatever decomposition is natural for the control system, while still retaining Subsumption's ability to implement behaviours incrementally and the natural robustness that this gives.

Lateral's structural flexibility is achieved by a corresponding flexibility in its priority system. The priority system in a behaviour-based architecture is critical since it determines how conflict between behaviours is resolved. In Subsumption, the priority of a module is determined by the layer it is in, so it is fixed at a level decided at design time. In Lateral, the priority of a module is instead determined by the importance of the task it performs. This is a dynamic property which can change as the robot's situation and goals change. While the hierarchy of behaviours in Subsumption is rigidly fixed, in Lateral it changes fluidly according to need using a system called "sponsorship".

A tool for simplifying the implementation of a design made using Lateral structures is also developed. This tool, Zac Script, is then used to implement the "sentry" behaviours designed in Chapter 5. A complete discussion of the nature and utility of Zac Script is given in Chapter 6.

The details of the Lateral architecture are presented in Chapter 3. It is compared with other robot architectures in general and Subsumption in particular, and the advantages it has for building a behaviour-based robot control system are elucidated.

1.2.3 Novel features

The main innovations of this section of the thesis are:-

- The flexibility of the decomposition of a robot's control system that can be supported.
- The new ways behaviours can be combined using "sponsorship".
- The Zac Script tool for encoding Lateral constructs in a straightforward and platform-independent way.

1.3 Overview of thesis content

The thesis is structured in the following way:-

- First, a review is made of the literature relevant to both threads of the thesis - the robot architecture, and the sentry application (Chapter 2).
- The new robot architecture, Lateral, is then described, justified, and compared with other architectures (Chapter 3). This is done before any discussion of the sentry application because that application is built using the Lateral architecture, so the nature of the architecture must be clear first. The architecture acts as a guide for and a set of constraints on the design of the sentry application.
- Next, the design of the sentry application is discussed. This involves two main issues - cartography (Chapter 4), and the choice of behaviours for the robot (Chapter 5). Both are examined and related to each other in this pair of chapters.
- The Zac Script tool for implementing behaviours in Lateral is then introduced (Chapter 6). The design of the sentry application and the discussion of the Lateral architecture are related to each other to show how the application is built from the architecture.
- An implementation chapter (Chapter 7) next shows how the entire architecture and application developed so far can be connected to a robot to become a functional control system. This chapter discusses the actual robot the work was implemented on, Khepera.

Figure 1-2 illustrates the structure of the main body of the thesis.

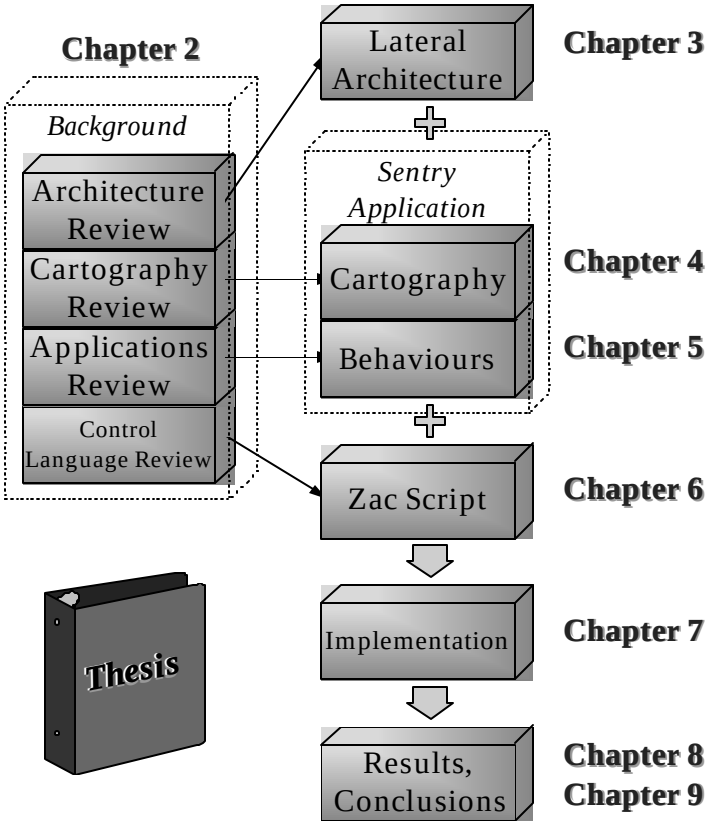


Figure 1-2: Road-map, structure of main body of thesis

Chapter 2	details the state of research relevant to this project. It describes the various classes of robot architectures, giving examples of each, and contrasts their functionality. It also examines approaches taken to map building and use by other researchers.
Chapter 3	develops the Lateral Architecture, a novel robot architecture for simplifying behaviour co-ordination.
Chapter 4	describes the design of a cartographic system suited to a robot performing "sentry duty"
Chapter 5	describes the design of behaviours for a robot performing "sentry duty", using Lateral.
Chapter 6	introduces a tool for aiding the implementation of a behaviour-based design called "Zac Script".
Chapter 7	discusses implementation of the sentry design on a physical and simulated robot.
Chapter 8	evaluates the performance of the sentry, and where possible makes comparisons with other research.
Chapter 9	discusses the project, and gives conclusions drawn from the work.
	References and appendices follow.

2. Background

This chapter presents a literature review of topics relevant to this thesis. It examines the theory of robot architectures, and different approaches taken to robot cartography. The overview of architectures provides the background necessary to understand the choices and decisions made when developing the Lateral robot architecture in Chapter 3. The review of cartography is needed for Chapter 4, where an approach to map building and use in a robot with short range sensors is described that builds on the ideas reviewed here.

2.1 Review of Robot Architectures

At the first International Symposium on Robotics Research, robotics was broadly defined as "the intelligent connection of perception to action" [[1]]. This connection is provided by the robot's control system (see Figure 2-1).

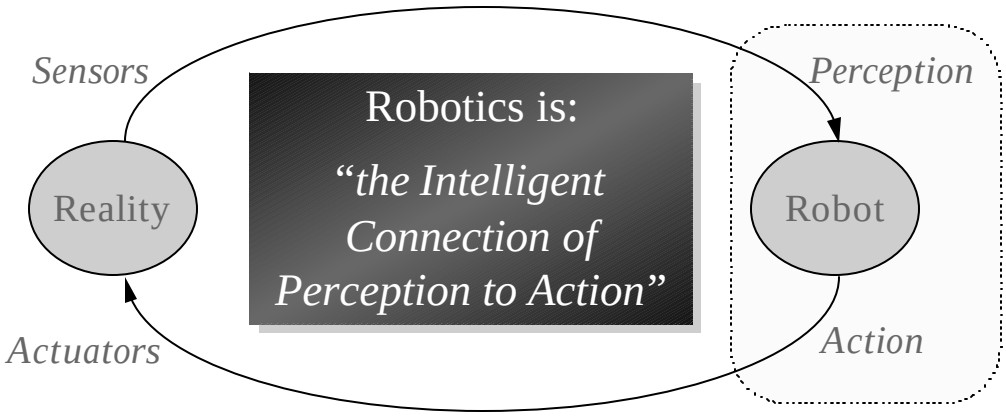


Figure 2-1: Definition of robotics

A robot architecture is a set of guiding principles used to structure a robot's control system. It does this both by providing a scheme for the organisation of the system, and by constraining the way in which the system can attempt to solve control problems. Hence the architecture is reflected not only in the building blocks from which the control system is constructed, but also directs its high level design. In summary, an architecture is responsible for:-

- providing a set of principles for organising a control system, and
- imposing a set of constraints on the design of that system.

Robots vary greatly in the hardware and software they use, and work done on one robot is often impossible to apply directly to others. Robot architectures, however, *are* reusable and for this reason research is often concentrated on them rather than on specific applications built from them. A wide range of robot architectures exist, varying in their domains of application and philosophical background. Some of the criteria they can be categorised under are as follows:-

- **Symbolic or non-symbolic** - an architecture may deal with atomic "facts" or may use distributed representations, such as neural networks and connectionist models generally.
- **Homogeneous or inhomogeneous** - architectures may require that the same single fundamental organisational unit be used throughout the control system, or may allow a range of such units. In the first case, it is implicitly assumed that a single form of building block is sufficient to construct any control system, including real-time human-level intelligence, while in the second case it is assumed that control systems may need to be composed of structurally different modules.
- **Deliberative or reactive** - an architecture may support protracted reasoning, or may be geared towards building systems that react directly to environmental cues. In the first case, the architecture is said to be knowledge driven, in the second case it is data driven.

2.2 Classical Robot Architectures

Robotics developed as a sub-field of Artificial Intelligence (AI), so it was natural that it was heavily influenced by the methodologies that had proved successful in that area. AI has evolved a useful toolbox of techniques applicable to problem solving and planning, whereby given a suitable description of the state of the world, powerful search algorithms can be applied to find the appropriate manipulations to move from that state to a desired goal state. This is strongly reminiscent of what robots need to do, and AI-based or "classical" robot architectures try to make use of such techniques. To do this, it is necessary to translate from sensor data to a description of the state of the world, and from manipulations on that description to physical actuator commands. In terms of the criteria listed in the overview, an architecture constructed in this way is:-

- *Symbolic* - AI techniques are based on manipulating systems of symbols.

- *Homogeneous*- symbols are used throughout the control system for all stages of reasoning- for describing the world, expressing plans, etc.
- *Deliberative*- AI is strongly biased towards describing a problem, carrying out extended reasoning, and generating a solution.

This initial decomposition leads naturally to a "pipelined" approach to interacting with the environment, where the overall system is divided into a number of subsystems arranged in series, as shown in Figure 2-2.

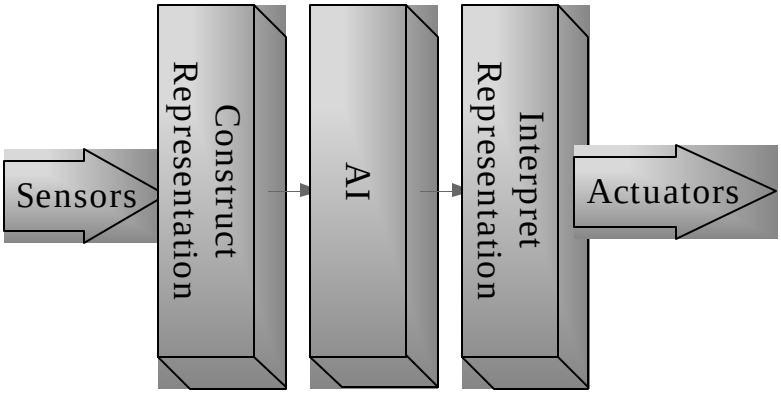


Figure 2-2: Applying AI to robotics

Typically the decomposition shown is further refined to consist of the following modules:-

1. **Perception**- this interfaces to the sensing devices connected to the robot.
1. **World modelling**- this subsystem uses the results of perception to update an internal model of the robot's environment, and to keep track of where the robot is with respect to that model.
1. **Planning**- this attempts to work out how it will achieve its goals given the current world state and the state of the robot.
1. **Task execution**- this breaks down the plan developed by the previous subsystem into detailed motion commands.
1. **Motor control**- this interfaces with actuators to express motion commands generated during task execution.

These modules still act in series, with each one in turn processing the results of the previous module, as shown in Figure 2-3.

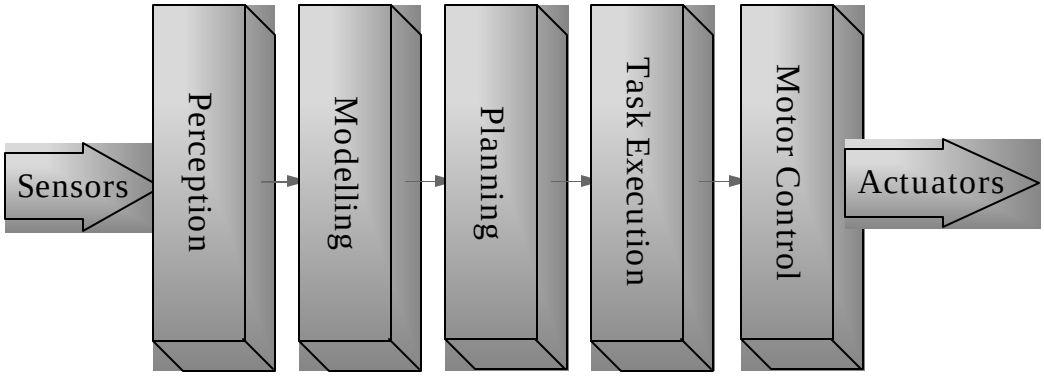


Figure 2-3: "Pipelined" approach of classical control

2.2.1 Domain of Application

Each of the subsystems enumerated above is a complex program, and all have to work together well for the robot to operate at all. However, some of the subsystems have many unsolved problems associated with them, such as perception and world modelling, that make them feasible only in very structured environments. The noisy and random nature of the real world overwhelms them. In particular, as the complexity of the environment increases, the time needed to perceive, model and plan about the world increases exponentially. Hence classical robot architectures are most successful in:-

- Very structured environments, such as industrial applications [[2]], or
- Simulated environments or "toy worlds" [[3]].

A specific example of a classical architecture will now be given.

2.2.2 An example architecture- RCS (Real-Time Control System), Albus [[4]]

This sophisticated architecture in the classical control tradition consists of a nested hierarchy of modules, with each layer distinguished by a characteristic bandwidth range, and with tasks at each level being decomposed into sequential sub-tasks. The components of this architecture are:-

1. **Sensory Processing modules**- These filter, mask and correlate sensor data, and perform feature detection, pattern recognition etc.
1. **Knowledge Database modules**- These embody knowledge about the logical and dynamic behaviour of the world.

1. **World Modelling modules** - These model the state of the world and predict the results of actions using the output of sensor processing and a knowledge database.
1. **Value Judgement modules** - These evaluate states and the reliability of state estimations, perform cost-benefit analysis, and may track the match between observations and predictions.
1. **Behaviour Generating modules** - These are concerned with decomposing tasks into achievable sub-tasks using any of a number of planning algorithms - simple look-up, state-space search, etc.
1. **A communications system** between the modules.

RCS is a reference model architecture for intelligent systems developed at the National Institute of Standards and Technology (U.S.) and has been deployed in various applications such as a motor assembly testbed, an intelligent workstation for deburring and chamfering components of jet engines, a control system for a U.S. postal facility, and in robots for floor-cleaning and hospital service. It is part of an overall effort to make industrial machinery more intelligent by providing an open control architecture supporting advanced AI techniques.

2.3 Reactive Robot Architectures

Robots built using classical robot architectures are capable of performing sophisticated reasoning, but only in well-structured environments. The first family of robot architectures to diverge from this pattern were called “reactive”. They attempted to operate in complex unstructured environments by *reacting* to environmental cues rather than trying to second-guess the environment by modelling it. As shown in Figure 2-4, this form of control is the exact opposite of classical control, at least along the dimensions of environmental and cognitive complexity.

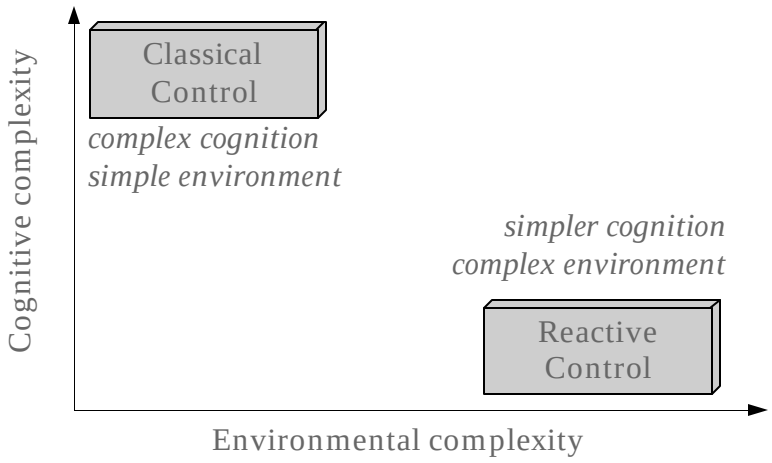


Figure 2-4: Reactive architectures versus classical control

This type of architecture was not developed explicitly, but rather emerged repeatedly from the work of researchers engaged in widely varying projects, working with totally different hardware and software, and using very diverse vocabularies to describe their methodology [[5]]. In the robot control systems that began to appear, the robot was controlled by programming it with a set of *condition Þ action* rules, or some equivalent abstraction, where the conditions are combinations of sensor readings and state variables, and the actions are simple functions performed on actuator settings or state variables. By coordinating these rules through state variables, robots are produced that, in their limited domain, perform very well and give the impression of advanced cognition simply because the complexity of their environment finds direct reflection in their actions. Robots controlled in this way, using *condition Þ action* rules, are by their nature reactive. There is little processing or modelling in the route from sensor data to actuator settings; the condition/action pairs are like reflexes in biological systems.

2.3.1 Domain of Application

Reactive robots have good real-time characteristics, which is an important advantage over classically controlled robots because *"An oncoming truck waits for no theorem prover"* [[6]]. They can keep pace with changes in a dynamic environment, since the path from change in sensor reading to change in actuator setpoint is short, providing a direct coupling of perception to action.

2.3.2 Example architecture - The Niche Robot Architecture, Miller [[5]]

This is a very typical reactive architecture. Niche robots are programmed using a set of *condition Þ action* rules, with the following semantics:-

- "Conditions" are tests of the state of sensor values and internal state variables. If they become true, the associated action is executed.
- "Actions" are simple manipulations of actuator settings or state variables.

There is no hierarchy mandated, but if the robot needs to switch between sets of rules under different conditions, it is useful to collect rules into "scenes", recognised by having a common component in their conditions, and to control switching between scenes using another batch of rules called the "sequencer" (see Figure 2-5).

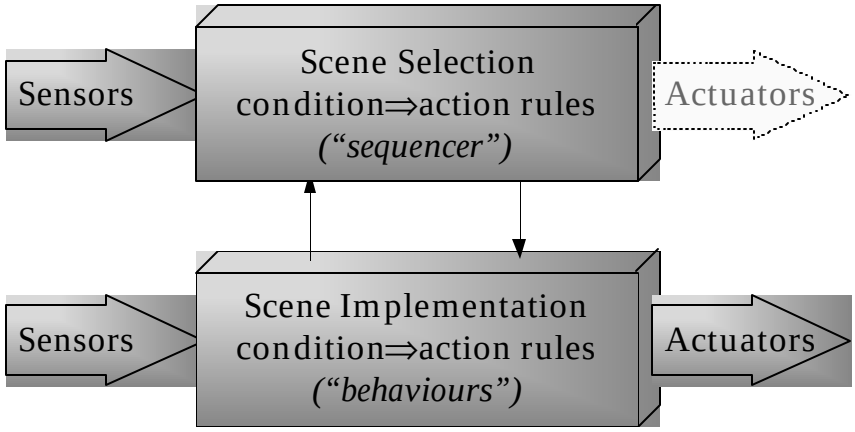


Figure 2-5: Niche Robot Architecture

Robots built using this architecture are suitable for applications where very tight real-time constraints must be met. They are relatively easy to design, but any given robot is very limited in scope (hence the term "niche" robots). Within their restricted domain the robots can perform very well and appear to exhibit a higher level of cognition than is actually the case. The architecture described here has been essentially re-discovered time and again by numerous researchers working in different contexts [[5]].

2.4 Behaviour-Based Robot Architectures

Behaviour-based robot architectures developed as a more sophisticated alternative to classical architectures than purely reactive architectures, which are limited in their ability to perform time-extended tasks [[7]], as Figure 2-6 illustrates.

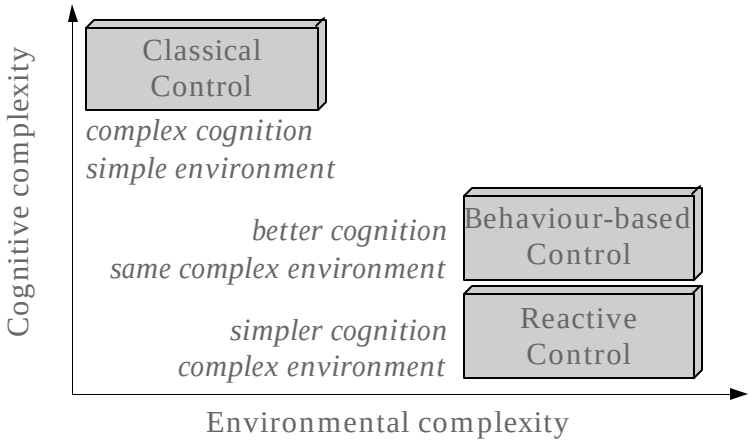


Figure 2-6: Behaviour-based robot architectures

Behaviour-based control architectures are characterised by the following principles and constraints [[8]] :-

- Modules are somewhat time-extended.
- Modules tend to be more reactive than deliberative.
- Modules tend to be relatively simple.
- Modules interact with each other through the environment, not the system, as much as possible.
- The world is considered its own best model- it is never out of date or inaccurate- so modules consult it directly whenever practical, rather than trusting the judgements of other modules.
- Modules uses distributed representations tailored to their particular needs, and do not share their representations with other modules.
- Execution of modules is not serialised.
- The system is incrementally implemented, with a working system existing from the very first module.

Behaviour-based architectures are dominated by the first and most well known of all such architectures, "Subsumption". This architecture will now be examined in detail, as the work presented in this thesis builds on this architecture.

2.4.1 Subsumption

Subsumption uses a "horizontal" decomposition of a robot's control system, rather than the "vertical" decomposition used in classic control. If a robot is drawn as a "black box" with sensor input entering on the left and actuator output exiting on the right, then the system decomposition used in classical control slices the black box vertically into a sequential pipeline of modules, with sensory information entering only one, and being processed by each in turn until it has passed through the entire pipeline and emerges as actuator commands. Subsumption, in contrast, divides the system "horizontally" into layers arranged in parallel rather than in series, each of which have simultaneous access to both sensors and actuators (see Figure 2-7).

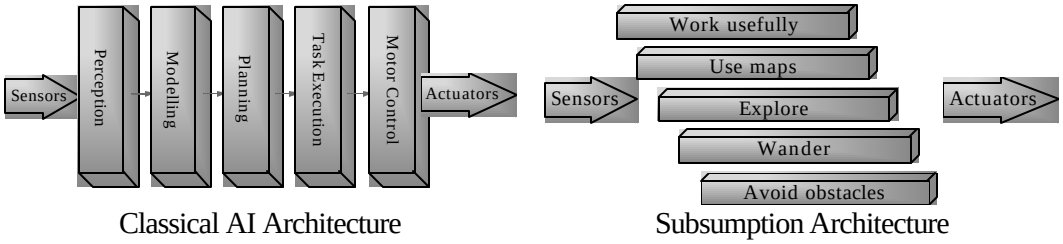


Figure 2-7: Classical AI architecture versus Subsumption architecture

Rather than using a functional decomposition, as is favoured in classical control, the system is divided into "task achieving behaviours". In general this type of horizontal decomposition is the one used in all behaviour-based robot architectures. Comparing the kind of modules present in classical control and Subsumption side by side as in Table 2-1, the classical modules are seen to be concerned with transformations on representations, while modules in Subsumption have direct expression as externally observable behaviour.

Table 2-1

Modules in Classical AI Architecture	Modules in Subsumption Architecture
Perception	Work Usefully
World Model	Use Maps
Planning	Explore
Task Execution	Wander
Motor Control	Avoid Obstacles

Each individual "horizontal layer" will contain elements of all the "vertical" tasks found in a classical system, but implemented in a form tailored to the requirements of the behaviour it is responsible for, such as the ability to move away from an obstacle, to move around an area in a random fashion, or to explore the robot's environment. Any given module need not tackle the whole business of getting the robot from A to B; instead, it only performs that part of perception, planning, etc., which is appropriate to the task it has to perform.

The horizontal decomposition used in behaviour-based systems introduces a new issue that does not affect classical control. Since more than one module can now potentially attempt to control an actuator at the same time, conflict is possible if two or more modules attempt to gain control of an actuator simultaneously. Every behaviour-based system must have some way of resolving this conflict, and it is the different approaches to doing this that distinguish behaviour-based architectures (of which Subsumption is just one) from each other. Subsumption resolves conflict by arranging layers in a fixed hierarchy, with higher layers given priority over lower layers, as explained in the following sections.

2.4.2 Levels of Competence

In Subsumption, a robot's control system is specified by defining a number of desired *levels of competence* for the robot. Each level of competence is an informal description of how the robot should behave for any environment it will encounter. The levels are arranged sequentially by their degree of sophistication, with each level enhancing the competence supplied by the level before it to provide a higher competence.

Table 2-2 shows an example set of competences used by Brooks [[7]].

Table 2-2

Level	Behaviour of robot
0	Avoid contact with objects (whether the objects move or are stationary)
1	Wander aimlessly around without hitting things
2	"Explore" the world by seeing places in the distance which look reachable and heading for them
3	Build a map of the environment and plan routes from one place to another
4	Note changes in the "static" environment
5	Reason about the world in terms of identifiable objects and perform tasks related to certain objects
6	Formulate and execute plans which involve changing the state of the world in some desirable way
7	Reason about the behaviour of objects in the world and modify plans accordingly

Each level of competence assumes the existence of all lower levels. For example, the exploration competence is free to move as it pleases, secure in the knowledge that the lower level competence designed to avoid contact with objects will prevent it from entering into collision with an obstacle. Once the obstacle avoidance competence is present, higher levels do not need to be aware of the problems of avoiding objects, because if there is something in the way, this competence will resolve that problem for them. Note that while higher level competences are aware of lower level competences, lower level competences are permitted no knowledge of higher level ones.

2.4.3 Layers of Control

Each level of competence is implemented incrementally by adding a corresponding *layer of control* to the robot. Suppose that a robot currently has layers of control implemented corresponding to all competences up to the n^{th} competence. Then the $(n+1)^{th}$ layer of control can be built. This layer is allowed to examine data from the n^{th} layer. It may also place data into the internal connections within this layer, overriding the normal data flow. The n^{th} layer remains unaware of the new layer above it, and runs as usual, except for the occasional intervention by the higher layer to make the refinements to its behaviour necessary for a higher level of competence. This is the essential characteristic of Subsumption (see Figure 2-8).

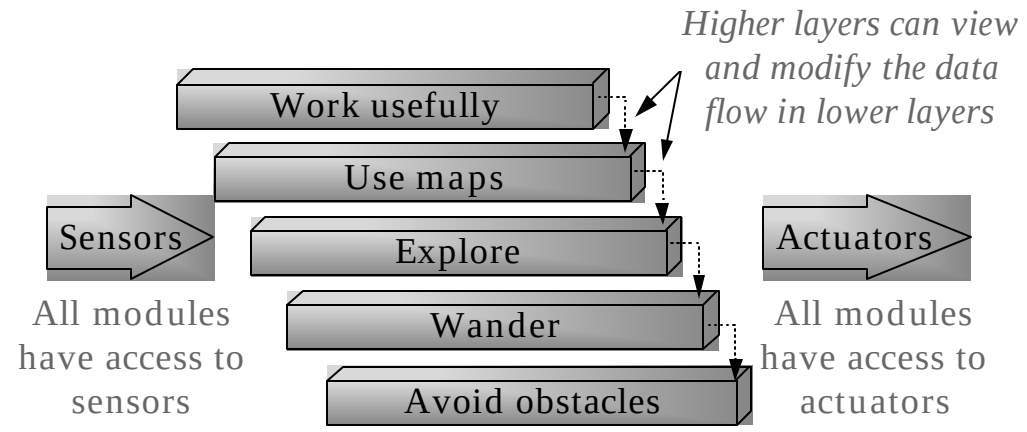


Figure 2-8: Subsumption architecture

A successful implementation of a layer of control is a functional robot exhibiting the new competence and with all other previously implemented competences retained.

2.4.4 Implementation of Individual Layers

How each layer is implemented is a separate issue to the main thrust of the Subsumption architecture specified above. The only constraint is that the implementation should allow higher layers to examine and modify internal dataflows in lower layers. Brooks [[7]] chooses to build layers from fixed topology networks of Augmented Finite State Machines, with communication via connecting "wires" (see Figure 2-9). The wires, since they carry dataflow, are the elements that higher layers interact with to enhance lower layers. The "Augmented" in Augmented Finite State Machines denotes that the state machines are not constrained to be pure state machines in the mathematical sense, but are permitted some internal storage (in the form of registers) and timers.

For a higher layer to examine data flow in a lower one, it attaches an extra wire to whatever it wishes to monitor. There are a number of schemes for overriding normal data flow. An extra wire can terminate at the target of another wire (an "input site"). If any message flows in the extra wire, it *inhibits* any message along the other wire for some pre-determined time (the "characteristic time" of the system). An extra wire can also terminate at the source of another wire (an "output site"). Any message flowing in the extra wire will not only inhibit the other wire, but will be inserted on to it, *suppressing* and replacing the normal dataflow. A variant of this type of connection appears in later versions of Subsumption [[9]], called *defaulting*,

where priority is given to the original wire, with dataflow from the new wire only being accepted when there is no data on the original wire for a given time.

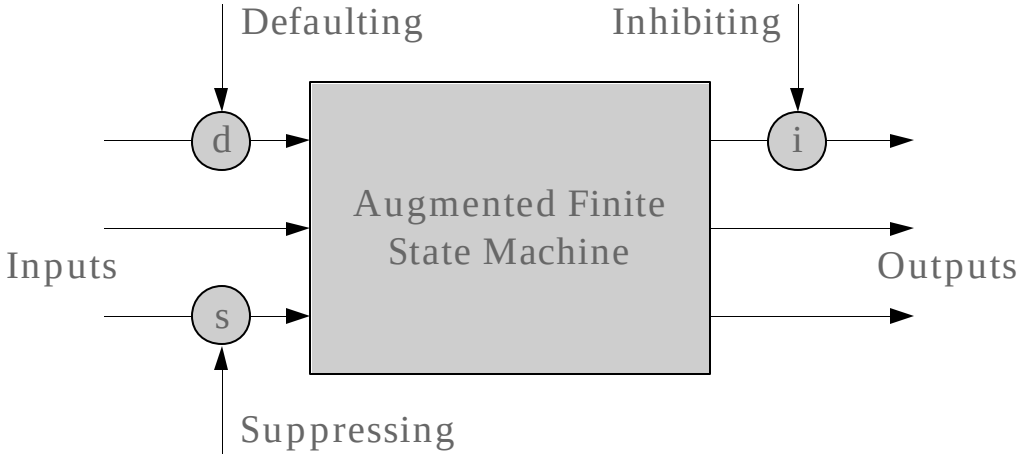


Figure 2-9: The use of wires in Subsumption

2.4.5 Development Strategy

Brooks strongly advocates a particular approach to the development of a Subsumption system beyond the specific details of levels of competence and layers of control [[7]] :-

- Implement the control system incrementally.
- Use “data driven” design.
- Avoid using simulation.
- Avoid using a shared world model.

Each layer should be implemented in turn, and debugged until flawless before proceeding to the next higher layer. This approach ensures that all bugs must belong to either the layer being added or the interface between that layer and the previous one. This incremental testing method is a software engineering principle recommended generally, not just within the specific context of robot control systems.

The control system should be “data driven” (reactive rather than deliberative). This means that the action of a part of the control system should be primarily a consequence of events or opportunities in the environment, and only secondarily dependent on interactions with the rest of the control system. Such a system avoids dependence on a centralised planner, since planning under these conditions will occur in a distributed fashion.

There should be no simplified test environments. Robots using Subsumption should be embedded in the real world, and interact with it directly, without depending on human assistance.

No central representation or "World Model" should be used. Brooks has claimed that in a Subsumption system, no variables should be instantiated, no rules matched, and no choices made. Certainly if representation is used, it should be distributed, and never shared between layers.

While Subsumption need not necessarily be implemented in the form of finite state machines, many of the criteria Brooks associates with his particular implementation should be retained. The implementation should avoid the use of global data and should not require "dynamic communication"– when a message is sent, there should be no need of an indication as to whether it has been received or not. This last point is a pragmatic one geared towards facilitating a fast control system.

2.4.6 Practical Arguments for Subsumption

Before examining the philosophical arguments for and against Subsumption, it is worth looking at its practical consequences. The most significant aspect of Subsumption is its incremental nature. This is the primary factor in its success in producing complete functioning physically realised robots.

Consider a robot in its initial state without any control software present. At this point the robot is essentially a hardware "trolley" carrying a collection of sensors and actuators. If signalled to move in a certain direction, it will do so, until it hits something and breaks.

Suppose a conventional AI module is added to the raw robot. A natural one to implement first is Motor Control. With the addition of this module, the nature of how the robot is commanded will have changed; it will now be easier and at a higher level of abstraction. But the nature of the robot has not changed. If commanded to move in a certain direction using the new interface, the robot will again move forward until it hits something and breaks. If another AI module is implemented also, Task Execution for example, then the robot becomes even easier to control. Now it could be given a more complex path to follow rather than moving in a straight line. But it will still smash into any obstacle it meets- it can't get any smarter until the Planning module is implemented, and that can't be done until the modules working from

Perception up are implemented. In short, the robot remains just as brittle as its raw hardware until all the AI modules of its control system have been implemented. Before anything works, everything must work.

If instead a Subsumption layer were to be added to the raw robot, the nature of the robot changes immediately. The most natural layer to add first is the Obstacle Avoidance module. Once this is added, if the robot is signalled to move in a given direction it will do so, until it meets something and stops to avoid a collision. The robot has become "smarter" immediately. As each layer is added in turn, the robot is given an increasing vocabulary of behaviours. But it is functional at each stage - the robot starts working from the time the first layer has been added.

2.4.7 Philosophical Arguments for Subsumption

Philosophically, Subsumption is rooted in a rejection of the *Symbol System Hypothesis*, or at least of the main body of work which developed in AI under its influence.

The Symbol System Hypothesis

This hypothesis asserts that "intelligence" arises through the manipulation of symbols in a domain-independent manner, where the meaning of the symbols is irrelevant to the reasoner [[10]]. "Meaning" enters the picture only when observers of the system identify symbols in the system with some corresponding entity within their own experience. The symbol system hypothesis is an implicit assumption behind much work in Artificial Intelligence. When applied to robotics, it is implicit in the hypothesis that perception and action, which are by their nature non-symbolic, should have symbolic interfaces so they can be reasoned about. Brooks suggests that this interface between the symbolic and the non-symbolic is critical, and has been mishandled by conventional AI. To reason at a certain level of abstraction, a system of symbols representing entities and their relationships at that level of abstraction is required. It is tempting to assign the task of generating such a representation to the pre-symbolic part of the system, i.e. the interfaces to the real world. But in Brooks' opinion this whole emphasis on abstraction from the real world is damaging because:-

- It directs researchers towards the elusive goal of building a general purpose system to deliver complete descriptions of the world in symbolic form, rather than more realistic task driven perception systems such as active vision [[10]].

- It builds into the system the assumption that knowable objective truth exists, with much added complexity needed to work around that assumption so the robot can deal with incomplete information about complex environments [ibid.].
- Brooks believes this process of abstraction is the difficult part of the problem posed by robotics, and that there is no clear dividing line between it and reasoning. Hence considering it just an "interfacing issue" is misleading. That this is so is attested to by the wealth of anecdotal evidence of robot simulators (which don't have to perform this interfacing) that performed fine but proved utterly incapable of being generalised to working in the real world [[11]].

Assuming that the state of the world can be known completely by a reasoning entity is acceptable when, for example, the entity is a compiler and the "world" is its host operating system. It is not so realistic when the entity is a robot, operating in an environment much more complex than that experienced by any disembodied software entity, and of which it can only sense a small slice at a time. Hence, for robots, the Symbol System Hypothesis, although not necessarily invalid, has proved misleading. In replacement, Brooks advances the *Physical Grounding Hypothesis*.

The Physical Grounding Hypothesis

The Physical Grounding Hypothesis states that "to build a system that is intelligent it is necessary to have its representations grounded in the physical world" [[10]]. The system must be connected to the real world - only embodied entities can be physically grounded, unlike with symbol systems. Physical grounding requires that all the robot's knowledge must be extracted from physical sensors and that its goals must eventually be expressed as physical action.

Brooks places great importance on embodiment. He suggests that concentrating on abstract "reasoning" is missing the fundamental nature of intelligence. The evolution of insects from the first single-celled entities took about 3 billion years; from there to the immediate predecessors of the great apes took about 430 million years; and from there to present day humans just another 18 million years. This last step in which "human intelligence" (language, reason, problem solving behaviour, etc.) developed seems small compared with the time taken to evolve to just before that level, and suggests that the more general type of intelligence concerned with simply existing and surviving in a complex world is much more important, and

much harder. By enforcing Physical Grounding and requiring robots to be physically realised, Brooks hopes to advance research in this neglected area which is often just "abstracted away" into non-existence. He believes that simulated robots have their intellectual work done for them by those creating their input, and that a fundamental paradigm shift is necessary for a robot that can cope for itself with the unpredictability of the real world seen through noisy, inaccurate sensors.

Brooks enumerates the following key aspects of his work that he views as crucial for making genuine progress in robotics research:-

- **Situatedness**- robots should deal with the real world, not with abstract representations of it.
- **Embodiment**- a robot is part of its environment; its actions change the environment and this effect is fed back to its sensors.
- **Intelligence**- the robot, as part of its environment, appears intelligent to an observer. That intelligence lies not only in the robot, but in its situation in the environment, the characteristics of its sensors, and how the robot's physical form interacts with the world.
- **Emergence**- external action may not be identifiable with any concrete entity or cause within the robot's control system, and instead the action may *emerge* from interactions of components both between themselves and with the world.

Brooks summarised the conclusions of his robotics research as follows:-

> *We have reached an unexpected conclusion (C) and have a rather radical hypothesis (H)*
>
> *C ¾ When we examine a very simple level intelligence we find that explicit representations and models of the world simply get in the way. It turns out to be better to use the world as its own model.*
>
> *H ¾ representation is the wrong unit of abstraction in building the bulkiest parts of intelligent systems."*

2.4.8 Experiences with Subsumption

MIT Robot "Allen" [[10]]

This early robot implemented obstacle avoidance, random wandering, and simple exploration where the robot targets a distant area and tries to reach it. Despite the robot's simplicity, it

exhibits all the hallmarks that have come to be associated with Subsumption-based robots- it is robust, autonomous (it does not need to be fed external goals), and solves tasks using a hierarchy of behaviours.

MIT Robot "Herbert" [[10]]

This robot was designed to wander through a lab and pick up empty soda cans. It did this in an interesting way, where the sequencing of behaviours needed to pick of a can was produced by environmental cues rather than an explicit internal sequencer. This allowed it to take advantage of opportunities in the environment by skipping behaviours that were redundant, and made it easier to recover when the unexpected happened because so few expectations were built into the robot. This is difficult in conventional AI, but very natural under Subsumption.

MIT Robot "Toto" [[8]]

Toto, like all of the Subsumption robots at MIT, implemented obstacle avoidance and wandering, but also demonstrated boundary following, landmark detection, map construction and path finding. It kept track of where it was by building an internal "map", so it could find its way back to a specific area if commanded to do so. The nature of this map-building is interesting, given the prohibition within Subsumption against building world models. First, landmarks are detected by a group of behaviours each of which monitors the robot's motion and the state of its sensors to detect a specific set of conditions that identify a particular noteworthy feature such as a "corridor". When one such landmark is detected, an empty "behaviour-shell" is associated with it. In effect, the robot acquires a new behaviour specifically geared to identifying that landmark, and new "behaviour-shells" will only be assigned if no existing behaviour recognises the robot's current location. Behaviours constructed in this way are attached to each other to reflect their observed topological relationships. The resulting collection of behaviours acts as Toto's map. Toto can plan a path towards a particular target by activating the behaviour associated with that target. The activation will spread through the topological links between behaviours until it reaches the one associated with the robot's current position, and from there the robot need only trace the trail backwards to arrive at its target. This method of map construction and use makes use of

"active representation"- it is procedural and distributed, rather than declarative and centralised as in conventional AI.

The Nerd Herd [[8]]

This experiment in group behaviour consisted of 20 mobile robots cooperating in collecting a scattered set of "pucks". In building the robots' control systems, the idea of "basis behaviours" was introduced, and the Subsumption methodology was extended to facilitate the grouping of such behaviours into composite behaviours. Two types of grouping were used, direct and temporal. Direct combination summed the effects of the behaviours per actuator, while temporal combination switched between behaviours, with only one being active at a time. The structure these ideas lead to is good for multi-robot scenarios where robot's behaviours interfere with each other. It is also a suitable foundation on which to apply learning, where the robot learns what behaviours are useful under what conditions.

The Reactive Accompanist [[12]]

The reactive accompanist is a program whose goal is to derive the chord structure of a melody (played live) in real time, mimicking the ability of musicians to "play along" unfamiliar melodies. In contrast to the examples of Subsumption given so far, the accompanist is not a traditional robot, although it has many of the properties of one. This program is interesting because it is implemented in a totally different software environment to the MIT robot series, yet retains all the key features of Subsumption.

2.4.9 Limitations of Subsumption

Brooks originally proposed that a Subsumption architecture be comprised of a hierarchy of control layers where each layer is capable of overriding all inferior layers at any time and taking control of the robot for as long as it wishes [7]. It then relinquishes control to whatever inferior layer was previously in command of the robot. The layer which "subsumes" control of the robot at any point need have no knowledge of what is currently controlling the robot at that point, and similarly the "usurped" module need have no information about the "subsumer".

It can be argued that this scheme has a number of weaknesses:-

1. In practice it has been found that this very loose coupling between layers is not sustainable. Layers often need to pass information back and forth, but "pure" Subsumption control only allows this on an ad-hoc basis.
1. The division between layers is quite rigid. Thus, if a layer 2 competence decides that it wants to control the robot, a layer 0 competence cannot resist it. Such a decision is potentially hazardous if the layer 0 competence has abilities that the layer 2 competence neglects. If layer 2 does not understand the behaviour required to retreat in the face of an on-coming truck, but layer 0 does, layer 0 cannot stop disaster from occurring. This leads to the requirement that each layer implement the full competence of all lower layers in addition to its own tasks, which is undesirable because it wastes resources and makes it difficult to alter a given competence at a later stage. The typical solution adopted by the MIT robots is for higher layers to "grope around" in the internals of their subordinates- which is also undesirable, this time from a software engineering standpoint. The difficulty seems to arise from the rigid and early decision about what competences belong "above" others.

2.5 Hybrid Robot Architectures

Reactive robot architectures avoid the need for explicit planning, at least for basic robot behaviour. At some level, however, it may prove beneficial to integrate AI-inspired modules into the robot for conventional goal-oriented behaviour (although Brooks denies this). Unfortunately, traditional planning methods are proving difficult to apply, as they are just not designed for use in "situated agents" (objects that are in continuous interaction with the physical world).

The most straightforward form of plan to generate and execute is a simple sequence of instructions, each of which is executed in order. Such plans are applicable to domains where operations never fail, or where it is acceptable for an entire plan to fail if one of its steps fails. This clearly does not correspond to a real unconstrained environment, but will correspond to the environment of an industrial robot (which explains the success of such robots).

There are two basic ways to improve on such plans. It is possible to retain the plan-making process as it is, but to use a more sophisticated **executive** or plan-user which follows the plan closely but has greater freedom in the detailed actions it follows. An example of this is to formulate a plan as if it were controlling a robot entirely, but to add reactive control to the robot so it can, if necessary, suspend plan execution to navigate small unforeseen obstacles etc. Such a capability is a definite improvement. However the robot is still limited by being strongly committed to a single inflexible plan.

A second approach is to change the nature of the plan, by enhancing it to take account of the possible failure of a step and provide alternative responses based on the various failure modes. Although such enhancements can be continued ad infinitum and so would seem to be capable of giving arbitrarily sophisticated behaviour, in practice it is difficult to second-guess the environment, since the number of possible results of any action is exponentially large.

Given the problems with these two approaches, a large number of alternative mixes have been developed. A sample of these will now be described.

2.5.1 "Reactive Deliberation" Robot Architecture, Sahota [[13]]

This robot architecture tries to integrate reactive and goal-directed activity. The control system is built from two distinct entities, an Executor and a Deliberator, with the only essential difference between them being that the two modules operate on different time-scales (see Figure 2-10). The deliberator decides what to do and how to do it, while the executor interacts with the environment in real-time. This is the basic idea between a large number of hybrids combining conventional AI with reactive approaches. In effect, the reactive part of the robot is used as a form of "sub-conscious" for more reflexive tasks with local solutions, while the conventional deliberative component acts as the "conscious" goal-oriented component.

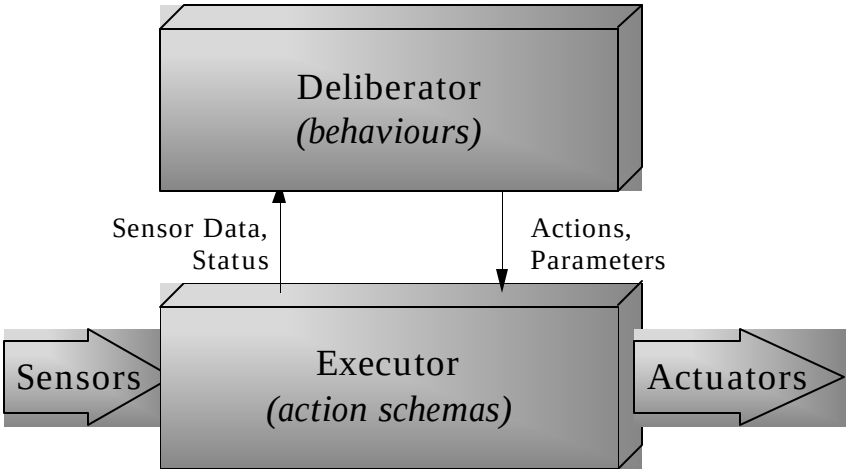


Figure 2-10: Reactive Deliberation

- **Executor**- this is a collection of "action schemas". Action schemas are units that, when provided with a set of run-time parameters, can accomplish specific tasks. The tasks themselves are independent of the robot's goals. The Executor partition of the system runs in real-time.
- **Deliberator**- this is a collection of "behaviours". Behaviours in this context are units that select an action schema, compute suitable run-time parameters to complete the schema, and make a "bid" based on a metric of the appropriateness of the schema given the robot's goals. Bids from different behaviours are compared, and a single action schema is selected for execution based on the results. The Deliberator partition operates on a longer timescale than the Executor.

The name of this robot architecture sounds like an oxymoron, given that reactivity and deliberation are so fundamentally different, but it is consistent with Artificial Intelligence nomenclature (e.g. "Reactive Planning"). This architecture has been used to build the control system of a pair of soccer-playing robots.

One constraint of the architecture is that it requires a strong system partition between modules implementing actions and modules that choose actions and the parameters for the actions. The first set of modules is run at real-time speeds, the other at a lower rate. This two-tier division requires all modules to be run at either of the two rates, with no in between. It is more useful if modules can be graded along a continuous spectrum from fast/reactive to slow/deliberative. Another constraint is that the architecture is built around the fundamental requirement that only one action is in progress at a time - so actions cannot be combined or actively compete.

2.5.2 GLAIR (Grounded Layered Architecture with Integrated Reasoning), Hexmoor [[14]]

This architecture divides the robot's control system into three levels (see Figure 2-11) :-

- **Knowledge Level** - this embodies deliberative modules, and deals with plans, beliefs, goals etc. Slower response time than other levels, but can implement complex control.
- **Perceptuo-Motor Level** - this embodies reactive modules that are in a tight interaction loop with the environment. Good response time, without complex control.
- **Sensori-Actuator Level** - this embodies reflexive modules, with very high response time, and very simple control.

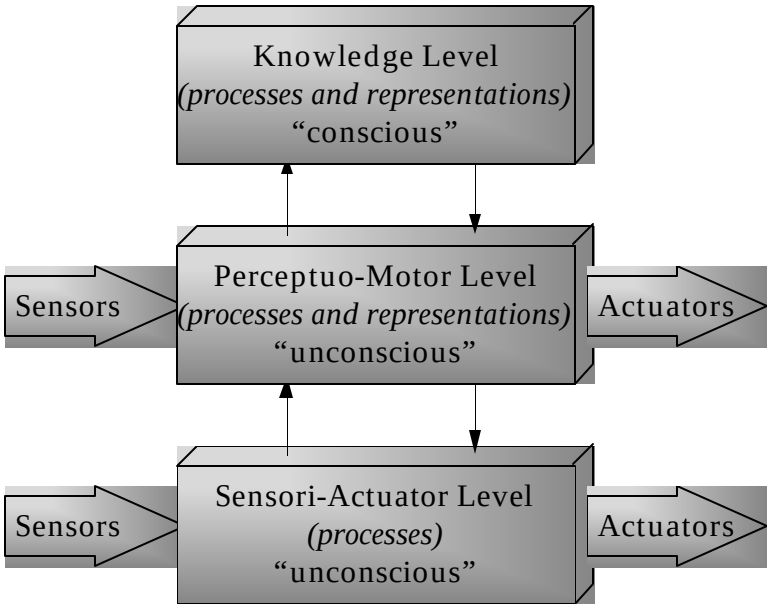


Figure 2-11: GLAIR Architecture

This is similar to the Reactive Deliberation architecture in that the system is partitioned into divisions operating at different speeds (in this case there are three divisions instead of two, allowing more flexible trade-offs to be made between speed and deliberation). Concurrency is allowed at the reactive and reflexive level, and conflicting actuator commands can arise. Therefore the architecture must provide an arbitration scheme for resolving such conflicts. The particular scheme used is to give commands from reflexive modules priority unless explicitly suppressed. This method of arbitration is less sophisticated than that allowed by Subsumption.

And, like the "Reactive deliberation" architecture, partitioning requires global decisions about speed versus deliberation trade-offs.

2.5.3 ACBARR (A Case BAsed Reactive Robotic system), Moorman&Ram [[15]]

This combines reactive techniques with case-based reasoning (see Figure 2-12). This scheme uses a reactive controller similar to that described for the "reactive deliberation" architecture in Section 2.5.1 (page 28), but the rest of the system is more sophisticated, providing support for:-

- Fine-tuning control parameters based on the success or difficulty a reactive "schema" is having in the environment.
- Recognising that a change in the environment makes a different set of parameters and active schemas appropriate, using a library of useful parameter/schema combinations.

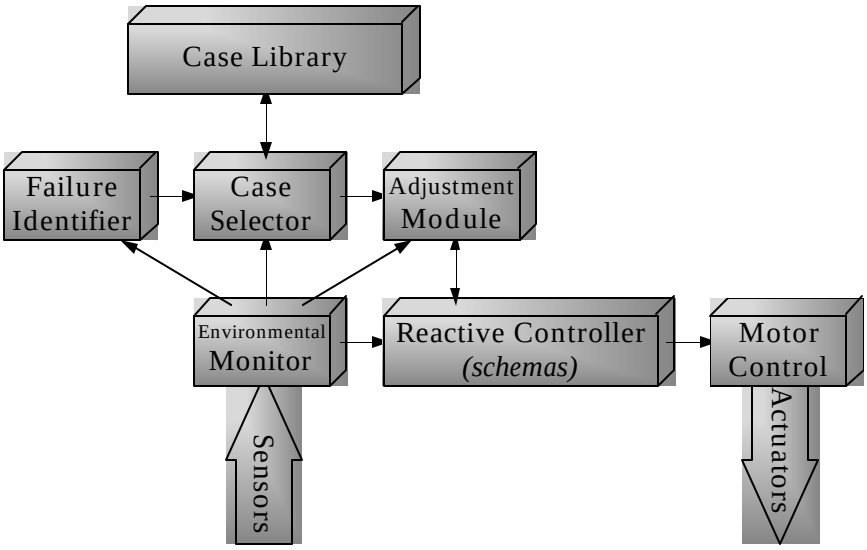


Figure 2-12: ACBARR

It is difficult to make a fair comparison between this architecture and those enumerated previously in this chapter, since it is quite different in its overall approach. If a "case" is thought of as an overall behaviour or strategy of the robot, then the criticism can be made that strategies cannot be combined in this architecture (although lower-level schemas can). Again this architecture draws a sharp artificial partition between the robot's actions and how those actions and their parameters are chosen.

2.5.4 Action Selection Network, Maes [[16]]

This technique for building "situated agents"[1] allows planning-like activity to occur without the presence of a classical controller. Instead, the "competence" modules (which are the objects of planning) are connected by a network which spreads an "activation level" dynamically between the modules in a way that combines deliberation with speed. Philosophically, this system is based on the notion that you should not "tell" the robot how to achieve goals, but instead let it find a control loop involving both the system and the environment which will converge towards the goal. Planning is therefore a situated dynamic activity.

The action selection network is a collection of "competences" which are individually characterised by enumerating the following:-

- A "precondition list" of propositions that must be true before the competence can be activated.
- A threshold specifying the activation level a competence must reach before it can be activated.
- An "add list" of conditions/propositions that become true after the competence has been activated and completed its action.
- A "delete list" of conditions/propositions that become false after the competence has been activated and completed its action.

A network is constructed of "successor", "predecessor", and "conflicter" links based on these descriptions.

- For every proposition in an "add list" of one competence and a "precondition list" of another, a successor link is added from the former competence to the latter. This models the fact that the first sets up a precondition of the second.
- A predecessor link is added everywhere there is a successor, but reversed.
- For every proposition in a "precondition list" of one competence and a "delete list" of another, a conflicter link is added from the former competence to the latter. This models the fact that the first is made unexecutable by the second.

[1] The term "agent" is very broad, and in general usage it refers to any software entity that is in some sense autonomous. Situated agents are software entities with a physical embodiment- i.e. robots.

Once the static network has been constructed, it is dynamically and continuously updated. If all the propositions in the "precondition list" of a competence have been met, then it is "executable"- there is nothing preventing it from being executed, except the presence of other similarly executable competences with higher activation energy levels.

- If a proposition in the "precondition list" of a competence is observed to be currently true, activation energy is inserted into that competence.
- If a proposition in the "add list" of a competence is a global goal of the robot, activation energy is inserted into that competence.
- If a proposition in the "delete list" of a competence is a protected goal of the robot (one it has completed and does not want to undo), activation energy is removed from that competence.
- If a competence is executable, then a fraction of the competence's activation energy is spread to its successors- this allows the system to anticipate in advance the consequences of a competence being activated.
- If a competence is *not* executable, then a fraction of the competence's activation energy is spread to its predecessors- this allows the system to encourage suitable competences to work to make a desirable competence executable.
- Every competence decreases the activation energy of its conflicters by a fraction of its own activation energy (with provision to prevent mutual inhibition). This allows competences to try to disable other competences that would undo their preconditions.
- A decay function is used to keep the overall activation level constant.

With the action selection network running, all that remains is to choose the "best" (highest activation) executable competences at any given time and activate them. The network can be tuned using various parameters controlling the spread of activation.

The major constraint of this system is that it implicitly requires that all actions of the system can be described symbolically in before-and-after terms. So although action selection networks move away from classical planning to an approach that utilises the robot's physical embodiment, they still retain the Symbol System bias.

2.5.5 Reactive Action Package (RAP) System, Firby [[17]]

The RAP system facilitates the reactive execution of symbolic plans (see Figure 2-13). Plans or "tasks" are implemented by giving a list of methods for performing the task, along with the conditions under which the methods are useful. Methods are either primitive actions or a list of sub-tasks. Suitable methods are attempted sequentially until they either finish or fail. On failure, alternate methods are tried in turn. Such a task description is called "situation driven".

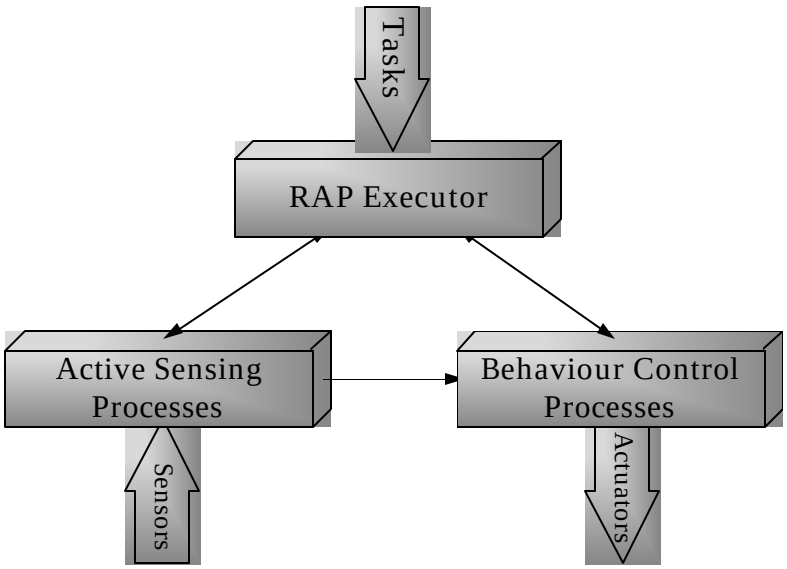


Figure 2-13: The RAP System

In the original RAP system, actions were assumed atomic - i.e. that they had a well-defined finish and that success and failure of the actions were equally well defined. It was also assumed that it is appropriate to complete one action before moving on to another. Both of these assumptions are limiting, so later versions of the system moved towards removal of the idea of "success" and "failure", and a change of focus from process steps to time-extended activities. This architecture again retains the need to operate at a symbolic level, with all the difficulties that entails (see Section 2.4.7, page 22).

2.5.6 ATLANTIS, Gat [[6]]

This is a combination of a reactive control substrate with a classical planning system. Planning is seen as an attempt to transform one world state to another using "operators" which map on to associated physical actions when executed. In the classical approach, operators are executed in an atomic fashion, so there is a strict one-to-one correspondence between

operators and actions. This is not amenable to continuous, overlapping, interruptible actions. ATLANTIS reallocates the role played by operators to "activities" and "decisions". Decisions are operators, but do not directly affect the world. Instead they start or stop "activities", which in turn affect the world. Activities are potentially long-running processes, and since atomicity is no longer an issue there may be several activities in progress at any time, interleaved or executing in parallel.

ATLANTIS has three main components:-

- **The controller**- this is concerned with activities that are mostly reactive. "ALFA" (A Language For Action) [[18]] was designed to aid the engineering of this component.
- **The sequencer**- this is concerned with controlling sequences of both physical activities and deliberation. It is designed around the idea of "*cognisant failure*". Rather than trying to design algorithms which never fail, it is of more practical worth to build algorithms which may fail but will detect that they have failed, so the system can take corrective action. The sequencer manages the activation of modules in the controller, monitors them, and providing them with suitable parameters for their operation. Decisions are based on tasks the robot has to perform. Like in the RAP system, tasks are described as a list of methods for performing the task, along with the conditions under which the methods are useful. Methods are either primitive actions or a list of sub-tasks. Tasks are attempted sequentially until they either finish or fail. On task failure, alternate methods are tried in turn. A resource list is attached to all activities to prevent conflict.
- **The deliberator**- this performs computationally expensive, long term tasks such as planning and maintaining a world model. Such computations are initiated by the sequencer and may be terminated by it if resources are scarce. The deliberator acts as an advisor only to the sequencer, with overall control of the robot remaining with the sequencer. ATLANTIS uses the idea of "plan-as-communication" rather than "plan-as-program" [[19]].

The ALFA language used in the controller is described in some detail in Section 2.6.1 (page 37). It is similar in ways to the language used with Subsumption. In particular, when priorities are used to resolve conflicts, those priorities are hard-coded as with Subsumption[2].

2.5.7 The Architecture Control Kit, Rosca [[20]]

This hybrid reactive/deliberative architecture is built from:-

- "Objects"- these are declarative entities, containing attributes, with knowledge sources attached, and connected with other objects. Objects are used to represent input sensors, and system "beliefs" in fixed symbolic form.
- "Knowledge Sources"- these are procedural entities, specifying how objects are processed. These can act like rules in a rule-based system, and may be expressed in the form:-

  WHEN (opportunity-test) IF (appropriateness-test) THEN (body)

If the value of an attribute changes, all knowledge sources connected to the object the attribute belongs to are activated. These knowledge sources may affect other attributes in the same or other objects in turn, causing a cycle to occur. This is the main execution loop.

The main advantage of this simple architecture is that making knowledge sources act in parallel is straightforward, and is just a question of ensuring they do not affect the same attributes. It is a conceptually simple architecture, combining general purpose computation and representation with a "triggering" system reminiscent of PLC devices for controlling industrial machinery- very much a reactive idea.

Again, this architecture is symbol-oriented, and thus difficult to ground.

[2] The priority system in ALFA is arguably an improvement on Subsumption, in that the hard-coded priorities are set at the point of connection rather than at the source, which means that the information for resolving conflicts is available locally - this is better from a software engineering viewpoint.

2.6 Review of Languages for Robot Architectures

Some of the architectures discussed above have languages associated with them to support some aspects of their organisation and structures. Examples of these are described in the following sections.

2.6.1 ALFA (A Language For Action) [[18]]

This language is used in the reactive controller partition of ATLANTIS (see Section 2.5.6, page 34), but can support other architectures such as Subsumption. A program in ALFA consists of a set of "modules" connected by "channels".

- A module is a control element that transforms a set of inputs to a set of outputs using either dataflow or state-machine computations. Dataflow computations have no history (and are therefore constant time functions), while state-machines may have history (and so may have time dependencies). Modules consist of internal registers, timers, and a set of methods that are tried in turn on failure.
- A channel is a control element that combines a set of inputs into a single output using purely dataflow computations. Inputs and the output come from and go to modules or the outside world. There are four ways inputs can be combined:-
  A) By selecting the minimum
  A) By selecting the maximum
  A) By selecting the average
  A) By selecting the highest priority input that is active, given a fixed priority ordering of the inputs

The idea of implementing a control system using objects corresponding to "channels" and processing "modules" is used in this thesis, but with the semantics of the channels considerably modified (see Section 3.3.1, page 48).

2.6.2 The Behaviour Language/"New Subsumption", Brooks [[21]]

This language is used to specify the augmented finite state machines needed in Subsumption. The smallest procedural unit is a "real-time rule", equivalent to reactive *condition Þ action* rules [[5]]. These appear either in isolation, or grouped into "behaviours".

Rules have two forms:-

- **Whenever** - a rule of this form continuously monitors for a condition, and executes a procedural body if it becomes true.
- **Exclusive** - this type of rule is composed of a set of Whenever rules, each of which is continuously monitored. As soon as one succeeds, it is committed to and the others are not checked again until the successful rule's body terminates.

When a number of rules are collected together to form a "behaviour", a list of inputs and outputs to that behaviour are specified. "Wires" are one method of communication between behaviours. A wire can transmit messages from a single source to multiply targets. Or, instead of transmitting copies of a message to its target, it can suppress, inhibit, or default its output.

- **Suppress** - sends messages to target and blocks messages from any other source to target.
- **Inhibit** - blocks messages from any source to target.
- **Default** - messages are sent to target if no messages have been received from another source.

Behaviours can be active or inactive. When inactive, rules tagged as "haltable" are stopped, and rules tagged as "inhibitable" are left running but have their outputs inhibited. Each behaviour has an associated "activation level" and "activation threshold" used to determine if it should be active. Activation levels can be managed in two ways:-

- **The Hormone System** - this uses entities called "conditions" which have an activation level that can be "excited" by any rule, and decays with time when not excited. Functions of these conditions can be used to control the activation levels of behaviours.
- **Spreading of Activation** - this uses an action selection network [[16]], as described in Section 2.5.4 (page 32).

The work described in this thesis tackles many of the same issues as those addressed in this language, such as the various ways "wires" can be connected, and how activation of modules should be managed - but the solutions advanced are quite different (see Chapter 3).

2.7 Review of Cartographic Schemes for Robot Navigation

In the following sections, approaches in the literature to map building and use are discussed. This review is needed in Chapter 4 as background for the cartographic issues associated with the sentry application.

2.7.1 Grid-based and topological methods

If the robot has very accurate knowledge of its position, it can construct accurate maps simply by representing the environment as a "grid", rather like a chess board. When a robot's estimate of its position is not perfect, as is normally the case, this representation scheme breaks down because it has no way to cope with uncertainty in the motion of the robot [[48]]. "Topological" maps try to overcome this difficulty by abstracting away from the environment. They try to represent which regions are connected to each other, rather than trying to capture the exact shape of objects. Such maps are much simpler to plan with. This type of map representation is useful for robots with long-range sensors, such as sonar or vision sensors [ibid.]. As will be argued in Section 4.3 (page 83), it is not useful for robots with proximity sensors alone, because such a robot cannot detect topological features in the environment directly.

2.7.2 Potential Fields, Arkin [[22]]

In this system, all the entities in the robot's environment are seen as generating individual "potential fields" which, when evaluated and summed at the robot's current position, give the direction and speed at which the robot should move. Each field is controlled by a "motor schema". As an example, for each static obstacle known to be in the environment, an "avoid-static-obstacle" motor schema is instantiated which gives strongly repulsive vector contributions when the robot is near the obstacle they are associated with (see Figure 2-14).

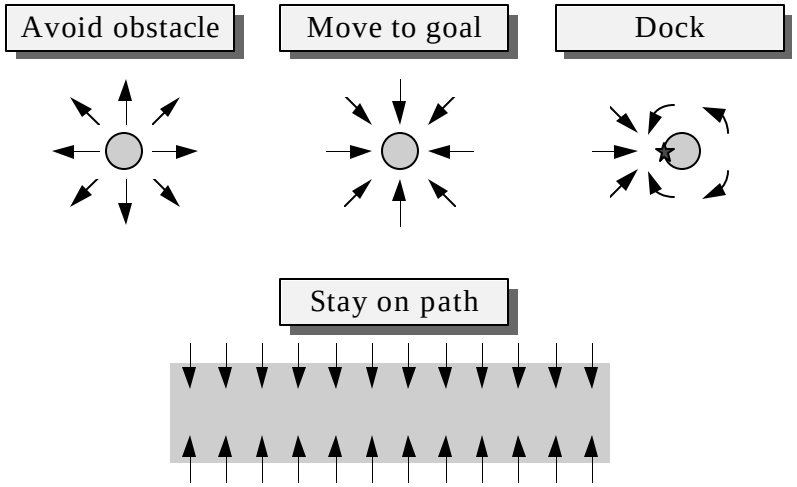


Figure 2-14: Potential fields of sample motor schemas (simplified)

The motion of the robot is determined by summing the required vectors of all motor schemas at its current position. A "noise" schema is added to remove problems of local minima. The actual global path the robot is to traverse is never calculated. Cartographic information in the form of a static map of the robot's environment is used to construct a set of linear path segments for a particular robot mission, then behaviours and schemas appropriate for implementing this path are chosen. The actual path is evaluated dynamically through the interaction of motor schemas as the robot moves.

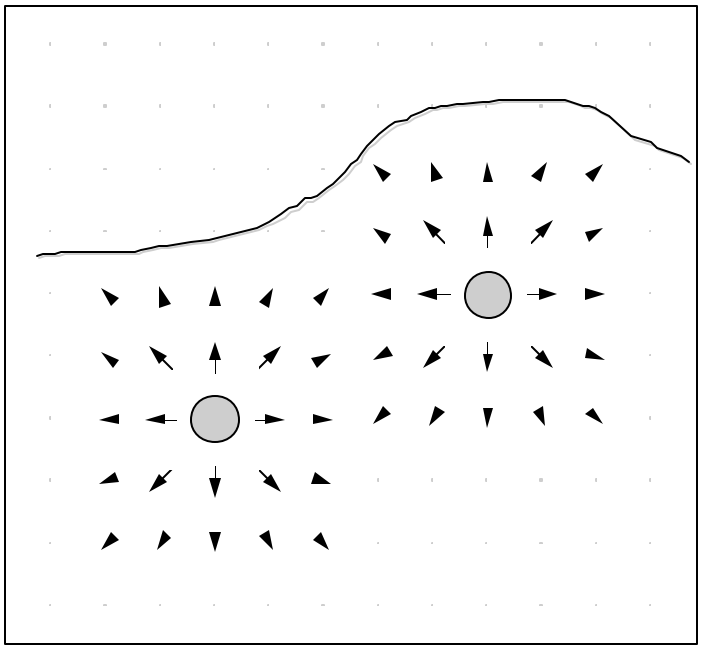


Figure 2-15: Path of robot resulting from combined potential fields

This system allows a reactive robot to negotiate complex terrain. The overall path followed can have the appearance of being the result of elaborate planning, but in reality arises from purely reactive decision making (see Figure 2-15). This system is suited to situations where the environment is static and all the obstacles within it have been mapped. It is not ideal for dynamic situations or situations with incomplete information.

2.7.3 Internalised Plans, Payton [[23]]

When plans are constructed using state-space search, the planning process generally finds an optimal path, returns this as the result, and if the assumptions under which the planning was done change, the process is repeated from the beginning. In particular, if the robot is forced to make even a minor deviation from the plan, it can only recover by generating a new plan for its changed circumstances. Internalised plans offer a way to avoid this. In the planning process, the cost from various intermediate points to the goal will typically be considered, and then an "optimal" route selected to minimise the cost. Instead of committing to such a route, internalised plans store this costing information for all points in the map, then construct a gradient field showing the optimal direction to move in when at a given point (see Figure 2-16). Now the robot always knows the best direction to move in, no matter where it happens to be. To reach the goal from any point, it simply needs to follow the gradient field. Because the field is generated from cost information, the path it directs the robot along will be optimal, rather than a heuristic guide as for potential fields (see Section 2.7.2). It also does not have the problem of local minima possible with potential fields.

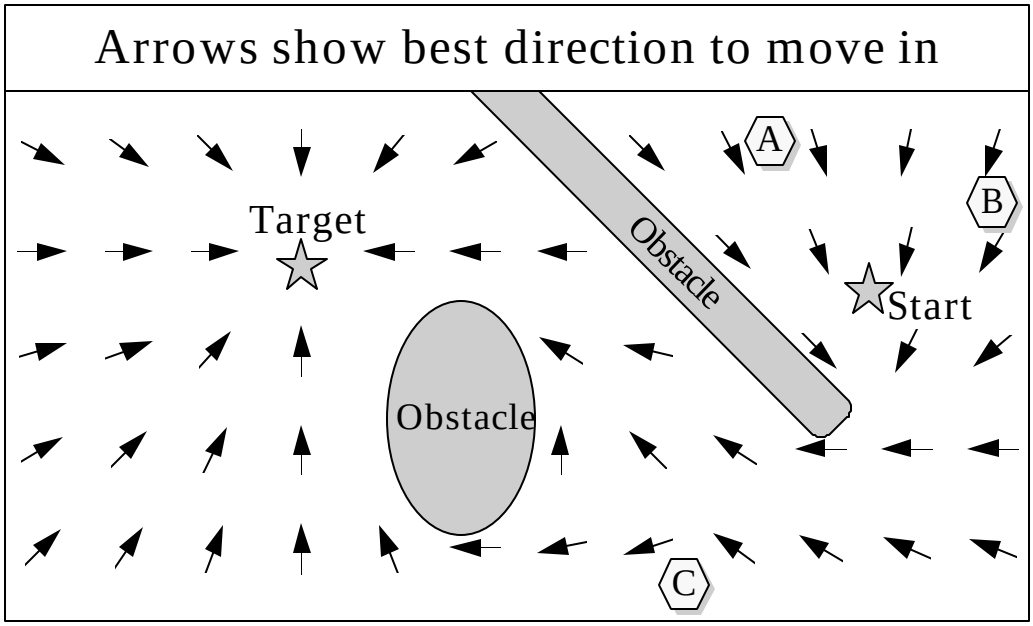


Figure 2-16: Internalised plan, gradient field

Figure 2-16 shows the gradient field associated with an example of an internalised plan. Note that the vectors are the result of planning, so for example the vectors at A, B or C show true optimal direction for the robot to move in at that point. This direction could not be calculated from purely local considerations, and so is superior to potential fields. The diagram shows that the technique has the useful feature that if the robot deviates from the optimal path to its goal, it can immediately continue to move towards the goal again without having to stop and re-calculate its plan. This is important for good real-time performance, since it allows the robot to make diversions around unforeseen obstacles and then return without interruption to approaching its target. One drawback of this technique is that it involves more computation than either the potential field approach or traditional planning. Also, if the target moves, the field must be recalculated. If an autonomous robot was entirely dependent on its internalised plan, then the burden of generating it whenever the target moved would threaten the robot's ability to operate in real-time. Internalised plans are best used as "advice"- information that improves the robot's behaviour when it is available but which the robot can function without. When used this way, the plan can be constructed by a background task without freezing the robot. The technique developed in this thesis for target-seeking has some similarities to internalised plans, and is described in Chapter 4.

2.8 Summary

This chapter has reviewed the literature relating to robot architectures and robot cartography. Representative examples of work from both fields have been presented- both are too large for exhaustive coverage. The review of robot architectures will serve as background to Chapter 3, in which a new architecture is developed that extends and improves on comparable work. The review of cartography provides context for Chapter 4, in which a cartographic system for a robot with proximity sensors is developed.

3. Lateral

This chapter presents Lateral, a novel behaviour-based robot architecture. Lateral extends the popular Subsumption architecture developed by Brooks [7]. It allows behaviours to be combined in a much more flexible way than is possible in Subsumption. The techniques used to allow this are examined here, and a number of comparative examples are presented. Also discussed are practical issues concerning the problem of implementing the Lateral architecture in a form that is "light-weight" enough to be used with lower-end robots with limited memory and simple on-board processors. The architecture developed in this chapter is used in Chapter 5 to build a specific robot control system.

3.1 Overview

Lateral is a behaviour-based robot control architecture. In other words, the robot's control system is composed of a collection of modules called "behaviours" that operate in parallel (see Section 2.4, page 15). This contrasts with the arrangement favoured by traditional architectures, where the control system is divided into a sequence of modules that progressively transform sensor input through various representations until actuator output is produced (see Section 2.2, page 9). As a consequence, behaviours all have access to sensors and actuators, rather than being "sandwiched" between modules that abstract away from perception and motor control. Behaviour-based architectures must provide a scheme to resolve conflicts in the case of different behaviours attempting to control the same actuator. It is here that Lateral diverges from Subsumption. The Lateral architecture does not require that behaviours be strictly layered in a rigid hierarchy to determine which should take precedence in the case of conflict, as Subsumption does (see Section 2.4.3, page 18). Instead it implements a dynamic priority system where the priority of a given behaviour is affected by the priority of any other behaviours that make use of it for their own purposes. As the demands of behaviours on each other change, the flow of priority between them changes to reflect that, and the effective precedence hierarchy of the behaviours in the control system alters dynamically.

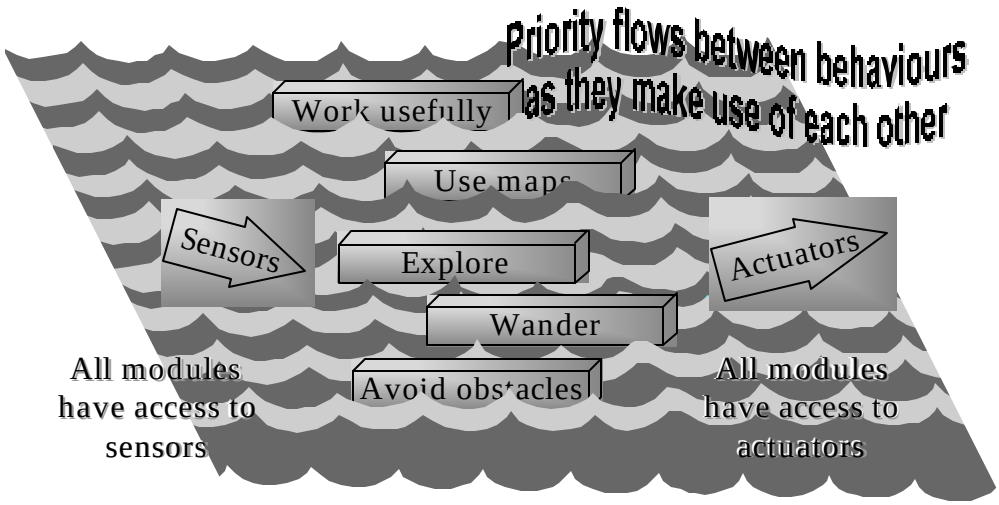


Figure 3-1: Lateral architecture

In Lateral, as the priority of a behaviour rises, its ability to exert influence over other behaviours increases. In Figure 3-1, the sea is used as a visual image to suggest the analogy of "waves" of priority lifting behaviours up to higher precedence. As one behaviour rises on a "wave" of priority, it can pull any behaviours it uses up with it to this higher level of priority by passing on its priority to them. It can do this selectively, by choosing how much of its priority it is willing to pass on or "sponsor" other behaviours with. The idea of sponsorship provides a principled way to resolve conflicts between groups of competing or co-operating behaviours. The architecture is thus more scaleable than Subsumption, which does not lend itself to managing groups of behaviours, as will be discussed in Section 3.4, page 59. This is the main benefit of Lateral- it extends the range of application of behaviour-based systems by improving their scalability (see Figure 3-2).

The ideas introduced here are presented in more detail in the sections that follow, with numerous examples to illustrate the nature of Lateral's priority system.

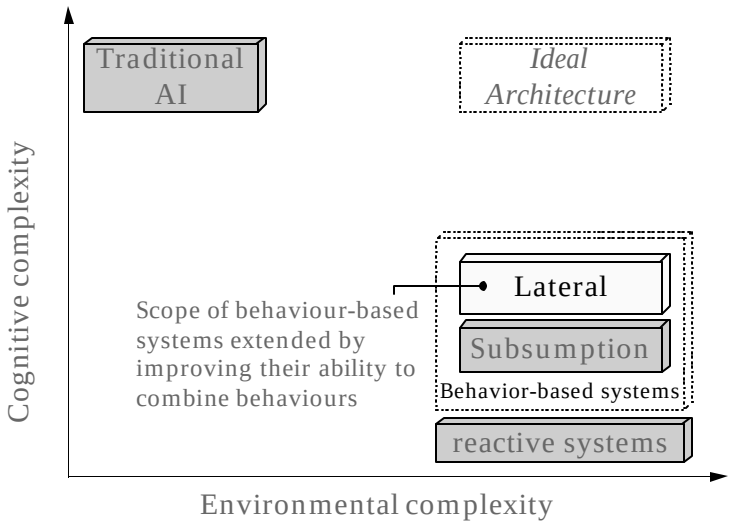


Figure 3-2: Lateral in relation to other robot architectures

3.2 Conflict Resolution in Lateral

To clarify the ideas presented in the previous section, consider the example control system shown in Figure 3-3.

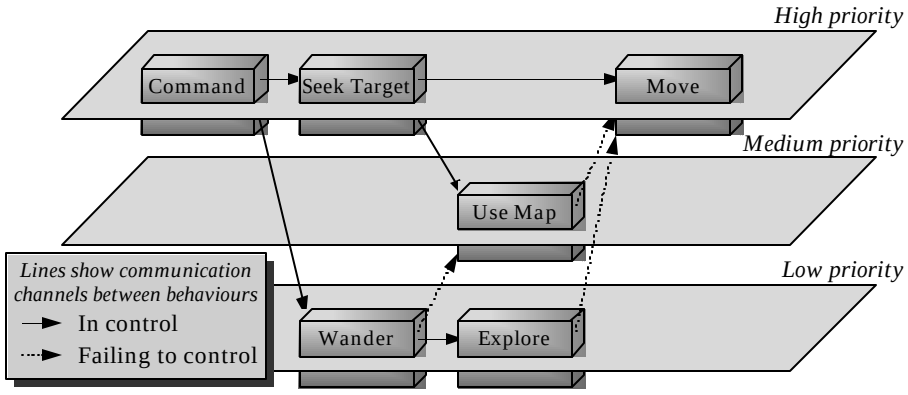


Figure 3-3: An example control system in Lateral

The "Command" behaviour shown has a high priority. Behaviours with high priority can take precedence over behaviours with lower priority when they compete with them for control of other behaviours they wish to make use of. They can also choose to sponsor selected behaviours they favour with their own high priority so they can rely on those behaviours taking precedence over competition as well. In the example, the "Command" behaviour chooses to sponsor the "Seek Target" behaviour, pulling it up to its own level of priority. As a

consequence that behaviour is able to win control of further behaviours ("Move" and "Use Map") despite competition by "Wander"- which does not have as much sponsorship.

The choices a behaviour makes about who to sponsor, and at what level, are allowed to change at any time in Lateral. Figure 3-4 shows the same example control system, but with the "Command" behaviour choosing to sponsor "Wander" rather than "Seek Target". Compared with Figure 3-3, the priorities of a number of behaviours have changed significantly. Some have lost priority, some have gained priority- and the changes are not confined to the behaviours that "Command" controls directly, but extend throughout the system.

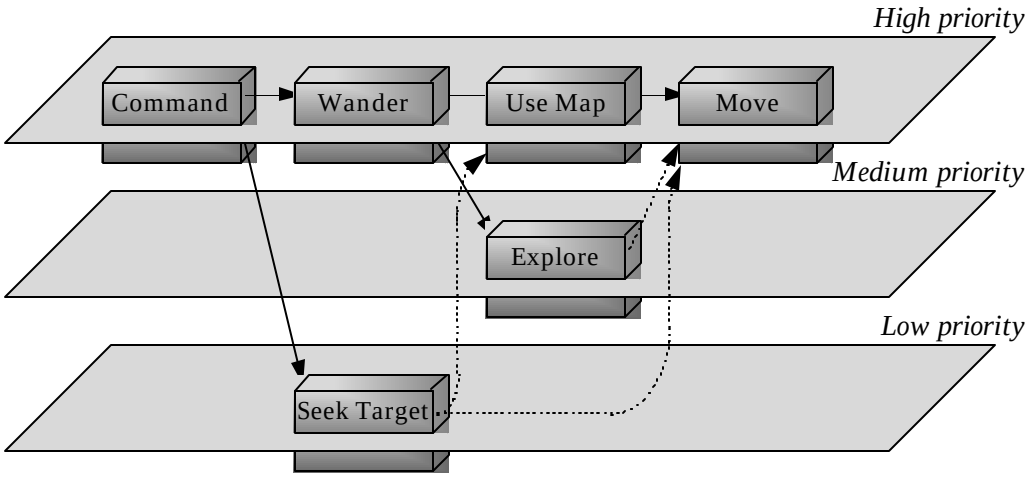


Figure 3-4: The example control system, with different sponsorship

While the potential advantages of a dynamic priority system over static systems such as Subsumption have long been noted [[47]], there have been a number of stumbling blocks to a practical realisation of that potential. The basic problem is that for a module to correctly choose its priority, it must know how important it is compared to all other modules in the system. Implemented directly, this would seem to require every module to have an excessive amount of global knowledge. The Lateral architecture side-steps this problem by implementing the priority system more within the channels of communication between modules than within the modules themselves. It uses the system of "sponsorship" mentioned above, where the priority of a behaviour module is set to the level that the highest priority behaviour using it requests it should be. This recursive definition of priority allows the *absolute* priority

of behaviours to be determined while only requiring that each behaviour make a *local* decision about the relative importance of the behaviours it uses[3].

The following sections introduce the elements from which the Lateral architecture is built, and how the priority system is implemented within them. The ideas discussed in this section will be returned to in detail in Section 3.3.2, page 55.

3.3 Elements of Lateral

The Lateral architecture consists of *behaviours* and *connections*. Behaviours implement competences of the robot, while connections channel information and commands between behaviours. Arbitration between conflicting commands is resolved within connections using Lateral's dynamic priority system. Priority flows from behaviour to behaviour through the connections, "piggy-backed" on the information flow. The details of this system will now be discussed, and then a comparison with Subsumption will be made to illustrate the advantages the system has.

3.3.1 Connections

All control information passed between behaviours in Lateral is channelled through objects called *connections*. It is important for behaviours to interact through such intermediaries so that the conflict resolution always necessary in a behaviour-based architecture can be enforced (see Section 2.4.1, page 16). Connections in Lateral are analogous to *wires* in the Subsumption architecture (see Section 2.4.4, page 19), except that as well as carrying values and messages from their source to their target, they also carry the priority of the originator of their content. This reflects the fact that the conflict resolution scheme in Lateral relies on dynamic priorities, in contrast to the simpler fixed priority scheme of Subsumption (see Section 2.4.3, page 18).

3.3.1.1 Advantages of explicit information channels

A benefit of having objects that explicitly channel the information flow between behaviours is that higher-level behaviours may now "tap in" to this flow and selectively modify it to suit their own purposes, *without* having to rebuild any of the behaviours or modify the information

[3] Normally modules would be connected in a tree structure, but cycles are possible and acceptable- see Section 3.6.1.

channels already present. This is fundamental to the operation of Subsumption- in fact this is where the idea of higher-level behaviours "subsuming" lower-level ones takes its name. Lateral retains this capability, and expands on it.

The two useful ways to "tap in" to the communications between behaviours are :-

- *Accessing* the information flow in a channel.
- *Selectively overriding* the information flow in a channel.

It should be possible to achieve both of these without having to rebuild any of the information channels being tapped. Also, it is useful if the tap itself can also be seen as an information channel, since then that tap can be accessed and overridden in turn by higher-level behaviours.

3.3.1.2 Implementing information channels as connections

In Lateral, connections implement the ideas presented in the previous section. A connection has a single *source* and a single *target*, both of which can be attached to other connections. At its simplest, the connection *accesses* the information flow at its source, and tries to *override* the information flow at its target with a replica of what it reads from its source. It will only succeed in overriding its target if the information flow it is copying from has a higher priority than the information flow in its target.

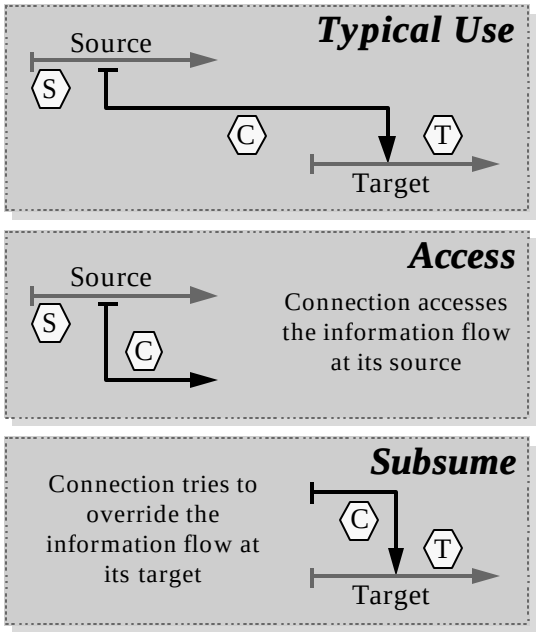


Figure 3-5: Basic use of connections

An example connection is shown in Figure 3-5. Connections are drawn as an arrow, with a bar to mark their source and an arrowhead to mark their target. In Lateral, the connection C in the diagram could itself be made the target of some other connection, and in this case it will access either its normal source *or* this new connection, depending on which has the higher priority. This is how overriding of an information channel occurs. In fact any number of connections can have the same target, in which case the information flow in the connection they target will be a copy of whichever of them has the highest priority- or a copy of its normal source if none of them were of a high enough priority to override that. This situation is shown in Figure 3-6. The connection C could also be made the *source* of some other connection. This has no effect on C, as the connection using it as its source will only access C and will not attempt to override it- connections only attempt to control their targets, not their sources. Any number of connections can have the same source. This is also illustrated in Figure 3-6.

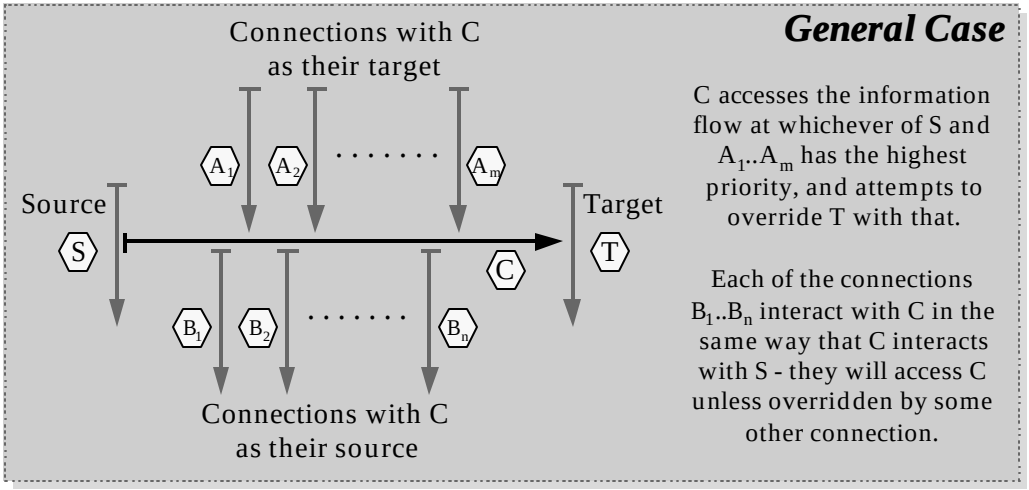


Figure 3-6: Using connections, general case

The priority of a connection is always the same as that of the connection from which it is reading. Hence a connection carrying high priority data will propagate that priority to any connection to which that data is passed.

3.3.1.3 Attaching connections to behaviours

So far, methods for attaching connections to other connections have been shown, but no way of attaching them to behaviours has been given yet. To achieve this, each behaviour is given two sets of connections, *inputs* and *outputs*. A behaviour can read and write directly to its input and output connections. Output connections can have their targets attached to other

connections, outside the behaviour, with which communication is desired- such as input connections in other behaviours. Output connections generally do not have their source attached to anything[4], because the behaviour that owns them is directly controlling the information they carry and so they need no source to read from. Input connections generally have neither their source nor their target attached to anything, leaving them free to be controlled by any external connection that attaches their target to them. Alternatively, they could have their sources attached to other connections which the behaviour wishes to monitor. These possibilities are illustrated in Figure 3-7.

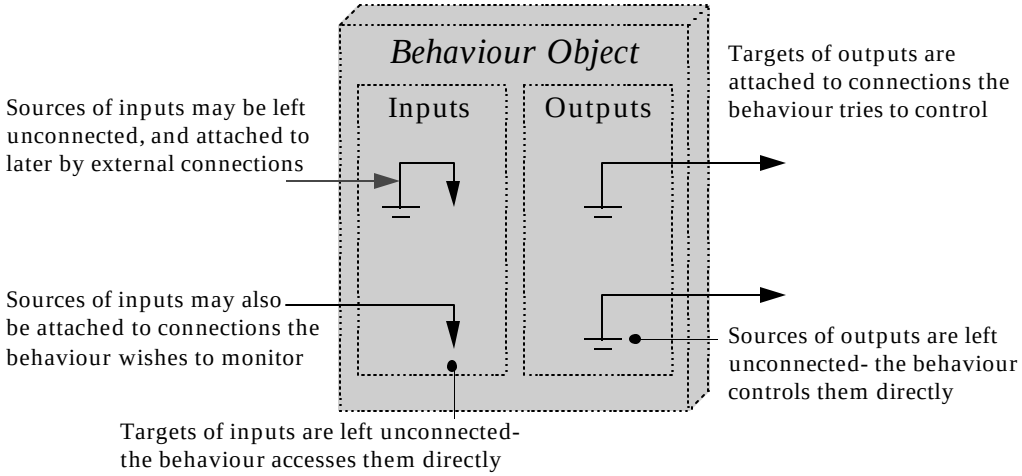


Figure 3-7: Input and output connections in a behaviour

Leaving both the source and target of an input connection unattached is a convenient arrangement when the semantics of the input are such that it is used to control some aspect of a behaviour, rather than provide raw data for the workings of the behaviour. In such cases, the behaviour should not be concerned with what is controlling it, only the task that it has to perform. Leaving the source unattached reflects this indifference, and means the behaviour does not have to be re-configured if it is later decided to change which behaviour controls it. An arrangement where the source of an input is left unattached and several external connections compete for control of it turns out to be such a common and useful arrangement that it is convenient to introduce a "short-hand" way of producing a diagram of the situation (see Figure 3-8).

[4] This is denoted in diagrams by replacing the bar at the source of the connection with the ground symbol from electronics. If the target of a connection is unattached, it is simply drawn curling back, pointing at nothing.

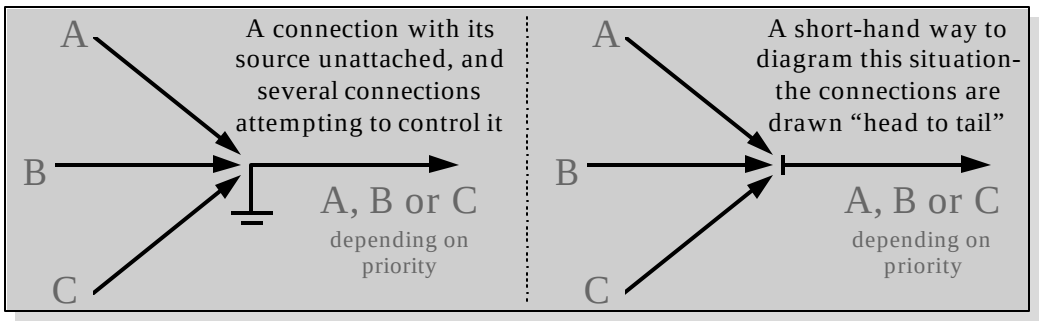


Figure 3-8: Special diagramming convention for inputs with no source attached

3.3.1.4 Dependencies between connections

At this stage the dependencies between connections will be looked at more closely, as a prelude to examining the full details of Lateral's conflict resolution scheme. The fundamental dependencies are as follows:-

- The information flowing through a connection C is drawn from either the connection to which its source is attached, or from any connection which has C attached as its target (since these may succeed in overriding the normal source if they have sufficient priority). In the rest of this chapter, the normal source of a connection C is labelled its "*primary source*", and any connections which have C attached as their target are called its "*secondary sources*". This is a reasonable label to apply, since these connections may effectively become C's source when their priority exceeds that of its primary source. The primary source of a connection will often simply be one that lower-level behaviours have configured it to read from, and the secondary sources could be overrides added by higher-level behaviours.
- The information flowing through a connection C may be passed on to the connection to which its target is attached, and to any connection which has C attached as its source. In the rest of this chapter, the direct target of a connection C is labelled its "*primary target*", and any connections which have C attached as their source are called its "*secondary targets*" (since they may copy from C if it has sufficient priority, effectively behaving as if they were C's target). The primary target of a connection will often simply be one that lower-level behaviours configured it to write to, and the secondary targets could be higher-level behaviours "listening in".

To summarise, primary sources and targets represent the communications channel that the connection was created to establish. Secondary sources and targets are added by behaviours which wish to monitor or influence that channel respectively. Secondary attachments can be made without having to re-configure the original connection (see Figure 3-9).

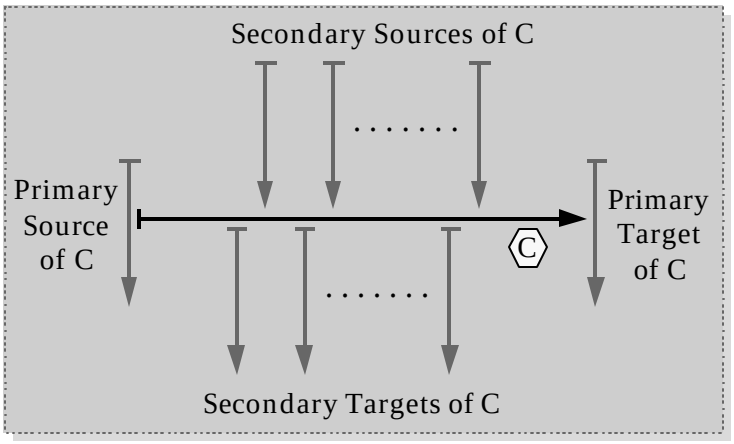


Figure 3-9: Connections in Lateral

Figure 3-9 illustrates the various ways that connections can be attached to each other.

3.3.1.5 Detailed conflict resolution between connections

Arbitration between connections attempting to control the same target is straightforward when they are at different priorities. The information flow that a connection carries is always a copy of the information flowing in its primary source- unless there is a secondary source at a higher priority, in which case the information flow in the connection is overridden by the secondary source with the greatest priority. Put simply, a connection copies the information flow from whichever source, either primary or secondary, has the greatest priority. There is no effective difference between primary and secondary sources under these conditions - the distinction merely reflects whether the source is the one originally configured, or a later override.

However, when a number of competing connections are at the *same* priority, it is not obvious what the correct way to arbitrate between them is. The normal result in Lateral is that messages from every member of a group of competing sources will be passed on when they have the same priority. This is called "sharing". However, connections can be assigned an extra "type" tag if it is important to explicitly control arbitration under these conditions. This tag is a feature of Lateral designed so that when connections are all at the same priority they may

interact in a way that is equivalent to wires in Subsumption. All the different ways wires can interact in Subsumption can also be used in connections (see Section 2.4.4, page 19). Connections can be tagged as either "Default", "Shared", "Preferred", or "Replace". The semantics of the different possibilities are given in Table 3-1. They are chosen to be consistent with corresponding ideas in Subsumption. Remember these distinctions are only relevant when connections are at the same priority.

Table 3-1

TYPE OF CONNECTION	DESCRIPTION
Default-connection	The connection passes on messages only if its target is not receiving messages from its primary source.
Shared-connection	The connection contributes its messages to be merged with those of its target's primary source. This is the normal type of connection.
Preferred-connection	The connection passes on messages if it has them, blocking its target from reading from its primary source. If not, its target is allowed to read from its primary source again.
Replace-connection	The connection passes on messages if it has them. Regardless of whether it has messages, it blocks its target from reading from its primary source.

Note that when this form of arbitration is in use, a distinction appears between primary and secondary sources of a connection. This mirrors the asymmetry in Subsumption between wires and taps on those wires. Suppression in Subsumption corresponds to a "preferred-connection", defaulting is equivalent to a "default-connection", and inhibition can be implemented using a "replace-connection" (the connection should however be enabled or disabled rather than sending or not sending messages as in Subsumption). See Section 2.4.4, page 19 for a description of these Subsumption constructs. If a number of secondary sources of the same priority are present, "replace-connection" is given highest priority, then "preferred-connection", then "shared-connection", and "default-connection" is given lowest priority. Streams of messages from connections of the same type, if present, are merged.

Effectively, the "type" tag allows a static priority scheme to be superimposed on top of the dynamic priority scheme used in Lateral. A useful consequence of this is that it allows the Lateral priority system to be used simply to merge a collection of Subsumption-like static hierarchies. The robot could then dynamically switch between these hierarchies as appropriate during its operation, with Lateral taking care of the details of selecting the correct behaviours and rearranging them into the new hierarchy. A more complete comparison of Lateral with Subsumption is given in Section 3.4, page 59.

The "type"-tag feature was included to make Lateral a true superset of Subsumption- it was not in fact needed for the robot application described in the latter part of this thesis. The control system for this application (see Chapter 5) was designed to use the full power of Lateral and not simply be a collection of Subsumption-like static hierarchies merged together.

3.3.2 Behaviours

The abstract nature of "behaviours" was discussed in Section 2.4, page 15. For the purposes of this chapter, a behaviour can be seen as a process running on a robot that implements some aspect of the robot's competences. Behaviours are executed in parallel, rather than sequentially as modules in traditional AI are. In Lateral, behaviours interact with the rest of the system by reading and writing information from a set of input connections and a set of output connections. The messages received through the input connections carry priority information about their sources as well as actual data, and this is used to calculate the priority of the behaviour itself. Before looking at how this calculation is done, it is worth examining what use is made of the result of the calculation. Firstly, in Lateral, the behaviour may independently choose to set its priority at some other lower level- the calculated priority is a maximum only. If the behaviour does not set its priority, it will be set to the maximum by default. Every output of the behaviour will carry the final priority chosen, and transmit it to any behaviours they are connected to in turn. The behaviour, again, can choose to override this default and set the priority of individual outputs at any desired level up to its own priority. It is also free to selectively enable or disable its outputs. A disabled connection effectively disappears, and means that the behaviour has no requirements of whatever it was connected to. Finally, the behaviour can also set a *priority factor* for each connection. This factor, which defaults to

unity, determines what fraction of its priority a connection may impart to the target behaviour it connects to (possibly through intermediate connections), as distinct from what priority it should have when competing with other connections for control of that behaviour (see Figure 3-10). This allows a "sponsoring" behaviour to use its maximum priority to take control of another behaviour and yet impart a lower priority to that behaviour (for example, so it will not compete with the sponsoring behaviour itself if they conflict elsewhere). An example of this will be given later in this section.

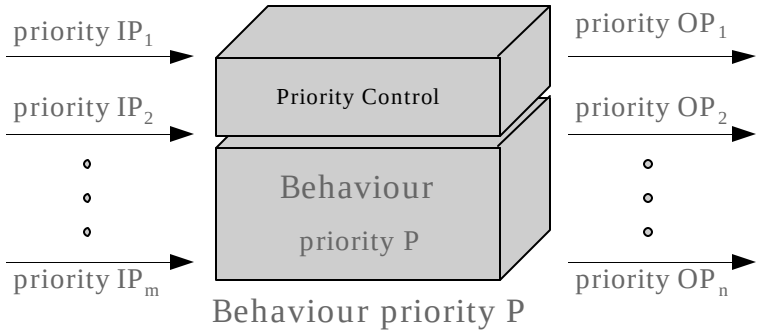


Figure 3-10: Priorities in behaviours

To calculate a behaviour's maximum priority, its input connection with the highest priority is found. That priority is multiplied by the associated priority factor carried by that connection, giving the amount of priority the most important user of the behaviour is willing to bestow. This is where the term "sponsorship" originated.

Priorities can change dynamically and are continuously recalculated. Behaviours in Lateral may have a special "priority control" section responsible for responding to changes in the input connections and propagating those changes to the output connections. If this section is not present, the behaviour is assigned the maximum priority it is sponsored, calculated as described above.

Consider the example shown in Figure 3-11. The "Command" behaviour is at a priority of 4. It has two outputs, attached to inputs of the "Seek Target" and "Wander" behaviours. It can control which of those behaviours finds expression in the action of the robot by choosing which to give the higher priority. In this case it sets its output to "Seek Target" at a priority

level of 2, and its output to "Wander" at a level of 4. These are the only inputs those behaviours have, so the priority of "Seek Target" and "Wander" become 2 and 4 respectively, and both accept any messages "Command" sends to them. "Wander" has an output to the "Explore" behaviour, which it chooses to drive at a low priority of 1. Since that behaviour has no other inputs, its priority becomes 1 and it is controlled by the output from "Wander". Both "Seek Target" and "Wander" attempt to control the "Use Map" behaviour simultaneously at their full priorities. "Use Map" is controlled by the connection with the highest priority- in this case, the one from "Wander". The connection from "Seek Target" is ignored, and only messages from "Wander" are accepted. Each of "Seek Target", "Use Map" and "Explore" attempt to control "Move". "Use Map" succeeds because it has the highest priority, and the other connections are ignored.

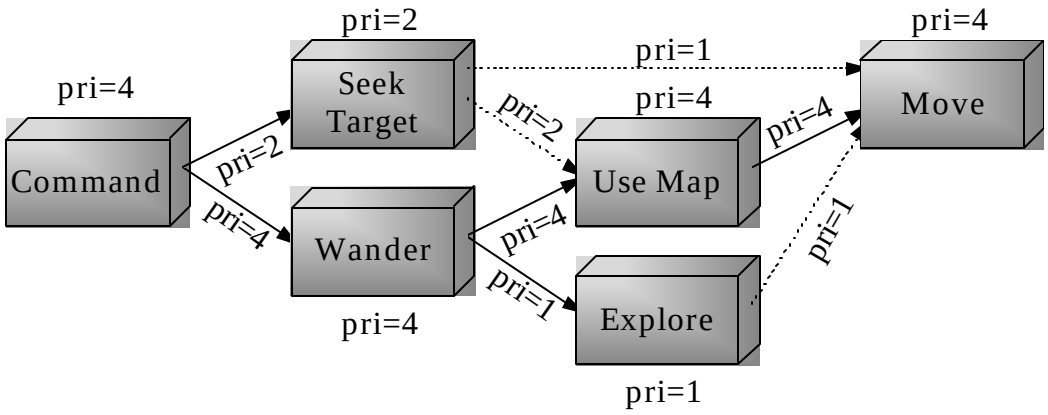


Figure 3-11: An example of calculating behaviour priorities

Figure 3-12 shows the same example, but with "Command" favouring "Seek Target" rather than "Wander". "Seek Target" now succeeds in gaining control of "Use Map". Note that "Command", by making a *local* decision about which of the behaviours it directly interacts with are more important, actually changes the action of behaviours it has no knowledge of- "Use Map" and "Move" are now being used for a different task.

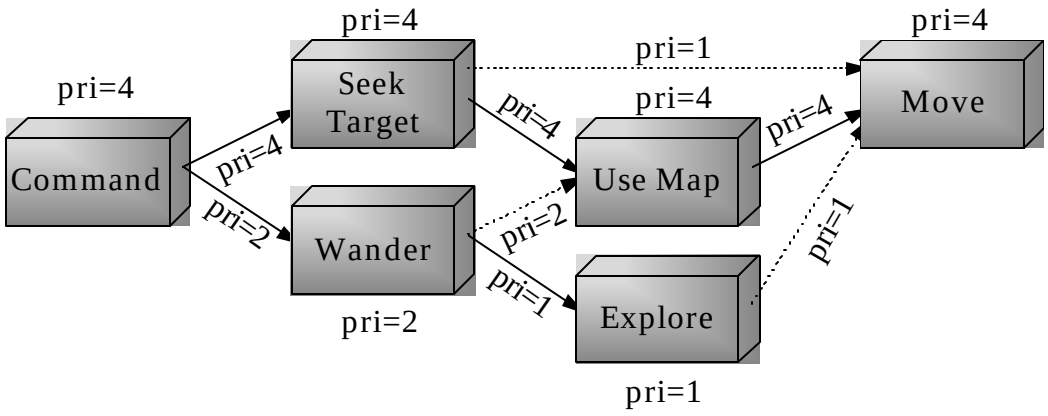


Figure 3-12: The effect of a change in sponsorship

A more sophisticated use of Lateral is shown in Figure 3-13. "Seek Target" is still being favoured with sponsorship, but it has now decided that although it still wants[5] to retain control of the "Use Map" behaviour, it wants to control the "Move" behaviour itself. This could be because it wishes to try to reach a target by some fast but unreliable guesswork, while waiting for "Use Map" to find a better path to the target. For its best chance of retaining control of "Use Map", it should place its full priority on the connection to that behaviour. However if it does that, "Use Map" will be able to compete with "Seek Target" itself for control of "Move", which "Seek Target" does not want to allow until it is confident that "Use Map" has a better chance of success than its guesswork. To avoid this, it seems that a lower priority should be used on the connection - but this does not reflect the fact that "Seek Target" wants to retain control of "Use Map" as much as it wants control of "Move", and it would leave "Use Map" open to being taken over by some lower priority behaviour. In this example, if the priority of the connection was set to 2, "Wander" could compete successfully for some control of "Use Map".

The solution is to use *priority factors*. "Seek Target" can set the connection to "Use Map" at its full priority of 4, but with a priority factor of 0.5 (for example). This means that the connection will compete for control of "Use Map" at a priority of 4, but when it has gained control of "Use Map", the behaviour only receives a priority of 2 (4×0.5). It therefore cannot compete with "Seek Target" for control of "Move", as desired. Priority factors are

[5] Anthropomorphisms are used here to avoid complicating the example with detail.

appropriate when a behaviour wishes to sponsor another behaviour, but is not willing to let it compete with activities of the sponsoring behaviour itself.

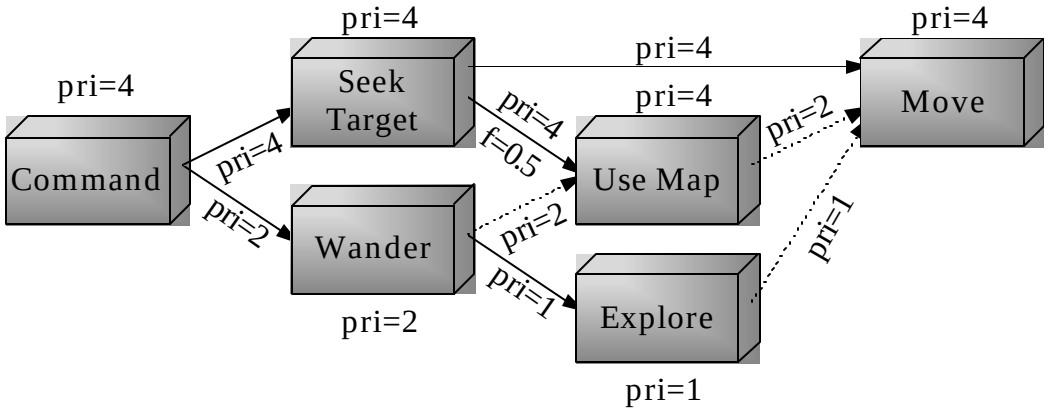


Figure 3-13: Use of priority factors

3.4 Comparison with Subsumption

Lateral is in a sense a superset of Subsumption, since a Subsumption-based control system could be implemented in Lateral by simply placing all behaviours at the same priority. Under this condition, connections behave very much like the wires in Subsumption. A set of behaviours operating at the same priority in Lateral is called a "priority plane". Within a single priority plane, the control system behaves similarly to how it would act under Subsumption, and the same linear enhancement of a behaviour through incremental levels of competence is possible. This linear enhancement is akin to single inheritance, since a basic behaviour is taken and enhanced to give it more functionality. However single inheritance is not always a good model for how behaviours relate. The existence of *multiple* priority planes in Lateral, with behaviours able to influence each other's priority as described in the preceding sections, means that behaviours may be related in ways other than simple inheritance. In the language of Software Engineering, links more like HAS-A relationships than the IS-A relationships of inheritance can be made between behaviours. Behaviours no longer need to be arranged as a strictly ordered series of enhancements.

For example, consider a Subsumption robot with a control system implemented using the levels of competence shown in Table 3-2, similar to the ones described by Brooks in [[7]].

Table 3-2

Level	Competence	Description of competence
0	avoid-contact	Avoid contact with objects
1a	wander	Wander aimlessly, moving at random
2a	explore-world	Explore the world by seeing places in the distance that look reachable and heading for them

And now consider the same robot, programmed to perform a different task with a new control system consisting of a different set of linear enhancements from the same "avoid-contact" competence (see Table 3-3).

Table 3-3

Level	Competence	Description of competence
0	avoid-contact	Avoid contact with objects
1b	follow-edge	Follow the edges of obstacles
2b	seek-goal	Move towards a goal by heading in that direction and skirting any objects in the way

Suppose that it became desirable to combine the abilities of both these control systems, to give a robot that can either explore the world on its own, or can be directed to reach specific targets. This is difficult to achieve in Subsumption. The problem is that both of the control systems above started from the same behaviour, the "avoid-contact" behaviour, and enhanced it in different ways. There is no easy way to fit these two different evolution paths into a linear layering of behaviours. One possibility is to take the control system that evolved the "explore-world" behaviour, for example, and then enhance that to include edge following, and then goal seeking ability (see Table 3-4).

Table 3-4

Level	Competence	Description of competence
0	avoid-contact	Avoid contact with objects
1a	wander	Wander aimlessly, moving at random
2a	explore-world	Explore the world by seeing places in the distance that look reachable and heading for them
3	explore-world-or-edge-follow	Explore the world by seeing places in the distance that look reachable and heading for them, or follow the edge of an obstacle
4	explore-world-or-seek-goal	Explore the world by seeing places in the distance that look reachable and heading for them, or move towards a goal

The problem is that neither edge following nor goal seeking have anything to do with exploring the world, and seeing them as enhancements of the "explore-world" behaviour is artificial and leads to unnecessary and undesirable dependencies between the behaviours.

Another possibility would be to make sure that the "explore-world" behaviour and the "seek-goal" behaviour are never active at the same time, so that effectively one control system is turned off and replaced with the other as needed. This will work[6], but places the burden of resolving conflicts back on the designer. The more the control systems try to share behaviours (such as "avoid-contact" in the examples above) the more complex the logic necessary to avoid conflict will be.

Lateral does not require behaviours to be shut off so as not to affect one another. Instead it moves them to separate "priority planes" based on the importance of what they are being used for, as opposed to assigning fixed priorities to the behaviours themselves (see Figure 3-14). A priority plane is simply the set of all behaviours and connections at a given priority. A behaviour on a lower priority plane is guaranteed not to affect anything on a higher priority plane. Since the priority of a behaviour is determined by the priorities of its input connections, a behaviour can be "pulled up" to a higher priority plane by an input connection having that higher level of priority. And since the priority of a behaviour's output connections reflects the behaviour's own priority, these connections may also be "pulled up" to the higher priority plane, and the behaviours they connect to, and so on.

[6] The idea of switching between different static hierarchies of behaviours to extend Subsumption is called "moods" [46]

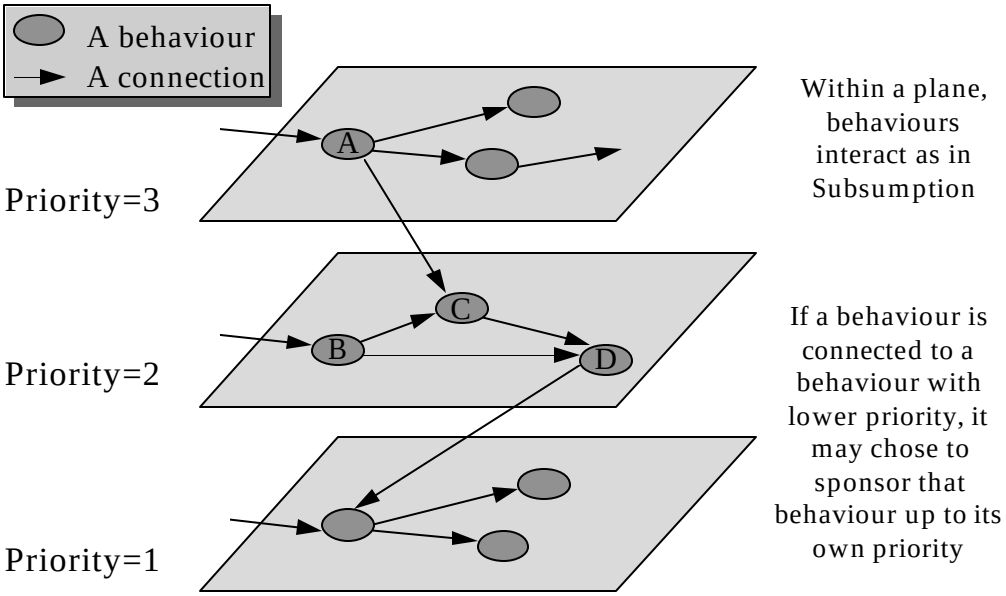


Figure 3-14: Priority planes

For example, if the robot is engaged in one task (which can use any number of behaviours), a higher priority task can "hijack" any behaviours it needs and not have to worry about turning off parts engaged in the lower task that do not concern it. This "hijacking" happens automatically through sponsorship. In the above diagram, behaviour "A" could choose to sponsor "C" to have a priority of 3. If so, "B" will no longer have control of it since "C" will now be on a higher priority plane. If "C" passes on its sponsorship to "D" in turn, then "B" will lose control of "D" as well. These behaviours have been "hijacked" for a higher priority task.

When this task finishes, its priority falls and sponsorship diminishes, so control will automatically return to lower priority tasks. Since behaviours are designed to be reactive, and to be able to deal with environmental interference, the temporary subversion of components to another task can be recovered from through mechanisms that are already naturally present. Interruptions are a normal part of the operation of a behaviour-based system, in contrast to being treated as abnormal exceptions as they are in traditional AI.

Returning to the problem of merging the two control systems described earlier, it should be clear that Lateral can handle this without any problem. The control systems can simply be merged directly, as they stand. Because the hierarchy is no longer linear, it can no longer be drawn clearly as a table, and is presented graphically in Figure 3-15.

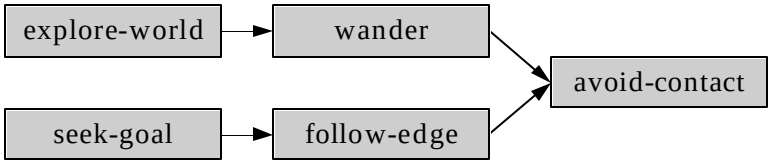


Figure 3-15: Control systems merged using Lateral

The two control systems can be implemented independently of each other, except where they share the common "avoid-contact" behaviour. Conflicts will be resolved according to the priority plane the behaviours are on at any given time. If "seek-goal" is at a high priority, it will pull "follow-edge" up with it, and that in turn will pull up "avoid-contact" (see Figure 3-16). "Avoid-contact" will therefore be guaranteed to be entirely unaffected by "explore-world" and "wander", since they are on a lower priority plane, and lower planes cannot affect higher planes.

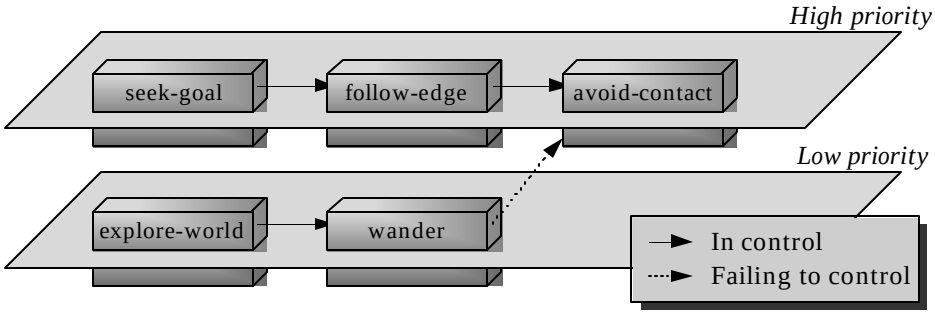


Figure 3-16: Merged control system, with seek-goal at high priority

If "seek-goal" becomes a low priority relative to "explore-world", the position will be reversed (see Figure 3-17). "Seek-goal" and hence "follow-edge" will be on a low priority plane, and "explore-world" will pull "avoid-contact" up to its higher plane, guaranteeing that it will be entirely unaffected by either "seek-goal" or "follow-edge".

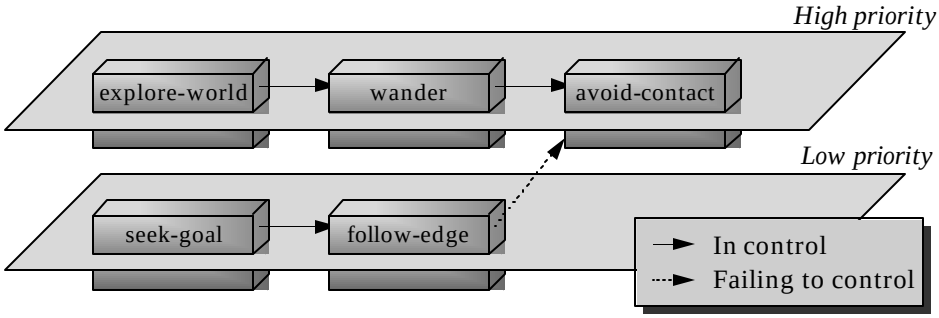


Figure 3-17: Merged control system, with explore-world at high priority

This is a very simple example of the advantages of Lateral in terms of the system decomposition it facilitates. More complete examples will be given in the next section.

The idea of priority spreading from module to module sounds superficially similar to the "spreading activation" system used in the New Subsumption (see Section 2.5.4, page 32). However, it is quite distinct. "Spreading activation" involves spreading an activation level between modules which, when compared with threshold levels, determines if they are active or not. It amounts to a way of automating the task of turning on and off modules to prevent them from inappropriately interacting or conflicting with each other. The Lateral system uses "spreading priority" to eliminate the need for such a system. It allows control of the relative weighting that the outputs of a behaviour have compared to other behaviours. This is fixed in the Subsumption scheme - the only way to change it is to turn a module off.

The Subsumption architecture is specified at a finer granularity than Lateral. It is described at the level of wires connecting a collection of augmented finite state machines to implement a single behaviour (see Section 2.4.4, page 19). While Lateral can support this level of granularity when incremental enhancement of a behaviour is desired, it is for the most part described here at a coarser level, with individual behaviours as the smallest unit. A behaviour's input and output connections comprise its public interface; no support is given in the architecture for accessing anything internal to a behaviour. This is good for modularity.

3.5 System Decomposition using Lateral

Further examples are presented in this section to clarify the influence of the choice of architecture on the system decomposition used for a given robot control system. The example is in fact a simplified extract from the robot application developed later in this thesis (see Chapters 4 and 5).

Suppose a simple robot is to be constructed which can move from its current location to a specified target location. The robot has a partial map of its environment in memory- but the map may be out of date, and searching the map to try and find a path to the target is a slow process. In such a scenario, the robot could combine two strategies in an attempt to reach its target. The first, *physical search*, is the most direct, and involves the robot simply moving towards the target and attempting to work its way around any obstacles it meets along the way. In the second, *map search*, the robot plots a path to the target using its internal map, and

when that is complete, attempts to execute that path. Attempting to reach a target by physical search alone will work well for situations with few obstacles between the current position and the target, but will be slow in more complex situations. Using just map search will perform better when there are complex obstacles to be negotiated, but will slow the robot down in simpler situations. By switching between the two types of search as circumstances indicate, the robot can gain the advantages of each. For this example, this combined search strategy will be called "smart search".

If such a robot were implemented using Subsumption, behaviours like the following would be implemented, each incrementally enhancing the one that came before it:-

- **Avoid**- this behaviour keeps the robot from bumping into anything.
- **Edge Follow**- this behaviour allows the robot to follow the boundary of an obstacle, while retaining the competence of the "Avoid" behaviour.
- **Local Motion**- this behaviour makes the robot move "sensibly"- so that it can manoeuvre gracefully both close to a boundary and in open areas.
- **Physical Search**- this behaviour gives the robot the ability to try to approach targets using a collection of techniques that do not require map information.
- **Smart Search**- this is the required behaviour of the robot, implementing both physical search and map search.

These behaviours are illustrated in Figure 3-18.

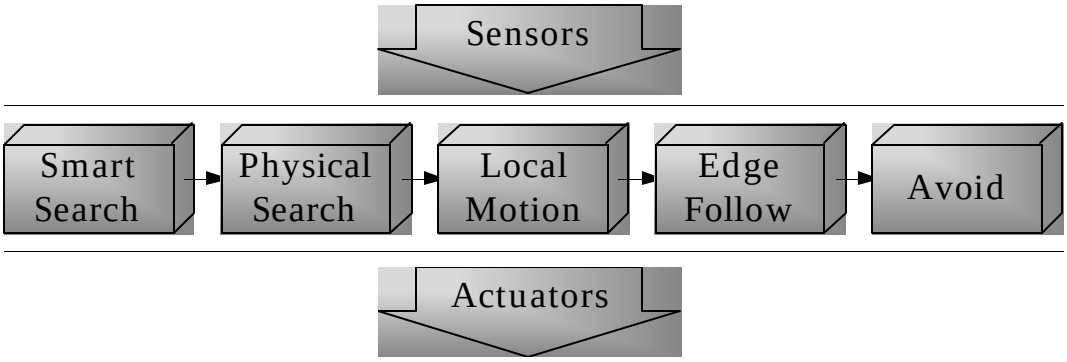


Figure 3-18: Decomposition using Subsumption

Physical and map search are distinct techniques, so it makes sense to implement them separately. But Subsumption only allows behaviours to be enhanced linearly, so one of the techniques must be picked as more "basic" and the other one added to it as an enhancement. For example, the choice of implementing physical search first in the decomposition in Figure 3-

18 was entirely arbitrary, and map search could have been chosen instead, as shown in Figure 3-19.

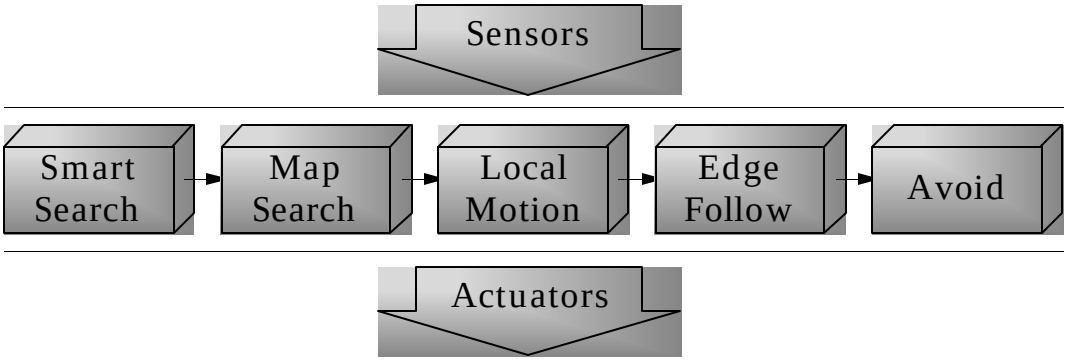


Figure 3-19: Alternate decomposition using Subsumption

Whichever technique is taken as basic, the other one must be derived from it even though they have no relationship other than that they work towards achieving the same task. This is undesirable from a software engineering point of view.

Because Lateral is not restricted to linear enhancement of behaviours, a more satisfactory and natural decomposition can be used, as shown in Figure 3-20. Physical and map search are implemented separately, and the "smart search" module need only implement the heuristics for switching between the two when circumstances dictate.

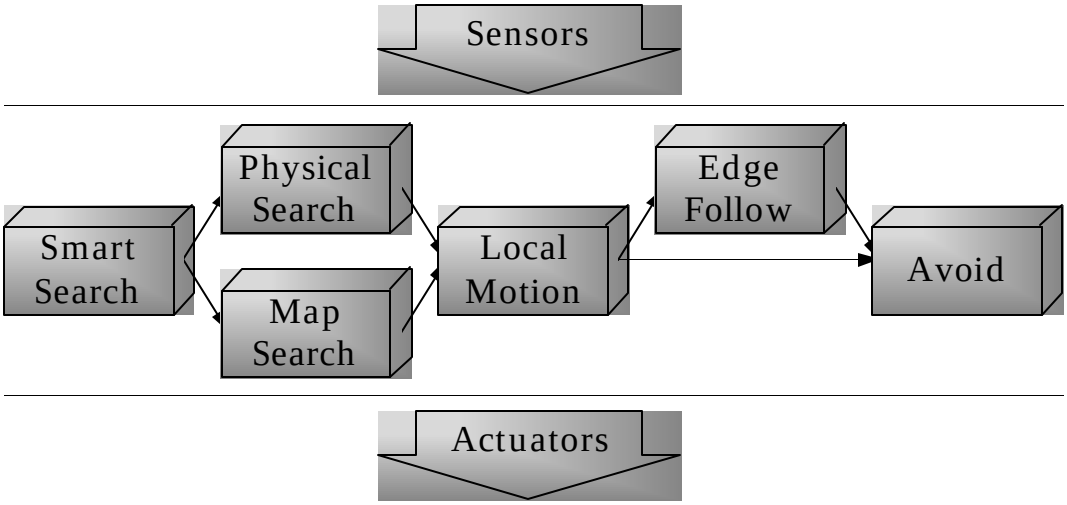


Figure 3-20: Decomposition using Lateral

This is more satisfactory, since each module need only know the minimum information required to accomplish its task. Note that in the decomposition, the dependencies between local motion, edge following, and avoidance have been similarly rearranged. This more accurately

reflects the nature of local motion, which sometimes makes use of edge following but does not need to use it when the robot is in open areas. "Making use" of a behaviour cannot be captured in Subsumption except by the crude approximation of linear enhancement.

To move away from simple examples, Figure 3-21 shows the main body of the decomposition used for the sentry application[7] (Chapter 5). The diagram is similar in nature to the ones seen already, but more complex. Implementing the decomposition in even approximately the form presented would not be feasible in Subsumption- it cannot be made linear without losing the advantages of using such a decomposition in the first place.

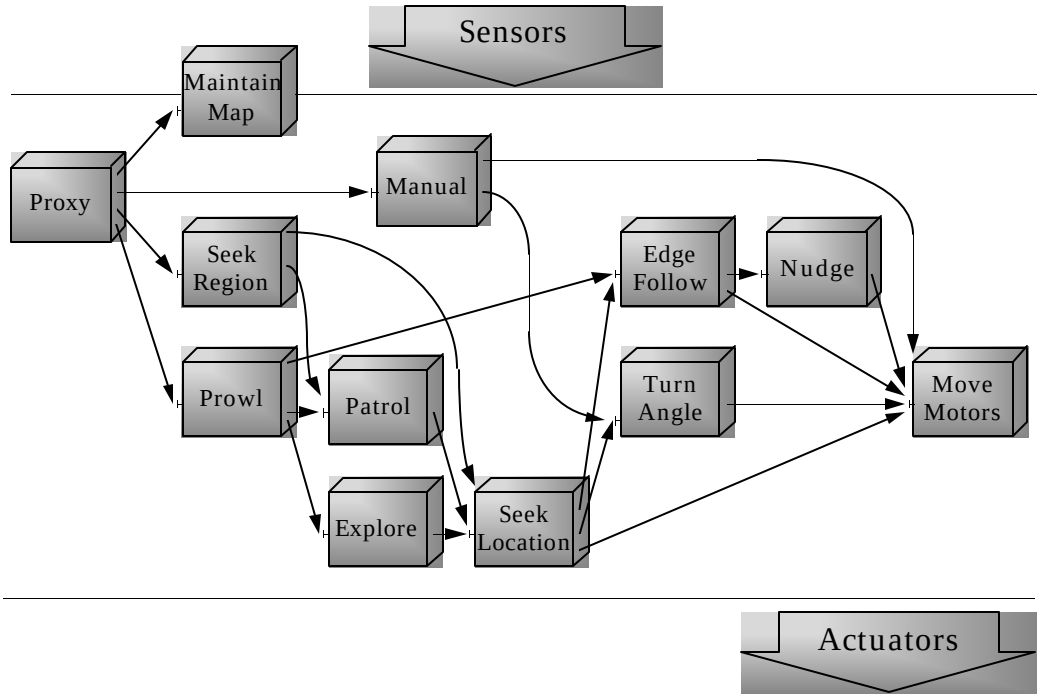


Figure 3-21: An actual decomposition

This decomposition is the source of the (very much simplified) example given at the very start of this chapter (see Section 3.2, page 46). For most purposes, the decomposition need operate on only two priority planes, high and low, indicating behaviours that need to be active and those that do not. The patrol behaviour may be sponsored at an intermediate level, since it can be used to perform the "map search" discussed earlier while a "physical search" is operating at a higher priority. The diagram of the decomposition differs from earlier diagrams

[7] Maintain-Map is shown straddling the sensor line because its outputs act as virtual sensors accessible to all the other modules.

in that it shows "Actuators" underneath the Motor behaviour. This reflects that fact that while in a behaviour-based system any behaviour can attempt to control the actuators, in this decomposition the motors are in fact always controlled through the Motor behaviour. This is done because that behaviour implements obstacle avoidance. Other behaviours are still technically free to attempt to drive the motors directly, but by doing so they do not have the protection of obstacle avoidance and risk damage to the robot. In effect, the Motor behaviour has become a "virtual actuator".

3.6 Implementing Lateral

The rest of this chapter is concerned with taking the abstract properties of the Lateral architecture discussed thus far and showing how they can be supported in an actual implementation. The most important single constraint on how the architecture can be implemented is the level of operating system support available to it. Autonomous robots generally have limited processing power and memory, and techniques that are perfectly adequate for use with simulated robots running on a workstation may be totally unsuited for downloading to a much less computationally powerful physical robot.

Lateral could be implemented most naturally under a full distributed system with asynchronous message passing. However, since the work in this thesis was to be implemented on a physical robot, it could not expect such sophisticated operating system support. In fact, an implementation that demanded *any* multi-tasking ability at all would contradict one of the aims of the work, which was to see what could be done with the cheapest robots with minimal processing power and minimal sensing equipment. Given that, it was decided to make the Lateral implementation itself responsible for running behaviours concurrently, without depending on operating system support. The implementation of multi-tasking could only be "bare-bones" however, to avoid swamping the limited memory of lower-end robots. It was obviously important to leave sufficient space for useful applications to be built. The approach used was to allow context-switching between behaviours at a coarse granularity only, in the following way:-

- Behaviours were required to be written in the form of augmented state machines, as in Subsumption (see Section 2.4.4, page 19).

- Code associated with a single state of a behaviour was always executed *without pre-emption*.
- Context switching between behaviours occurred only at state transitions or on completion of the code associated with a state.

The advantages of this approach were that it made for a very light-weight implementation with low memory overhead, and it required no platform-dependent code to be written[8]. The disadvantage is that it places an extra burden on the programmer to ensure that code within states completes or transitions in a timely fashion and does not "hog" the processor. This was considered an acceptable trade-off, since using this light-weight system meant that less time had to be spent "re-inventing the wheel" implementing a multi-tasking system, and more time could be spent concentrating on the novel features of the Lateral architecture. Also, the burden on the programmer in practice is less than might be imagined, because states of behaviours written to be reactive naturally have very short duration so that they can respond to the environment as it is now rather than as it was at some time in the past.

For clarity, the implementation of Lateral developed in this thesis is given its own name, "*Zac*", separate from the architecture itself. This is to clearly distinguish the general concepts behind the architecture from the particular way chosen to implement them. Restrictions had to be placed on the support for Lateral developed in this thesis so as to satisfy the constraints the work took place under, particularly the desire for the work to be applicable to lower-end robots. These limitations are not intrinsic to the architecture - hence the careful distinction being made.

3.6.1 Executing Behaviours: The Scan Cycle

To run the control system in the Zac implementation of Lateral, every active behaviour (defined as a behaviour with non-zero priority) is executed in turn for one state. When all the behaviours have been executed, the cycle is repeated for another state. This process is called the "scan cycle". Before a behaviour is executed for another cycle, its input connections are updated, and its priority is recalculated accordingly as described in Section 3.3.2, page 55.

[8] This was an important pragmatic consideration, since when the work for this thesis was begun, the robot it was to be implemented on was not known.

Updating a connection means applying the arbitration rules of Lateral to determine which of its sources it should read from (see Section 3.3.1.5, page 53).

The order of evaluation of behaviours is chosen so that if there is a chain of dependencies between any behaviours, effects from the start of the chain will reach behaviours at the end of the chain within a single cycle. To clarify what this means, consider an example where a behaviour A depends on a behaviour B (i.e. information flow from B is channelled to A through some sequence of connections), and B in turn depends on C, and C depends on D. If these behaviours are executed in the order A, B, C and then D, three scan cycles will be needed before a change in an output from D can affect A, since the change will only propagate up the chain at a rate of one behaviour per cycle - as shown in Figure 3-22.

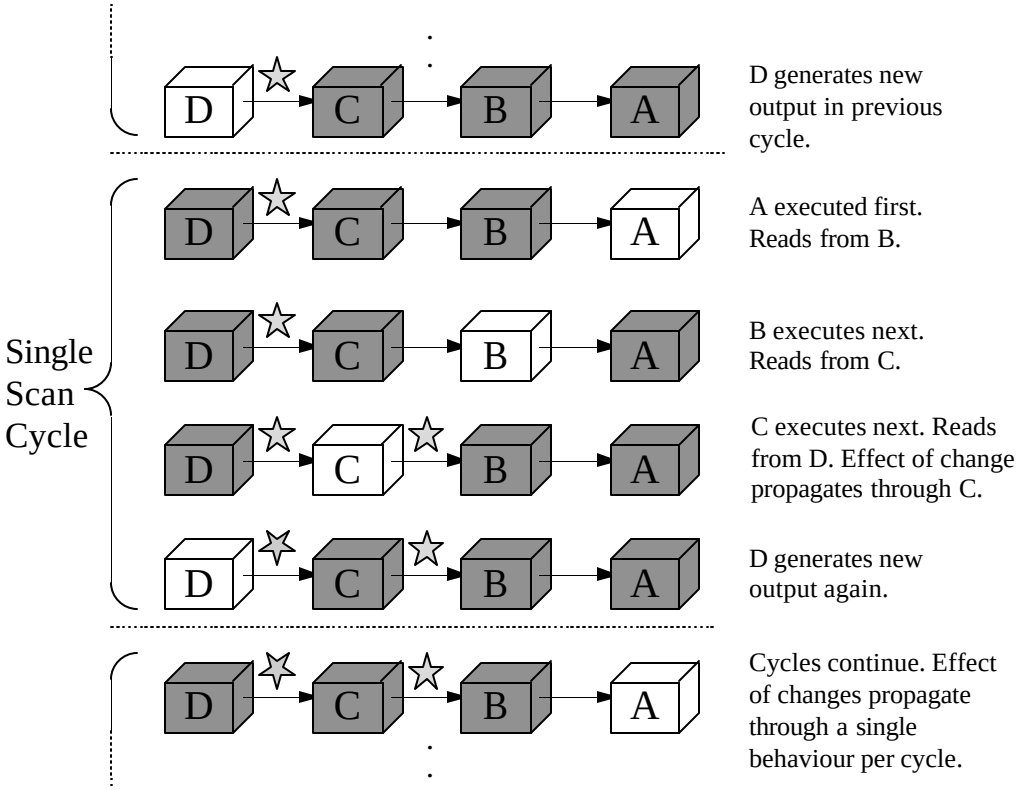


Figure 3-22: Propagation of effects with bad choice of execution order

In contrast, if the behaviours are executed in the order D first, then C, then B, and finally A, then the action of D will affect all the other behaviours within a single cycle, as shown in Figure 3-23. This will be the order chosen by the implementation of Lateral developed in this thesis. How the proper ordering of behaviours is deduced is described in the following section. The algorithm used will order any tree-like hierarchy of behaviours so that a single cycle is sufficient

to propagate effects from the top of the tree to the bottom. If there is a cyclic dependency (e.g. A depends on B, B depends on C, and C depends in turn on A) the order of execution chosen will be such that it takes a single cycle for effects to propagate from one behaviour in the loop around to that behaviour again.

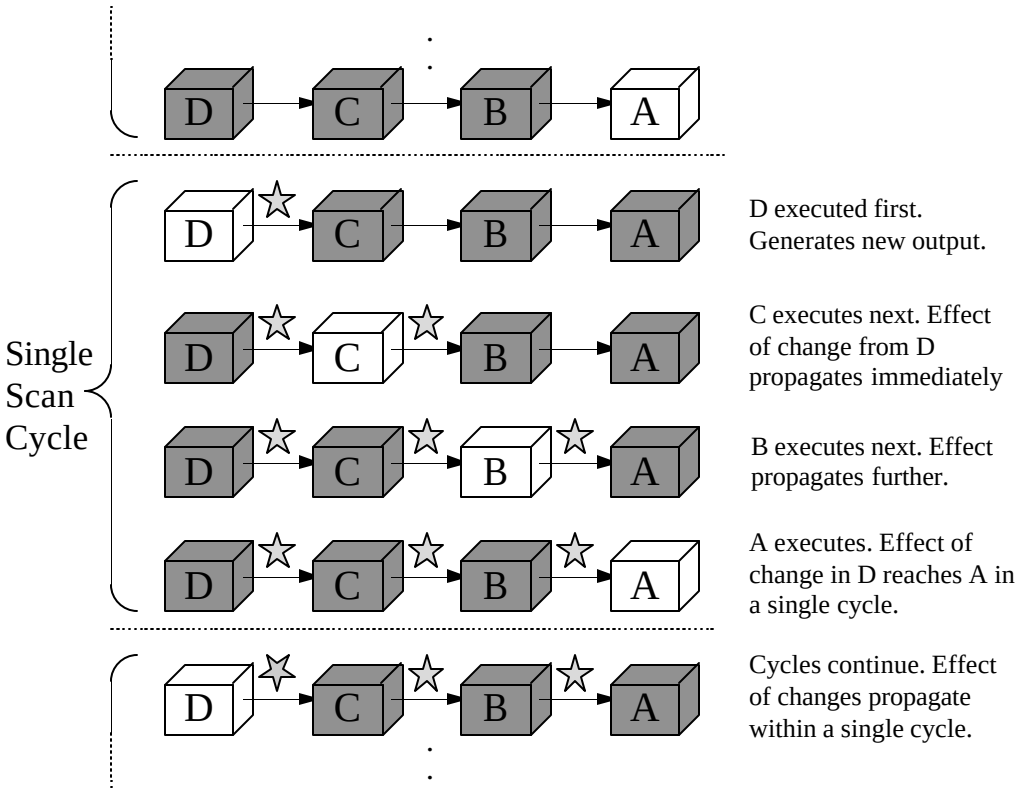


Figure 3-23: Propagation of effects with good choice of execution order

3.6.2 Updating Connections: The Pull System

Before a behaviour is executed, its input connections are updated so that it is operating on up-to-date information. In Lateral, updating a connection means applying the arbitration rules of the architecture to determine which of its sources it should read from, and then accessing that source (see Section 3.3.1.5, page 53). A suitable data structure for implementing connections is given in the next section that makes it possible to apply Lateral's arbitration rules quickly and efficiently.

The implementation of Lateral developed in this thesis makes use of the presence of connections to determine the best order in which to execute behaviours to optimise the speed at which effects spread through the system. As well as being active instruments of communication, connections serve to make the dependencies between behaviours explicitly

accessible to the architecture. By applying the following two rules, it is possible to ensure that all behaviours are executed in the optimum order:-

1. To update a connection, update all its sources first. Omit any that have already been updated in the current scan cycle or which are in the process of being updated. Then, if the connection is an output of some behaviour, update that behaviour using rule 2 (unless that behaviour has already been updated in the current scan cycle or is currently in the process of being updated).
1. Before updating a behaviour, update all its input connections first in the manner described in rule 1. Then execute it.

Note that the rules recursively invoke each other as dependencies are traced back though connections and the behaviours met along the way are updated. The conditions in rule 1 check for cycles and avoid looping. These two rules together are known as the "pull system", since when a behaviour is to be executed, it "pulls" through its input connections to ensure that everything it depends on is executed first. A simple example is given in Figure 3-24.

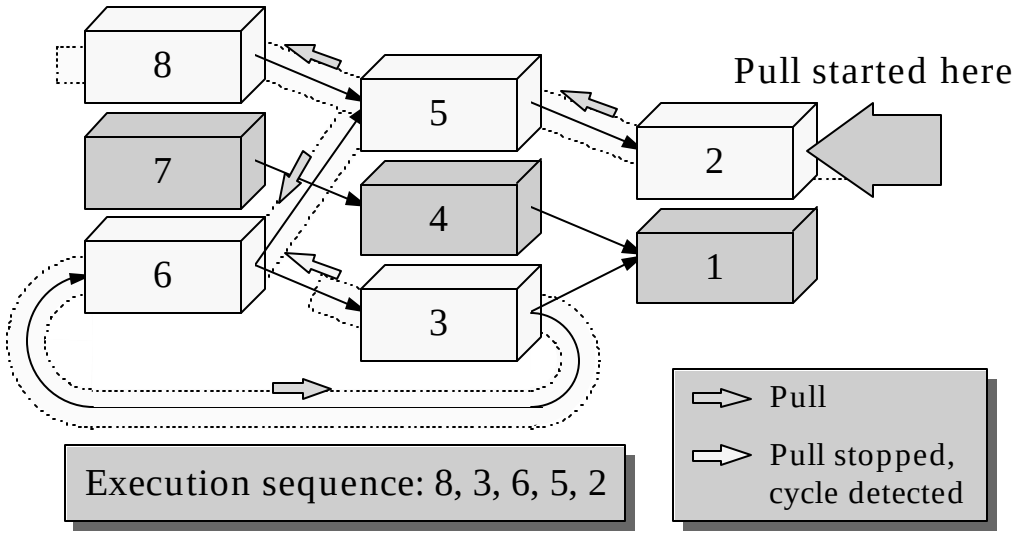


Figure 3-24: Pull system in action

Here behaviour "2" is to be executed. It first updates its input connection, which traces back to behaviour "5". Rule 2 is then invoked to update behaviour "5" before behaviour "2" can be executed. Behaviour "5" depends in turn on behaviours "8" and "6", so these are updated before either "5" or "2". Behaviour "8" has no dependencies, so it may be executed immediately. Behaviour "6" depends on behaviour "3", so "3" must be updated before "6". Behaviour "3" depends cyclically on behaviour "6" again, but rule 1 will detect this cycle and

cut it off[9]. Behaviour "3" has no other dependencies, so it will now execute. This allows behaviour "6" to execute in turn. All the dependencies of behaviour "5" have executed, so that behaviour may now be executed itself. Finally, behaviour "2" is free to execute.

3.6.3 Traversing Connections: The Mesh Structure

To implement Lateral efficiently, it is important that the various relationships between connections can be determined quickly and without search. This is because these relationships are used extensively both in the arbitration rules for updating connections, and in the "pull system" for executing behaviours in the optimum order. This section introduces the idea of a "mesh", a very flexible data structure developed for this architecture which is sufficiently powerful to allow all the relationships possible between connections in Lateral to be determined efficiently.

In Figure 3-25, the logical relationships between connections in Lateral are shown for reference purposes, copied from Figure 3-2 on page 46. Each connection has a single *primary source* and *primary target*. It can also have *secondary sources* (connections which have that connection as their primary target) and *secondary targets* (which have that connection as their primary source). Also, every input or output connection belongs to a behaviour called its *owner*. Intermediate connections outside of behaviours have no owner.

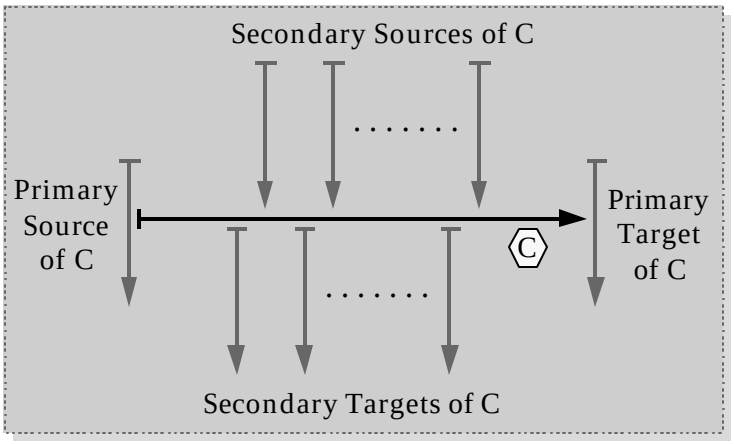


Figure 3-25: Relations required in Lateral

[9] The cyclic dependency still exists, but propagation from behaviour "6" to behaviour "3" will occur in the *next* cycle rather than the current one.

Since the pull system and the priority system in Lateral both rely on repeatedly evaluating these relationships, it is important that all of the following operations can be performed quickly without search:-

- Finding the primary source of a connection.
- Finding all secondary sources of a connection (all connections that have the given connection as their primary target[10]).
- Finding the primary target of a connection.
- Finding all secondary targets of a connection (all connections that have the given connection as their primary source[11]).
- Finding the behaviour that owns a connection, if it is an input or output connection.
- Finding all connections with the same owner as a given connection

The data structure used in this thesis to implement a connection so that all the above relationships can be easily determined is called a "mesh link". A collection of such links is called a "mesh". Each link[12] has a direct hook to its owner, its primary source, and its primary target. That much is straightforward. Each link has in addition a hook to a *single* secondary source, and a *single* secondary target. It also acts as an element in three separate linked lists- the list of links with the same primary source as the given link, the list of links with the same primary target as the link, and the list of links with the same owner behaviour as the link. Note that having access to these lists means that only a hook to a single secondary source or target need be maintained within a link- the rest can be found by traversing the list of links with the same primary source or target as that link. In Lateral, the secondary sources and targets are only ever examined as a group- so it is not necessary to be able to find a *particular* secondary source or target without search, only to find the *group*. The lists are stored as double-linked lists for ease of manipulation, as shown in Figure 3-26.

[10] If this seems confusing, remember that the primary targets of connections are chosen freely, and do not affect either the primary source or the primary target of the connections to which they are attached- just as one electrical wire can be attached to another without changing what that wire is connected between. Hence if the primary target of A is B, that does not mean that the primary source of B is A (and in general it will not be).

[11] If this seems confusing, the explanation is analogous to that given in footnote 10.

[12] In this thesis, the terms "connection" and "link" are essentially synonymous. The distinction is made as a reminder that connections could be implemented in many ways, with the "link" structure being just one of those ways.

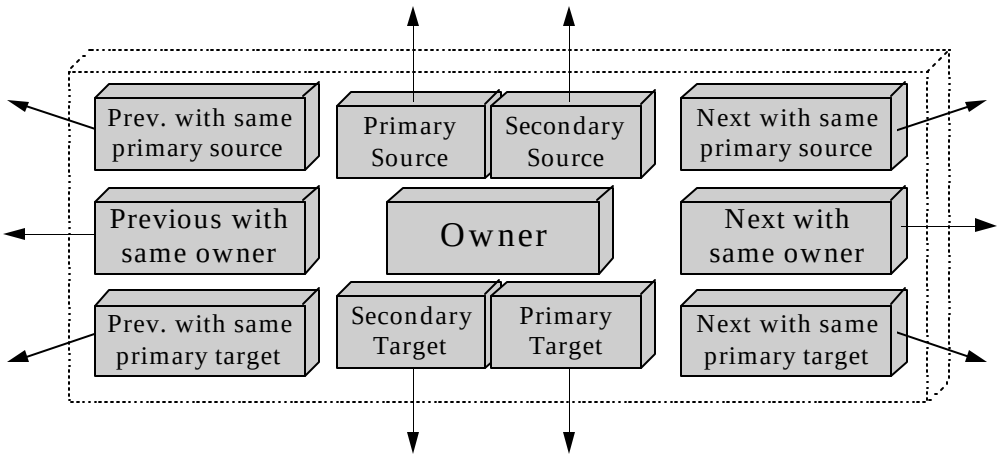


Figure 3-26: Connection implementation- a link

To demonstrate the use of mesh links, consider the set of connections shown in Figure 3-27. Primary sources, secondary sources, primary targets and secondary sources are all present.

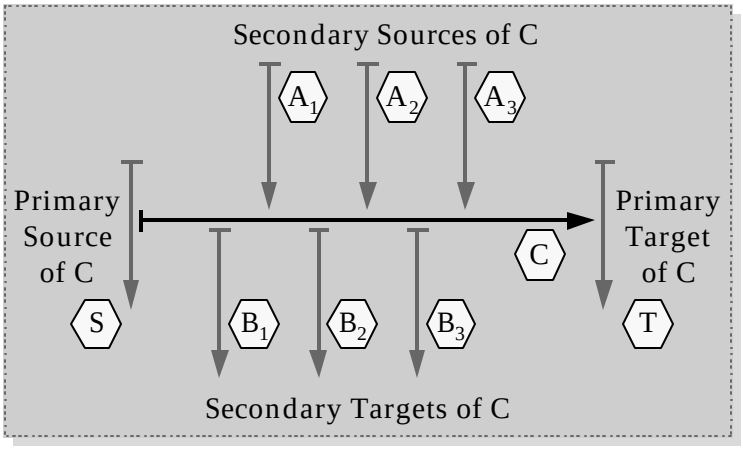


Figure 3-27: An example group of interacting connections

Figure 3-28 shows how the connections attached to C in the various different ways can be determined. The primary sources and targets are found directly. The group of secondary sources is found by first following the hook that C has to one of them, and then tracing around the double-linked list of all links with the same primary target as that link. The group of secondary targets is found in an analogous way.

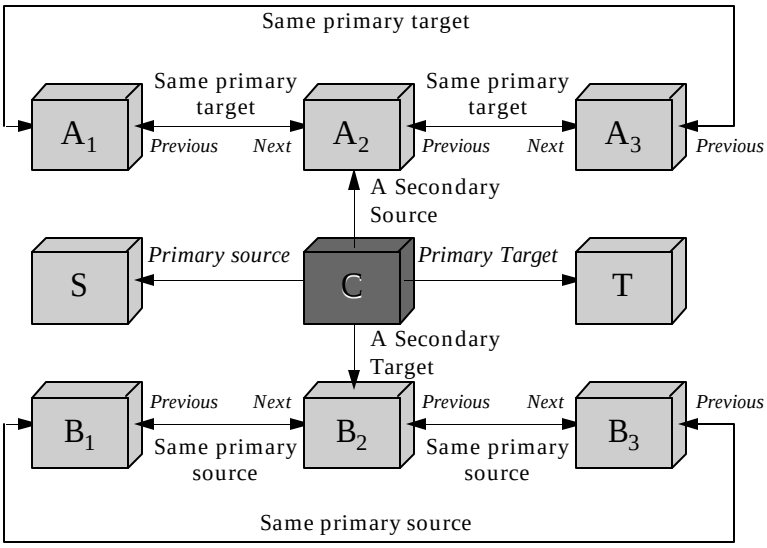


Figure 3-28: Evaluating the relationships between connections

The advantage of this arrangement, as opposed to each link maintaining its own list of secondary sources and targets, is that it is more efficient in terms of memory, and much simpler to maintain.

3.6.4 Coding Lateral through C++

It was considered desirable to make the Zac implementation of Lateral portable across robots with different on-board processors[13]. C++ is a popular programming language for which cross-compilers are available for the processors commonly used in autonomous robots, so this was the implementation language chosen. The reasons for this decision are discussed in more detail in Chapter 6. The C programming language would have been an equally good choice, but the object-oriented nature of C++ was a good match for Lateral's design as a collection of behaviour and connection "objects".

The approach adopted in using C++ to implement Lateral was to embed the functionality of Lateral behaviours and connections in corresponding behaviour and connection C++ classes. Then a control system for a robot implementing a particular set of competences could be constructed by deriving appropriate new behaviours using inheritance and attaching connection

[13] This was particularly important since, as remarked earlier, the robot that the work in this thesis was to be implemented on was not known until that work was quite advanced.

objects between them as necessary. The result would automatically support the Lateral's priority scheme, the pull system, and all the functionality described in this chapter.

The "mesh link" data structure for storing connections discussed in the previous section was implemented as a class called "MeshLinkBase" (see Figure 3-29). This class was only concerned with modelling the abstract relationships between connections, and not with a connection's role as a channel of communication. From this, a class called "ZACMeshLinkBase" was derived which implemented Lateral's priority scheme, the pull system, etc. Inheriting from this, a template class was set up that allowed a set of classes to be derived to implement links carrying particular types of data. This use of templates allows the compiler to check that only links carrying the same kind of data are attached to each other[14]. Also it allows the programmer to easily derive a new type of link that carries a different kind of data needed for a particular application.

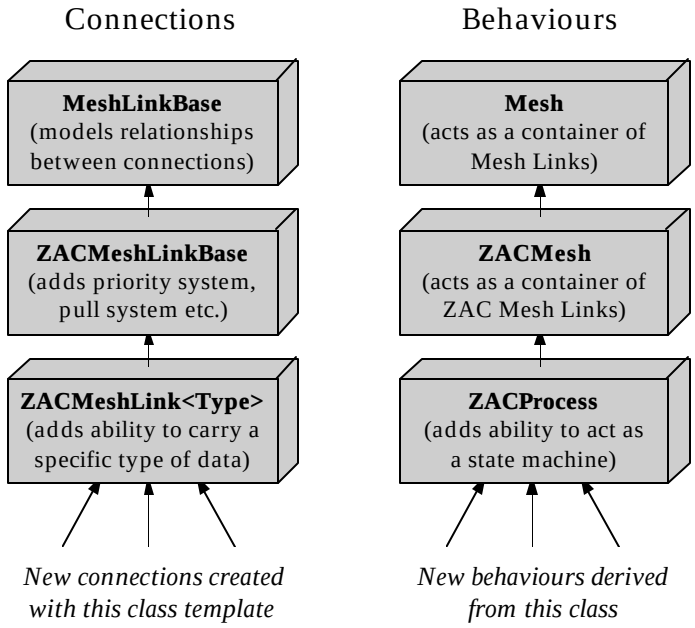


Figure 3-29: Behaviours and connections - meshes and links

The base class for implementing behaviours was called "Mesh", and simply acted as a collection of mesh links. Like the base class for connections, this class was only concerned with modelling the abstract relationships in the control system. For a behaviour, this means

[14] This use of templates to make a "type-safe shell" is particularly effective when, as in this case, the class template is a minor enough extension to the base class for all its member functions to be made inline. The compiler then need

storing a set of input and output connections. This is why it is appropriate for the base class of behaviours to be a simple container for a collection of mesh links. Next, the class "ZACMesh" was derived which could co-operate with the "ZACMeshLink" class to implement Lateral's priority system, the pull system, etc. Finally, the ability to act as an augmented state machine was added, giving the "ZACProcess" class. This class represented a complete behaviour, except that it lacked any connections and an actual state machine to execute. The programmer could derive new behaviours by simply inheriting from this class and adding a state machine (implemented by overriding a virtual function) and appropriate connections (member variables derived from the ZACMeshLink class). Such new behaviours would be automatically executed and managed by the Lateral support inherited from the ZACProcess class.

While it is straightforward to use the classes described here to build a working control system, it was found that the constructs and modularity of Lateral were different enough from those of C++ to result in a significant amount of repetitive, inelegant, clumsy code being necessary to express various common structures, e.g. state machines. A tool called the "Zac Translator" was developed to automate the generation of this code. With this tool present, Lateral constructs could be written in a specially designed syntax called "Zac Script", and the translator would then convert those constructs to pure C++ for compilation. This tool is described in more detail in chapter 6. The Zac Translator was also important for pragmatic reasons. The implementation of the Lateral architecture presented here evolved over several design cycles, changing frequently. These changes would normally have required application code written using the architecture to be updated too- but instead, if the source code was written in Zac Script, only the translator needed to be modified.

3.7 Summary

This chapter has introduced the Lateral architecture, a behaviour-based robot architecture that extends on the successful Subsumption architecture developed by Brooks [[7]]. The novel features of Lateral were discussed, in particular the flexibility it brings to how behaviours can be organised. Further ideas were presented on how to actually implement support for the Lateral architecture that would be light-weight enough for downloading to lower-end robots

not generate code for a separate set of functions for each class (important when memory is limited), but type-checking is still as strong as if it did.

with limited memory resources. These ideas lead to "Zac", a particular implementation of Lateral. Finally a suitable way to use C++ as a vehicle for implementing Lateral in a portable fashion was examined.

Many of the chapters to follow rely on the work presented here. In Chapter 5, Lateral is used to build a set of behaviours. In Chapter 6, the Zac Translator mentioned in Section 3.6.4 is presented in detail. Then in Chapter 7, a complete system decomposition for the entire control system of a robot is presented, with the elements discussed here finding expression in the components of the system in Section 7.6, page 227.

4. The Cartographic System

This chapter presents a cartographic system capable of building and maintaining an effective map of the robot's environment with the use of short-range proximity sensors only. While the approach developed has some common ground with the techniques reviewed in Section 2.7 (page 39), the unique challenges which the absence of long-range sensors pose for map-building are resolved by using a novel representation scheme specifically tailored to the uncertainties involved. At the same time, the nature of the cartographic system developed here is carefully chosen to be consistent with behaviour-based design principles, which place severe constraints on the use of representation (see Section 2.4, page 15).

The first section in this chapter establishes why a cartographic system is of such importance to an autonomous robot. Following that, the use of maps is reconciled with the limits behaviour-based design places on the role modelling can play in a robot's control system. A form of map representation which meets these constraints is then developed. Techniques are presented to make the representation practical for use on robots with limited memory and relatively slow processors. The issue of recognising landmarks in the environment with proximity sensors only is then examined in detail, because it is crucial to the success of the cartographic system. Finally, a set of services is described for allowing a behaviour to interact with the robot's map without compromising the behaviour's essentially reactive nature.

4.1 Motivation for using maps

An autonomous robot needs to maintain a map of its environment for a number of reasons. One of the most important is that if the robot fails to maintain a map, it will be prone to cyclic behaviour. As an example, consider Figure 4-1. Here the robot is shown trying to move in a particular direction, but being blocked by an obstacle. The robot then tried to move around the boundary of the obstacle until it is free to move in the desired direction again. Unfortunately in this case the shape of the obstacle is such that following its boundary leads the robot into turning back on the path it has already followed. If the robot does not have a way of detecting that this has happened, it may enter a behavioural cycle in which it repeats the same motion again and again.

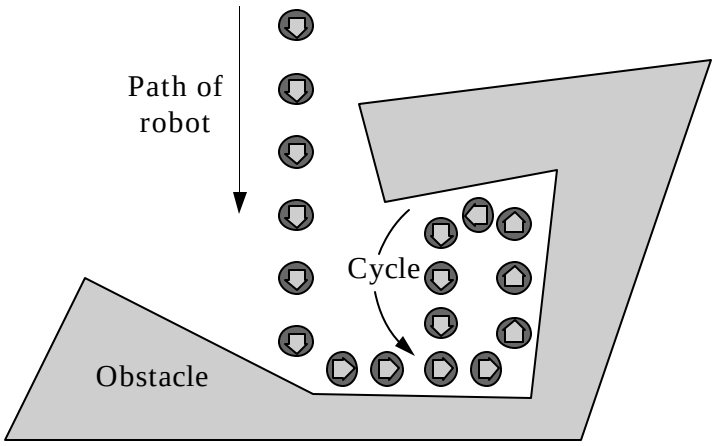


Figure 4-1: Robot without map caught in behavioural cycle

Maintaining a map based on what the robot experiences as it moves would give the robot the "memory" needed to detect and break out of cycles like this. Also, as the robot explores more of its environment and becomes better informed, its map becomes a useful tool in planning efficient routes to targets. It allows the robot to anticipate obstacles in the way and avoid entering into situations such as the one considered above by just bypassing the obstacle altogether.

Another basic use of maps is to prevent the robot losing track of its position relative to the rest of the environment. While a reasonable estimate of the robot's position and the direction it is facing in can be maintained by tracking its movements and integrating them, as will be shown in Section 7.4.1 (page 217), errors in the feedback from the robot's motors, and environmental interference, will accumulate over time to make these estimates increasingly inaccurate. By detecting landmarks in the environment and comparing them with the map, the robot can compensate for this accumulating error and keep better track of where it is relative to its environment.

For the specific "sentry" robot application developed in this thesis, maintaining a map is particularly important. The map is used to locate any areas in the environment that the robot has not yet explored so that the robot will not neglect to patrol these regions. The map is also fundamental to supporting algorithms which ensure that the robot patrols its entire territory in a timely, reliable fashion (see Section 5.3.2, page 143).

Having justified the need for maps, the following section will reconcile their use with the constraints placed on the design of a behaviour-based robot, particularly in terms of the use it may make of representation.

4.2 Using maps in a behaviour-based robot

At first glance, using maps seems to violate many of the criteria outlined in Section 2.4 (page 15) for a behaviour-based system, particularly the following:-

- *"Modules tend to be more reactive than deliberative"*- Building and maintaining a map seems very much an act of deliberation. Also, any module that uses the map will not be deciding its actions based entirely on the current state of the environment, and so that module will not be purely reactive in nature.
- *"Modules tend to be relatively simple"*- Inserting and extracting information to and from a map would seem to require a good deal of complexity in any module involved in doing so.
- *"Modules use distributed representations"*- Modules in a behaviour-based system should use representations tailored to their particular needs, and should not share these representations with other modules. A group of modules using information from a map would appear to be in contradiction of this.
- *"The world is considered its own best model"*- Modules should consult direct sensor data whenever practical, rather than relying on information from other modules. The world itself is never out-of-date or inaccurate, whereas derived sources of information may well be either or both.

The above considerations do not in fact rule out the use of maps- rather they place limits on what the nature of that map can be and particularly how it can be used. Examining once more the points listed above, these limits are as follows:-

- *Constraint: "Modules tend to be more reactive than deliberative"*- Construction and maintenance of the robot's map should not require any time-extended computation. It should be possible for the robot to update its map in real time, quickly integrating its perception of the immediate surroundings into the map as it moves. Equally importantly, it should not take excessive deliberation to actually interpret the map- the robot should be able to make use of the map immediately.

- *Constraint: "Modules tend to be relatively simple"*- It should be possible for a behaviour that needs to use the map to do so at the simplest possible level that is adequate for the task it is performing. The complexity of the map should not be reflected in the behaviours that use it.
- *Constraint: "Modules use distributed representations"*- While a map is by its nature dependent on the use of representation and modelling[15], the information the map provides should be accessible without shared representation. This is similar to the last point. Behaviours should be able to extract information from the map without needing to know anything about its structure or how it is maintained.
- *Constraint: "The world is considered its own best model"*- The world should be consulted in preference to internal information sources "whenever practical". While surprisingly complex tasks can be achieved simply by reacting to the current state of the environment, those tasks are limited to ones that can be done without any form of "memory". For any other tasks a behaviour-based robot has no choice but to make use of some form of map. This is acceptable as long as it only resorts to using the map if there is no way of extracting enough information from the immediate environment to perform its task- i.e. it still consults the world itself "whenever practical".

These requirements constrain both how the cartographic system should be implemented, and the nature of the services it should provide. Implementation issues are tackled in the next section, where a map representation scheme is developed that is consistent with the design principles of a behaviour-based system outlined here. Interfacing with that representation scheme to provide useful services is discussed later in the chapter.

4.3 "Marker"-based map representation scheme

This section develops a suitable representation for modelling the environment in a form that meets the design constraints outlined in the previous section.

The two most widely used map representation schemes in robotics are grid-based and topological (see Section 2.7, page 39). A grid-based scheme assumes the world is arranged in

[15] Subsumption-inspired cartographic schemes such as that used in Toto [8] claim to avoid representation by building a network of behaviours instead of a map, but this seems to be simply another implicit form of representation- although with many advantages over traditional symbolic models.

a manner somewhat analogous to a chess board, with all the objects in it arranged in definite locations that can be known exactly by the robot. If the robot can know its own position precisely, and is able to sense objects perfectly, there is no problem with this, but otherwise the scheme becomes essentially unworkable. It is very difficult to allow for any uncertainty in the robot's position in it, and limitations in the robot's sensing capability are particularly troublesome. Hence it is unsuitable for use with an autonomous robot, although it may work perfectly well in a simulated environment.

Topological schemes, in contrast, depend less on the exact location of the objects in the robot's environment, and concentrate more on trying to capture the basic shape of the environment in terms of its essential topology. In other words, the robot tries to determine which areas can be reached from each other, which are cut off from each other by obstructions, etc. As such the precise details of the robot's surroundings are not as significant, only the overall form, so uncertainty in the robot's position becomes less important. However the price paid for this is the extra computation necessary for recognising abstract topological features in the environment. For a robot with no long-range sensors, such features cannot be directly observed. The only way to observe them is indirectly, by keeping track of the discernible attributes of the immediate environment, combining that information into a model of the overall environment, and then analysing that model for topological features. In other words, in the absence of long-range sensors it is effectively necessary to build a map before topological features can be detected, so such features cannot be used in building the map in the first place.

The map representation scheme this section will develop is quite different from either of the grid-based or topological approaches. Instead of seeing a map as a model of the environment, it is viewed more as a "record of experiences". At any particular moment, the robot experiences the environment in its immediate vicinity, as perceived through the robot's sensors. It also experiences feedback from its motors indicating any movement it is making, which can be used to calculate its position. It is clear the most well informed robot conceivable is one which archives all this information exhaustively, continuously recording the state of the sensors along with the associated position of the robot, and never discarding any of this data. Such a

robot is an upper bound on the information a cartographic system can have[16], since it contains every piece of data ever available to the robot. An attractive feature of recording data in this form is that it makes no assumptions about the environment- no attempt is made to build a model of what is actually in the environment. The approach used in this project takes this trivially simple idea of archiving all the robot's experiences as a starting point, and modifies it into a form that is actually practical to implement.

Firstly, a complete archive of sensor state and motion feedback for all the time the robot is in operation is obviously hopelessly impractical- the robot would quickly "drown" in the information deluge, with too much data to process. To reduce this burden, it would seem reasonable to eliminate any records in the archive that apply to the same position of the robot as a new record being added, since the new record will have the most up-to-date information about the state of the environment at that position. However there is a problem with this. The robot's position is deduced by integrating the motion feedback from the robot's motors. Errors in the motion feedback are unavoidable, so the position estimate will gradually drift from the robot's true position. Hence two records showing the same "nominal position" of the robot cannot be assumed to correspond to the same physical position if there is a significant time interval between when they were recorded. However, if the robot were in some way able to use features in the environment to keep its estimate of its position consistent, then it would be acceptable to keep only the most recent record corresponding to a given position and discard all previous records for that location.

At this point an initially *unjustified* assumption is made that this is indeed the case- that the cartographic system eventually constructed will be able to keep track of its position in some way. Given that, we can discard as out-of-date any entries in the archive that are marked at the same position estimate as the current position when we add a new record. This step will be justified in Section 4.4 (page 95), where it is shown that the representation scheme which the assumption made here leads to will in fact allow the robot to keep track of its position by detecting stable features of the environment called *landmarks*.

Another change to the archiving strategy needs to be made before it becomes practical. Obviously records of sensor state versus robot position cannot be made for every continuous

[16] Of course, trying to make use of this huge archive of data sensibly would be another matter entirely.

point along the robot's path, as this would still demand unlimited storage space. Instead, it is acceptable to record "samples" at positions with a certain minimum distance between them- so the robot has information about representative points in every area it has been. The minimum distance between records is chosen based on how much memory is available to store the samples- the smaller the distance, the more samples in a given area and the more memory that will be required (see Section 4.3.2, page 93).

At this point, the "record of experiences" has been re-cast in a form that is actually practical to implement. The records of sensor state versus nominal position are called *"markers"* in the rest of this chapter. The example shown in Figures 4-2, 4-3, and 4-4 reviews and clarifies the proposed nature of the "marker"-based representation scheme.

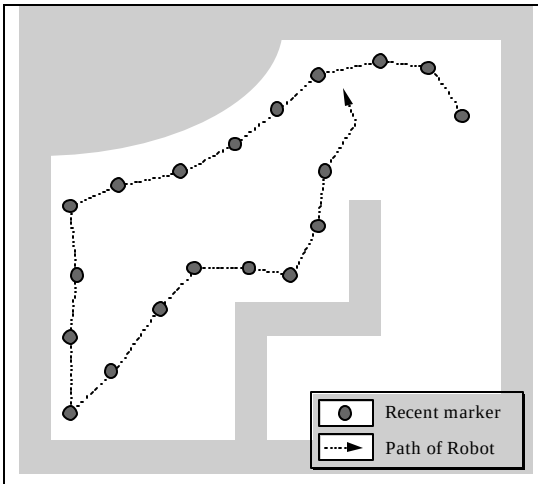


Figure 4-2: Laying markers

The diagram opposite shows the robot moving through a simple environment. As the robot moves, it records markers at fixed minimum distances from each other along its path. Recording a marker is called "laying" it, as if the robot were laying down some physical trace in the environment as it moved. The information in these markers details the readings from the robot's sensors the last time it was in the area they were laid

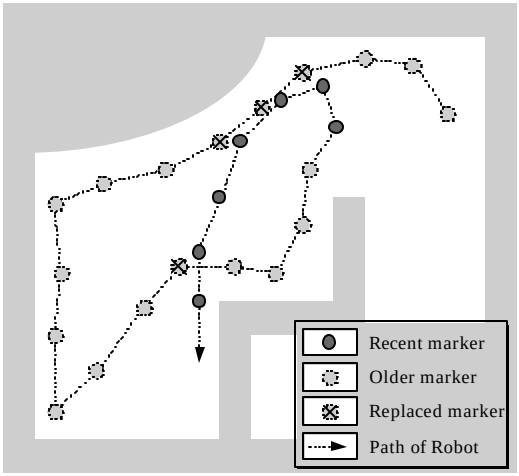


Figure 4-3: Replacing markers

When the robot is in an area where it has laid markers before, those markers are removed and replaced. The new markers contain the most up-to-date information that the robot has about the environment in that location, and hence the older markers become redundant. If the environment should have changed since the robot was last in the area, the new markers will reflect the present reality. Hence the ability to cope with a dynamic environment is built in at a very basic level to this representation scheme.

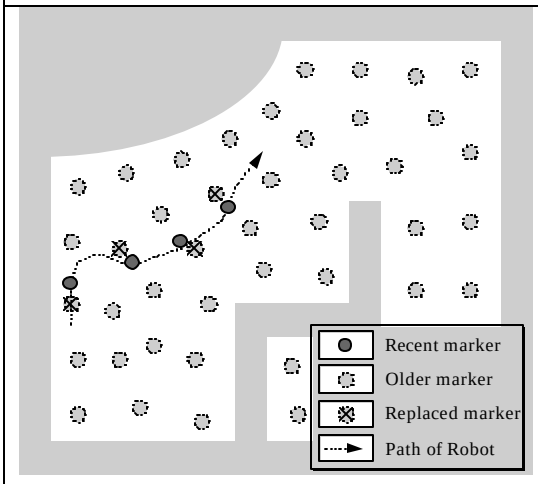


Figure 4-4: A map as a collection of markers

The diagram opposite is for a considerable time afterwards, when the robot has exhaustively explored the environment. It now has markers distributed evenly at representative points throughout its territory. The robot still continues to replace markers as it moves, keeping them up to date with any changes in their area. The markers as a group are effectively a map of the robot's environment.

By using markers, the robot can deduce what its sensors were reading the last time it was in a particular position. But it is not as easy to work the other way, and use markers to estimate the robot's position from its current sensor readings. This is because there may be many markers with the same set of sensor readings associated with them, and the robot will not be able to distinguish them. The robot's sensors are not rich enough to record its environment at a level of detail that would make this distinction possible - for example, all the markers that are away from boundaries look the exact same to the robot: zero proximity readings on all sides. This is why it is important to try to isolate special "landmark" features in the environment that the

robot *can* distinguish from other areas and use to keep track of its position. Although most markers of themselves cannot be used as landmarks, there are some markers- or *groups* of markers- that can. These will be discussed in Section 4.4 (page 95).

The robot's "map" in this form of representation is simply a collection of markers. Since the collection is potentially large, it is important that it be structured in a way that lets the robot extract the information it needs from the map in a timely fashion, without having to do any computationally expensive searching. The following section discusses a method for achieving this by structuring markers into a hierarchy of "neighbourhoods".

4.3.1 Neighbourhoods

As the robot moves, it needs to constantly discard outdated markers and replace them with ones that reflect the current state of the environment. To do this, the robot must be able to determine which markers were laid in the same locations as the new markers it is currently laying, so that it can eliminate them. Hence, it is vital that the robot can quickly find all the markers whose positions are close to the current location of the robot. As well as this, it is useful to be able to determine which markers are in the general locality of the robot- even if they are not in its immediate vicinity- so that the robot can gain an impression of what the environment beyond its sensor reach is like in the area through which it is moving. In general, the closer a marker is to the current position of the robot, the more relevant it is to the robot.

"Neighbourhoods" are structures for filtering groups of markers by their distance from the robot so that it can find out about its locality in a timely fashion without searching the entire archive.

The ideal situation would be if a fully sorted list of markers could be maintained such that at any instant the markers are ordered by their distance from the robot. The robot could then easily access the markers relevant to it. A difficulty with this is simply that it takes too long to do. A fast sort algorithm such as a heap sort has a complexity of $O(n \log n)$. This is not bad, but would be a drain on the resources of the robot- acutely so for the lower-end robots with limited processors the work in this thesis is aimed at. A more serious problem is that the robot will not stop moving and wait for the sort to complete- since that could lead to pauses of the order of minutes for a typical robot. Therefore, the "metric" by which the information is being

sorted- distance to the robot- is continually changing as the robot moves. Fast clever sort algorithms are not designed with this in mind, and so their performance would be difficult to predict under these unique conditions.

Investigation with Khepera, the robot the work in this thesis was implemented on, showed that, practically speaking, the map maintainer could rely on being allowed on the order of 10 "updates" to the sorting process per second. Within an update, only a handful of markers could be processed, or system performance would degrade badly. Given these constraints, it is not possible to maintain an ordered list of markers using normal sorting algorithms unless the robot is limited to moving extremely slowly.

However, the robot does not actually need to sort its markers fully. It is important that the robot can identify markers that are in its immediate vicinity, or its general locality. Beyond this, it does not need to know which markers are closer than others. This would suggest a "bucket sort", where all markers are examined and placed in one of three "buckets" labelled "immediate", "near", or "distant", depending on how close they are to the robot. In a conventional bucket sort, all the markers would be examined in turn and classified one by one. Experimental tests proved this to be impractical because the sort couldn't keep up with the rate at which the robot could move- the full sort took on the order of a minute, while the set of markers that should be in the "immediate" bucket changed every few seconds (and the "near" bucket every few tens of seconds). The reason for this high rate of change is because, while a marker that is distant from the robot tends to stay distant for some time, a marker that is in the immediate vicinity of the robot at one moment may not be so in the next. The robot only has to move a small distance to completely change which set of markers are in its immediate vicinity. A similar argument applies to the "near" bucket, except the robot has to move further for a given marker to no longer be near it, so the rate of turnover of markers is not as high.

A significant improvement to the sorting process can be achieved by observing that in general the number of markers in a bucket will be roughly proportional to the area it encompasses. Hence there will be fewer markers in the "near" bucket than there are in the "distant" bucket, and fewer still in the "immediate" bucket. If the robot were to devote the same length of time to sorting each individual bucket, then the "immediate" bucket would be worked through at the fastest rate (since it has the fewest markers), the "near" bucket would be processed somewhat more slowly, and the "distant" bucket would be worked through slowest of all. Effectively, the

rate at which all the markers would be updated in a given bucket will be higher the fewer markers there are in that bucket. But the buckets with the fewest markers are the ones closest to the robot, which are also the ones with the greatest rate of change. Hence this is very desirable behaviour since it means that the robot will process the buckets at a rate that mirrors the speed at which changes take place in them.

So the sort can be improved by simply modifying it so that in one "cycle" it takes a single marker from each of the buckets, sorts them, then repeats the cycle[17]. Hence the same amount of time is spent sorting each bucket. Because there are just a few markers in the "immediate" bucket, the bucket sort cycles through them very quickly, so they are updated at a rate that can keep up with the robot's movement. The "near" bucket has some more markers, and takes slightly longer to pass through, which is acceptable because the turn-over of markers is not as fast as in the "immediate" bucket. The "distant" bucket has many markers in it, and so is passed through most slowly. In practice experimental tests proved it was passed through *too* slowly- markers that should be moved to "near" were not detected quickly enough. For that reason, an intermediate bucket between "near" and "distant" was inserted, called "local", to hold markers that were indeed distant from the robot, but still near enough to be worth monitoring more closely so that they could be moved to the "near" bucket in a timely fashion when appropriate.

The "buckets" discussed above will be called "*neighbourhoods*" in this chapter, since they represent a collection of markers all at the same general distance range from the robot (see Figure 4-5).

[17] The robot will not run out of markers because, when it has processed all the markers in a given bucket, it should simply start over and begin processing them all again. This is because the robot will have moved in the meantime, and the markers may now need to placed in different buckets.

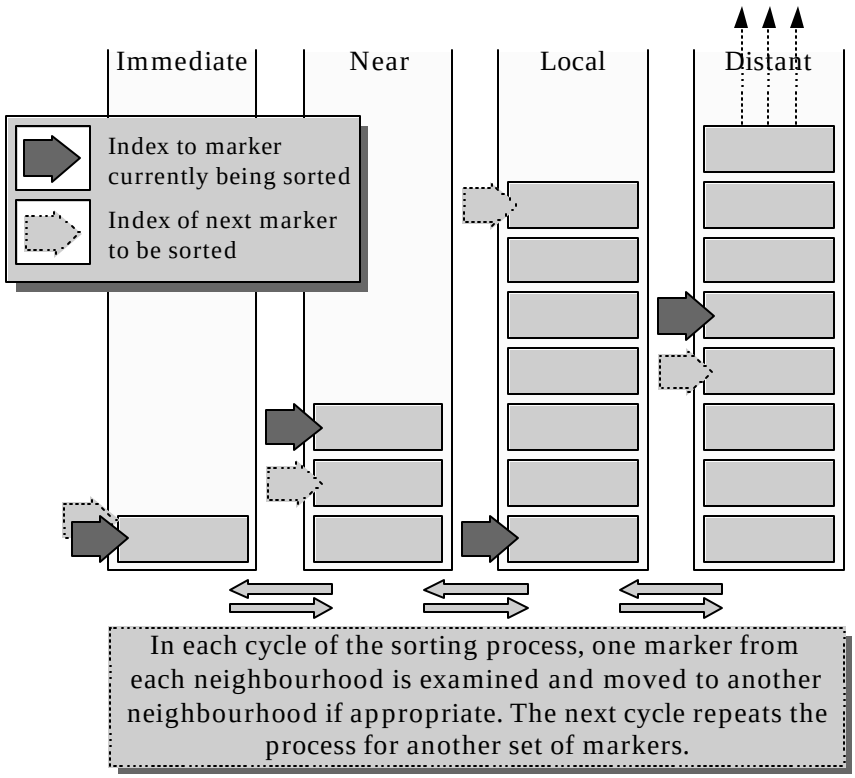


Figure 4-5: Sorting markers using neighbourhoods

The properties of neighbourhoods will now be formalised. Each neighbourhood has an associated nominal distance range (see Figure 4-6). The robot attempts to keep markers whose positions are within that distance range from the current robot position in the appropriate neighbourhood. The distance range associated with a neighbourhood is called *nominal* because in general the distance of some markers in a neighbourhood from the robot's current position will in fact lie outside the neighbourhood's specified distance range. This happens when a movement of a robot changes the distance from the robot to a marker sufficiently to make it inappropriate for the neighbourhood it is in. The marker will be re-classified into the correct neighbourhood the next time it is examined by the sorting process.

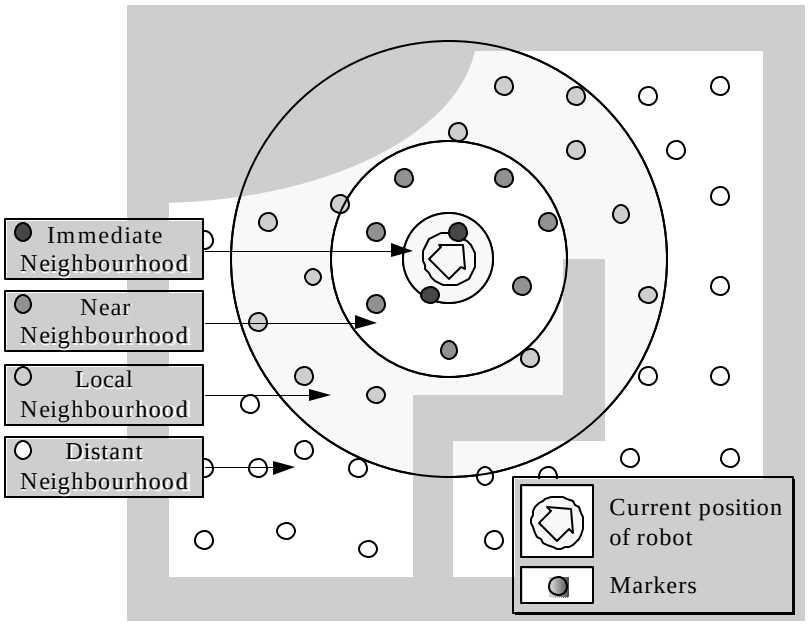


Figure 4-6: Distance ranges associated with neighbourhoods

If the robot were stationary long enough for the sorting process to place all the markers in their appropriate neighbourhood, then the neighbourhoods would contain the following:-

- **Immediate Neighbourhood** - this contains any marker whose position is very close to the current robot position, and which would need to be discarded if a new marker was laid at that point. The minimum distance enforced between markers keeps the number of markers here very low- either zero, one, sometimes two. The size of the immediate neighbourhood should correspond to this minimum distance (see Section 4.3.2).
- **Near Neighbourhood** - this contains any marker located in the general surroundings of the robot. These markers let the robot gain an impression of the environment just beyond the reach of its sensors. The minimum distance enforced by the marker laying system will also keep the number of markers in this neighbourhood relatively low. For the choice of minimum distance used in this work, the number of markers here was kept at about half a dozen.
- **Local Neighbourhood** - this contains any markers which do not qualify as "Near", but which might do so in the foreseeable future. It acts as a buffer zone between "Near" and "Distant", increasing the speed at which it is possible to detect markers that have become "Near" because of movement of the robot. The extent of this neighbourhood is not critical, but if possible it should be chosen so that its distance range is greater than the distance the

robot can move in the time it takes to sort through all the markers in the neighbourhood once. For the speed at which the robot this project was implemented on could process markers compared with the speed it could move through the environment, this condition was satisfied by choosing a distance range that kept on the order of 20 markers in this neighbourhood.

- **Distant Neighbourhood** - this contains any other markers too distant from the robot to fall into any of the other neighbourhoods.

The process of sorting markers is a continuous one, because effectively the "goal-posts move" during the sort because the position of the robot keeps changing. Hence the robot cannot expect to ever have all the markers ordered correctly unless the robot is stationary for an extended period. But the neighbourhood system allows the robot to concentrate its efforts on those areas that are most relevant to it- the immediate and near neighbourhood- and process distant markers at a slower rate.

The system developed here is dedicated to making sure the robot always has good *local* knowledge of its environment. It is not very suitable if the robot needs to examine the map at particular positions distant from it. No way could be found to allow that and still have acceptable run-time performance[18]. Such an ability is not generally desirable in a behaviour-based robot anyway- the robot is mostly concerned with supplementing its immediate perception of the local environment with information from the map about that same locality.

4.3.2 Marker Laying System

The marker laying system is designed to try to keep exactly one marker in the "immediate" neighbourhood. This enforces a minimum distance between markers so that the robot does not become swamped with many markers all corresponding to the approximately the same location. When a marker is laid, all other markers in the "immediate" neighbourhood are considered to be superseded by this marker, and are deleted. A new marker is laid every time the robot moves beyond the chosen minimum distance from the last marker laid.

[18] In Section 4.5.4 (page 115) a method of searching the robot's map even at locations distant from the robot is presented, but it is not a technique that can be applied in real-time, only as a background task.

This marker laying system leads to the memory requirements of the cartographic system growing with the ratio between the area of the robot's environment and the area a single marker covers (i.e. the "level of granularity" at which the environment is mapped). A good choice for the area a single marker should cover is a fairly large fraction of the area of the body of the robot itself, since features smaller than the robot have little impact on it. In this project, markers represented an area of about one third of the robot's body. The minimum distance between markers is deduced as the radius of the area a single marker is chosen to represent.

4.3.3 Adding annotations to markers

Markers are essentially records containing data derived from the robot's sensors- its "experiences"- along with the position that the robot was in when it laid the marker. In practice it is useful to also include a timestamp to indicate when the marker was laid, but that is all the information specifically needed by the basic marker representation scheme.

However, as discussed in Section 4.3, the validity of the marker representation scheme depends on the ability of the robot to recognise landmarks in its environment. This will be examined in the next section. For now, it is important to observe that the landmark system will need some way to store data about landmarks it has detected, so that the next time it meets them it will have something to compare them against. Detecting landmarks is of no use unless deductions can be drawn from them[19]. The way this is handled is by letting the landmark system "hook on" its own data to the marker. Keeping this data with the marker, rather than setting up separate storage for it, means that the landmark system is effectively allowed to "annotate" the robot's map, and have those "annotations" brought back to its attention when the robot returns to the area where they were made.

This ability to add "annotations" to the map by including data in markers is quite useful for other components of the cartographic system as well. In Figure 4-7, the full structure of a marker is shown, including all the annotations added by various parts of the cartographic system discussed later in this chapter. The diagram is given simply to demonstrate that a

[19] In fact for one of the two types of landmarks that will be examined the basic marker data is sufficient, but the other will indeed need to store extra data characterising the landmark it detects (see Section 4.4.3, page 108).

marker acts as a single "hook" onto which diverse information for different parts of the cartographic system can be hung- the actual data stored in the annotations are not relevant yet.

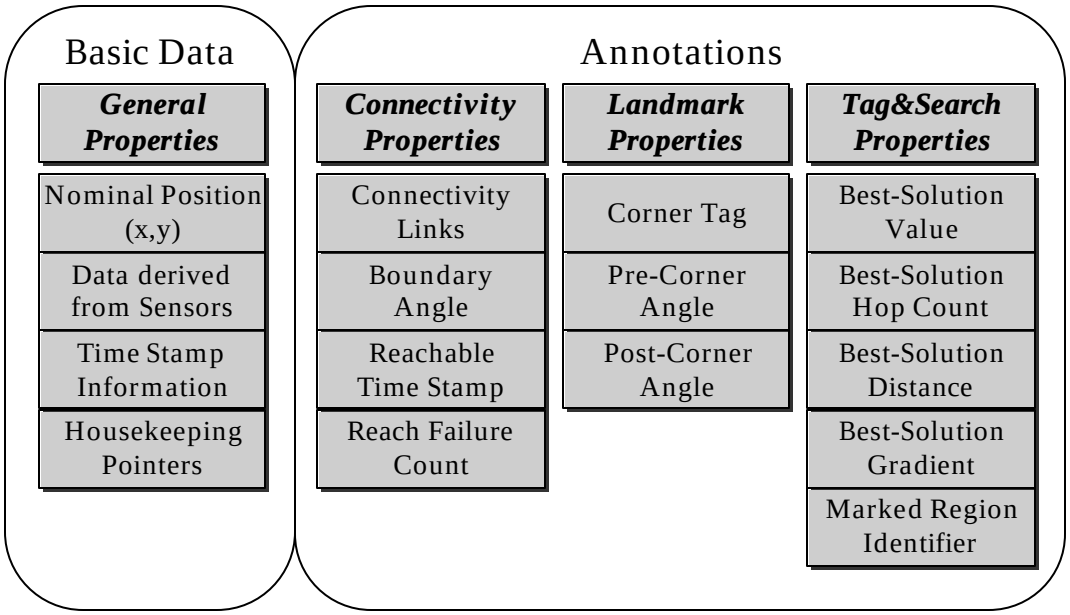


Figure 4-7: Marker structure with annotations

This completes the discussion of the marker representation scheme. Remember however that an initial assumption was made when developing the scheme that has not yet been justified. The assumption was that a landmark recognition system could be built using the marker representation that would be able to keep the robot's estimates of its position and direction consistent over time. The next section demonstrates how this can be done, and so confirms that a map composed of markers can be maintained for extended periods.

4.4 Using Landmarks

It is important that the robot has some means of recognising landmarks in its environment- otherwise the robot will lose track of its position and direction, and be unable to continue to maintain its map. A fundamental assumption was made in Section 4.3 (page 83), where it was assumed that the cartographic system, when constructed, would be able to keep track of its position by using landmarks. That confidence in the robot's estimate of its position was a

necessary starting point for the marker-based map representation scheme. Now that assumption needs to be justified.

The robot's estimate of its position is generated by tracking the movements of the robot and integrating its motion to give its position[20]. The robot determines its motion by feedback from its motors. However, this feedback will contain some error due to occasional motor slippage and environmental interference with the robot's progress. Since the motion feedback is integrated to give position, these errors will also be integrated, and the robot's estimate of its position will grow less accurate over time as the error builds up. The robot's estimate of the direction it is facing in is generated in a similar way, by keeping track of every turn the robot makes. The direction estimate will also grow less accurate over time for the same reasons as the position estimate.

The purpose of building a landmark recognition system is to find stable features of the environment that can be used as reference points to deduce how much error there is in the robot's position and direction estimates, and make the appropriate corrections to compensate for that error. This section presents a scheme for recognising landmarks in the environment with proximity sensors only, using the marker system developed in the previous sections.

A robot with proximity sensors only cannot simply "look" at an object and recognise it. It can only sense the small portion of an object that is in its immediate locality. And even that portion may not be sensed very accurately. For example, the proximity sensors of the robot that the work in this thesis was implemented on gave a distance reading that was non-linear, noisy, influenced by ambient light, and the colour, texture, and other features of the object. Such readings are not even remotely suitable for direct use in landmark recognition. There are simply no stable features to recognise. As an example of the difficulties, the same object in the same position relative to the robot may give varying proximity readings depending on how sunny the day is. The basic problem is that, because of all the uncertainties involved, there is no way to take a reading from a proximity sensor and actually convert it to an exact distance measurement to the object.

However, such non-ideal sensor data *is* sufficient to allow the robot to perform a simple task like following the boundary of an object. The details of how this is done are discussed in

[20] The details of how this is achieved will be described in Section 7.4.1 (page 217)

Section 5.2.3 (page 129), but essentially the robot simply comes close to the object, turns at right angles to its boundary, then moves forward continuously, turning to the left or right as it detects the edge coming closer or falling away (see Figure 4-8). The robot never needs to know the exact distance it is from the object, only whether that distance is increasing or decreasing. This *relative* information can be extracted reasonably reliably from proximity readings, even though the *absolute* distances the readings represent cannot.

The reason this is relevant to a discussion of landmarks is that when the robot is following the edge of an object, the path it moves along will trace the outline of that object's boundary (see Figure 4-8). The distance the path is from the object will depend on the exact nature of the object's surface- but whatever the distance is, it will be consistent, since the nature of surfaces tends to be remain constant. So if the robot follows the same boundary twice, the path it follows will generally be consistent. Hence recognisable features such as corners and edges that appear in the path will reappear in the same places the next time the robot follows the boundary. Therefore these features can act as landmarks for the robot.

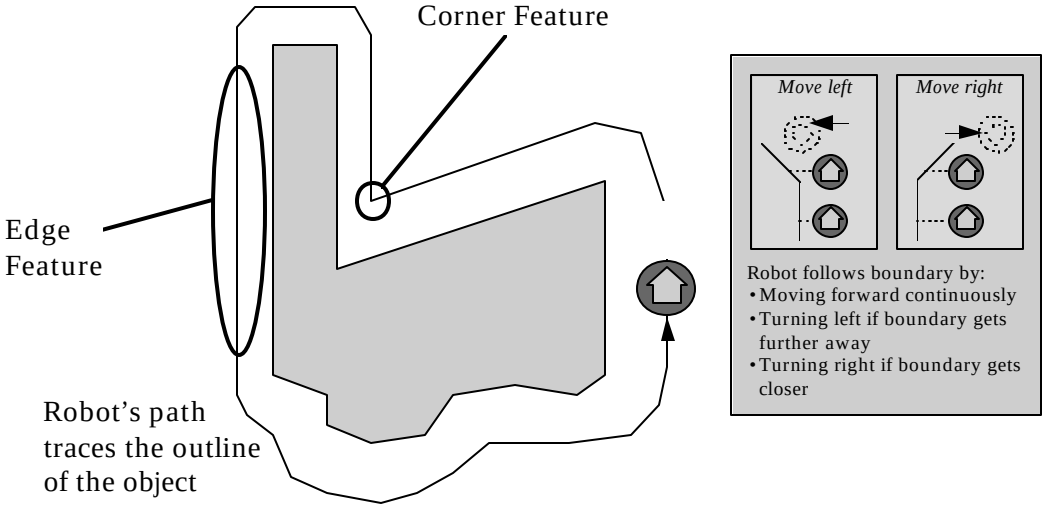


Figure 4-8: Tracing the outline of an object

The sections that follow discuss how the ideas developed here can be implemented using the marker-based map examined earlier.

4.4.1 General strategy for trusting landmarks

There are two sections of a boundary which are particularly well suited to act as landmarks- corners and straight edge sections. These segments of the boundary have well-defined, stable

features from which consistent information can be extracted, as will be shown soon. With careful use of this information, it is possible to compensate for accumulating error in the robot's estimates of its position and direction. Before looking at the details of how this can be done, some general difficulties with landmarks need to be addressed.

While the idea of landmarks is that they provide reference points that the robot can use to keep track of its position, it is important to remember that the robot's environment is not static. It cannot be assumed that a change in the apparent position of a landmark automatically means that the robot's estimate of its position has become inaccurate - it may equally well be that the landmark has simply moved. A change in the position of a landmark and an error in the robot's estimate of its position are indistinguishable while the robot is close to that landmark. However, an important point to realise is that they *can* be distinguished once the robot moves away from that area. A change in the position of a landmark will not change the position of any other landmark, whereas a drift in the robot's sense of position will change the perceived position of *every* landmark. One is a local change, the other is a global change. So if the robot finds that every landmark it meets seems to have moved, then it becomes more and more likely that the movement is only apparent, caused by a drift in the robot's estimate of its position. Essentially, the "consensus" of the landmarks in the environment is used to determine what has occurred in a particular part of that environment. This use of landmarks ensures that the map is kept self-consistent by essentially averaging error over the entire map[21].

A strategy for applying corrections to the robot's estimates is now presented that follows from the above considerations. It is first assumed *for the purposes of calculation* that apparent changes in the environment are entirely due to drifts in the robot's position and direction estimates and not to changes in the position of landmarks. Then, having computed the corrections to the robot's estimate of its position and direction that would be appropriate under that condition, only a *conservative fraction* of each correction is actually applied. If the apparent changes in the environment are in fact due to drifts in the robot's estimates, then every landmark the robot meets will continue to apply these corrections until they build up sufficiently to compensate for the drifts. If, on the other hand the apparent changes are real,

[21] Note that this averaging process, while it keeps the map self-consistent, does not prevent the coordinate system of the map drifting over time. This point has no bearing on landmark recognition, but does have bearing on the nature of the services the map can provide - see Section 4.5.1, page 113.

and due to an actual change in the position of a landmark, the correction will not be reinforced at any other landmark. In fact, at other landmarks the "incorrect correction" will be compensated for, since the error introduced into the robot's estimates will seem just like a natural drift from accumulating error and can be corrected as such. The "conservative fraction" mentioned earlier should be one chosen so that if a correction is applied in error, it is small enough to be recovered from through the same process that deals with normal drift. Suitable values will be quoted for straight-edge and corner landmarks when they are discussed. It is important to note that corrections to the robot's estimate of its *direction* must be made particularly conservatively, because small erroneous corrections to the direction will be multiplied into very large errors in its position as the robot moves. Erroneous corrections to the position estimate do not become amplified in this way[22].

This ability of the landmark system to "heal itself" if errors are introduced while attempting to make corrections is quite general, so long as the errors are not large enough to prevent landmarks being recognised. One useful consequence of this is that it is acceptable to make approximations when calculating the corrections to the robot's estimates from a landmark. The difference between the approximation and the exact answer can be seen as introducing an error component into the corrections the robot makes which may add to the drifts in the estimates rather than removing them. This will simply appear as an extra component in the position and direction drift calculated in future corrections and be eliminated.

The following sections now look at the details of detecting particular landmarks and using them to make appropriate corrections to the robot's position and direction estimates.

4.4.2 Straight-edge landmarks

This section discusses how straight sections of an object's boundary can be used as landmarks.

As the robot moves, it lays markers representing what it senses, replacing markers already present that represent what the robot sensed last time it was in the area (see Section 4.3.2,

[22] If you aim a cannon at a target, but shoot a few degrees in the wrong direction, you will miss the target by a distance that keeps increasing the further the cannonball goes. If, on the other hand, you aim the cannon exactly on target, and move it a few paces left or right before firing - still facing the same direction - then the cannonball will miss the target by just the distance you moved no matter how far away the target is.

page 93). If the robot follows a straight edge section of a boundary that it has passed before, it should find that the markers it is laying are collinear with the ones being replaced. In practice this may not happen, either because the robot's estimate of its position may have drifted due to accumulating error, or its direction estimate may have drifted, or both. Of course, it is also possible that the position of the edge may have changed. This possibility will be ignored for now, and then accounted for later. The logic of doing this was given in the discussion in the previous section.

It is possible to assess the amount by which the robot's estimates have drifted since the last time it passed a particular straight edge segment. This is done by working backwards from an examination of the apparent difference between the edge as it is perceived currently with how it was perceived in the past, as recorded in the markers the robot laid at that time.

The effect of a drift in the robot's position estimate is shown in Figure 4-9. The robot is shown following the same edge segment twice. As the robot follows the edge for the first time, it lays markers in a straight line at regular intervals along it. When the robot returns to the edge at a later stage, any drift in the robot's position estimate will cause the markers it lays this time around not to be collinear with the ones laid originally[23].

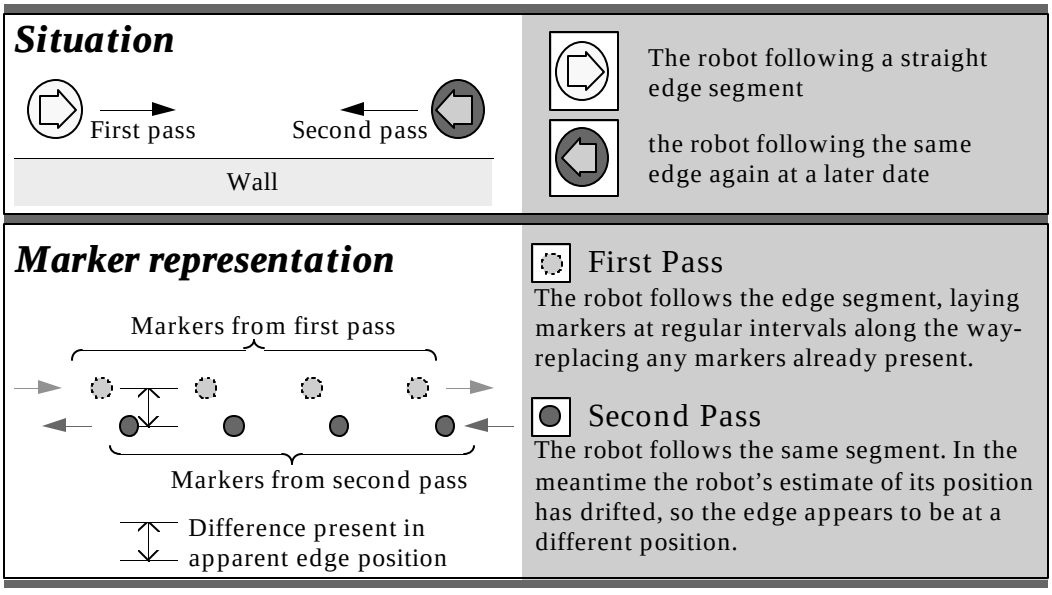


Figure 4-9: Effect of position estimate drift

[23] Unless the drift happened to take place along the direction of the edge itself. This possibility will be discussed soon.

If the robot finds itself following an edge while replacing markers that suggest the past existence of an edge very similar to the one it is following[24], but slightly offset, that is a good indication that the robot's position estimate has drifted and the two edges are in fact the same. Of course, it is necessary to be cautious in deducing corrections from this since the edge may in fact have moved in the environment. This issue is discussed in more detail further on.

The amount of information the robot can extract about the drift in its position estimate varies. If the drift occurs in the same direction as the edge itself extends in, then it will be undetectable, since the markers the robot lays will still be collinear with the ones laid previously. If the drift was at right angles to the edge, it will be detected in full. In general, the component of the drift perpendicular to the edge can be deduced.

In Figure 4-10, the effect of a drift in the robot's *direction* estimate is portrayed. This type of drift is more serious, since even a small drift in direction can produce a large drift in position as the robot moves.

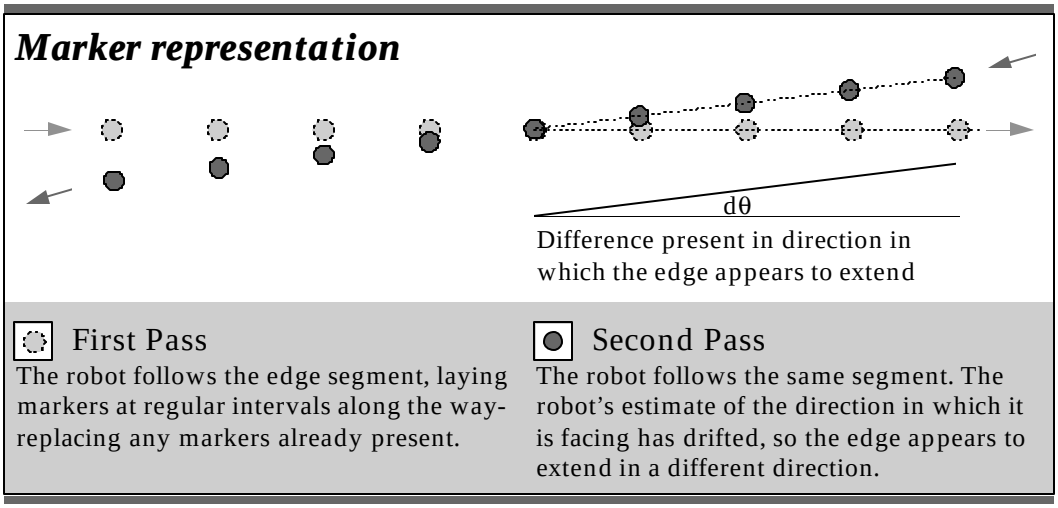


Figure 4-10: Effect of direction estimate drift

The drift in the direction estimate can be found in full from the difference in the direction the edge appears to extend in currently compared to the direction in which the previous set of markers laid alongside the edge extend.

In general both position and direction drift may be present. Figure 4-11 illustrates this situation. Note that the point at which the drifts are being calculated is labelled. This is because direction

[24] The robot can tell if the markers it is replacing indicate the presence of an edge by examining the proximity sensor data recorded in them.

drift causes a drift in position that changes as the robot moves, so the position drift calculated will depend on where along the robot's path it is measured. The direction drift can be found by comparing the direction the two sets of markers are aligned in as before. The detectable component of the position drift at a particular point along the robot's path can then be calculated by drawing the imaginary line that the robot would have followed if the direction drift had been corrected at that point, and then finding the perpendicular distance between that line and the one formed by the recorded markers.

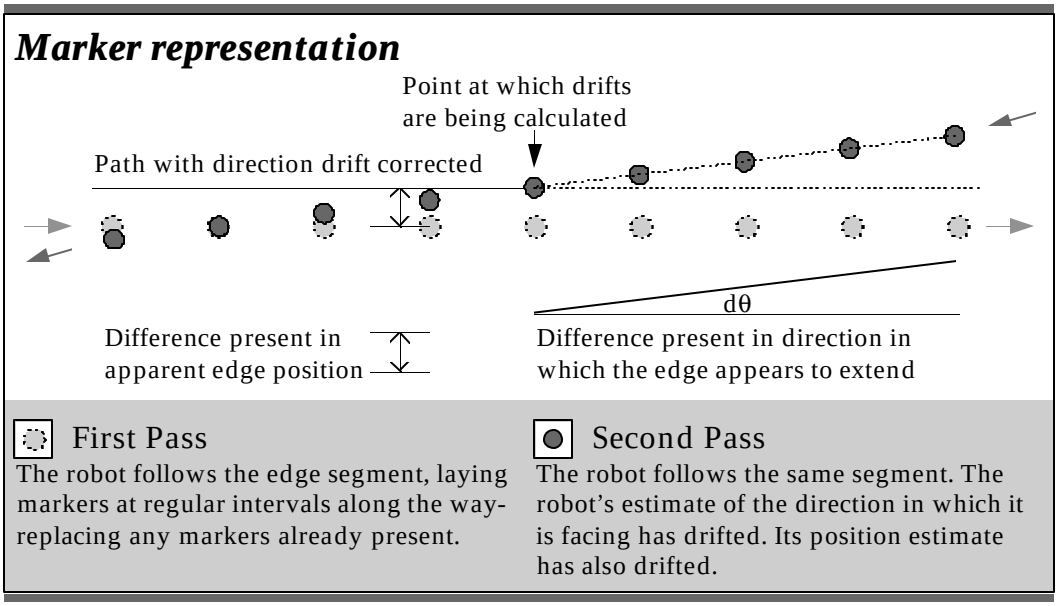


Figure 4-11: Combined effect of direction and position estimate drift

The practical issues involved with actually performing the calculations for the drifts in the robot's position and direction estimates will now be examined.

Calculating drifts involves continuously comparing markers recently laid with the markers they replaced. This can be done by maintaining a list of the last *n* markers laid and the markers they replaced. By applying metrics to these markers it can be determined if they indicate that the robot is following a straight edge that it has met before. If this is found to be the case, the two sets of markers can be compared to calculate drifts in the robot's position and direction estimates as outlined earlier.

Because of the steady computational burden these continual comparisons make on the robot, it is crucial that the robot can make the relevant calculations quickly and efficiently. This was

done by carefully choosing metrics that made little use of transcendental functions[25], and by limiting the number of markers that were compared to a manageable number. It was decided to involve only the last *three* markers laid (and the markers they replaced) in calculations. This decision was based on a number of factors:-

- The more markers used, the longer an edge segment needs to remain straight for it to be accepted as a landmark. Choosing a low number of markers implies that even if the edges of boundaries are only straight in small sections, these sections can still be recognised.
- Using a low number of markers also allows edges that have a small curve to be used as landmarks- since these are close enough to a straight line for practical purposes over short distances.
- The mathematics involved is particularly simple for the three-marker case, minimising the computational burden on the robot[26].

Figure 4-12 shows the information the robot has available to it when it maintains a list of the last three markers laid (p_1, p_2 and p_3) and the markers that they replaced (q_1, q_2 and q_3). The marker p_1 is the one most recently laid.

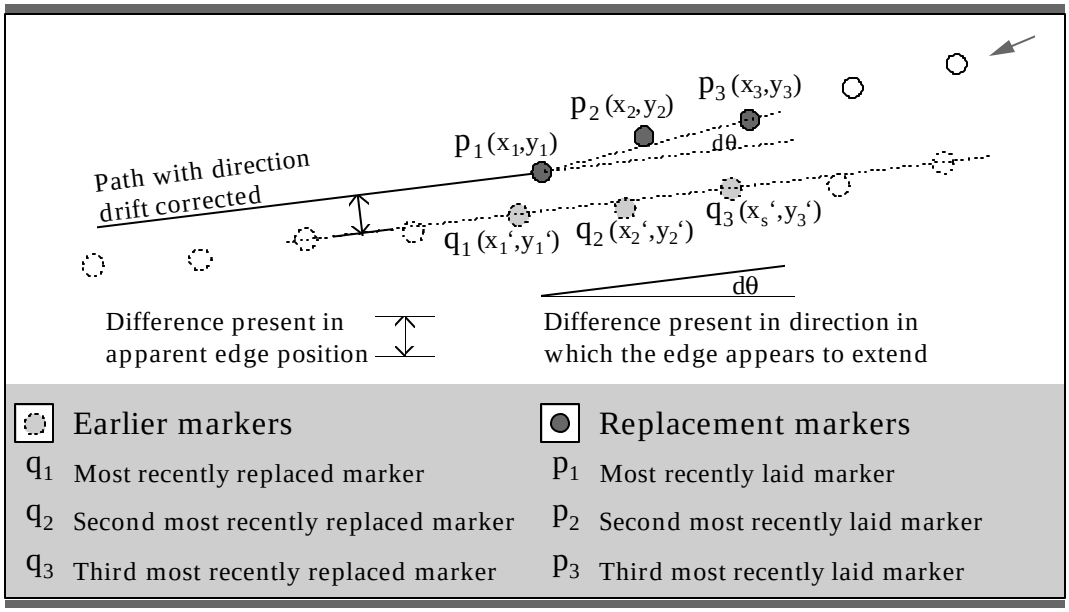


Figure 4-12: Computing drifts by comparing three markers

[25] Functions such as sine, cosine, square root etc. take longer to calculate than multiplication, division, addition, etc. The robot is unlikely to have a maths co-processor.

[26] In fact for the physical robot used in this project, three markers was the maximum that could be used without unacceptable runtime performance degradation.

Before comparing the two sets of markers, it is important to evaluate how confident the robot can be in the corrections to the drifts in the robot's position and direction estimates that it deduces from them. Firstly, it is important that each set represents a straight edge. This can be tested by measuring how close to being collinear each set of markers are. There are many possible metrics that can be used- the one described here was chosen because it avoided the use of transcendental functions.

Consider the angle θ formed between the most recent marker laid, the second most recent marker, and the remaining marker. This situation is shown in Figure 4-13. The closer θ is to 180°, the closer the three markers are to being collinear.

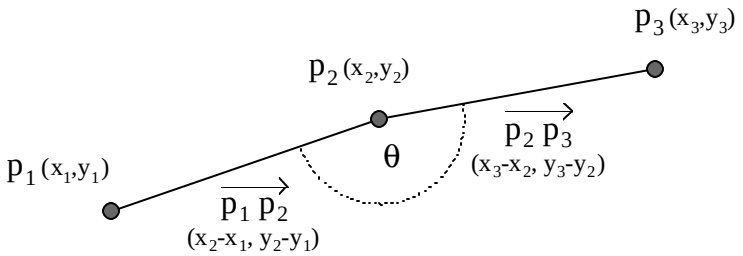


Figure 4-13: Testing the collinearity of a set of markers

The inner product of the vectors formed between the markers at the extremes and the marker in the centre is:-

$$\overrightarrow{p_1p_2}.\overrightarrow{p_2p_3} = \left|\overrightarrow{p_1p_2}\right|\left|\overrightarrow{p_2p_3}\right|\cos\theta$$

Rearranging for cos θ gives:-

$$\cos\theta = \frac{\overrightarrow{p_1p_2}.\overrightarrow{p_2p_3}}{\left|\overrightarrow{p_1p_2}\right|\left|\overrightarrow{p_2p_3}\right|} = \frac{(x_2-x_1)(x_3-x_2)+(y_2-y_1)(y_3-y_2)}{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}\sqrt{(x_3-x_2)^2+(y_3-y_2)^2}}$$

Squaring both sides will remove the square roots on the right- square roots are undesirable since they are transcendental functions and relatively expensive to calculate.

$$\cos^2\theta = \frac{\left[(x_2-x_1)(x_3-x_2)+(y_2-y_1)(y_3-y_2)\right]^2}{\left[(x_2-x_1)^2+(y_2-y_1)^2\right]\left[(x_3-x_2)^2+(y_3-y_2)^2\right]} \qquad ...(4.1)$$

At an angle of 180° the markers are collinear, and $\cos^2\theta$ is at its maximum value of unity. The further the markers are from being collinear, the further the angle θ will deviate from 180°, and

the further $\cos^2\theta$ falls from unity[27]. Hence the fraction on the right of the above equation can be used as a metric of the "straightness" of the line formed by the markers p_1, p_2 and p_3. The same metric can be applied to the markers q_1, q_2 and q_3.

As well as being collinear, it is important that the two sets of markers form lines that extend in approximately the same direction. The further their directions diverge, the less confident the robot can be that the differences can be accounted for by drift in the robot's direction estimate, and the more likely it is that the difference is caused by an actual change in the environment. The direction of a set of markers is here approximated as the angle formed between the markers at the extremities as shown in Figure 4-14. The approximation is acceptable because, if it is not reasonably accurate, the collinearity metric will have generated a low value so the robot will know not to have confidence in the markers as a landmark.

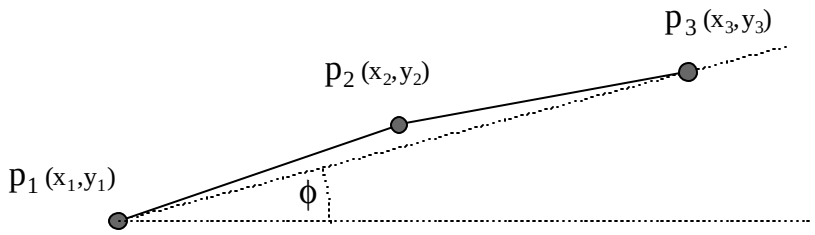


Figure 4-14: The direction indicated by a set of markers

Note that calculating this angle unavoidably involves the use of a transcendental trigonometric function, inverse tangent:-

$$\phi = \tan^{-1}\left(\frac{y_3 - y_1}{x_3 - x_1}\right) \qquad ...(4.2)$$

The same calculation can be performed for the markers q_1, q_2 and q_3. Then the closer the difference in the angles calculated is to zero, the more confident the robot can be that the lines represent the same feature. It would be reasonable to combine this angle confidence metric with the collinearity one given in Equation 4.1. However a better alternative is to only calculate the angle metric if the collinearity metric is satisfactory, since this reduces the average

[27] Of course, $\cos^2\theta$ also approaches unity as θ approaches 0°- i.e. when the robot turns about-face. This undesired possibility could be tested for, but instead a more general idea for eliminating spurious landmarks was used. Straight-edge landmark detection was only enabled when the robot was following an edge smoothly, and making no sharp turns. The "curvature" virtual sensor to be described later in Section 4.5.2, page 114 was used to check for this condition.

computational load on the robot. A cut-off confidence of 0.9 for the collinearity metric was found to work well. A cut-off confidence of ±10° was similarly placed on the angle metric, so that if it was not satisfactory the robot would perform no further computation. These numbers were chosen based on an estimate of the rate at which position and direction drift occurred on the particular robot used in this project. The use of such "magic numbers" could be avoided by simply not using cut-off confidence thresholds, and factoring the metrics into an overall confidence in the final calculated corrections to the robot's drifts. This was not done because it required the robot to complete all stages of the calculations, rather than only performing them when it would be able to have high confidence in the results. The cut-offs reduced the computational burden on the robot significantly with only a small price in missed opportunities to fix drifts.

The actual corrections to the robot's estimates of its position and direction are now calculated. The drift in direction is simply the difference in angles calculated earlier, as shown in Figure 4-15.

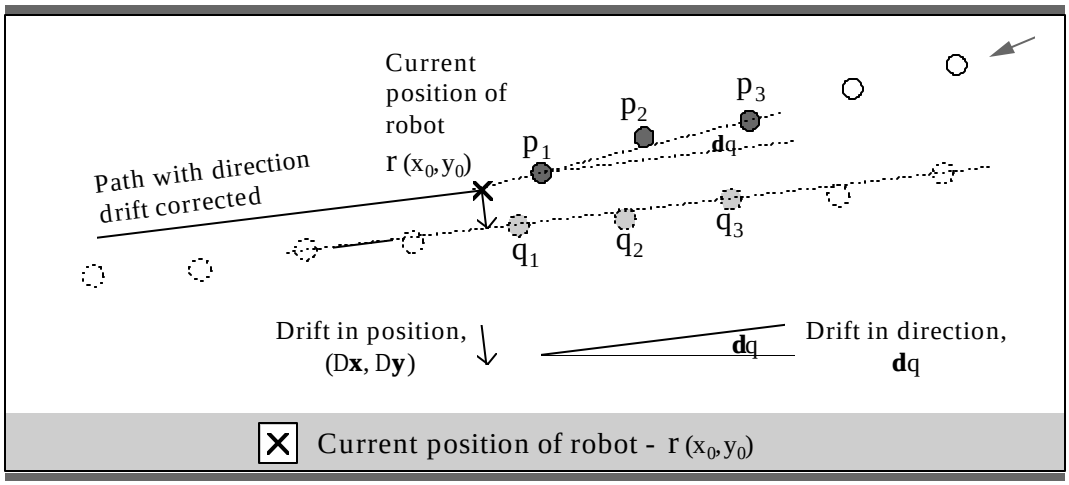


Figure 4-15: Calculating drifts in position and direction estimates

The observable component of the drift in position is found by considering what minimum displacement (Δx, Δy) is necessary to translate the robot's current position onto the recorded location of the edge it is following. The situation is also shown in Figure 4-15. This displacement can be estimated by finding the intersection of the line formed through q_1 and q_3 with the perpendicular dropped from r.

The slope of the line through q_1 and q_3 is :-

$$m = \frac{y_3' - y_1'}{x_3' - x_1'}$$

Hence the equation of that line is:-

$$y - y_1' = m(x - x_1')$$

The perpendicular to this line through the point (x_0, y_0) is:-

$$y - y_0 = -\frac{1}{m}(x - x_0)$$

These lines intersect at :-

$$x = \frac{m^2 x_1 + x_0 - m(y_1 - y_0)}{1 + m^2}$$
$$y = \frac{m^2 y_0 + y_1 - m(x_1 - x_0)}{1 + m^2}$$

Hence the displacement is :-

$$\Delta x = x - x_0 = \frac{m^2 (x_1 - x_0) - m(y_1 - y_0)}{1 + m^2}$$
$$\Delta y = y - y_0 = \frac{-m(x_1 - x_0) + (y_1 - y_0)}{1 + m^2}$$

Δx and Δy are the appropriate corrections to the robot's position estimate, if it is assumed the boundary has not changed since the robot last saw it. In practice this correction is made with one quarter weighting so that the map is changed conservatively, and can be repaired if the correction turns out to be misguided. The correction to the robot's direction estimate was trusted at one half weighting because experimental tests found it to be very reliable. However it was found that any fractional weighting for both position and direction estimates that was not too large produced perfectly acceptable results. This is consistent with the remarks made in Section 4.4.1 about the resilient nature of a landmark system.

As discussed at the start of this section, straight-line landmarks do not allow the full component of the position drift to be calculated, only the component perpendicular to the surface of the edge. The robot could only detect the position drift fully if *two* straight-line landmarks have been passed that are at a 90° angle to which other. For practical purposes, it is reasonable to consider two straight-line landmarks at an angle of 45° or greater to each other to be enough to extract a single fairly accurate "position fix" from the environment[28].

[28] This is the assumption made by the "confusion" virtual sensor to be discussed in Section 4.5.2.

4.4.3 Corner landmarks

Consider the situation where the robot is following an idealised boundary as shown in Figure 4-16. The boundary is straight initially, then turns, then continues straight again in another direction. The robot's movement will reflect the shape of the boundary. By monitoring its motion, the robot can detect corners such as the one in the boundary shown here, and use them as landmarks. This section discusses how this can be accomplished.

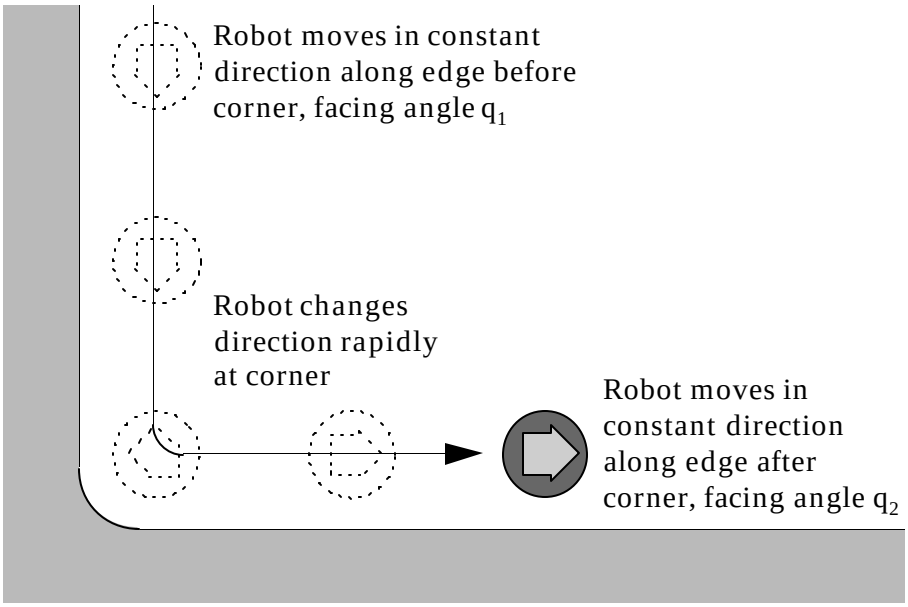


Figure 4-16: Robot moving around a concave corner

It is possible to estimate how sharply an edge is turning simply by measuring how quickly the direction of the robot is changing as it follows that edge. Therefore there is no difficulty in detecting and characterising curves in the boundary. The problem is making use of that information. There is no special point along a curve that the robot can distinguish from all others and use as a landmark. However, the sharper a curve is, the less distance it can extend. A gentle curve can extend a great distance, but a sharp curve must end quickly or the boundary will turn back in on itself. If a curve is sharp enough, then the distance it extends will appear as a point to the robot. More specifically, if the distance a curve extends is close to the granularity at which the robot is mapping its environment (see Section 4.3.2, page 93), then that curve can be used to isolate a point in the environment that can be treated as a landmark.

For the robot used in this project, a "corner" was considered to be any curve that made the robot turn 30° or more in one second of motion. Such a curve appeared as a well-defined point. Note that concave corners are useful, convex ones are not. The reason is that the distance the robot has to travel to turn a convex corner cannot decrease lower than a fixed limit caused by the shape of the robot itself (see Figure 4-17).

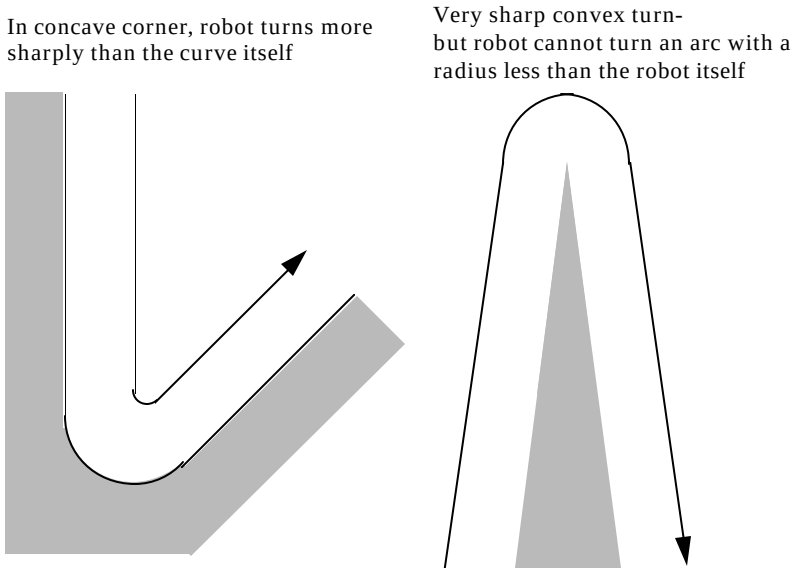


Figure 4-17: Concave versus convex turns

A marker with a special "corner" tag (see Section 4.3.3, page 94) is laid at the location of every concave corner the robot meets. Whenever the robot meets a corner, it first checks if any of the nearby markers were tagged in this way. If there is such a marker, then it is possible that it represents the same corner that the robot is at now - any apparent difference in position being caused by the robot's estimate of its position drifting due to accumulating error[29]. However, pairs of corners often appear in close proximity (such as at the end of a cul-de-sac; see Figure 4-18) so it is dangerous to assume the markers represent the same corner without making further tests.

[29] Or, of course, the corner may have just moved.

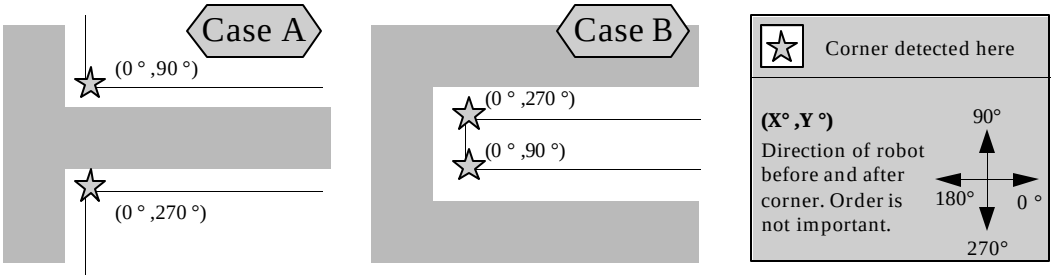


Figure 4-18: Distinguishing close corner landmarks

This difficulty is resolved by storing the direction the robot is moving in before and after the corner as an "annotation" in the marker representing the corner (see Section 4.3.3, page 94). This ensures that the robot could distinguish between the corners in Figure 4-18, for example. In practice, the situation in Case B is the only one in which the robot could potentially confuse corners. In Case A the distance between the corners is twice the radius of the robot plus the width of the boundary, which would make position drift an unlikely explanation for the large difference in corner positions. In Case B there is no such limit on the distance between the corners[30], so confusion may occur.

Of course, corners do not always occur with perfectly straight edges before and after them, so the direction of the robot as it enters and leaves the corner will not generally be well defined. However the directions do not have to be at all accurate to perform the discrimination between nearby corners described above. Case B occurs when two concave turns are met one after the other, and in any such case one of the directions will be approximately the same (the direction lying along the shared edge segment between the corners), and the others will differ by an amount equal to the sum of the angles the corners turned. Each corner must turn a minimum of 30° to be classified as a corner in the first place (see earlier discussion in this section), so the total difference in the directions the corners do *not* share must be at least 60°. In fact in the case shown in the diagram it is 180°. Hence corners such as those in case A will always have one direction that differs by at least 60°. Therefore the directions do not need to be known at all accurately, since there is a tolerance of ±60° allowed. In this project a conservative tolerance of ±15° was chosen, and the direction of the robot a second before and a second after it turned the corner were used as the directions recorded with the corner

[30] Although if they come very close they will appear as one single turn to the robot- it will no longer be able to distinguish them.

landmark. This proved perfectly satisfactory for distinguishing between corner landmarks that would otherwise be confused.

Once a corner has been recognised, it can be used to correct the robot's position estimate. Any difference between the position at which the robot turns the corner and the position of the marker laid at the corner by the robot the last time it turned it may be due to a change in the environment or to a drift in the robot's position estimate. It was decided to give the current and previous corner position equal weighting in calculating the corrected position of the robot, but any conservative ratio produced satisfactory results.

Note that corner landmarks cannot be used directly to correct the direction estimate. The directions of the robot before and after the corner are not known accurately enough for this. However, *pairs* of corners can be used indirectly to correct the direction estimate. Consider the situation shown in Figure 4-19. Here the robot has passed two corners. At the first corner it met, it found a discrepancy between its current position estimate (b_1) and the position at which the corner had been detected the last time the robot passed it (a_1). It used this discrepancy to compute a corrected position, c_1, that was simply the average of a_1 and b_1. At the next marker, it repeated the same process. However it can then be observed that when the robot passed these corners last, they were in the positions a_1 and a_2, but this time round they have appeared to be in locations c_1 and b_2[31]. This indicates a direction drift of $\Delta\theta$ as shown, which can be applied as a correction (weighted conservatively, as always).

[31] b_2 is used in the calculation rather than c_2 because the position correction applied at the second corner actually obscures the direction drift by compensating for some of the position drift it caused as the robot moved from c_1 to b_2. Remember that as the robot moves, any error in its direction estimate is reflected as a growing position drift. To compute the error in the direction estimate, all the position drift should be taken into account.

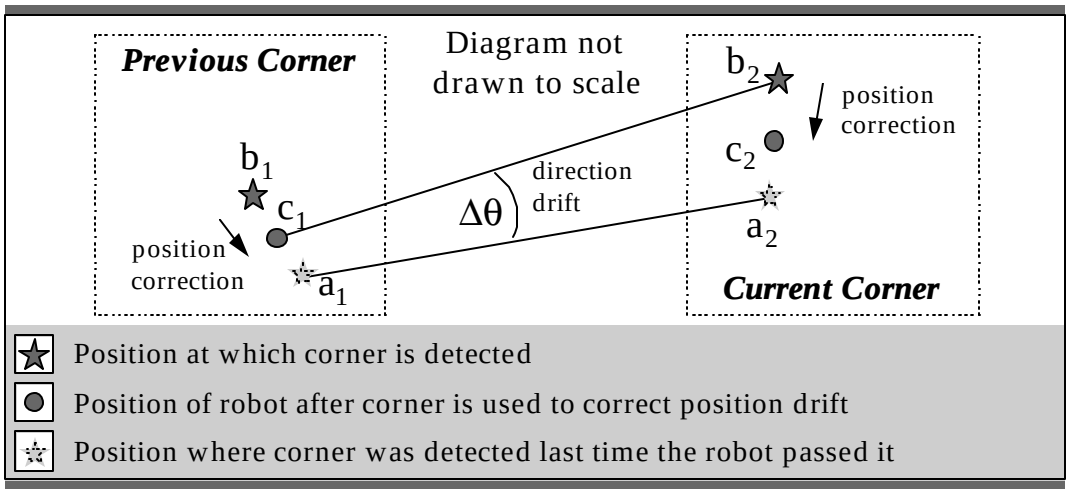


Figure 4-19: Sequential corner landmarks

The discussion of the use of landmarks is now complete. The two types of landmarks developed were found in experiments to work well, and they demonstrate that the marker representation scheme is in fact workable by showing that the robot can keep its position and direction estimates consistent relative to its environment. See Chapter 8 for experimental results that illustrate the operation of the landmark system in practice.

The use of landmarks concludes the examination of how the robot's map is built and maintained. The following sections look at how the map can actually be accessed by behaviours that need to use it.

4.5 Interacting with the map

The chapter so far has concentrated on the issue of building and maintaining a consistent map of the robot's environment. Equally important as building the map is how to go about actually making use of it. This section discusses how the cartographic system can provide useful services to behaviours that wish to interact with the map.

The most important constraint placed on the services that the map may provide and still be "behaviour-based" is that the services must not require a shared representation between the user and the provider of the service (see Section 4.2, page 82). A suitable set of services to meet this constraint and still provide all the necessary functionality described in Section 4.1 is not obvious, and was evolved in a number of iterations of exploratory research. The following

set of services were found satisfactory in providing an interface to the map with little or no shared representation:-

- **Position and Direction Service** - this gives behaviours access to estimates of the robot's position and the direction it is facing relative to its environment.
- **Virtual Sensors** - these provides useful statistics about the state of the robot and its environment that are derived from the map but appear to behaviours in a form that is exactly analogous to physical sensors.
- **Tagging Service** - this allows locations of special interest on the map to be marked so that they can be referred to later, without requiring the user of the service to know anything about how the map is represented.
- **Goal Seeking** - this allows the map to be used to plan an efficient route to a target. Again this is supported without requiring the user of the service to know anything about how the map is represented.

These services are now individually examined in detail.

4.5.1 Position and Direction Service

The most basic function of a map is to keep track of where the robot is and what direction it is facing in relative to its environment. This service makes the cartographic system's own position and direction estimates available to behaviours. These estimates are given as a simple coordinate and angle. No shared representation is required between this service and its user. However there is a shared "understanding" that the estimates quoted are relative to a coordinate system that may drift with time, so that the readings should not be retained and used over an extended period. In particular, if a behaviour wishes to make note of a special location it wishes to return to later, it should not just store the estimated position of that location, since the coordinate system of the map may drift with time and render the position estimate useless. Instead, the Tagging Service described in Section 4.5.3 should be used.

As described in Section 4.3 (page 83), the position and direction estimates are generated by following the movements of the robot and integrating its motion to give its position. The details of this for the particular robot the work in this thesis was implemented on will be discussed in Section 7.4.1, page 217. The robot then uses landmarks to cross-check and correct its

estimates of the robot's location and direction (see Section 4.4, page 95). Although using landmarks necessarily involves considerable complexity, that complexity is entirely transparent to the user of this service. In fact if the use of landmarks was removed and simple motion integration alone was used for estimating the robot's position, a behaviour using this service would be entirely unaffected- except of course that its performance would be degraded because the information it is working with grows less accurate as time passes.

4.5.2 Virtual Sensors

Often behaviours do not need detailed qualitative information from the map, just answers to simple quantitative questions like "how familiar does the current location of the robot seem?" or "how confident is the robot of its position estimate?". These statistics can be provided in the form of "virtual sensors" that appear to behaviours just like physical sensors, but are internally generated. The virtual sensors provided are as follows:-

- **Familiarity**- this sensor indicates whether the robot is currently in a region that the cartographic system recognises, and if so how long has it been since it has last there. This gives the robot a sense for how recently it has passed through a particular area, or if the area is totally unknown to it. Familiarity is generated very simply from a comparison of the current time with the timestamp of the nearest marker to the robot which has not been laid recently, if one is present. This sensor results in a simple number, yet cannot be supported without the full effort of the cartographic system.
- **Confusion**- this sensor measures how uncertain the cartographic system is about the accuracy of its best guess at the robot's position. This uncertainty grows with the length of time the robot is moving after it has last managed to get a fix on its position from a landmark. This gives the robot a sense of how long it has been in motion without getting some fix on its location. Confusion is zeroed when the robot meets a corner landmark, or when two edge landmarks are passed with edges that are at an angle of at least 45° to each other (see Section 4.4.2, page 99).
- **Curvature**- The robot monitors the rate at which it turns and produces a virtual sensor proportional to that rate of change. This can be used to detect when the robot is moving smoothly, and when it is turning sharply.

- **Curiosity**- As the robot moves, it watches out for any nearby areas in which it has never been. Such areas can be detected simply by the absence of any markers there. When the robot notices such an area, it generates a vector pointing in its direction called the "curiosity" virtual sensor.

The estimates generated by the position and direction service could also be seen as a pair of virtual sensors, but they have been treated separately because of their special place in the cartographic system.

4.5.3 Tagging Service

It is useful to have a system whereby certain locations can be marked on the map as "special" (so, for example, goals could be set for the robot to navigate between). Storing such locations separately from the map would require that the map maintain a coordinate system that remains the same over all time, so the same real-world location would always have the same coordinate. In contrast, storing the locations with the map means that the cartographic system only needs to ensure that the overall map remains consistent, without necessarily maintaining an absolute coordinate system over time. This is what results from the approach to the use of landmarks described in Section 4.4 (page 95), so the tagging service was implemented this way[32]. A behaviour can hand the cartographic system a "tag" to assign to the current position of the robot, and from then on that position on the map can be accessed through the tag without having to worry about drifting coordinates. A second advantage is that this avoids shared representation- the behaviour can specify a region it is interested in without knowing anything about how it is represented.

4.5.4 Goal Seeking

It is also useful to have a service that uses the map as a resource to plan an efficient route to a given target location. This is a particularly difficult situation in which to avoid shared representation between the service and the user. First there is the problem of how to set the

[32] This is why the robot's position and direction estimates are specified as being "relative to the rest of the environment". The co-ordinate system of the map is allowed to drift over time, with landmarks being used to keep the co-ordinate system of the robot's position and direction estimates in step with that drift.

target without requiring the behaviour using the service to have access to the map. This is achieved by using the Tagging Service described above. A limitation of this is that targets may only be places the robot has passed through at some point in its history- but this is reasonable since the map would be of no use for planning routes to areas in which the robot has never been[33]. Another problem is how to communicate the route this service generates back to the behaviour that uses it without a shared representation of locations and paths. The solution adopted was to implement a virtual "*scent*" sensor. This sensor was generated in such a way that the robot could reach the target simply by moving in directions of increasing "scent", and moving away from directions in which the scent's intensity decreased. This is a simple reactive strategy, with no shared representation with the cartographic system needed. The "scent" sensor represents the gradient of the cartographic system's estimate of some cost function from the robot's current location to the target, but none of the complexity of calculating that function is visible to the user. This idea works out to be something quite similar to the "Internalised Plans" technique discussed in Section 2.7.3, page 41. The details of how planning is achieved within the marker map representation scheme are now examined.

Markers are stored in a system of neighbourhoods designed to efficiently filter out which markers are close to the current position of the robot, since these are the most relevant to it for most purposes. However, in planning routes to arbitrary targets, information about markers distant from the robot's current position is needed. Specifically, it is important to be able to determine which markers can be reached from each other, and how great a distance the robot has to travel to do so. The use of neighbourhoods only allows the robot to determine which markers are close to the current position of the robot, and it cannot be used to determine which markers are close to some other arbitrary marker.

To solve this problem, extra "connectivity" data is added as an annotation to markers (see Section 4.3.3, page 94). As the robot moves, the neighbourhood system determines which markers are close to its current position. When the robot lays a marker and moves away from it, that information about adjacent markers can be captured and stored in the marker as connectivity data. It is then possible for planning to be done at a later stage, using these "frozen

[33] This is true for autonomous robots, which are entirely responsible for generating their own map of the environment and hence cannot know anything about places they have never been. It would not apply if the robot had a built-in map of some form.

images" of the information calculated by the neighbourhood system. Figure 4-20 shows an example of a collection of markers with connectivity information overlaid that would be suitable for planning routes with.

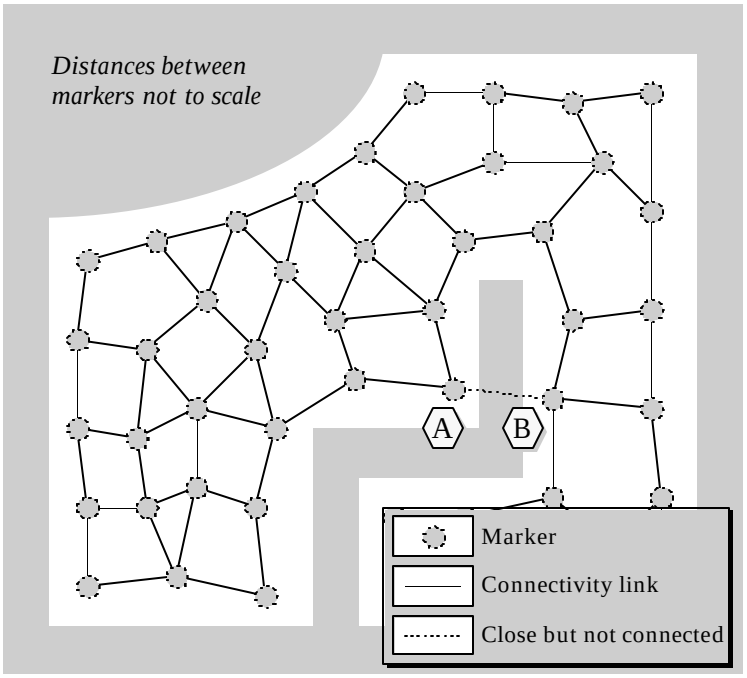


Figure 4-20: Storing connectivity data with markers

Figure 4-20 draws attention to the fact that the robot cannot assume that, just because two markers are close to each other, it is possible to move from one to the other- they could be on opposite sides of a wall, for example, as is the case with markers A and B in the diagram. Care must be taken to detect such conditions. Connectivity links should only lead from one marker to other nearby markers which the robot can move to *without hitting an obstruction*- otherwise they will be misleading and useless for planning routes with. Hence two markers being close to each other is a necessary condition for them to be considered connected, but it is not sufficient.

One sufficient reason for considering two markers to be connected is if they are laid one after the other by the robot (see Figure 4-21). Such markers represent successive points along the path of the robot, so it is reasonable to assume that they can be reached from each other since the robot has actually just done so.

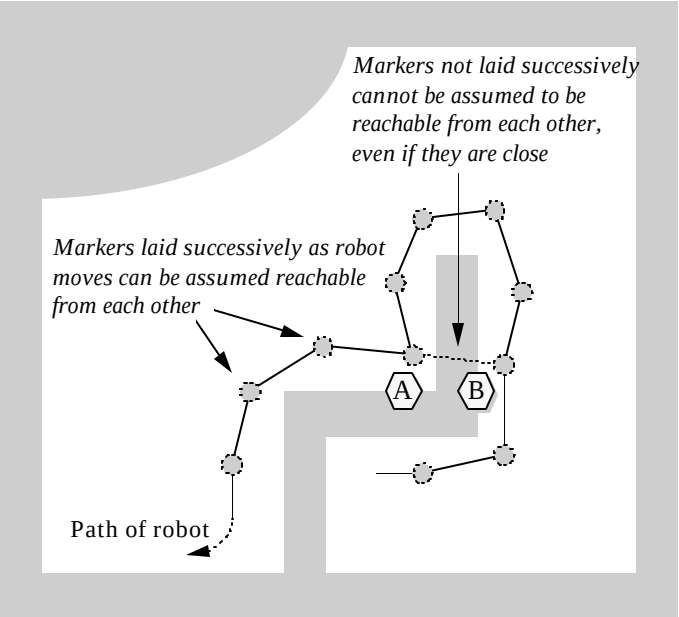


Figure 4-21: Conditions on connectivity of markers

From the connectivity information this gives, the robot can deduce further appropriate connections to make. Two markers that are close to each other but were laid at different times can be deduced as being connected if markers close to one of them are connected to markers close to the other. Figure 4-22 illustrates a common situation where this form of deduction becomes useful. The robot is shown following a path that leads it back into an area where it has already laid markers. When the robot lays marker A, it is quite close to marker B- but they were not laid successively. If the two markers are close enough that the robot can be sure there is no undetected boundary between them, then it can assume they are connected. But the robot has only short-range sensors, so the markers could be quite close and yet the robot cannot determine from its sensors if they are reachable from each other. Assuming this is the case here, the robot will not add a connectivity link between A and B. However, when the robot reaches C and lays a marker there, it finds that there is a marker already present for that area. Since the marker being laid and the marker being replaced represent the same physical area, connectivity data from the replaced marker can be transferred over to the new one (with some caution, as will be discussed in a moment). At that point there is a path from markers A to B through a small number of other markers. This, in combination with the fact that A and B are close, gives the robot enough confidence to mark them as connected (again, subject to some conditions that will be discussed).

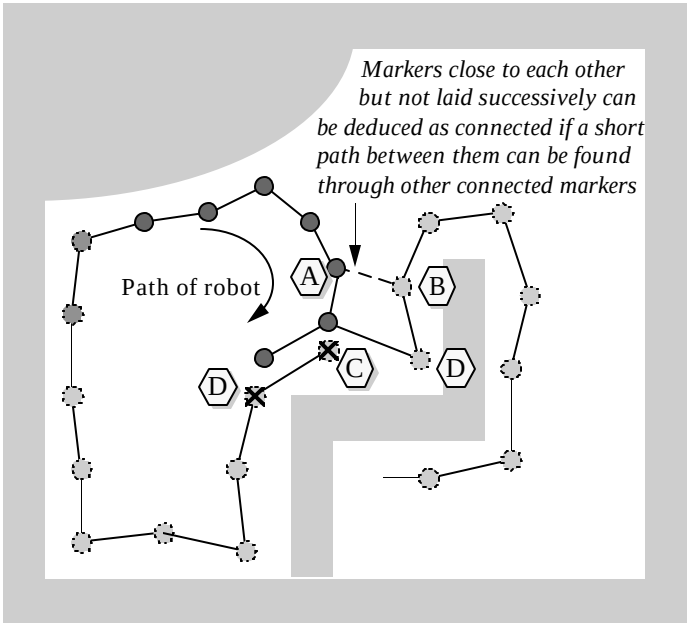


Figure 4-22: Determining connectivity indirectly

Connectivity information is different to other data about the environment stored in markers because, by its very nature, it cannot be derived from purely local considerations. In contrast, information about whether an area is beside a boundary, for example, can be constructed entirely from immediate sensor data. It makes sense for the robot to always discard such data when it is replacing markers, and derive it anew for the current state of the environment. The robot should always trust its sensors over any other source of information it has (as discussed in Section 4.2), so the map should be overwritten with actual sensor data whenever possible. However, when working with connectivity data, information is lost if the old markers are simply discarded because connectivity information cannot be reconstructed completely from immediate sensor data. If the new marker being laid at C in Figure 4-22 had replaced the older marker without copying its connectivity data, the robot would no longer know that the area the markers represent is connected to D. Therefore, connectivity data should be transferred across to new markers from the markers they replace.

If the links are simply copied to the new markers, the system works reasonably well for a while. However the markers the robot lays are not constrained to be in the exact same location as the markers they replace- and in general they will not be. Hence as the robot moves back and forth across the same area, and the links are copied between successive replacements to the original marker, it is quite possible that the position of the marker holding

the links may have drifting quite a distance from the position of the original marker. The links will then give a totally inaccurate picture of the reality.

To step around this problem every marker is given a "Reachable" timestamp. When the robot passes close enough to a marker to be confident that there is no boundary between the robot's current position and the marker, this timestamp is set to the current time. This it taken to indicate that this marker was reachable by the robot at the given time. The timestamp is actively spread to any markers that are linked to it (with some time subtracted to represent roughly how long it would take to get to that marker). This is taken to indicate that those markers could have been reached at the calculated time if the robot had chosen to do so. These timestamps continue to spread from marker to marker. Then, when the robot replaces markers, connectivity can be reconstructed by comparing "Reachability" times with other markers in the locality (see Figure 4-23). If two markers are close in position and both could have been reached within a short time from the robot's current position (as indicated by them having reachability timestamps close to the current time), then those two markers can be considered reachable from each other and have their connectivity links updated appropriately. This is robust, and not subject to marker drift as simply copying connectivity data would be.

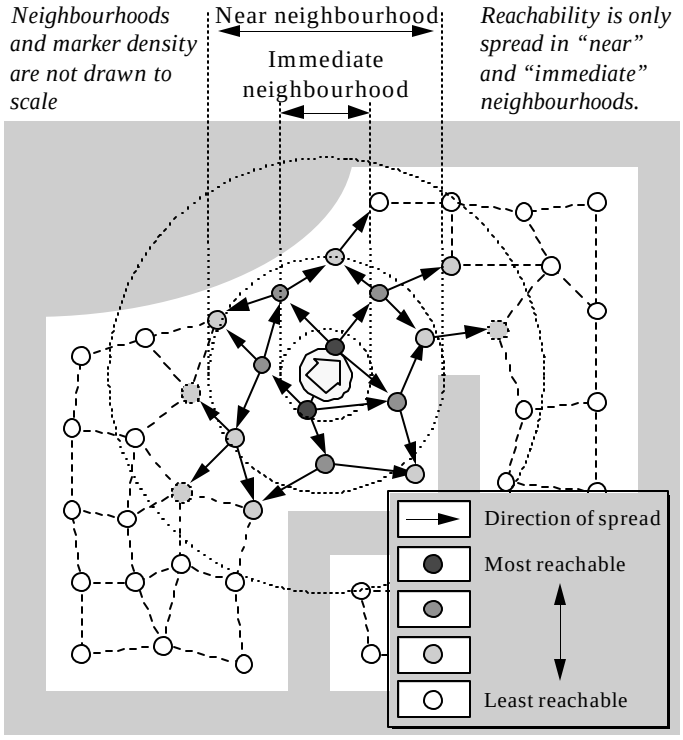


Figure 4-23: The use of reachability

To limit the computational burden on the robot, reachability is only spread within the immediate and near neighbourhood of the robot (see Section 4.3.1, page 88). Any markers outside of these neighbourhoods are not reachable from the robot's current position within a short time, so it is reasonable to eliminate them from consideration anyway.

There is one special case that needs to be catered for to ensure the correct operation of the reachability system. It is possible that reachability could spread to a small extent around a narrow wall as shown in Figure 4-24.

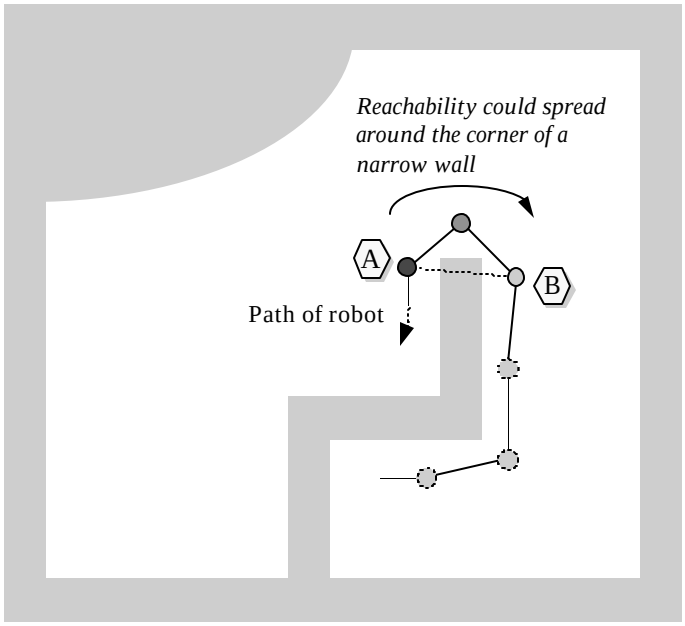


Figure 4-24: Connectivity spreading around a narrow wall

This is not an error, since it is true that markers A and B can be reached easily from each other, so connecting them would not mislead the robot if it was planning a route- it would not make the robot choose a path leading to a dead-end. This is all the robot needs from the map- its own basic competences will let it navigate minor obstacles. However, it is simple to introduce a heuristic to stop connectivity creeping around a wall, by simply disallowing links to be made between close markers that are both on a boundary, with the boundaries facing in opposite directions away from each other- i.e. on opposite sides of a narrow wall.

There is one final practical consideration that is useful for storing connectivity information efficiently. It has been shown that if the robot is aware that two markers are linked, it may deduce that other markers are reachable from each other by spreading reachability. Hence not every link between markers needs be stored, only a sufficient number to allow reachability

spreading to deduce the rest. Due to the memory limitations of the robot this work was implemented on, the number of links from each marker was limited to four. By applying criteria that favoured storing links to markers lying in different directions over links to markers clustered in the same direction, four links were found to be more than adequate for correct functioning of the reachability system (i.e. by spreading reachability, the robot successfully avoided losing connectivity information when it replaced old markers with fresh ones). The first diagram in this section, Figure 4-20 on page 117, in fact showed connectivity data constructed with a maximum of four links from each marker.

Given that connectivity links are being maintained, goal seeking is straightforward to achieve by any standard search technique. One simple way it can be done is to assign a "hop count" to every marker to represent how many other markers the robot would need to pass through when going from that marker to the target. Obviously the hop count of the target is zero. Any markers linked to the target will take on a hop count of one, and markers linked to them in turn will take a hop count of two, etc. In general, a marker M determines its hop count by finding which of its links leads to the marker with the lowest hop count. It should assign itself a hop count of one greater than that, and tag the link as shown in Figure 4-25.

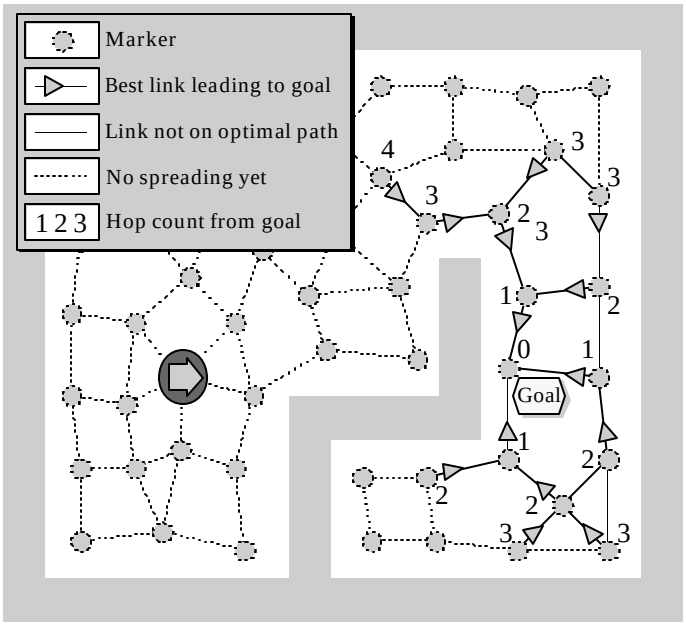


Figure 4-25: Goal Spreading in action

Once this process "spreads" out to markers in the vicinity of the robot, it can simply follow all the tagged links to the target. This is called "Goal Spreading". It can be improved by using cumulative distances between markers on the route to the goal instead of simple hop counts, or a combination of both (as was used for the robot this work was implemented on). The computation required for Goal Spreading is extensive and takes time, although the work done for each marker is quite simple. The calculations should be performed as a background task so that they do not affect the real-time performance of the robot.

There is one final improvement that can be made to Goal Seeking. If *multiple goals* were set for the robot, everything described so far would still work- Goal Spreading would simply spread paths out from each goal, and whichever reached the robot first would be the one it would move towards. This is suitable for a situation where a number of goals are equally acceptable to the robot. It is simple to extend Goal Seeking so that it can also handle cases where some goals are more desirable than others. This is done by associating a "desirability" factor with the goals when they are set, and spreading that factor to markers that point along routes to that goal. In other words, when a marker scans the markers it is linked to for the one with the lowest hop-count/cost to a goal, it should only consider those with the highest "desirability" present, and then accept that desirability level for itself. Hence as paths to more desirable goals propagate, they can "take over" markers that were leading to less desirable goals- even if those goals were closer.

The results of goal spreading are made available through the "scent" virtual sensor, which simply gives a vector corresponding to the direction of the tagged link of the marker nearest the robot's position, if goal spreading has reached that link. The image is of a scent released at the goal spreading outwards until the robot picks it up and follows it to its source. This involves no shared representation between the service and its user. Another advantage of the use of this sensor rather than returning an explicit optimal path is that if the robot wanders off course, the path does not have to be recalculated.

4.6 Summary

In this chapter, a complete cartographic system has been developed that is capable of constructing and maintaining a map of the robot's environment in real-time, and with the use of

short-range proximity sensors only. The representation scheme used was based on units called "markers". The map consisted of a collection of these markers- records of the robot's experiences at particular locations- rather than being an explicit model of its environment. Strategies for maintaining this collection of markers in a state that reflected the condition of the environment were discussed in detail. Then the issue of landmark recognition was addressed, showing that the robot could maintain an estimate of its position that remained consistent relative to its environment over time. This is the ultimate test of a cartographic system. Finally, a set of services were presented that allowed the map to be used without requiring any knowledge of how it is represented.

The following chapter relies heavily on the work presented here. In it, a set of behaviours implementing a robot "sentry" are developed. These behaviours use the services provided by the cartographic system to perform many of the activities discussed in Section 4.1- checking and escaping from behaviour cycles, backing out of dead-ends, planning routes to targets, patrolling and exploring the environment, etc.

5. Sentry Behaviours

This chapter develops a set of behaviours for a robot which together allow it to act as a "sentry", exploring and patrolling its environment autonomously. The overall sentry-like behaviour of the robot is produced by combining a set of lower-level behaviours performing simpler tasks, which are in turn combinations of still simpler behaviours. The ability to combine behaviours in this way is supported by the Lateral robot architecture, developed in Chapter 3, and is the major advantage of this architecture.

For convenience, the behaviours in this chapter are grouped into three broad categories- "motion" behaviours, "informed" behaviours, and "user interfacing" behaviours. The chapter begins by giving an outline of the role each of these groups play in the overall behaviour of the robot. Each of the groups is then examined in turn. The functionality of each behaviour is described both in terms of its individual actions and its influence on other behaviours.

5.1 Overview

The basic "sentry-like" action of the robot is implemented within a set of five behaviours called the "informed behaviours". These behaviours are called "informed" because cartographic information is vital to their successful operation. One of these behaviours, the *map maintaining* behaviour, is responsible for actually generating and updating the robot's map. The others- *prowling*, *patrolling*, *exploring*, and *location seeking* behaviours- make use of the map to implement various algorithms necessary to perform sentry duty.

A lower-level set of behaviours called the "motion behaviours" are concerned with direct control of the robot's movement. These are the *edge following*, *turning*, *nudging*, and *motor control* behaviours. They control the movement of the robot in a more immediate and "reactive" way than the higher-level behaviours that use them, and have no need of cartographic information.

Finally there is a set of high-level "user interaction behaviours" that allow the robot to be controlled by external commands, rather than operating autonomously. Some of these

behaviours use cartographic information, some do not[34]. Behaviours under this classification are *proxy control*, *manual control*, *reporting*, and *region seeking*. These are the highest-level behaviours, since they control which of the other behaviours are allowed to act.

The full suite of behaviours that will be described in this chapter is shown in Figure 5-1. The diagram illustrates that the decomposition is certainly not strictly layered as would be required under Subsumption.

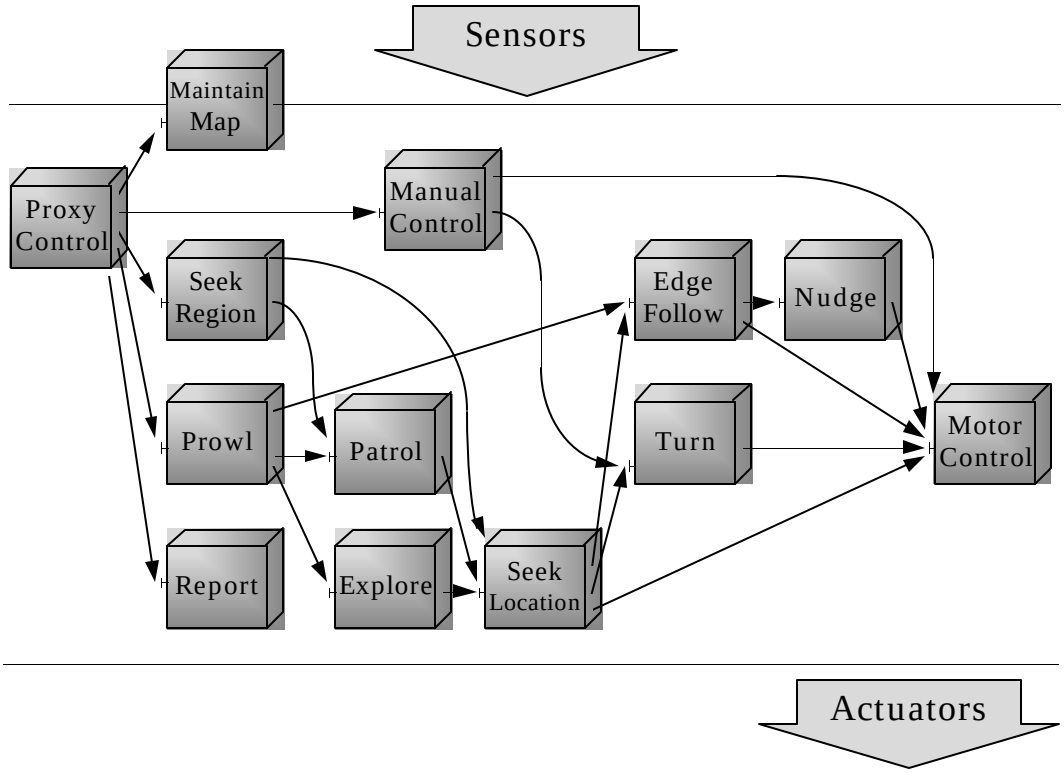


Figure 5-1: The complete set of behaviours implemented

Roughly speaking, the highest level behaviours (to the left) are for user interaction, the lowest level (to the right) are motion behaviours, and the informed behaviours lie in between the extremes. Each of the groups of behaviours will now be examined in turn, starting with the low-level motion behaviours.

[34] The grouping of behaviours adopted in this chapter is for convenience only, and there is some overlap between the categories.

5.2 Motion Behaviours

This set of behaviours dictate the robot's movement without any reference to the cartographic system, taking only the state of the robot's immediate environment into consideration. The place of these "motion behaviours" in relation to other behaviours is shown in Figure 5-2.

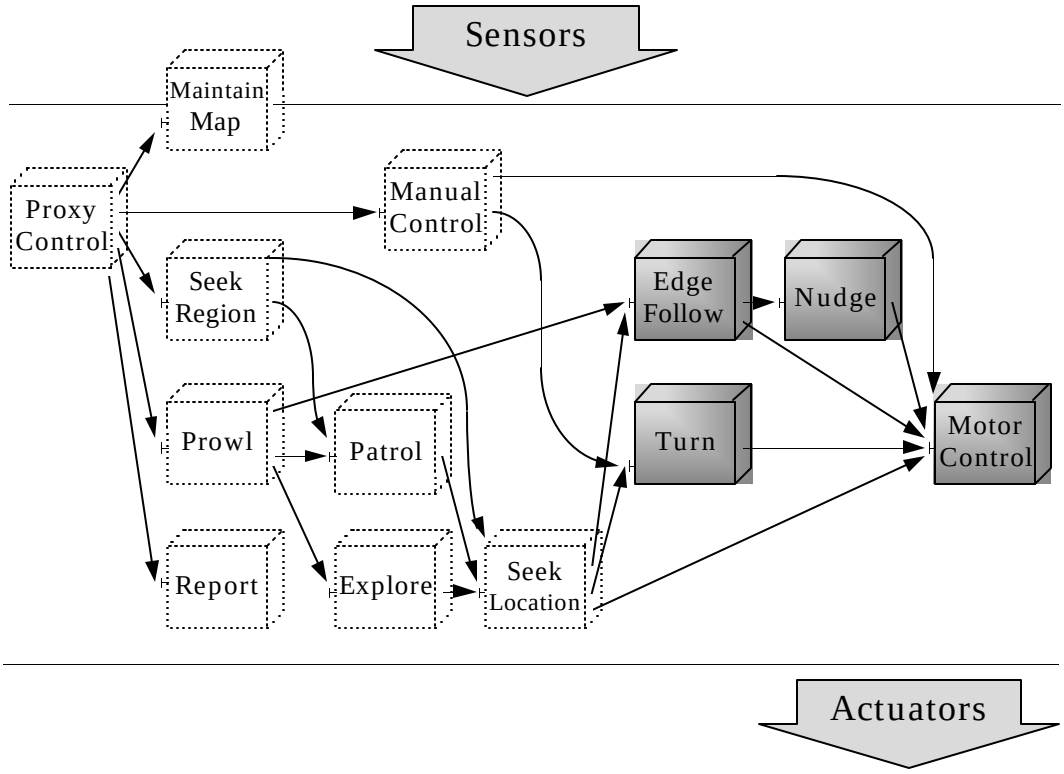


Figure 5-2: Motion behaviours

All commands to the robot's motors are channelled through the *motor control* behaviour.

The *turning* behaviour and the *nudging* behaviour are both concerned with changing the direction of the robot- the turning behaviour makes the robot face in a specified direction, while the nudging behaviour allows it to turn very smoothly. Both of these behaviours are quite simple. The *edge following* behaviour, on the other hand, is relatively sophisticated. This behaviour is concerned with tracing around the boundary of an obstacle, and since this activity is fundamentally important to detecting landmarks (see Section 4.4, page 95), it is carefully crafted to operate as smoothly as possible. These behaviours will now be described individually.

5.2.1 Motor Control Behaviour

This behaviour is the channel through which all commands to the robot's motors are sent. It interfaces directly with the robot's kernel to control the motor[35]. It has a single input, carrying a setpoint for the robot's motors. This setpoint is expressed in two components, "speed" and "nudge" (see Section 7.5.2, page 225 for a discussion of why this is useful). "Speed" controls the rate of the forward motion of the robot, while "nudge" controls the rate at which the robot turns. For each of the behaviours in this chapter, a state machine will be given, but for this behaviour, the state machine is entirely trivial (see Figure 5-3). There is no need for any state information- the behaviour simply passes its input on to the robot kernel.

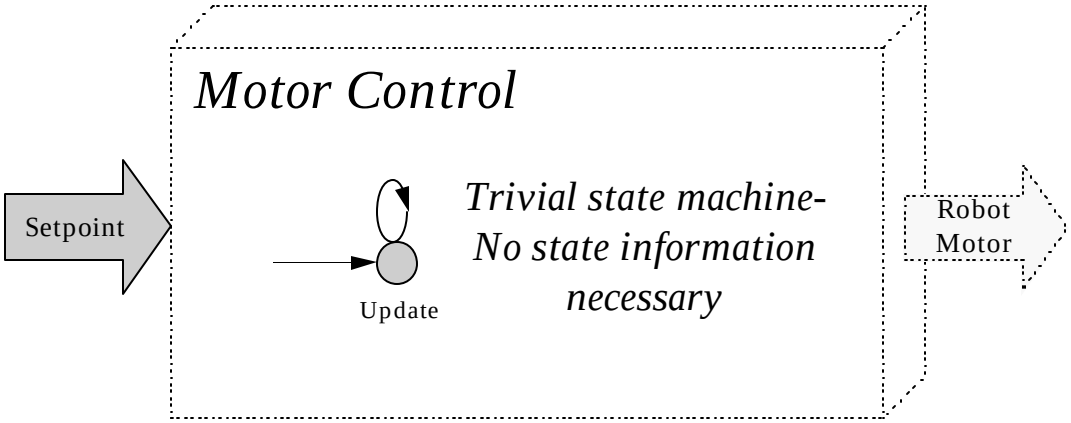


Figure 5-3: State machine for motor control behaviour

The reason that this behaviour is used rather than sending commands directly to the motors is that it prevents different behaviours from sending conflicting commands. By sending the commands to the inputs of a behaviour, Lateral's priority system will be invoked to resolve any conflicts that might occur. There are extra benefits as well. The motor control behaviour can detect when there is no behaviour controlling its inputs, and bring the robot to a stop. If commands were sent directly to the robot, there would be no way to detect when nothing is controlling the robot's speed, and the robot would simply continue to move at whatever rate it was last commanded to move at.

This behaviour also contributes to implementing the "confusion" virtual sensor (see Section 4.5.2). Whenever the robot is in motion, the behaviour increases the value of this sensor

[35] See Section 7.3, page 213. As will be discussed in this section, the robot kernel has "common sense" built in to it to prevent the robot from moving in a direction that collides with an obstacle, so there is no need to implement obstacle avoidance again within this behaviour.

slightly. Hence the longer the robot is in motion, the higher the robot's "confusion" or uncertainty of its position becomes. Confusion is reset to zero by the cartographic system when it detects suitable landmarks (Section 4.4, page 95). The rate at which confusion increases is chosen to suit the rate at which accumulating error builds up in the robot's position and direction estimates.

5.2.2 Nudging Behaviour

For the robot on which these behaviours were implemented, motor speed could only be set in discrete increments. The result was found to be too jerky for the edge following behaviour (to be discussed next), so this simple but useful behaviour was introduced to allow the robot to turn more smoothly. It does this by duty cycling the motor speeds between two setpoints for a variable "mark-space" time ratio. This allows fine tuning of the rate at which the robot turns, which leads to much better behaviour when edge following.

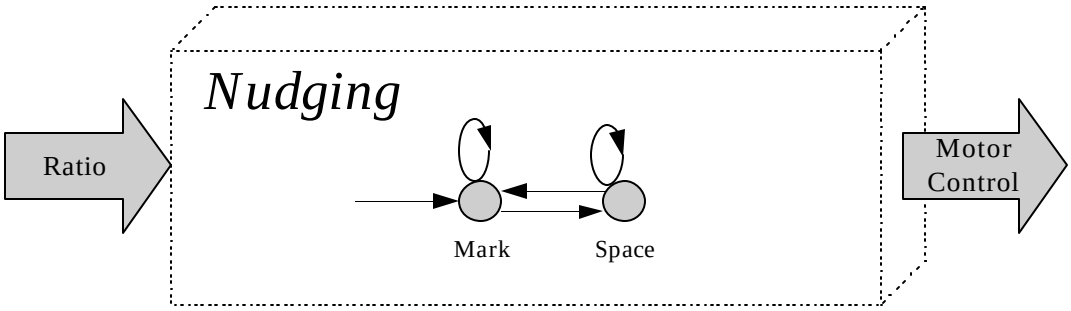


Figure 5-4: State machine for nudging behaviour

The input to this behaviour is a desired mark-space ratio (see Figure 5-4). The output goes to the motor control behaviour. This behaviour passes whatever sponsorship it receives from its input on to its output unchanged, since it is essentially just converting a motion command from one form to another- therefore the priority of the commands it issues should be the same as the priority of the commands it receives.

5.2.3 Edge Following Behaviour

This behaviour makes the robot move around the boundary of an obstacle, turning as the boundary's edge turns. It is important that this behaviour is very robust, and that the robot follows the shape of the edge accurately, because the robot tracks its motion while edge

following to indirectly determine the shape of the boundary it is moving beside (see Section 4.4, page 95). Detecting landmarks therefor depends on a well-behaved edge following behaviour.

The basic strategy for following an edge with proximity sensor information is straightforward. For example, if the robot is following an edge on its left hand side it need simply do the following:-

1. Move forward if there is nothing directly ahead.
1. If the edge is greater than a desired distance away, veer towards the edge somewhat[36].
1. If the edge is less than a desired distance away, veer away from the edge somewhat.
1. Repeat indefinitely.

If the edge does not turn too sharply, this simple algorithm works satisfactorily, as illustrated in Figure 5-5. This kind of motion is labelled "waddling" for the purposes of this section.

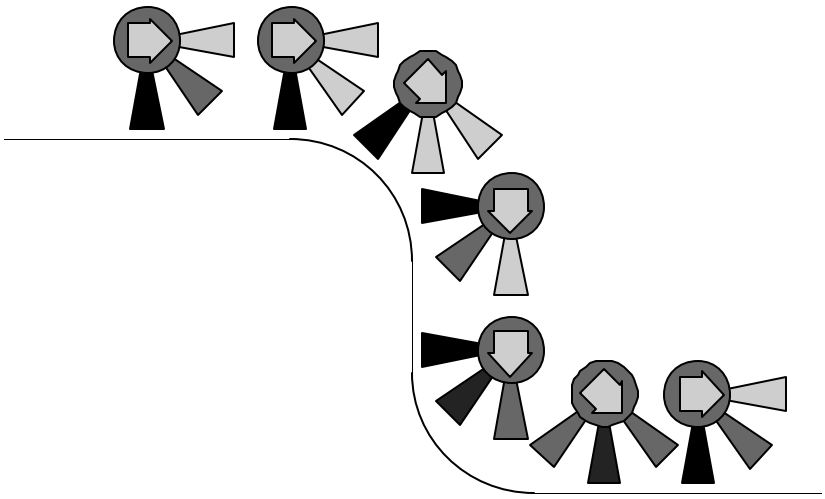


Figure 5-5: Following a reasonably smooth edge

If, however, the edge turns sharply at a convex corner, the robot may not veer fast enough to react to the change, and could lose sensor contact entirely with the boundary. As will be seen further on, the readings from the robot's proximity sensors dwindle very rapidly as it moves away from a boundary, so it is very easy for the robot to lose contact with the edge. It is important that it can re-establish contact with the edge gracefully. The simple algorithm above may work as it is- but it might lead to the robot losing the edge entirely if it veers too slowly, or

[36] The robot in fact has a desired "proximity reading" rather than actual distance which it seeks to maintain. This was discussed in Section 4.4 (page 95), and will be clarified further on in the discussion of this behaviour.

the robot might spin around backwards if it veers too quickly. A specially adapted strategy called "capturing" is implemented for this situation. Here the robot veers rapidly initially in the hopes of re-establishing contact immediately, but if that does not succeed it recovers itself and executes a graceful sweep in search of the edge.

If the edge turns sharply at a *concave* corner, on the other hand, there is no danger of the robot losing contact with the edge. This is because the edge actually starts obstructing the robot's path- rather than diverging away from it, as it did at a convex corner (see Figure 5-6). Hence the simple algorithm described earlier will work. However it may be quite jerky since the robot will be continually trying to move forward at any chance it gets, and being continually frustrated until it veers enough for the edge to no longer be obstructing its way forward. Another specially adapted strategy, labelled "turning", is implemented for this situation. This simply stops the robot from attempting to move forward, turns until the way forward is clear, then reverts to the robot's original behaviour. The desired action of the robot under capturing and turning is shown in Figure 5-6.

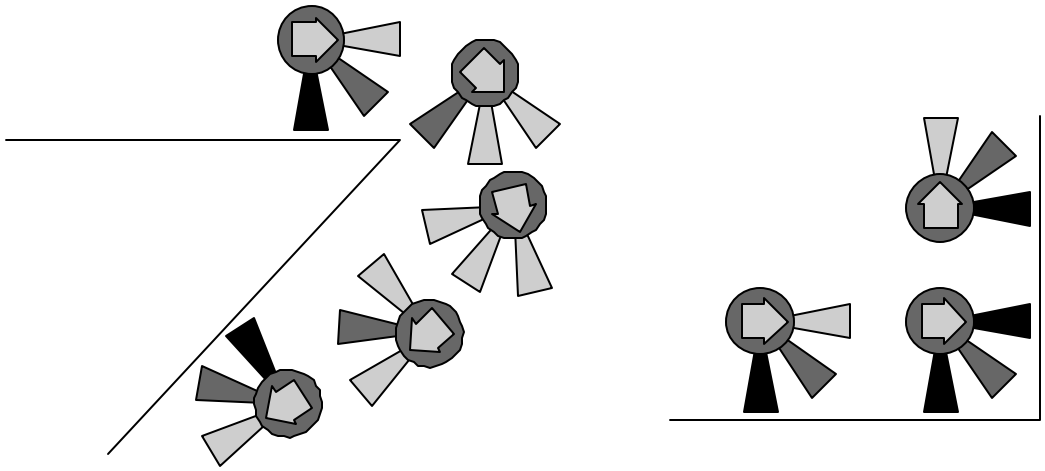


Figure 5-6: Moving around sharp corners

An appropriate state machine for edge following is given in Figure 5-7.

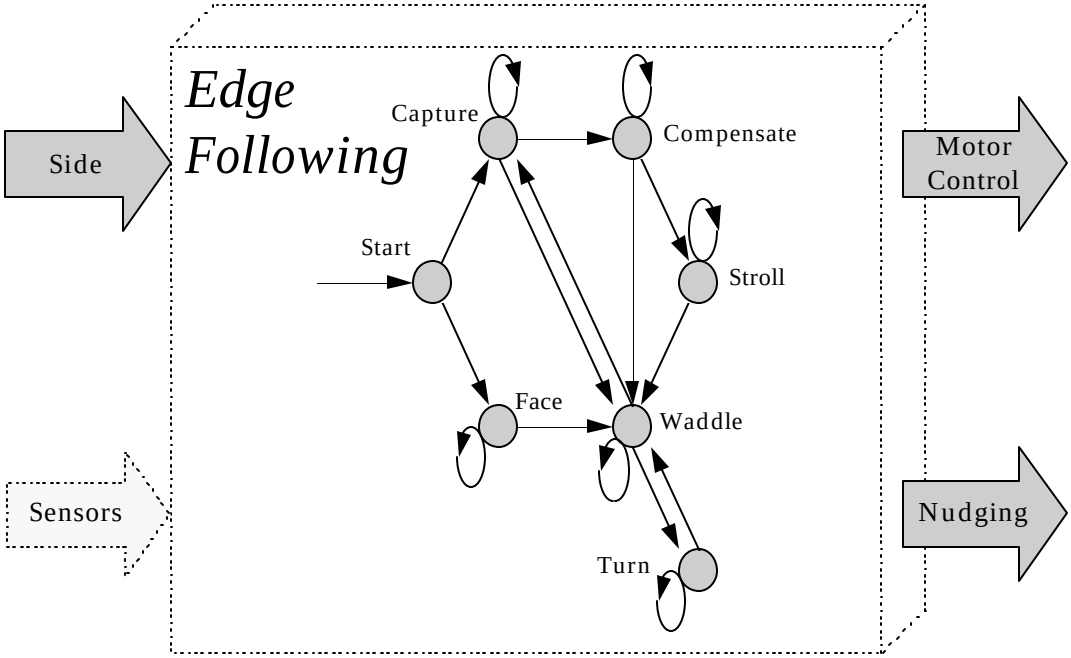


Figure 5-7: State machine for the edge following behaviour

The behaviour takes a single input to determine whether the robot should follow boundaries with its left or its right side facing them. It has two outputs, one to the motor control behaviour, and one to the nudging behaviour. It passes on its sponsorship to the motor control behaviour at all times, to keep the robot moving forward or turning as appropriate. When following a smooth edge, it passes on full sponsorship to the nudging behaviour to turn the robot more finely than is possible directly through the motor control behaviour.

The states in Figure 5-7 have the significance shown in Table 5-1.

Table 5-1

State	***Action***	***Sponsorship to..***
Start	The edge following behaviour is initialised	None
Face	The robot gracefully turns until it is at right angles to the edge before starting to follow it	Motor Control
Waddle	The robot follows a smooth edge	Motor Control Nudging
Turn	The robot turns in a sharply concave corner	Motor Control
Capture	The robot tries to find an edge after it has lost sensor contact with it	Motor Control
Compensate	The robot abandons trying to find an edge, and turns back to the direction it was travelling before it lost it	Motor Control
Stroll	The robot moves in a straight line, looking for an edge to follow	Motor Control

The design so far would be applicable to any robot with proximity sensors. However, for very smooth edge following, it is necessary to take the exact sensing capabilities of the robot into consideration, at a greater level of detail than proves necessary for any other behaviour.

The proximity sensors of the robot used to implement this work have a non-linear relationship with distance, as illustrated in Figure 5-8. Note that the "proximity" readings depend on the colour and texture of the object being detected as well as its distance from the robot.

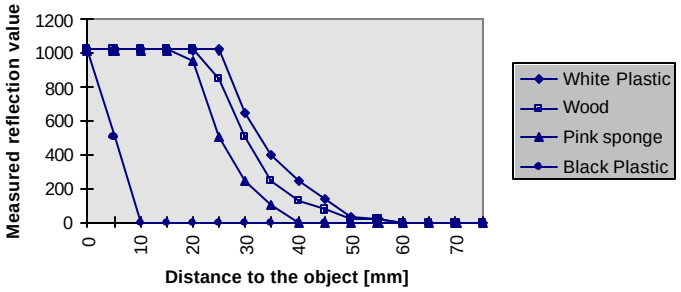


Figure 5-8: Distance to object versus proximity sensor reading (from Khepera User Manual [[24]])

For smooth, accurate edge following, it is important to operate within the sloping section of the graph shown. This is because the characteristic is flat if the robot is either too close or too far

from the edge. In those sections, therefore, the robot receives no feedback on how the distance to the edge is changing- so the distance from the edge could change by a large amount before the robot could detect that and attempt to compensate. Within the sloping section, the robot can detect very small changes of the distance to the edge and compensate immediately.

The relationship between the proximity sensor reading and the actual distance to an edge varies widely depending on the object being sensed. Black objects are essentially invisible - since the robot has to almost collide with them before it can detect them. For lighter shaded objects, once the proximity sensor value falls off from its maximum of about 1000, the robot is guaranteed to be at least 20mm from the edge. The reading falls off rapidly after that. For edge following, a reasonable setpoint for the proximity reading to be maintained at is around 400 to 500. For most objects the robot can detect, this will keep the robot at a conservative distance while still being well inside the sloping section of the characteristic. 425 was the figure chosen for the actual implementation, but anything from 400 to 500 did indeed prove perfectly satisfactory.

Another consideration is how quickly proximity readings fall off as the angle between the sensor and the object being sensed increases. This relationship is shown in Figure 5-9. After 45° the sensor reading has approximately halved. After 90°, the reading has fallen to zero.

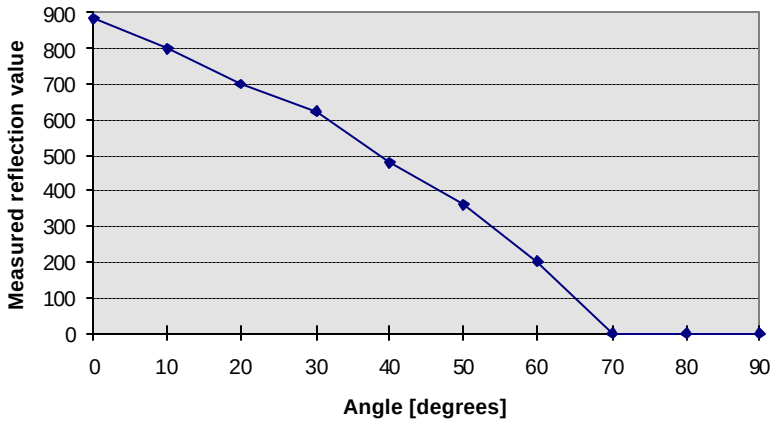


Figure 5-9: Response of a proximity sensor to an object at different angles (from Khepera User Manual [[24]])

This information will now be used to determine how quickly the robot should veer as the distance to the boundary varies, such that the proximity reading of the sensor facing the edge will remain close to a setpoint value $d_{setpoint}$ of, for example, 425 (the value suggested earlier).

The sensors of the robot on which this work was implemented are arranged about its circumference as shown in Figure 5-10. When following an edge, three of these sensors are particularly useful- the one facing the edge directly, here called the "SideSense", the one facing at a 45° angle to that, called the "DiagonalSense", and the one facing forward on the same side of the robot, called the "ForwardSense".

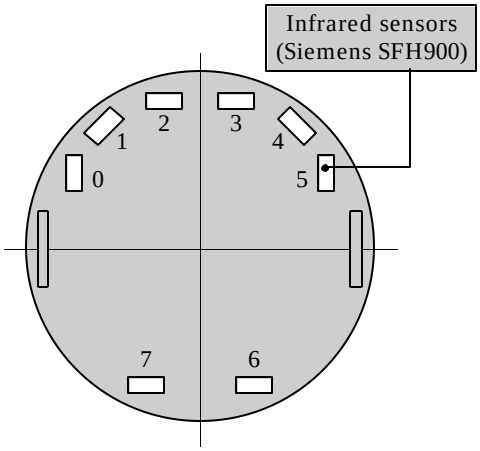


Figure 5-10: Arrangement of sensors

If the robot is moving parallel to a straight edge at the desired proximity level of $d_{setpoint}$, then *SideSense* will have a value equal to $d_{setpoint}$ itself, *DiagonalSense* would have a value of approximately half that (since it is at 45° to the boundary- see Figure 5-9), and *ForwardSense* would have a value of approximately zero (since it is at 90° to the boundary). If these values change, then the robot is not parallel to the edge at the desired distance. The robot's task is therefore to try to maintain these values by veering towards or away from the edge.

- If *SideSense* is greater than $d_{setpoint}$, then the robot is currently too close to the edge and should veer outwards.
- If *DiagonalSense* is greater than $\frac{d_{setpoint}}{2}$, the robot is heading too close to the edge and should veer outwards. This sensor gives an indication of what the robot's position *will shortly be* rather than what it is now. This reading is more important than *SideSense*. For example if *SideSense* shows that the robot is currently slightly too close to the edge, but DiagonalSense shows that the robot will soon be too far from it, the robot should start to turn inwards in preparation, rather than veering outwards and making the situation worse.

- If *ForwardSense* is significantly above zero, then the robot is heading straight for the edge (or some other obstruction) and it should start turning outwards to avoid it. This takes precedence over either *SideSense* and *DiagonalSense.*

It is quite possible that the sensors might not be consistent in which direction they suggest the robot should veer, and that is why it is necessary to know the relative importance assigned to each as described above. The simplest way to implement this hierarchy of importance is to generate a weighted sum of the three sensor readings, with the weights for each sensor being chosen to reflect its relative significance:-

$$w_1 SideSense + w_2 DiagonalSense + w_3 ForwardSense$$

Then, the difference between this and its ideal value of

$$w_1\left(d_{setpo\,\mathrm{int}}\right) + w_2\left(\frac{d_{setpo\,\mathrm{int}}}{2}\right) + w_3(0)$$

gives a measure of how much the robot should veer. A good choice of weights was found to be w_1=1, w_2=2, and w_3=4. This reflects the fact that the *DiagonalSense* reading is more significant than the *SideSense* reading, and the *ForwardSense* reading is more significant than either of them.

This metric was developed by assuming that the edge the robot was following was perfectly straight, and that the robot was facing in the wrong direction and trying to correct that. However, if the edge is *not* straight, any turn in the edge can be compensated for just as if it was the robot which had turned instead. Hence the above metric is in fact suitable to drive the motor of the robot while it is following a curved edge, although some minor modifications are necessary:-

- The units must be scaled appropriately
- It is a good idea to apply a non-linear function to the metric, so that the robot veers more gently when close to the setpoint and more sharply when far from it. The simple linear metric given would lead to a "wobble" around the setpoint if implemented as it stands, due to overshoot in the robot's motors.

A good metric for the robot used in this project was :-

$$metric = \left[\frac{SideSense + 2DiagonalSense + 4ForwardSense - 850}{200}\right]^3$$

In the edge following behaviour, this metric is used to control the robot's direction through the nudging behaviour (see Section 5.2.2, page 129). The nudging behaviour allows finer control over the rate at which the robot turns than would be possible if the motor was controlled directly.

By simply moving continuously forward and veering according to this metric, the robot will follow a boundary smoothly. While the discussion so far has assumed that the boundary is continuous, the robot will in fact be able to follow boundaries with small discontinuities. This is because, as seen earlier in Figure 5-9, the proximity sensors respond to any objects in the general direction they point, not just to objects directly in that direction- so they will simply be unable to detect small features of the boundary, and will instead return an average distance to the boundary. For edge following, this is useful because it makes the above algorithm more robust.

5.2.4 Turning Behaviour

This behaviour turns the robot to face in a given direction. This is a very simple behaviour - it merely compares the direction setpoint it is given with the robot's estimate of the direction it is facing in currently, and turns to decrease the difference between the two. A certain amount of sophistication is given to the behaviour so that it turns "gracefully", at a rate that slows as the difference between the actual direction and the direction setpoint decreases, so that the robot will not overshoot.

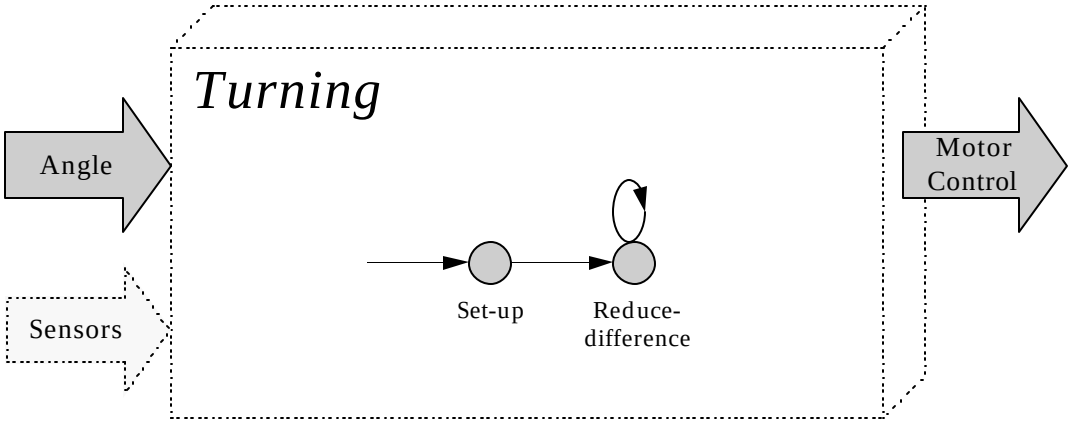


Figure 5-11: State machine for turning behaviour

This behaviour has a single input determining the angle the robot should face, and a single output going to the motor control behaviour (see Figure 5-11). It passes on any sponsorship it receives.

This is the last of the low-level motion behaviours. The higher-level "informed behaviours" are now examined.

5.3 Informed Behaviours

This collection of behaviours consists of the behaviours required to allow the robot to act as an autonomous sentry. Cartographic information is vital to the successful operation of these behaviours. The robot's map is generated and maintained by the *map maintaining* behaviour, and is used by the other behaviours in this group- *prowling*, *patrolling*, *exploring*, and *location seeking*. These behaviours are shown in relation to all other behaviours in Figure 5-12.

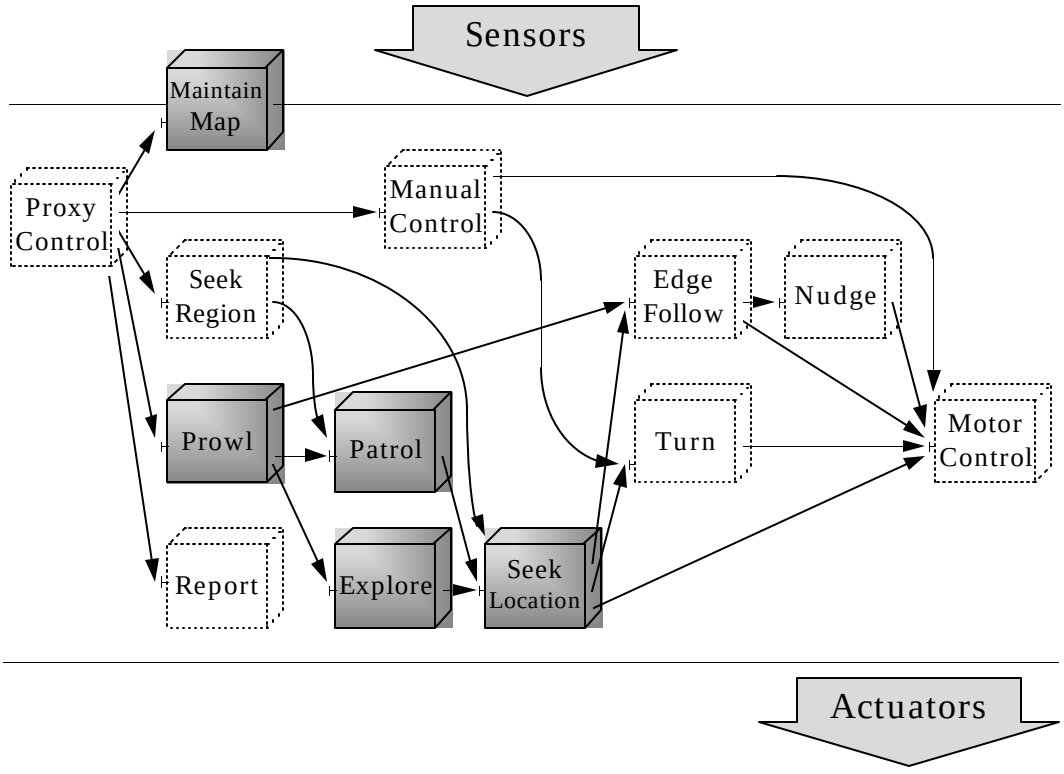


Figure 5-12: Informed behaviours

The basic "sentry-like" action of the robot is orchestrated by the *prowling* behaviour, the highest level autonomous behaviour of the robot. Prowling combines the actions of the other

lower-level behaviours to meet a number of simultaneous goals necessary for good sentry behaviour. These behaviours will now be described individually.

5.3.1 Location Seeking Behaviour

In the behaviours that follow, the robot will often need to move towards a given marker. It is convenient to control the robot's motion using a behaviour which can be fed the co-ordinates of such a marker, and which will do everything necessary to get as close as possible to that target- navigating around obstacles, backtracking out of dead-ends, etc. Since the robot's estimate of its position may have drifted since the co-ordinates of the target marker were set, it can only hope to approach the approximate locality of the marker, but, especially over short distances, this is perfectly acceptable. None of the behaviours that use location seeking will rely on it being entirely accurate.

The behaviour starts by moving directly towards its target. If it strikes a boundary, it will start following that boundary in whichever direction seems "best"- whichever direction seems to require the least deviation from the robot's current path, at least in that locality (since that is the only area it can sense or evaluate from the map). It will continue to follow the boundary until conditions become suitable for it to resume its path towards the target. This will occur if the boundary turns sufficiently to no longer be an obstruction. With this simple strategy, the robot is able to negotiate many obstacles. However, it may lead to cyclic behaviour for obstacles with certain shapes, such as the one in Figure 5-13.

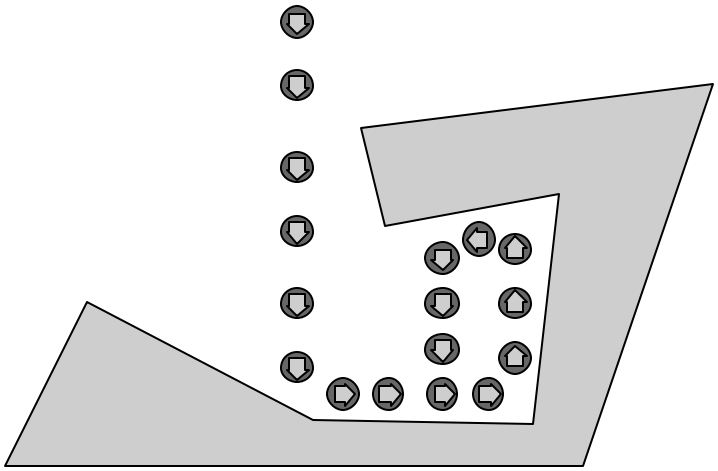


Figure 5-13: Obstacle makes robot loop back on its path

A loop can occur if the robot meets an obstacle, follows its boundary until it again becomes possible to move in the target direction, and then runs into the same obstacle, hence entering a cycle. One simple solution to this, using the "familiarity" virtual sensor generated by the cartographic system (see Section 4.5.2, page 114), is to make the robot alternate in the direction it chooses to move in after it hits a boundary at a familiar location[37], as shown in Figure 5-14.

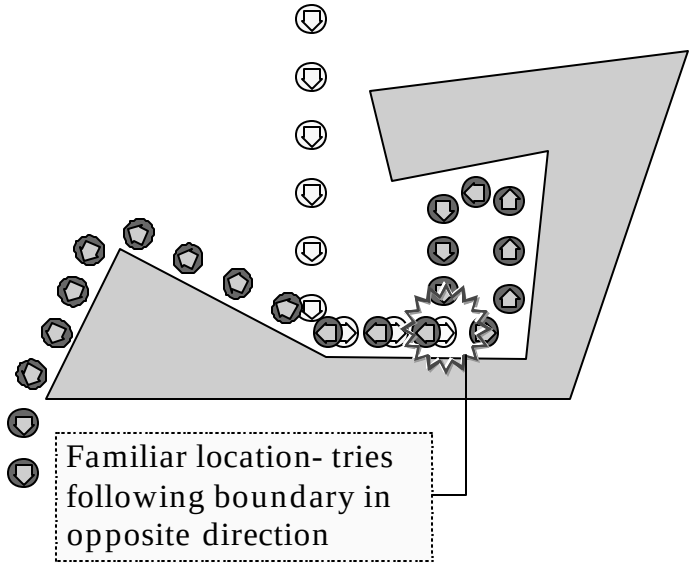


Figure 5-14: Robot uses familiarity to avoid cyclic behaviour

This simple improvement is enough to allow the robot to navigate most common boundaries. Looping can still occur, however, in situations like the one shown in Figure 5-15, where the obstacle resembles a "cave".

[37] That is, any location whose familiarity indicates that it was visited after the robot started seeking its current target.

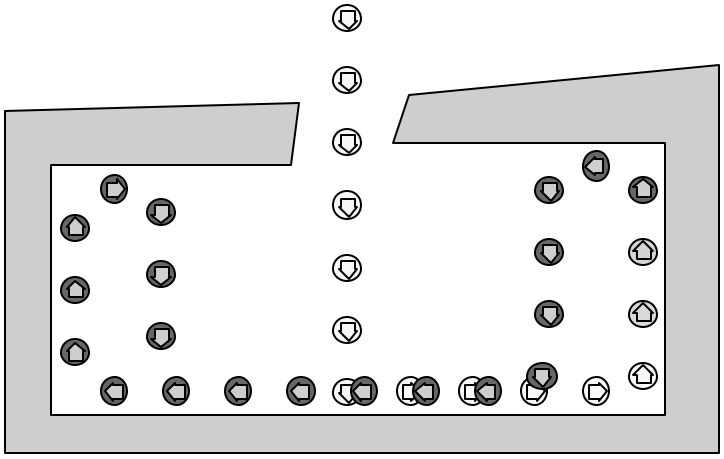


Figure 5-15: Behaviour cycle in "cave" -shaped obstacle

Here, because the boundary-following strategy turns the robot back on its path twice, it again enters a loop. Cases like this can be handled using the familiarity sensor in a more general way than described above. The robot should take an opportunity to return to moving in the direction of its target only if it is following a boundary in *unfamiliar* territory. Then, if the robot is following a boundary in *familiar* territory- territory that it has moved through before while seeking the current target- it "knows" that it should not return to moving in the direction of its target even if it seems desirable to, since this is what it would have done last time it was there. Instead it should wait until it reaches unfamiliar territory again, and then turn whenever appropriate. This results in an expanding search that can get the robot out of awkward situations like the "cave" obstacle, as shown in Figure 5-16.

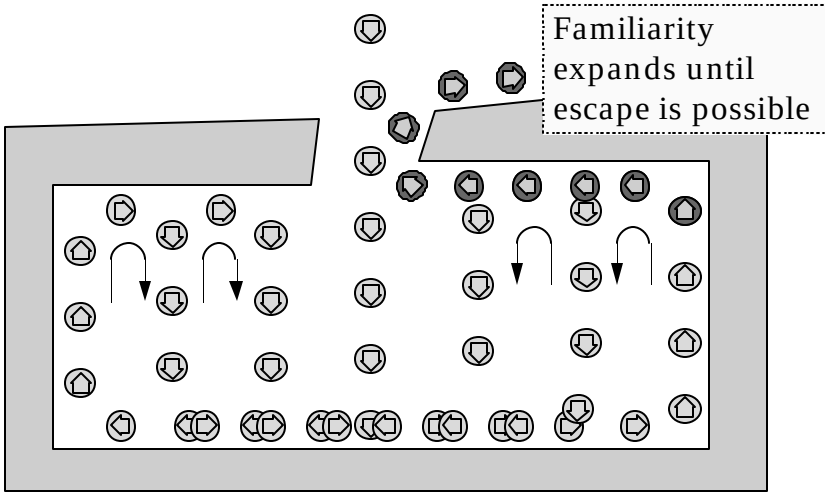


Figure 5-16: Robot escapes from cave-shaped obstacle

It is easy to see that this approach should never result in looping behaviour. If the robot makes a decision to turn at a particular location, and that decision results in it looping back to that same location, the next time around that area will be familiar to it so it will not turn there again.

While the obstacles the robot has been shown navigating are large scale, this behaviour is in fact mostly used for moving to places near the robot's current position. It is given a high level of intelligence so that the robot's behaviours will be robust in the face of changes in the environment, without the robot having to exhaustively check for such changes every single time it prepares to make a movement. The region seeking behaviour that will be described in Section 5.4.1 (page 158) enhances this behaviour for moving over long distances.

A suitable state machine for implementing this behaviour is shown in Figure 5-17.

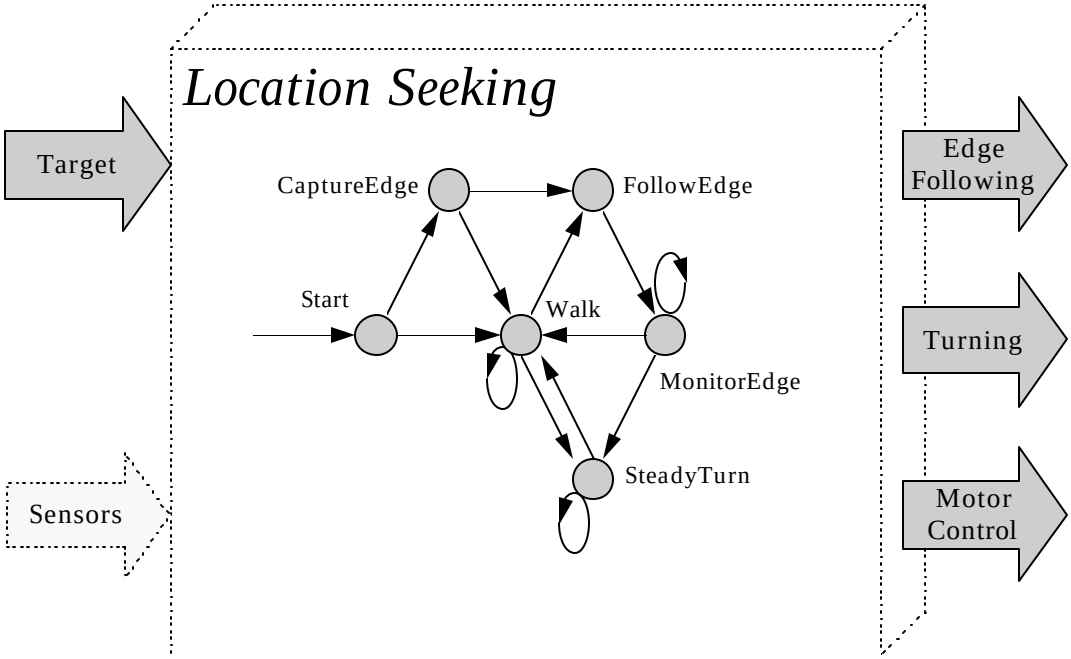


Figure 5-17: State machine for location seeking behaviour

The behaviour has a single input, specifying the location of its target. It has outputs to the edge following, turning, and motor control behaviours. It decides which of these to sponsor according to its needs at a given time. If it is following a boundary, it sponsors edge following. If it wishes to leave that boundary, it sponsors turning to change the robot's direction to point towards the target. And if it is simply moving forward in an open area, it controls the motor directly by sponsoring the motor control behaviour to move forward at the desired speed (see Table 5-2).

Table 5-2

State	***Action***	***Sponsorship to..***
Start	The behaviour is initialised. Provision is made so that if location seeking is sponsored repeatedly to move between markers close to each other, the movement of the robot will be smooth and not jerky.	None
CaptureEdge	The robot evaluates the shape of a nearby boundary to determine whether to leave it gracefully if it is not an obstacle, or follow it smoothly if it is.	None
FollowEdge	The robot starts following an edge, setting up timers, checking the familiarity of the region to avoid looping behaviour etc.	Edge Following
MonitorEdge	The robot follows an edge, watching out for opportunities to leave the boundary and approach the target directly that it has not tried before.	Edge Following
SteadyTurn	The robot turns gracefully to face towards the target.	Turning
Walk	The robot moves towards the target directly in a free area.	Motor Control

5.3.2 Patrolling Behaviour

The patrolling behaviour is concerned with ensuring that the robot repeatedly re-visits every area that it has ever passed through before. It is the main sub-activity of the "prowling" behaviour to be discussed in Section 5.3.4. Prowling is taken to be the entire co-ordinated activity of patrolling familiar territory, exploring new areas, and ensuring that the cartographic system gets a chance to find and use landmarks. Hence the patrolling behaviour need not concern itself with any of these other issues.

The difficulties encountered in trying to implement patrolling are as follows:-

- The robot's map is always changing, so approaches that assume the environment and the robot's representation of the environment are static will fail.

- The patrolling activity may be interrupted at any time- for example, if the robot needs to start following a boundary to find a landmark before it loses track of its position. Hence approaches that assume they have full and continuous control of the robot will fail.

In this section, an approach called the "static patrolling strategy" is given that would work well for a static map with uninterrupted operation, but is not guaranteed otherwise. Another simpler approach called the "dynamic patrolling strategy" is described which is less efficient but which *will* work for a dynamic map and interrupted operation. Then a scheme is presented which merges the two approaches. This "combined strategy" normally behaves much like the first approach (giving good runtime behaviour most of the time), but there is also an influence from the second approach that builds up over time and will "rescue" the robot if the first approach fails.

Static patrolling strategy

The cartographic system maintains a map of the robot's environment in the form of a collection of markers, with links formed between adjacent markers (see Section 4.5.4, page 115). These markers possess a timestamp that indicates when the robot last visited the area they are associated with. The robot can make use of these features of the cartographic system to repeatedly patrol the environment using an algorithm that is very simple, yet guaranteed to patrol *every* part of the robot's map reachable from the robot's starting point. However this guarantee only holds if the markers and links between markers do not change while the robot is executing the algorithm. Hence this is a "static strategy" for patrolling. A "dynamic strategy" will be given in the next section which does not have this limitation, and then the two strategies will be merged into a combined strategy that has the advantages of each.

The static strategy is to use the following extremely simple algorithm for controlling the robot's motion:-

1. Set the timestamp of the marker associated with the robot's present location to the current time.

1. Choose the marker linked to the current one which has the *least recent timestamp*[38]. Move to that marker, and repeat from step 1. If more than one marker has the same timestamp, choose any one of them.

This will be shown to cause the robot to repeatedly visit every marker in its map which is reachable from its starting point, assuming *markers and links are static*. Figure 5-18 shows an example of the algorithm in action for a small number of markers. For this example, all the markers are initially given a timestamp of zero. In practice, no two markers will ever have the same timestamp since the robot cannot be in two places at the one time, but the algorithm does not depend on that.

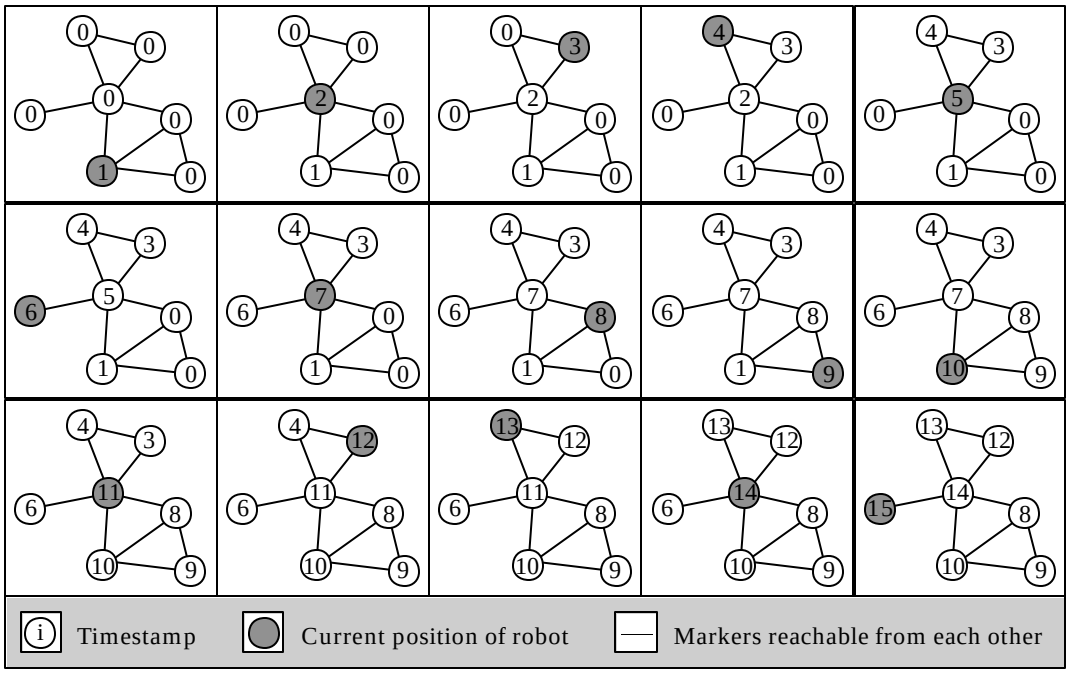


Figure 5-18: Static patrolling strategy in operation

The diagram shows that the robot does indeed repeatedly visit each marker, without neglecting any of them. While the operation of the algorithm may be intuitively quite clear, it is not immediately obvious that it will work under all situations, and that there are not some pathological cases that could lead it to neglect markers. Therefore the algorithm will now be analysed in some detail to show that it is in fact guaranteed to function correctly.

[38] If there is no marker linked to the current one, this is a degenerate case where there is nothing for the robot to patrol- so it should simply stay at its current position.

For this discussion, "patrolling a marker" is formally taken to mean that the robot visits that marker repeatedly, with a finite time interval between each visit. "Neglecting a marker" is the logical opposite - meaning that the robot after some point fails to visit that marker for an unbounded length of time.

The algorithm will be shown to work by demonstrating first that if the robot patrols any marker A, it will also patrol any marker B linked to A. Now that B is known to be patrolled, this argument can be repeated, so any marker C linked to B will also be patrolled. Therefore, by repeating this argument as many times as required, any marker which can be reached through any number of intermediate links from the marker A can also be shown to be patrolled. It will then be demonstrated that the robot patrols at le ast one marker, and hence it must patrol every marker reachable from that marker.

Consider any marker A that the robot is known to patrol. Let B_1, B_2, ... B_n be all the markers linked to A, as shown in Figure 5-19. When the robot visits A, the next marker it chooses to visit will be whichever of these markers has the least recent timestamp.

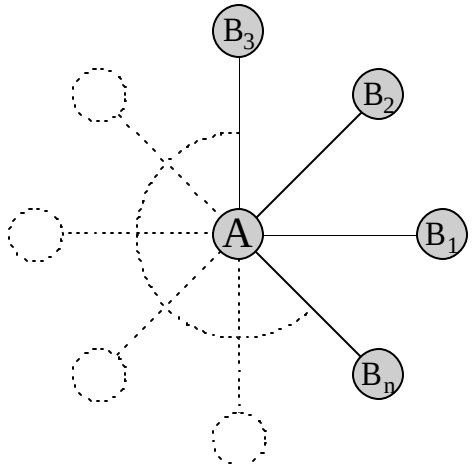


Figure 5-19: A group of markers linked to a patrolled marker

Label the marker that the robot chooses to move to next "B_i". When the robot moves to B_i, the timestamp of that marker will be set to the current time. In the succeeding visits of the robot to A, it will not choose to leave through B_i again at least until all the other nodes connected to A have been visited - since until then at least one of those nodes will have a less recent timestamp than B_i and therefore that node will be selected by the robot in favour of B_i.

This observation will now be used to show that each of the *n* markers connected to A will be visited at least once in the time interval between *n* successive passes of the robot through A.

This can be seen by considering the consequence of assuming that this is *not* the case, and that there is at least one marker B_k which is not visited in this time interval.

Consider the scenario where the robot has passed through A *n-1* times since B_k was last visited, and has just arrived back at A for the n^{th} time. Each of the *n-1* times the robot passed through A, it must have chosen a different next marker to visit- since each time it left A, the timestamp of the next marker would have been set to the current time and hence would become more recent than that of B_k, and so could not be chosen again at least until B_k was. The robot therefore will have left A through *n-1* different markers, and it cannot leave through any of these again until B_k is visited, since they will have more recent timestamps than that marker has. Hence when the robot leaves A for the n^{th} time it must leave through B_k, because it is guaranteed to have a less recent timestamp than any of the others. This shows that it is impossible for the robot to every fail to visit a marker linked to A for a time interval greater than *n* passes of the robot through A[39]. Hence, since there is a finite interval between the robot's visits to A, and *n* is finite[40], every marker connected to A will also be patrolled, because the robot is guaranteed to visit them with a finite interval between visits.

It is now necessary to show that there is at least one marker that the robot patrols. If the converse was true, and the robot patrolled *no* markers, then that would mean that there is no marker that the robot visits repeatedly, with a finite time interval between each visit. This implies that for each marker, there is some finite time beyond which the robot fails to visit that marker for an unbounded length of time. Therefore, by taking the maximum of these times for all the markers, there is a finite time after which the robot will never visit *any* of the markers. But this does not make sense- the static strategy will always give the robot a marker to move to[41]. Therefore the initial assumption must be wrong, and there must be at least one marker that the robot patrols. Let this be the marker A in the argument given earlier. It can therefore be shown that all the markers linked to A are patrolled. By repeating the argument, the markers linked to these markers in turn can also be shown to be patrolled, and so forth.

---

[39] Note that this does not imply that the marker will be visited *through A* in that interval. If the marker were to be visited through some other route, the argument given does not apply, and it is quite possible that the marker would not be visited through A at all. The argument only applies when a marker is being neglected, and places an upper bound on how long it may be neglected for.

[40] This is guaranteed by how the links are formed- in fact, *n* is typically four (see Section 4.5.4, page 115).

[41] Unless there are no markers at all- in which case, the robot has no territory to patrol.

Therefore it can be shown that all the markers which can be reached through any number of intermediate links from the marker A will be patrolled. Since A must have been reachable from the robot's initial position, or else the robot could never have reached it to patrol it, this shows that the robot will patrol all markers reachable from that initial position.

The static strategy, while experiment shows it to generally give good runtime behaviour, is not safe to use as it stands for patrolling. The robot's map was assumed static, but in actual operation it is dynamic- markers are removed, added, and frequently have the links between then modified. This occurs even if the environment itself is static, due to the nature of the marker laying system (see Section 4.3.2, page 93). Once the markers and their links can change while the robot moves, the guarantees for the static strategy are not so strong. Dynamic effects could potentially lead to looping behaviour. The static patrolling algorithm is also difficult to apply, because it relies on the map being static at the robot's position- and this is precisely where the map is most in flux, since the robot continually updates the map at its current location as it moves.

Dynamic patrolling strategy

Another possible strategy is as follows. The cartographic system could allow the robot to detect the marker with the least recent timestamp in the entire map. If the robot moves according to the rule "always move towards the location of the least recent marker in the map", then it will not neglect any location in the map- since if it did, the marker associated with that location would become the least recent in the map and the robot would then devote its efforts to reaching that location.

This simple strategy will work in a dynamic map without any problem, since the least recent part of the map will not be in flux until the robot actually gets there, at which point the robot just moves on to the next least recent marker. It also works better when interrupted- since it has a "global" target rather than a "local" target, changing the position of the robot through some other activity has less of an effect. It is guaranteed to work if the interruption does not move the robot further away from the target than it already is- if that were to happen, there would be some potential for a behavioural cycle where the robot approaches the target, triggers some condition that interrupts the robot and moves it further away for whatever reason, then the robot happens to be approach the target again along the same vector, and the

same condition triggers, etc. When the prowling behaviour is discussed in Section 5.3.4, it will be seen that for the most part it alternates between patrolling activity and moving around boundaries - and when it interrupts patrolling, it will circle boundaries completely and returns to the same point before resuming patrolling again. This effectively prevents the possibility of a behavioural loop, since the robot is left at the same distance from the target after the interruption.

The problem with the dynamic patrolling strategy is that it becomes inefficient. In practice, it encourages the map to fragment into scattered areas of widely varying ages, with the robot "bouncing" back and forth between areas that are close in age but far apart in position. This is because there is nothing in the strategy to bias the robot towards preferring targets close to its current location. Also, while in the static strategy every motion of the robot is chosen to patrol a location, in this strategy only the end-points of the robot's motion contribute to patrolling - the path it takes is not controlled. This is inefficient.

Combined Strategy

A much improved strategy can be formed by combining both the static and dynamic approaches using the "Goal Seeking" ability of the robot (see Section 4.5.4, page 115). Goal Seeking allows multiple targets for the robot to be set with different levels of "desirability". Patrolling can be achieved by setting *all* the markers to be targets, with their desirability made to be higher the less recent their timestamps are. If the robot then simply follows the vector indicated by the "scent" virtual sensor, it will automatically patrol the markers in a way that combines both the static and dynamic strategy, as will now be explained.

In Goal Seeking, markers are continuously updated in an effort to find the shortest path from them to the most desirable goal. This is done in an iterative way, and takes time. Information "spreads" outwards from the goals through the links between markers until it reaches the robot. It is possible that information from one goal will reach the robot first, and then later be superseded by information from a more distant but more desirable goal. This is why making every marker a goal implements patrolling. Information from the nearest markers to the robot's current position reaches it first. The most desirable markers were configured to be those with the least recent timestamp, so the "scent" at the robot's position will lead it to move to these, just as the static patrolling strategy would have it do. But if the robot fails to patrol a distant

marker, information from it will eventually be spread to the robot. Since that marker will be less recent and therefore more desirable than the markers in the robot's vicinity that it *has* been patrolling, the "scent" will lead it to now go towards that distant marker- just as the dynamic patrolling strategy would have it do. So the combined strategy has both the efficiency of the static strategy, and the long-term guarantees of the dynamic strategy.

Implementation of patrolling

While the discussion of this behaviour has been quite extensive, its implementation is comparatively straightforward because it simply sits on top of the cartographic system. It is only necessary to configure Goal Seeking to derive markers' desirability from their timestamps, and then the behaviour can follow the "scent" virtual sensor in a reactive way.

The state machine for this behaviour is very straightforward (see Figure 5-20). The robot cycles between waiting for information from a goal to "spread" to it, and moving towards that goal. To move towards a goal, the behaviour simply sponsors an output to the location seeking behaviour and passes the goal on to it.

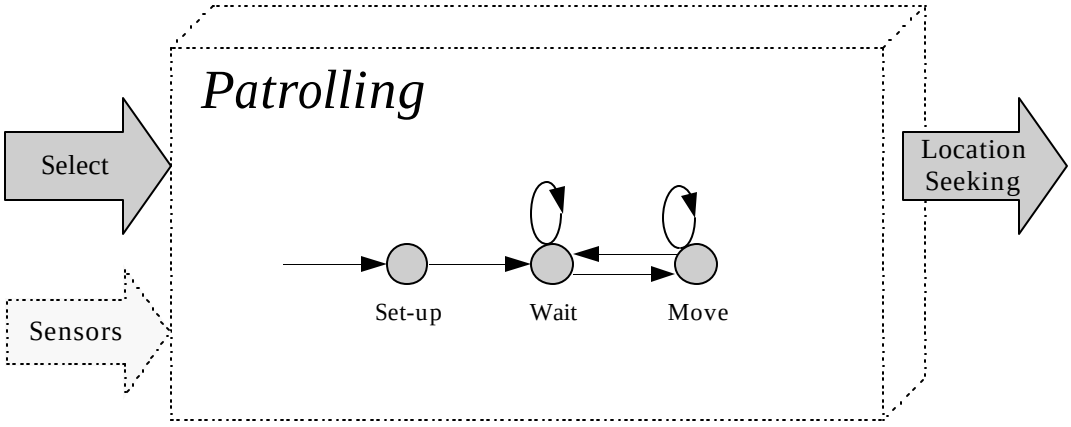


Figure 5-20: State machine for patrolling behaviour

The behaviour does not need any inputs, as it is complete in itself. However, an input is provided so that sponsorship can be passed to the behaviour- effectively to "select" the priority level at which the behaviour operates.

5.3.3 Exploring Behaviour

This is a straightforward behaviour which simply moves the robot into an area that it has never explored before. Again this is a behaviour that sits on top of the services of the cartographic

system. The "curiosity" virtual sensor described in Section 4.5.2 (page 114) can detect directions in which the robot has not explored. Such areas are indicated by a lack of any markers in a particular direction, and the absence of any boundary blocking the robot from moving that way. When such a condition is detected, it is passed on to the rest of the control system through the curiosity sensor as a simple vector. The exploration behaviour is given this vector to venture along by the prowling behaviour (which is responsible for deciding if the robot can afford to engage in exploration at the moment or not). The exploration is considered complete if the vector has brought the robot to a new boundary[42]. This is a particularly simple behaviour- it starts moving, and keeps moving until it detects a new boundary, and then stops (see Figure 5-21). Again it moves by passing the vector it is given on to the location seeking behaviour, and sponsoring that behaviour to move on its behalf.

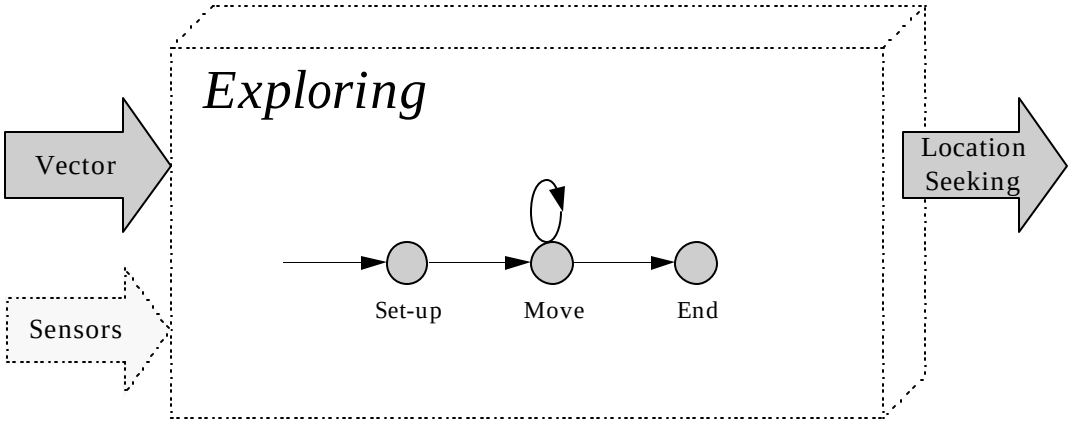


Figure 5-21: State machine for exploring behaviour

5.3.4 Prowling Behaviour

"Prowling" is a high-level behaviour of the robot designed to implement its overall "sentry-like" functionality. It orchestrates the action of a set of lower-level behaviours to meet a number of requirements necessary for good sentry behaviour. The nature of these requirements will now be examined, to motivate the decisions taken in the design of this behaviour. Acceptable sentry-like behaviour requires the following of the robot:-

- The robot must find and explore any areas in which it has never been.

[42] It may also be terminated prematurely by the prowling behaviour to prevent the robot losing track of its position, but the behaviour itself need make no provision for that.

- The robot must repeatedly patrol back and forth through every area it has ever explored in a timely fashion. This is the basic action of a sentry, and is also necessary so that the robot can detect if any part of its territory has changed.
- The robot should not spend an extended length of time exploring at the cost of neglecting patrolling, nor should it concentrate on patrolling to the exclusion of exploring. Either of these would be undesirable in a sentry.
- The robot must be able to keep track of its position relative to the environment, otherwise it will not know where it needs to patrol and where it needs to explore.

This last requirement is an important one, and will now be examined in detail.

Constraints on robot behaviour while prowling

To act as a sentry, the robot needs to build and maintain a map of its environment. A suitable cartographic system for achieving this was described in Chapter 4, and is implemented within the *map maintaining* behaviour (see Section 5.3.5, page 156). This section examines the constraints that must be met by the overall behaviour of the robot so that the cartographic system may perform correctly.

No matter what the behaviour of the robot is, if it needs to keep track of its position then there are some concessions it must make to allowing time for finding landmarks. When moving in open areas away from any boundary, the robot receives no information from its proximity sensors. This means that the robot has no external references it can use to help keep track of its position. It is forced to rely on simply integrating its motion to get its position. As discussed earlier (see Section 4.4, page 95), this unavoidably introduces accumulating error in the robot's estimate of its position relative to its environment. The longer the robot remains in motion, the greater the uncertainties in its position become. Therefore, if the robot needs to keep track of its position relative to its environment, then it must be designed to behave in a manner that leads it to avoid operating in open spaces for extended lengths of time.

When moving in an area near a boundary, the robot receives information from its proximity sensors which may give external cues or landmarks the robot can compare against an internal map and use to complement position tracking derived merely from motion integration (again, as detailed in Section 4.4, page 95). These landmarks are recognisable boundary features such as corners, and to detect them from proximity data, the robot needs to physically follow

the boundary so that it traces the outline in its own motion. If the robot does not follow the boundary, the information it receives from its sensors is virtually useless without complicated processing- both for interpreting data to determine what it says about the environment, and in correlating that with the map to deduce position corrections. Therefore the type of information the robot can extract from the environment varies both with the area if finds itself in and with the robot's behaviour, as follows:-

- **Absence information**- when in an open area, the only information the robot can deduce is that nothing is present in its current locality. All such places look the same. so they give no help in indicating the robot's position.
- **Presence information**- when in an area close to a boundary, the robot knows that something is present close to its locality. This is normally still not enough to indicate the robot's position, since such places generally appear quite similar to other places further along the same boundary.
- **Trace information**- when in an area close to a boundary, and moving so as to follow that boundary, the robot gains information about the shape of the boundary. This *can* give an indication of the robot's position when compared with previous traces, as described in Section 4.4 (page 95).

The fundamental requirement cartography places on the robot is that it must avoid being in motion for an extended length of time without correcting its position estimate from landmarks to keep the estimate consistent with its environment. This implies that:-

- The robot should never be in motion in *open areas* for too long. The robot can detect no landmarks in open areas, so errors in its estimate of its position and direction will accumulate without any way to correct them. It is important to find a landmark before the error grows to such an extent that the map becomes useless due to the excessive uncertainties involved. The amount of time the robot can stay in motion in open areas safely depends on how quickly error accumulates in its position and direction estimates- for the robot this work was implemented on, anything over about a minute was dangerous.
- The robot should never move through previously *unexplored areas* for too long, whether a boundary is near or not. The robot may detect landmarks in a place it has never been before, but it has nothing to compare them with, so it is again important that the robot turns back before accumulating uncertainty endangers the usefulness of its map. The upper limit

on the time the robot can move through an unexplored region is the same as the bound on the time it can spend in open areas- the problem is the same in both, a lack of any way to compensate for errors in the robot's position and direction estimates.

- The robot must frequently engage in behaviour that allows the robot to get trace information from its environment, so that the cartographic system will be able to detect and use landmarks.

This last requirement in practice means that, however prowling is implemented, a large fraction of the robot's time should be spent following boundaries, since this is the condition under which a robot with proximity sensors can detect landmarks (see Section 4.4, page 95).

Suitable algorithm for prowling

The following basic algorithm for implementing prowling meets the requirements listed above:-

1. When the robot meets a boundary, it should navigate around the circumference of that boundary in its entirety, arriving back at roughly the same point at which it started.
1. Once this is done, the robot should try to reach parts of its territory which it visited less recently than its current location- in other words, start patrolling the environment. If it meets any boundaries after departing from the one it is currently on, it should revert to step 1, then try again to reach parts of its territory less recently visited.

This simple strategy will make the robot spend a good deal of its time following boundaries, which is desirable for the proper functioning of the cartographic system. With the design for patrolling described in Section 5.3.2 (page 143), it also guarantees that the robot will not neglect any part of its territory.

It is a good idea for a robot to completely circle any boundary it meets, when practical. If the robot follows only part of the boundary, then the boundary will be split into segments of different age[43] in the map. The robot, when patrolling those areas, will generally retain this segmentation- because when, for example, it moves between a part of a boundary that was patrolled a long time ago into a part that was patrolled recently, it will most likely move off to some more critical area before coming back to finish the boundary off. If any further segmentation occurs for any reason, that will be retained too. So the boundary will become

[43] The "age" of an area refers to how long it is since the robot visited that area last.

more and more "fragmented", and the robot will in the long run end up patrolling in sections too short for landmarks to be detected. Completely circling the boundary avoids that problem. It also avoids potential looping behaviour where the robot "bounces" between two boundaries without ever finishing either. As well as this, the robot is well informed about where it should move next, since it has the opportunity to measure the familiarity of all the areas surrounding the boundary. Another very important reason to completely circle the boundary is that it makes it possible to give guarantees about the robot's overall behaviour (see the "dynamic strategy" discussion in Section 5.3.2).

Implementation of prowling

The strategy outlined above is only a starting point for good sentry behaviour. Numerous refinements need to be superimposed on top of the basic strategy to determine when the robot should explore areas outside of the robot's current territory, and how to respond to difficulties in patrolling a boundary and changes in the environment. A suitable state machine diagram is sketched in Figure 5-22.

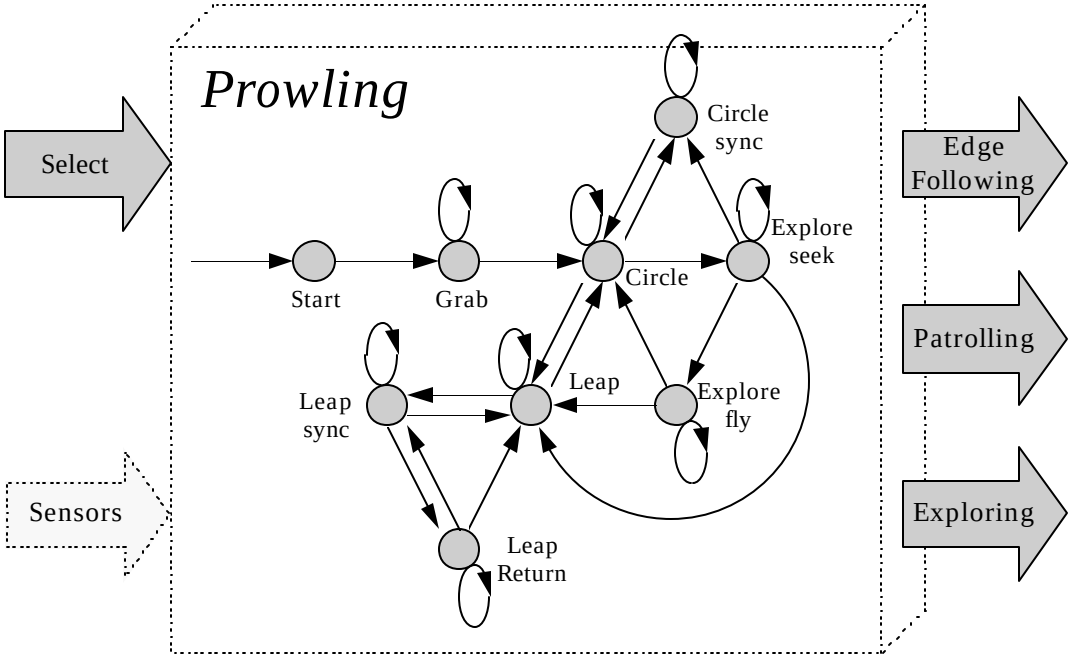


Figure 5-22: State machine for prowling behaviour

The diagram shows that the prowling behaviour has a single input through which it may receive sponsorship. It has three outputs, feeding to edge following, patrolling, and exploring. Prowling

can choose to pass on its sponsorship to these behaviours as it sees fit. Table 5-3 shows the meaning of the different states in the behaviour's state machine, and when the different outputs are sponsored.

Table 5-3

State	***Action***	***Sponsorship to..***
Start	The prowling behaviour is initialised	None
Grab	The robot finds a boundary from which to start	Edge Following
Circle	The robot navigates around a boundary completely	Edge Following
CircleSync	If the robot gets confused while circling a boundary, it turns back to familiar territory in search of landmarks	Patrolling
ExploreSeek	Given the opportunity, the robot will follow a boundary it has not met before	Edge Following
ExploreFly	Given the opportunity, the robot will explore an open area (away from any boundary) it has not met before	Exploring
Leap	The robot leaves the current boundary in search of "older" areas to patrol	Patrolling
LeapSync	If the robot gets confused while in an open area, it will try to find any boundary at all in a search for landmarks	Edge Following
LeapReturn	If the robot gets confused while in an open area, and cannot find any boundary nearby, it will turn back to familiar territory in search of landmarks	Patrolling

5.3.5 Map Maintaining Behaviour

This is the actual module within which the bulk of the cartographic system is implemented. It is a direct implementation of the marker representation scheme described in Section 4.3, page 83. In every cycle, this behaviour scans a single marker from each of the four neighbourhoods (see Section 4.3.1, page 88), and updates the relevant virtual sensors. The state machine for this behaviour is trivial- the behaviour essentially stays in the same state throughout its

operation (see Figure 5-23). The complexity of this behaviour lies in what it does within that state. Chapter 4 almost in its entirety can be taken as a description of this behaviour.

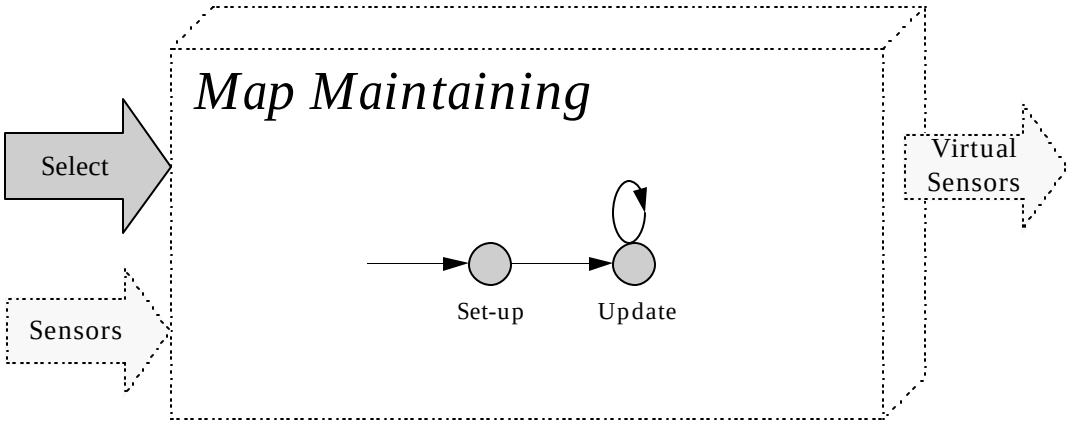


Figure 5-23: State machine for map maintaining behaviour

This completes the discussion of the informed behaviours. The higher-level "user interaction" behaviours will now be described.

5.4 User Interaction Behaviours

This collection of behaviours allow the robot to be guided by external commands, whether those commands are to enable the robot's own autonomous sentry-like behaviour, or to impose specific goals on the robot, or to control the movement of the robot directly. The place of these "user interaction behaviours" in relation to other behaviours is shown in Figure 5-24.

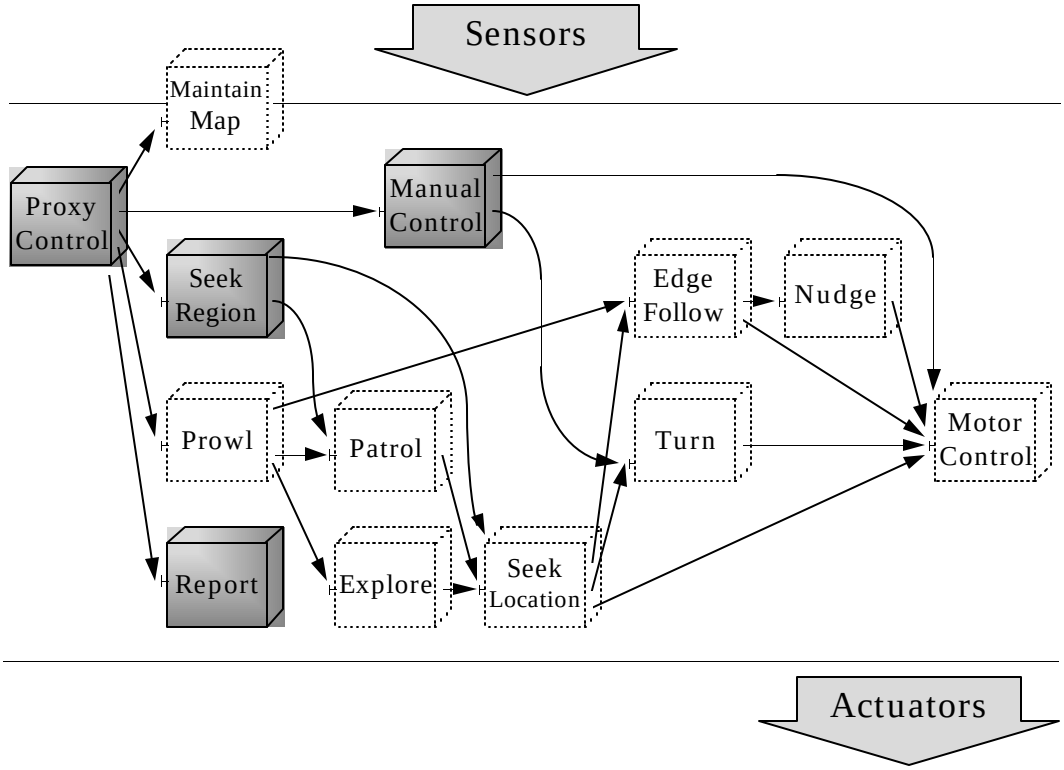


Figure 5-24: User interaction behaviours

These behaviours are for the most part quite simple - most of the hard work has been done by lower-level behaviours.

5.4.1 Manual Control Behaviour

This behaviour allows an external user to control the motion of the robot directly by speeding it up, slowing it down, and turning it left and right. The state machine for this behaviour is extremely simple - it simply listens for commands and changes speed and angle setpoints as appropriate. It has outputs to both the motor control behaviour and the turning behaviour. It uses the motor control behaviour to set the forward speed of the robot, and uses the turning behaviour to set the direction the robot should be facing. It passes on its full sponsorship to both behaviours.

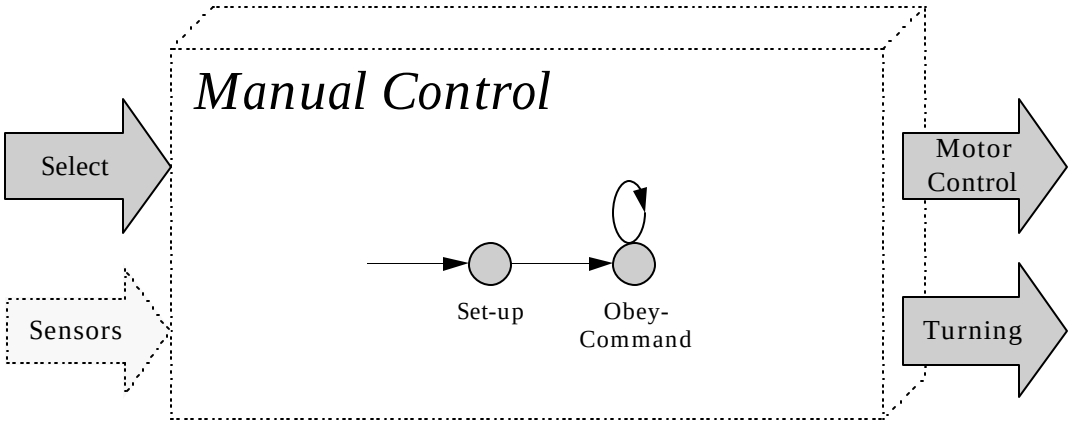


Figure 5-25: State machine for manual control behaviour

It is interesting to note that the turning behaviour used by this behaviour also has its own output to the motor control behaviour. Since that output will have the same priority as the one from the manual control behaviour itself, commands from both sources will be merged. If the manual control behaviour attempted to control the speed the robot turned at through its output to the motor control behaviour, this would conflict with its use of the turning behaviour at the *same level of sponsorship* to also turn the robot. The motor control behaviour would accept the two conflicting streams of command, and the rate at which the robot would turn would oscillate between the two- effectively averaging them. In situations where there is danger of such conflict, behaviours should be sponsored at different levels to establish which takes precedence.

5.4.2 Region seeking behaviour

This behaviour lets the user command the robot to move towards targets set previously (using the proxy behaviour, see Section 5.4.4). To do this, it simply passes the co-ordinate of the target on to the *location* seeking behaviour. That behaviour is designed to perform a very similar task, but over shorter distances. Because it works with absolute co-ordinates, it is not accurate over long distances- since the robot's position estimate may drift during the journey- so the robot will only move towards the general *region* of the target[44]. The location seeking

[44] This will usually be reasonably accurate since, because since targets are marked on the map with the "tagging service" (see Section 4.5.3, page 115), their location will be kept consistent with any drifts in the robots co-ordinate system. Hence the error is bounded by the drift the robot has suffered since it was last at the target, rather than the drift it has suffered since the target was originally set.

behaviour will get the robot to its target quite quickly, unless the robot has to back out of dead-ends (see 8.2.3, page 248). The robot can "hedge its bets" by using the Goal Seeking service of the cartographic system in the background to search for a route to the target (see Section 4.5.4, page 115). This searching process is slow, but if the robot is delayed backing out of dead-ends as it tries to get to the target the search will have time to succeed, and the robot can then switch to simply following the "scent" virtual sensor to the goal (see Figure 5-26). It can always revert back to using location seeking if the path found by goal seeking proves to be inaccurate due to changes in the environment.

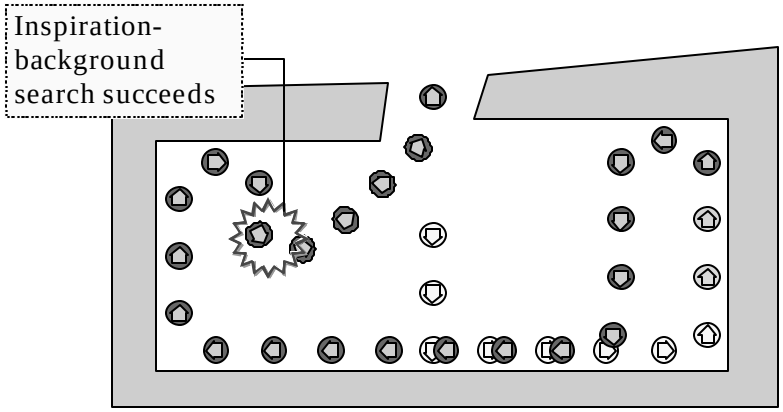


Figure 5-26: Use of background search

Goal seeking is only suitable for use as a safeguard, not as the robot's main strategy for finding a route to the target. Lower-end robots have insufficient processing power to implement such a search in anything approaching real-time, so they would have to "sit and think" before moving at all[45]. Also, since planning based on a map constrains the robot to moving through regions it has already explored and mapped, opportunities may be missed that the "physical search" approach could have taken advantage of.

A suitable state machine for this behaviour is shown in Figure 5-27. The behaviour simply switches between using physical search and map search as appropriate. It begins using physical search, then switches to map search if that succeeds before the robot reaches its target. It may need to switch back to physical search if the environment has changed and the map is found to be no longer accurate.

[45] For the robot this work was implemented on, a feature was added so that the search could be speeded up a hundredfold by an external request from the user, with significant degradation of the real-time performance of the robot (it grinds to a halt for a few moments).

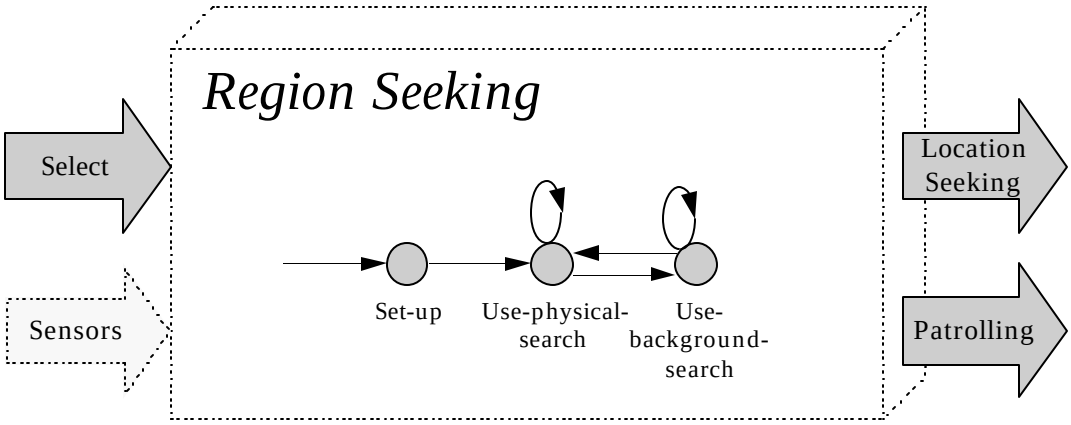


Figure 5-27: State machine for region seeking behaviour

The behaviour has an output to the location seeking behaviour to perform physical search. It actually "hijacks" the patrolling behaviour to perform the background map search, by superimposing "infinite desirability" in terms of goal seeking on the target marker. The patrolling behaviour will then, in the long run, lead the robot to that target. The region seeking behaviour passes on its full sponsorship to both location seeking and patrolling, but places a low "priority factor" on the output to the patrolling behaviour (see Section 3.3.2, page 55). This means it will compete for control of patrolling at its own level of sponsorship, but having received control it will only pass on a low level of sponsorship to patrolling. Hence it gains control of patrolling, but ensures it will not compete with physical searching. This is the desired configuration because, while the robot wants the patrolling behaviour to be managing goal seeking, it does not want it to control the movement of the robot. Once the "scent" of the target is detected, then the sponsorship for location seeking will be dropped and given to patrolling instead.

5.4.3 Reporting Behaviour

This behaviour collects statistics about the performance of the robot and sends them out along the serial connection to the robot every second, so that the status of the robot can be monitored. The statistics include:-

- Number of scan cycles per second (see Section 3.6.1, page 69).
- Average time per cycle for the last second.
- Longest time of a cycle in the last second.

- Number of connections active.
- Current confusion level of the robot.
- Current familiarity level of the robot.
- Whether the robot is currently following a boundary.

Optionally, exhaustive information on the status of the robot's behaviours and connections can be reported. The estimated position of the robot is also reported at frequent intervals, along with the position at which various significant events occur at (finding a landmark, laying a marker, etc.). This can be used for graphically visualising the activity of the robot (see Sections 7.8.1 and 7.8.2, page 229). The state machine for this behaviour is trivial (see Figure 5-28).

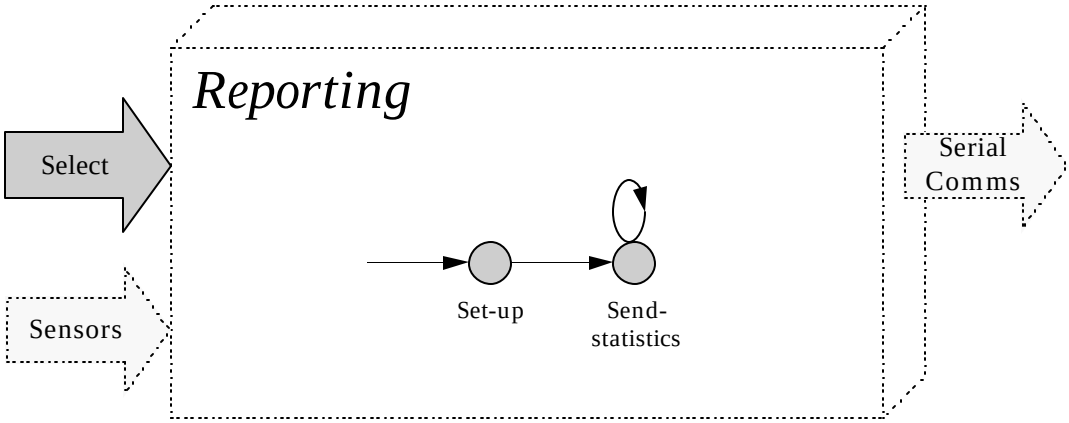


Figure 5-28: State machine for reporting behaviour

5.4.4 Proxy Behaviour

This is the highest-level behaviour present in the robot. It responds to external commands, supplying sponsorship to other behaviours on behalf of the user. The robot is designed to be able to operate autonomously, but this behaviour allows it to also respond to a simple set of commands across a serial connection. These were useful for performing experiments with the robot. Table 5-4 gives a list of the commands implemented.

Table 5-4

Command	Description
Prowl	Sponsors the prowling behaviour, which starts the robot performing its basic autonomous "sentry-like" activity.
Patrol	Sponsors the patrolling behaviour alone, making the robot continually move through every part of its territory. If this is used for an extended length of time instead of prowling, the robot will lose track of its position because the behaviour makes no provision for seeking landmarks.
Edge	Sponsors the edge following behaviour, to make the robot follow the edge of any nearby obstacle.
SetMark	Takes note of the current location of the robot using the Tagging Service (see Section 4.5.3, page 115).
SeekMark	Sponsors the region seeking behaviour to move the robot towards a location tagged at an earlier stage.
Explore	Chooses whether the "curiosity" virtual sensor should be enabled or not. If disabled, the robot will never move out of its current territory to explore its environment.
Renew	Controls whether the marker laying system is enabled (see Section 4.3.2, page 93).
SendMap	Requests the robot to transmit its internal map on the serial connection for external inspection.
Think	Increases the rate of "goal spreading" a hundredfold for a short interval. This has the consequence of degrading the robot's real-time performance for that interval.
Conquer	Requests that the robot take over the world[46].
Manual	Sponsors the manual control behaviour so that the robot will follow external motion requests.
Halt	Puts the robot into an idle behaviour.

[46] Not fully implemented as yet.

Basic commands such as pausing, resetting, and restarting the control system are implemented in the robot's kernel (see Section 7.3, page 213) rather than in this behaviour so the user is guaranteed to always be able to stop, pause, or restart the control system, even if it hangs or crashes.

The state machine for the proxy behaviour is shown in Figure 5-29. It has several outputs, some of which it sponsors continuously- map maintaining and reporting- and the rest of which it chooses to sponsor according to commands it receives. It has a single input called "root". A connection is attached to this input at an arbitrary fixed priority, and it is from this single source that priority is distributed to all the behaviours in the control system.

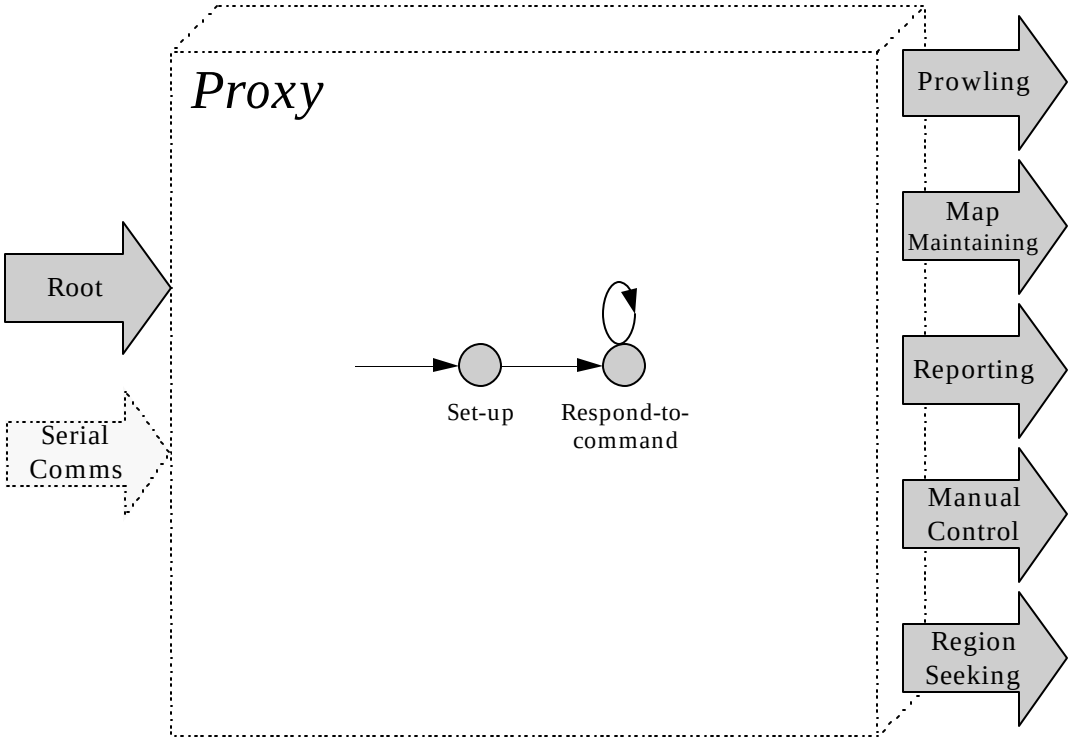


Figure 5-29: State machine for the proxy behaviour

5.5 Summary

This chapter described the design of a set of behaviours for the robot which together combine to make it act as a sentry, allowing it to explore and patrol its environment autonomously. The behaviours influence each other by controlling how they choose to pass on the sponsorship they receive. This sponsorship flows from the highest-level behaviour down to the lowest, with

each behaviour along the way free to pass on its share of sponsorship any way it chooses. Each behaviour makes a local decision, and from that the overall hierarchy of control at any instant is determined. Nowhere in this chapter was any mention made of any absolute level of priority- all that is needed is that each behaviour decides which of its outputs to sponsor, and at what fraction of its own level of sponsorship.

There is an interesting point about the Lateral architecture that the behaviours developed here bring into focus. Consider the edge following behaviour. In constructing this behaviour, a large amount of detailed consideration of the robot's sensing ability was required. But as soon as it was constructed, other behaviours could use it for their own purposes without needing to know how it worked. The location seeking behaviour was one of the behaviours that used it. This behaviour in turn had its own different set of concerns about the topology of obstacles, but once constructed it could also be used by other behaviours without them having to share its concerns. Location seeking was used by patrolling, patrolling by prowling, and prowling by proxy- none of them needing to be concerned with how the behaviours they used did what they did. This shows that the Lateral architecture supports a modular design, and suggests that it should be a scaleable architecture, since the complexity of lower-level behaviours does not cause complexity to accumulate in behaviours that use them as it tends to do in Subsumption (see Section 2.4.9, page 26).

6. Zac Script

This chapter presents an extension to the C++ programming language called *Zac Script* which simplifies the process of implementing control systems designed using Lateral's behaviour and connection constructs. The first section in this chapter clarifies the nature, purpose and utility of Zac Script. An outline is then given of the process whereby Zac Script can be converted to executable code using a tool called the *Zac Translator*. This is followed by a section describing the details of how Zac Script is used to specify behaviours and connections. The complete syntax of the language extension is then presented formally, element by element. An extended example is presented to demonstrate how a full a hierarchy of behaviours can be implemented using Zac Script. Finally, the limitations of Zac Script in its current form are examined.

6.1 Overview

In Chapter 3, issues related to implementing the Lateral architecture using the C++ programming language were discussed (see Section 3.6, page 68). Structures for a "light-weight" version of Lateral suitable for running on lower-end robots were presented. This was called the "Zac" implementation of Lateral[47]. It was noted that, while C++ is an excellent choice of implementation language for portability reasons, the constructs of Lateral are different enough from those native to C++ to make coding them in C++ a somewhat difficult and tedious process. Hence a special language extension to C++ for Lateral constructs seems justified. The following sections examine the arguments in favour of such a language extension in detail, explaining:-

- Why robot architectures may require new language structures.
- Why it was decided to implement Lateral structures by mapping them on to C++.
- Why it was decided to add new syntax to C++ and to automate this mapping, rather than choosing to directly code Lateral constructs in C++ itself.

[47] "Zac" is derived from the first name of Isaac Asimov, a science fiction writer known to many as the "Father of Robotics".

6.1.1 Languages for robot architectures

Robot architectures are a collection of constraints and guiding principles for organising a control system, as was described in Chapter 2. "Constraints" and "principles" are high-level concepts, and are of more help in the design of control systems than in their implementation. However, when an architecture has well-defined structural elements associated with it, such as the wires and augmented finite state machines of Subsumption, then it can guide implementation as well as design. One way of providing this guidance is to encapsulate the structures of the architecture in a "language" supporting the architecture's constructs, and then implement all or part of the robot's control system in that language. Examples of languages developed for robot architectures include the Behaviour Language [[21]] for Brook's Subsumption and ALFA (A Language For Action) [[18]] for Gat's ATLANTIS. The reason why it is useful to provide explicit support for the constructs of an architecture rather than just converting them to elements of some existing language is simply that this conversion can be difficult or tedious to perform. The structures and modularity of a robot architecture can be quite different from those supported by pre-existing languages. This was the case for Brook's Subsumption, and it is also the case for Lateral. In particular, the idea of using "connections" as explicit representations of the communication channels between modules (see Section 3.3.1, page 48) does not have a direct analogue in the popular programming languages. It can of course be emulated in them all, but this is exactly the extra layer of difficulty and tedium that introducing extra language constructs avoids.

6.1.2 Mapping Lateral structures to C++

It was desirable that, whatever the method used to implement Lateral, it should be applicable to as wide a range of target platforms as possible. There are few hardware standards yet in the field of robotics, so tying Lateral to one particular configuration would limit its use[48]. A good way to achieve this platform independence is simply to map the structures of Lateral on to a language that is widely supported, and then compile the result for the appropriate platform.

[48] As mentioned in Chapter 3, this was also important for pragmatic reasons - since when the work for this thesis was begun, the robot it was to be implemented on was not known.

C++ is an excellent choice for the target language. Compilers for C++ are available for most platforms - for example, GNU CC is a free C/C++ compiler and cross-compiler which can be configured to work with a wide variety of processors (see Appendix A3). An extra bonus of mapping Lateral structures in this way is that it is then possible to "borrow" all the constructs native to the target language, such as expression evaluation, and avoid "re-implementing the wheel". The reasons for choosing C++ over C was that the object-oriented nature of C++ is a closer match for Lateral, and hence Lateral constructs can be mapped on to it with less effort.

6.1.3 Supporting Zac Script through C++ extensions

It was decided to create a collection of extensions to the C++ language called "Zac Script" that support Lateral constructs directly, and can be mapped on to C++ automatically by a translator tool. A natural question, given the arguments advanced in the previous section in favour of C++ as a target for mapping Lateral on to, is whether the advantages such an approach can bring are sufficient to make it worth using instead of coding directly in C++ itself. If the benefits are not significant enough, then there will be more effort in constructing support for the language extensions than those extensions actually save. The full list of advantages of using Zac Script instead of C++ alone is as follows:-

- Converting state machines and connections to their C++ implementations by hand is tedious because, although it is quite straightforward to do, it results in a significant amount of repetitive, inelegant, clumsy code. It is useful to automate this.
- Simple but tedious "housekeeping" functions are also needed to generate interfaces to behaviours so that they can be correctly managed at run-time by the robot's control system (see Section 6.3.3, page 179). It is helpful to automate this as well.
- Once Lateral structures have been mapped on to C++, they become difficult to edit or maintain, because of the extra level of detail involved. For example, changing a state machine implemented in C++ may require re-ordering of identifiers and dealing with forward references. Hiding such details makes maintaining and modifying code simpler.
- Separating Lateral structures from how they are implemented in C++ means that the implementation of constructs can be changed quite radically without requiring source code

to be re-written[49]. If changes are made to how Lateral constructs are implemented, only the translator used to convert source code to C++ need be modified to reflect the new mapping.

The last two points were considered particularly important. By allowing explicit syntax for Lateral constructs, the detail of how they were mapped on to C++ was hidden- with a benefit of increased clarity and ease of maintenance of source code, and the pragmatic advantage that the mapping could afterwards be changed without having to rewrite all source code. All these considerations led to the decision to introduce explicit language extensions into C++ to support Lateral constructs. These extensions were called "Zac Script". Header files and libraries, the conventional ways to extend the C++ language, were not sufficient to encode the significantly different structures of Lateral, so a tool called the Zac Translator was developed that essentially acted as a pre-processor, taking source code written in Zac Script and translating all the extended syntax into pure C++. The translation could then be compiled, linked with suitable libraries, downloaded to the robot, and executed. This process is outlined in the following section.

6.2 Outline of translation process

In Figure 6-1, the process whereby a set of behaviours and connections eventually becomes executable code is outlined.

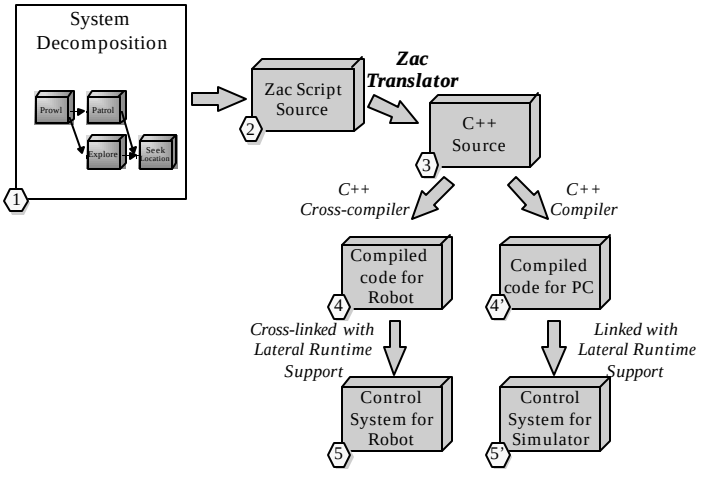


Figure 6-1: The path from Lateral structures to executable code

[49] This was particularly important for pragmatic reasons, since the "sentry" application described in the previous chapter was developed side by side with the Lateral architecture implementation, and could not wait for it to be

The steps in the process are as follows:-

1. Initially, the control system is designed in terms of behaviours and the connections between them.
1. This design is then expressed in Zac Script, which has special syntax for representing behaviours and connections.
1. Next, the Zac Script source code is converted into pure C++, mapping any of the Lateral constructs on to C++ structures. This translation is performed by the Zac Translator.
1. The resultant C++ code is compiled appropriately for whatever target platform it is aimed at, either for a physical robot or a simulated one on a PC.
1. The compiled code is linked with a module containing the runtime support necessary for the Zac implementation of the Lateral architecture (see Section 3.6, page 68). Modules specific to either the simulator or the physical robot are also linked in, and the resultant executable code will exhibit the behaviours specified in the original design.

An introduction to the actual language extensions to C++ present in Zac Script is now given. As the extensions are introduced, their equivalents in C++ will be presented.

6.3 Writing Zac Script

Zac Script is an extension of C++. As such, a file containing nothing but C++ statements is an acceptable Zac Script source file.

The most important new construct that is available for source code written in Zac Script is the "BEHAVIOUR" construct, which is intended to be used to implement actual robot behaviours. The behaviour construct appears in the following form:-

```
BEHAVIOUR <Name>
{
        <Input Connections>
        <Output Connections>
        <Variables local to behaviour>
        <Functions local to behaviour>
        <State machine implementing functionality of behaviour>
};
```

---

mature.

Every behaviour can have a set of input connections, a set of output connections, and a state machine that implements the actual functionality of the behaviour. These elements are described in the sections that follow. It is also useful for behaviours to be able to allocate local storage, and define local "helper" functions. To avoid having to invent new syntax to deal with this, behaviours are defined as extensions of the C++ "class" construct. Any code allowed within a class in C++ is therefore also allowed inside a behaviour and will have the same semantics. For example, the meaning of the following code segment is essentially unchanged if "BEHAVIOUR" is replaced by "class":-

```
BEHAVIOUR MakeTea
{
      // member variables
      int numCups, numBags;

      // member functions
      void PlugInKettle();
      int AreBagsAvailable() { return numBags > 0; }
};
```

There is one subtle difference however. A class in C++ is used as a "pattern" from which to create several objects of the same form. Behaviours in a robot, on the other hand, are generally unique- there is no need for more than one instance of them. Hence when the Zac Translator is converting a behaviour to C++, it will by default cause an *intermediary* class to be constructed with the member variables and functions described in the above code, and then create a *single instance* of that class called by the behaviour name, "MakeTea" in this case. The intermediary class will be derived from a shared base class for all behaviours that implements the functionality of Lateral. This base class, ZACProcess, was introduced in Section 3.6.4 (page 76). The Zac Script code fragment above would be converted to C++ code of this form:-

```
class ZACProcess_MakeTea : public ZACProcess
{
        // member variables
        int numCups, numBags;

        // member functions
```

```
        void PlugInKettle();
        int AreBagsAvailable() { return numBags > 0; }
};
ZACProcess_MakeTea MakeTea;   // create a single behaviour object
                              // called "MakeTea"
```

The services that behaviours inherit from ZACProcess will be described in examples to come, and are listed exhaustively in Appendix C3.

Within the Behaviour construct, it is possible to specify a state machine implementing the functionality of the behaviour, and to set up connections to other behaviours. These elements are now described.

6.3.1 Specification of state machines

In the Zac implementation of Lateral, behaviours are designed as augmented finite state machines. The "augmented" part of "augmented finite state machines" simply means that the state machines are permitted to have memory. This memory can be created in Zac Script by simply placing variables within the behaviour. A special syntax for streamlining the process of coding the actual state machines themselves is available in the behaviour construct. Consider the following example:-

```
#include <iostream.h>
BEHAVIOUR Greeting
{
      // member variables and functions could go here

      // state machine starts here
      @SayHi
            cout << "Hello World" << endl;
};
```

If this code is translated, compiled, linked with the appropriate runtime support (see Section 6.2, page 169), and executed, it will print "Hello World" to standard output. The behaviour has a state machine with a single state called "SayHi" that prints the given greeting. States are distinguished from normal C++ code by the "@" symbol preceding them. The symbol is

followed by a label identifying the state, and then arbitrary code to perform whatever that state actually needs to do. A less trivial state machine is shown in the following code fragment:-

```
BEHAVIOUR MakeTea
{
      // member variables
      ...
      // member functions
      ...

      @PlugIn
            if ( numCups >= numPeople && AreBagsAvailable() )
            {
                  FillKettle();
                  SwitchOnKettle();
                  PutTeabagsInCups();
                  NEXT WaitBoil;
            } else NEXT DrinkMilk;
      @WaitBoil
            if ( Boiled() )         NEXT Prepare;
            else                    NEXT WaitBoil;
      @Prepare
            PourWater();
            AddMilkAndSugar();
            // finished
      @DrinkMilk
            cout << "Wouldn't you much prefer some milk?" << endl;
            // finished
};
```

All the variables and functions referred to in the code are assumed to be local variables and helper functions of the behaviour. Note that any code valid within a normal C++ function can be used within a state. There is an extra element allowed, the "NEXT" statement, which specifies the identifier of the next state the robot will enter. Every state is assigned a unique name so that it can be referenced in this way. By default, if no NEXT statement is executed within a state, the state machine will be halted.

When the above state machine is executed, the "PlugIn" state will run first. From there the behaviour may transition to "WaitBoil" or "DrinkMilk". The "WaitBoil" state loops until Boiled() is true, then moves on to "Prepare". Both the "Prepare" and "DrinkMilk" states contain no transitions in the form of NEXT statements, so the state machine will be halted after either of them are executed.

The next section shows how the syntax in the above code fragment is converted to C++ by the Zac Translator.

6.3.2 Implementation of state machines

A state machine within a behaviour is converted to C++ by building a function to implement it as a "switch" statement. The base class from which all behaviours are derived contains a virtual function called "ZAC_Run" which is expected to be overridden to perform the functionality of the behaviour. The function is passed the current state of the behaviour, and returns the next state to which it should transition. Its prototype is as follows:-

```
virtual int ZAC_Run ( int ZAC_state );
```

The Zac Translator generates this function from the state machine, and takes care of informing the Lateral runtime support module of the existence of each behaviour, so that their state machines will be automatically executed (see Section 6.3.3). The code generated by the Zac Translator for this function is a switch statement, with each "case" corresponding to a single state in the state machine. Unique identifiers are generated from all the state names to use as case labels *before* the compiler meets the switch statement. This means that forward references between states will not cause difficulty. To illustrate these ideas, the code generated by the Zac Translator from the state machine example in the previous section will now be examined[50]. Firstly, code is written to set up unique identifiers for use as case labels in the switch statement implementing the state machine.

[50] This code is taken directly from the output of the Zac Translator, with explanatory comments added.

```
enum
{
      ZAC_LINE(MakeTea,PlugIn),      // ZACLine_MakeTea_PlugIn
      ZAC_LINE(MakeTea,WaitBoil),    // ZACLine_MakeTea_WaitBoil
      ZAC_LINE(MakeTea,Prepare),     // ZACLine_MakeTea_Prepare
      ZAC_LINE(MakeTea,DrinkMilk),   // ZACLine_MakeTea_DrinkMilk
      ZAC_LINE(MakeTea,ZAC_STATE_COUNT)
};
```

ZAC_LINE is a trivial macro to create unique identifiers based on its arguments. It is used to make the code in the switch statement easier to read. ZAC_LINE(x,y) simply corresponds to the identifier ZACLine_x_y. An extra identifier of the form:-

```
ZACLine(BehaviourName,ZAC_STATE_COUNT)
```

is automatically added at the end of every identifier list that the Zac Translator creates to give the number of states in that behaviour. Once the identifiers have been set up, the function implementing the state machine can be generated. It is passed the state to run as an argument, and should return the desired next state of the behaviour. It consists of a "switch" based on the state to run, with each case being of the form:-

```
case ZAC_LINE(BehaviourName,StateName):
      // Code from state is copied to here
      break;
```

The code from the state is copied into the case statement almost verbatim. The only special provision that has to be made is for the "NEXT" statement. This should set the state that the function will return as the desired next state of the behaviour. This is done by setting up a local variable ("ZAC_next") and translating

```
NEXT StateName;
```

into

```
ZAC_next = ZACLine(BehaviourName, StateName);
```

The entire function will be as follows:-

```
int ZACProcess_MakeTea::ZAC_Run ( int ZAC_state )
{
      // Local variable to store the next desired state. The
      // default value ZAC_LINE_DEFAULT stops the state machine.
```

```
int ZAC_next = ZAC_LINE_DEFAULT;

// switch to the correct state to run
switch ( ZAC_state )
{
// Code corresponding to the “PlugIn” state
case ZAC_LINE(MakeTea,PlugIn):
      // C++ code from state gets copied directly
      if ( numCups >= numPeople && AreBagsAvailable() )
      {
            FillKettle();
            SwitchOnKettle();
            PutTeabagInCup();
            // “NEXT WaitBoil” is translated into the following
            ZAC_next = ZAC_LINE(MakeTea,WaitBoil);
      } else ZAC_next = ZAC_LINE(MakeTea,DrinkMilk);
      break;

// Code corresponding to the “WaitBoil” state
case ZAC_LINE(MakeTea,WaitBoil):
      if ( Boiled() )   ZAC_next = ZAC_LINE(MakeTea,Prepare);
      else              ZAC_next = ZAC_LINE(MakeTea,WaitBoil);
      break;

// Code corresponding to the “Prepare” state
case ZAC_LINE(MakeTea,Prepare):
      PourWater();
      AddMilkAndSugar();
      // finished
      break;

// Code corresponding to the “DrinkMilk” state
case ZAC_LINE(MakeTea,DrinkMilk):
      cout << “Wouldn’t you much prefer some milk?” << endl;
      // finished
      break;

// Code corresponding to the any invalid state
default:
      // Special return code to indicate an invalid state.
      ZAC_next = ZAC_LINE_NOTSET;
```

```
            break;
      }  //  End of switch on ZAC_state
      // return next state the behaviour should execute
      return ZAC_next;
} // End of ZACProcess_MakeTea::ZACRun
```

As can be imagined, it is easier to write and modify state machines in the syntax that Zac Script allows than it would be to use switch statements directly. Also, now that state identifiers are handled automatically, it is possible to introduce convenient short-cuts into how a state machine can be specified:-

- To transition from one state to the state immediately after it in the list of states, it is useful to be able to issue a "NEXT;" command without specifying the state's identifier explicitly. The fewer times a state is referred to by name, the less code that has to be changed if that state is renamed. Hence the following code is allowed:-

  ```
  @FirstAction
        DoSomething();
        NEXT; // equivalent to writing "NEXT SecondAction;"
  @SecondAction
        DoSomethingElse();
  ```

  When used this way, NEXT is translated as

  ```
  ZAC_next = ZAC_state + 1;
  ```

  i.e. it simply moves on sequentially to the next state. If this is used in the last state of the state machine, it will be halted.

- States often need to loop continuously, waiting for some specific condition to occur. With the Zac Translator, The "@" symbol can be used within a state to refer to the identifier of that state. For example, a loop could be implemented as follows:-

  ```
  @LoopingState
        DoSomethingThatNeedsRepeating();
        NEXT @; // equivalent to writing "NEXT LoopingState;"
  ```

  Writing loops this way means that if the name of the state is changed later, no code in its body is affected.

- Frequently a state machine will have a number of trivial intermediate states that are not

worth naming. However the switch statement in C++ will require the states to have unique names. To get around this, if the "@" state marker is repeated where the name of the state should be, the Zac Translator will automatically substitute a unique identifier for that state. The following code will then be legal:-

```
@@    // No name given to this state
      DoSomethingTrivial();
      NEXT;
@@    // No name given to this state either, but it is
      // distinguished from the previous state for C++
      DoSomethingEquallyTrivial();
      NEXT ImportantState;
```

- Often the logic of a state can be expressed most clearly by specifying a *default* next state that the robot will transition to if no other NEXT statement is executed. It would be attractive to be able to write code like the following:-

```
@@
      DEFAULT NEXT DefaultAction;
      // DefaultAction will be transitioned to unless one
      // of the following conditions succeeds
      if ( condition1 ) NEXT Response1;
      if ( condition2 ) NEXT Response2;
      ...
```

  However, by referring back to how NEXT is implemented, it can be seen that if two NEXT statements are executed, the last one to execute will be the one that actually chooses the next state. Hence the DEFAULT keyword in the above code could simply be omitted, and the code would work as it stands. The DEFAULT keyword does make the logic of the state clearer, so the Zac Translator will accept it and simply treat it as a piece of documentation.

Portions of the state machine example given earlier could have been written using these short-cuts, in the following way:-

```
...
@WaitBoil
      DEFAULT NEXT @;
```

```
                    // This sets the default next state to be
                    // the "WaitBoil" state itself
        if ( Boiled() )
              NEXT; // This will transition to the next state
                    // in the list of states- the anonymous one
                    // directly following

@@                  // Anonymous state
        PourWater();
        AddMilkAndSugar();
        // finished
...
```

As a final detail for completeness, the Zac Translator recognises one special state, the "CONTROL" state. If such a state is included in the state machine, it will be called once every scan cycle (see Section 3.6.1, page 69) through the ZAC_Run function *in addition* to the particular state the behaviour is in at that time. This is useful to allow the behaviour to react to changes in priorities at the rate they occur.

6.3.3 Discussion of initialisation

Thus far it has not been specified how the state machines written in behaviours actually ever come to be executed. The Lateral runtime system must be notified of the existence of each behaviour in some way, so that it may drive their state machines. The Zac Translator handles all the details of performing this notification, so that the programmer who generates the behaviours need never even consider the issue, and can treat behaviours as units of execution in their own right. The Zac Translator generates a single "initialisation function" in each source file to inform the Lateral runtime system of the behaviours contained in that source, and to initialise those behaviours. This function is called automatically on program start-up without the programmer needing to know of its existence. The operations that the initialisation functions perform in effect construct the "Lateral Object Hooks" interface to be discussed in Section 7.7 (page 228).

6.3.4 Specification of connections

A behaviour in Lateral has a set of input and output connections that it can use for interacting with other behaviours (see Section 3.3.1.3, page 50). In Zac Script, a special syntax is introduced to make it easier to set up these connections. The form of the syntax is as follows:-

```
BEHAVIOUR Example
{
      INPUT ( ConnectionType, ConnectionName, Source );
      OUTPUT ( ConnectionType, ConnectionName, Target );
      ...
};
```

Input and output connections are special objects belonging to the behaviour they appear within. As for all variables in C++, they have a type and a name. The type refers to the nature of the data the connection can carry. The name allows the connection to be accessed within the behaviour's state machine and helper functions, just like any member variable. Beyond the attributes of normal member variables, however, connections also have information about what they are attached to. Input connections may read from a specified source (another connection), and output connections may write to a specified target (also another connection). As an example, consider the following two behaviours, "ShowLevel" and "SetLevel".

```
BEHAVIOUR ShowLevel {
      // input connection called "in_level", carrying integer data,
      // and left unattached
      INPUT ( int, in_level, NULL );
      @display
            if ( in_level.Delta() ) // check if input has changed
            {
                  // display value stored in input
                  cout << in_level.Value();
            }
            NEXT @;                 // repeat state forever
};

BEHAVIOUR SetLevel {
      // output connection called "out_level", carrying integer data,
      // and attached to the input of ShowLevel
      OUTPUT ( int, out_level, ShowLevel.in_level );
```

```
      @start
            out_level.Set ( 0 );    // Initialise the output to zero
            NEXT;
      @set
            // Increase the output by one, and loop forever
            out_level.Set ( out_level.Value() + 1 );
            NEXT @;
};
```

ShowLevel has a single input connection called "in_level", and SetLevel has a single output connection called "out_level". This output connection of SetLevel is attached to the input connection of ShowLevel[51]. The state machine in SetLevel repeatedly increments its output, while ShowLevel displays its input whenever it changes. Because the output of SetLevel is attached to the input of ShowLevel, this results in ShowLevel displaying the incrementing values of SetLevel's output. This could also be achieved by attaching ShowLevel's input to SetLevel's output, but this is done less commonly because it requires the behaviour being controlled to "know" about the behaviour controlling it (see Section 3.3.1.3, page 50).

Notice that the connections are accessed within their respective behaviours by name, just as normal variables are. When the output of SetLevel is being attached to the input of ShowLevel, it identifies the input as "`ShowLevel.in_level`". In general an input or output connection of a behaviour can be referred to in this way, as:-

"<BehaviourName>.<ConnectionName>"

All connections have a set of operations that can be performed on them. The full set of operation is listed in Appendix C3. The ones used here are given in Table 6-1.

Table 6-1

[51] Hence out_level becomes a *secondary source* for in_level (see Section 3.3.1.4, page 52). If further outputs were attached to in_level, it would apply Lateral's priority rules to determine which to read from (see Section 3.3.1.5).

Operation	**Action**
Set	This places the data from its argument in the connection (the assignment operator can be used for this as well).
Value	This returns the data in the connection (the type cast operator for the appropriate data type can also be used for this).
Delta	This returns true if the data in the connection have changed since the last time the function was called.

It is possible for connections to exist that are not owned by a behaviour. These are called *intermediate* connections, and are specified as follows:

```
CONNECT ( ConnectionType, ConnectionName, Source, Target );
```

Intermediate connections are specified outside of any behaviours, at file scope. Again, they have a type and a name, and both a source and target connection can be specified. They are referred to by their name, without any qualification, just like a global variable in C++.

6.3.5 Implementation of connections

All connections are implemented as objects derived from the ZACMeshLink base class (see Section 3.6.4, page 76). This class gives support for all the operations that the Lateral runtime module needs to manipulate connections. It also defines the operations that can be applied to connections when they are accessed within behaviours. A full list of these operations is given in Appendix C3.

Because of the similarity of connections to normal variables in C++, it would be relatively easy to use them directly without any special syntax. The only difficulty is how to set the source and target of the connections. In C++ classes, member variables can only be initialised in the class's constructor. Therefore input and output connections would have to be declared first as member variables, and then have their source and targets set separately in a constructor. It was found to be easier to understand the relationships between behaviours if the declarations and attachment information for their connections were side by side at the start of a behaviour, rather than being dispersed. For this reason, the special syntax described in the previous section was used. Connection declarations and attachments were kept together, with the Zac Translator automating the "donkey work" of splitting them up for the C++ compiler.

6.3.6 Running the translator

The Zac Translator is the tool developed for converting Zac Script source code to pure C++ code. It was constructed using FLEX[52] and a recursive-descent parser. The translator works by building several files from a single source file, corresponding to the declarations, definitions and initialisations necessary to implement the Lateral constructs in C++. These files are then concatenated into a single C++ translation of the source. The intermediate files appear only because by writing the translator this way it can be made single-pass. For example, when translating a state machine, a list of identifiers for all the states must appear in the C++ translation before any code for the actual states, otherwise forward references will not be allowed by the compiler. But the translator does not have a list of all the states until it has finished parsing the behaviour, so it will not be able to insert this list until it is too late. To get around this, the translator places the declarations for the states of all the behaviours in one file, and the actual code for the states in another, and then after scanning the entire source code it will simply append the file containing the code to the end of the file containing declarations. A similar argument holds for the generation of a file containing initialisation code. The declarations, definitions, and initialisations combine to form a single C++ source file. The translator also generates a file giving declarations for any intermediate connections in the source, and this forms a header file. This header file allows intermediate connections to be attached to by behaviours outside of the file in which they were created.

[52] "Fast LEXical analyser"- this general utility for performing lexical analysis of files is ideal for building scanners for parsers. FLEX or the earlier LEX are available under most distributions of UNIX.

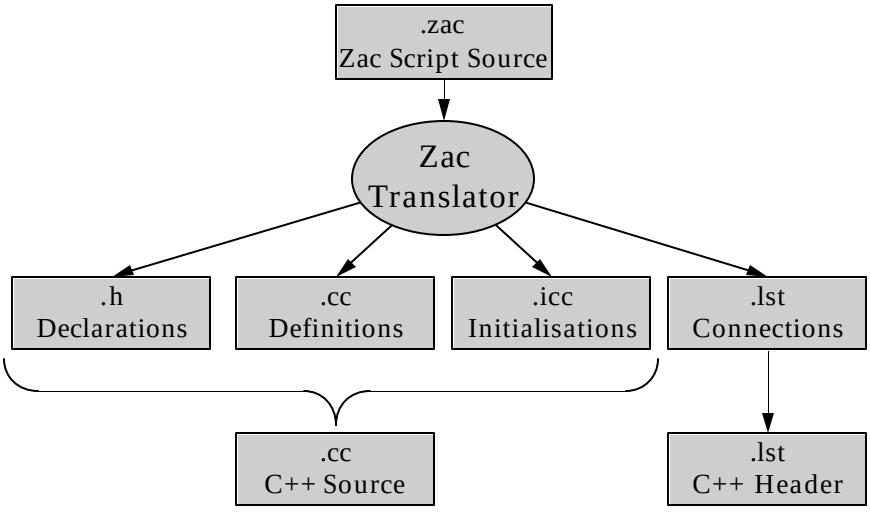


Figure 6-2: C++ translation of Zac Script source

Figure 6-2 outlines the stages involved in translating Zac Script source to C++. The contents of the various files are detailed in Table 6-2.

Table 6-2

Stage	File	Contents	Description
Source	.zac	Zac Script source	Source file written in C++ with syntax extensions supporting Lateral constructs.
Intermediate	.h	Declarations	Declarations generated for states, behaviours, connections, etc.
	.cc	Definitions	Body of state machines, behaviours, definitions of connections, etc.
	.icc	Initialisations	Initialisation functions for behaviours, automatic implementation of the Lateral object hook interface.
	.lst	Intermediate Connections	A list of declarations for the intermediate connections specified in the source file.
Target	.cc	C++ source	The translated version of the source file, now in pure C++.
	.lst	C++ header	A copy of the list of declarations for the intermediate connections specified in the source file. Suitable for use as/inclusion in a header file.

6.4 Syntax of Zac Script

The chapter so far has given an overview of the important components of Zac Script and how they are implemented. In this section a complete systematic description of the syntax of Zac

Script is given. The syntax is first presented in EBNF notation[53], and then each major element of the grammar is described in turn.

It might seem that, since Zac Script extends on C++, a parser for Zac Script would also need to parse C++. However the full grammar of C++ is very complex, and parsing it is difficult, so no attempt was made to do so- it would certainly have been more trouble than the language extensions were worth. Instead the grammar and keywords of Zac Script were carefully chosen so that the translator could "skim" over C++ source code, pick out the constructs it is responsible for, and parse the bare minimum necessary to convert them to C++. For example, the character "@", used as the state identifier, is not used at all in C++, so its presence unambiguously indicates the start of a state machine.

The full grammar of Zac Script is given in Table 6-3. Compilers generally divide their source code into units such as identifiers, numbers, white-space, etc. The Zac Translator does the same when it is parsing the constructs it is responsible for, but otherwise it treats whole sections of C++ code as "white-space" which it just passes on unmodified in the translation. Such a block of C++ is called a "<c-unit>" in the table below, and it is bounded by any keyword from Zac Script, or alternatively any element within C++ that indicates the end of a construct, such as a closing parenthesis or closing brace that was not opened within the unit. A "<c-term>" is the same as a <c-unit>, except it is used when parsing elements of an argument list, so it is also bounded by a comma. The translator is free-format; however provision is made to allow formatting from the source to be copied (with modifications when appropriate) to the translation, so that the translation will be human-readable.

Table 6-3

[53] Extended Backus Naur/Normal Form, a tool for describing context-free grammars.

Syntax of Zac Script in EBNF notation		
<Program>	:==	{ <Lateral-unit> | <c-unit> }
<Lateral-unit>	:==	<Behaviour> | <Connection> | <Object>
<Behaviour>	:==	"BEHAVIOUR" <Identifier> "{" <Behaviour-body> "}"
<Behaviour-body>	:==	{ <Behaviour-unit> } [<State-machine>]
<Behaviour-unit>	:==	<c-unit> | <Behaviour-connection>
<Behaviour-connection>	:==	(Input>|<Output>|<Input-status>|<Output-status>) ";"
<Input>	:==	"INPUT" "(" <c-term> "," <c-term> "," <c-term> ")"
<Output >	:==	"OUTPUT" "(" <c-term> "," <c-term> "," <c-term> ")"
<Input-status>	:==	"INPUT_STATUS" "(" <c-term> "," <c-term> "," <c-term> ")"
<Output-status>	:==	"OUTPUT_STATUS" "(" <c-term> "," <c-term> "," <c-term> ")"
<State-machine>	:==	<State> { <State> }
<State>	:==	"@" <Identifier> { <c-unit> | <Next> }
<Next>	:==	["DEFAULT"] "NEXT" [<Identifier>] ";"
<Object>	:==	"OBJECT" "(" [<c-term> { "," <c-term> } ")" ";"
<Connection>	:==	("CONNECT" | "CONNECT_STATUS") "(" ... ")"...

The major elements of the grammar are now examined in turn.

6.4.1 Program syntax

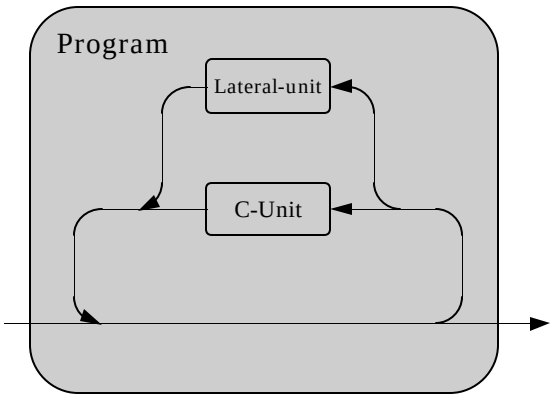


Figure 6-3: Syntax of a Zac Script program

Zac Script source code consists of blocks of pure C++ code ("c-units") and blocks of code written in an extended syntax representing Lateral elements such as connections and behaviours (see Figure 6-3). C++ elements are not parsed, simply copied verbatim to the translation. Only units of code representing Lateral elements are parsed.

6.4.2 Syntax of Lateral elements

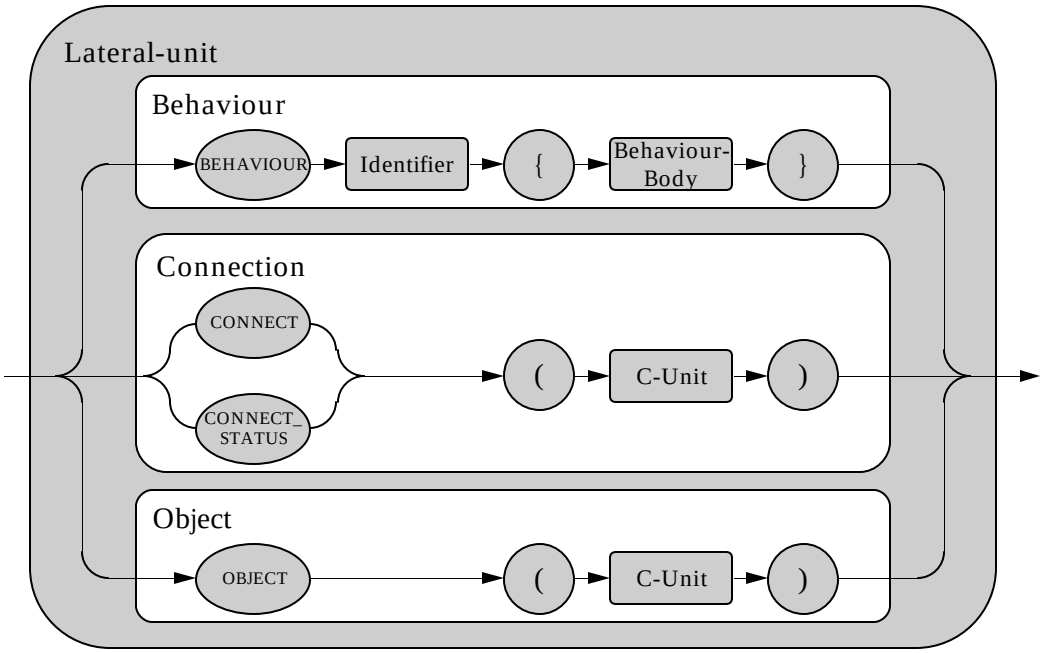


Figure 6-4: Syntax of Lateral elements in Zac Script

The three main types of Lateral elements are shown in Figure 6-4. These are :-

- **Behaviours**- These represent structures for implementing behaviours of the robot. They start with the keyword BEHAVIOUR, followed by the body of the behaviour delimited with braces. This construct was discussed in Section 6.2.

  Connections - These represent *intermediate* connections that do not belong to any behaviour. Input and output connections appear inside the body of behaviours and are parsed differently. Intermediate connections are specified with the keyword CONNECT, followed by arguments inside parentheses indicating their type, name, and target or source as discussed in Section 6.3.3. There is a variant on normal connections called "CONNECT_STATUS" which does not carry priority information and so does not have the overhead associated with doing so. This construct was introduced for the sake of efficiency only, and will not be described in detail as it is simply a restricted version of normal connections. The structure of the arguments to CONNECT is not parsed, because the translation can be arranged so that the C++ compiler itself will check the arguments in the translated version of the construct. Whenever possible, checking argument counts and

types is left to the C++ compiler to keep the translator simple, and because the compiler will be able to generate more informative error messages[54].

- **Objects**- This special construct is not part of Lateral, and was introduced for purely pragmatic reasons. A statement like:

```
OBJECT ( int, status, 1 );
```

is directly equivalent to simply writing

```
int status = 1;
```

except that in the first case the Lateral runtime system takes over responsibility for performing the initialisation. This has the practical advantage that the control system can be reset to its original state by the runtime system, without having to stop and restart the whole program. For the robot the work in this thesis was implemented on, restarting a program could only be done by downloading the program to the robot again- a slow operation, hence the usefulness of the OBJECT statement.

This construct is specified with the keyword OBJECT, followed by a type, variable name, and initial value for that variable inside parentheses. As for connections, the structure of the arguments is not checked or enforced by the Zac Translator, since that can be done by the C++ compiler working with the translated version.

[54] Errors generated by the compiler while compiling the C++ translation are referred back to the appropriate line in the original source. This is possible because the Zac Translator automatically inserts #line directives as it is translating. These directives give the compiler the information it needs to generate error messages that are relevant to the Zac Script source rather than the C++ translation.

6.4.3 Syntax of behaviour body

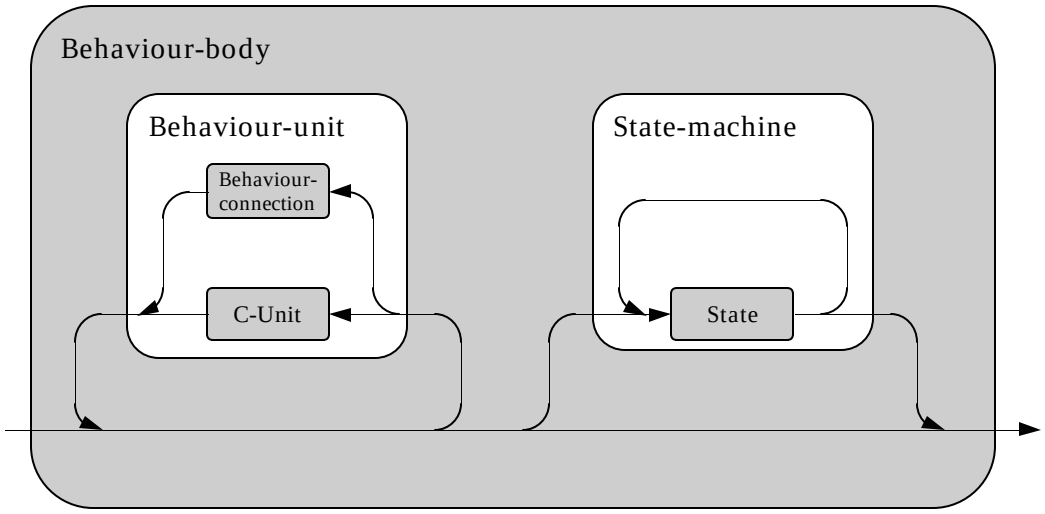


Figure 6-5: Syntax of the body of a behaviour

The body of a behaviour is similar to that of a C++ class (see Section 6.2). Any construct allowed in a C++ class is also allowed in a Behaviour- member variables, member functions, member classes etc. (see Figure 6-5). There are also further special elements allowed within the body of a behaviour - input and output connection to other behaviours, and a list of states specifying the behaviour's state machine, as described in Section 6.3.1.

6.4.4 Syntax of input and output connections

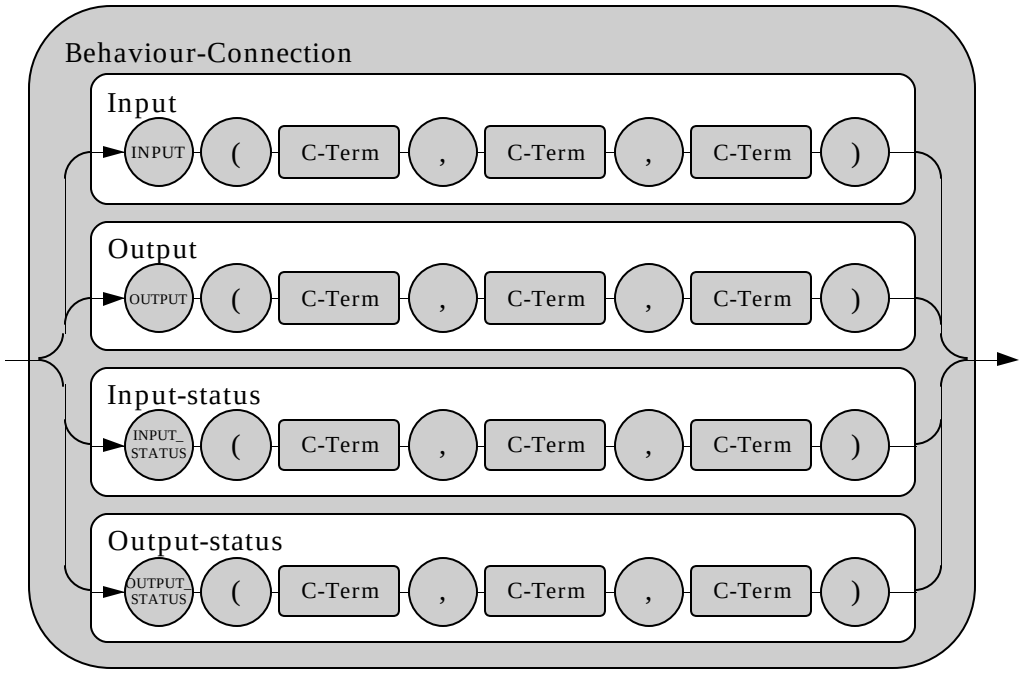


Figure 6-6: Syntax of input and output connections

Every behaviour can have a set of input and output connections that it uses to communicate with other behaviours. These connections were discussed in 6.3.3. Although in Figure 6-6 the arguments the connections take are shown for reference, these are not in fact parsed at this stage- rather they are passed on to the C++ compiler to be checked. As for intermediate connections, there are alternative INPUT_STATUS and OUTPUT_STATUS connections that do not carry priority information. These construct were introduced for the sake of efficiency only, and will not be described in detail as they are simply very restricted versions of normal connections. As their name suggests, they generally carry status information that does not need an associated priority.

6.4.5 Syntax of states

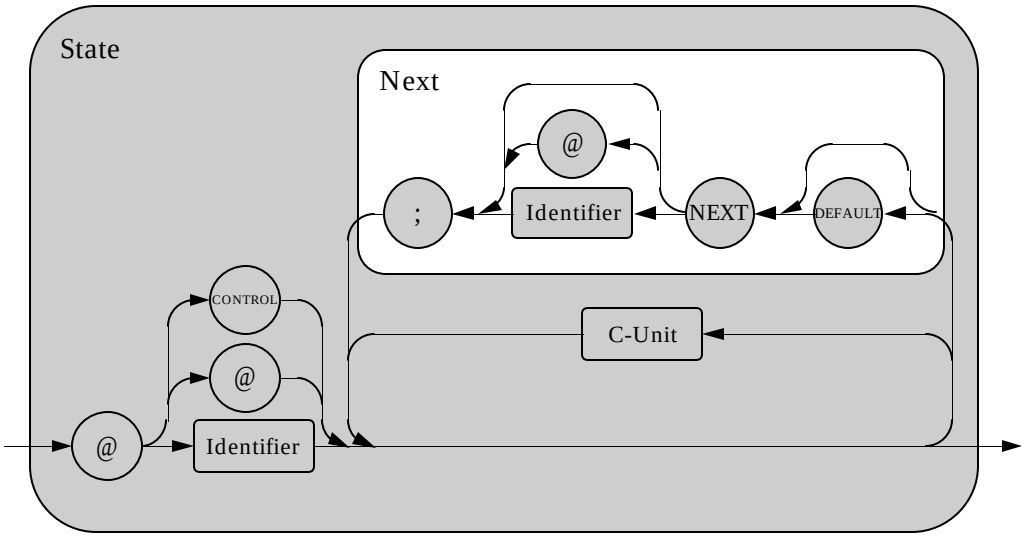


Figure 6-7: Syntax of a single state in a state machine

The behaviour's state machine can be detected by the appearance of the "state keyword", which is the "@" symbol (see Figure 6-7). This keyword is followed with a state identifier- or the "@" can be repeated for an anonymous state, or the "CONTROL" keyword is used for the special control state (see Section 6.3.1). Any C++ code that can appear within a function can be placed within a state. A statement starting with "NEXT" or "DEFAULT" is parsed to give a command indicating a state transition.

This concludes the formal specification of Zac Script. An extended example of its use will now be given.

6.5 Extended Example- the Robot Wheelbarrow

In the remainder of this chapter an extended example of the use of Zac Script is presented, clarifying how it is applied in practice for specifying a robot's control system in terms of behaviours and connections. The target behaviour for the robot in this example is to act as a "robot wheelbarrow". The robot is taken to have four proximity sensors- one to the front, one at the rear, and one to either side. To start the wheelbarrow moving, the operator simply approaches its rear sensor. As soon as this occurs, the robot will move in the direction it is facing until it detects an obstacle with its front sensor, at which point it stops. The robot can be turned left and right while it is moving by approaching the sensor on the opposite side to the

desired direction- as if it was being physically pushed. To stop the robot, the operator can simply stand in front of it. Once the robot stops, it will not start again until the operator approaches its rear sensor again.

This overall behaviour will be implemented by combining two simpler behaviours, one which simply moves forward, and one which turns. The two behaviours will be developed individually as a "MoveForward" and "Turn" behaviour, and then a higher level behaviour called "Push" will be constructed that uses them both to give the required overall behaviour of the robot. Finally, a further behaviour called "ImprovedPush" will be built to demonstrate the use of Lateral's sponsorship system.

For the purposes of this example, the following services will be assumed available:-

- A low level "Motor" behaviour is assumed present, with two input connections- "speed" to control the rate at which the robot moves forward, and "nudge" to control the rate at which it turns[55]. The units of these inputs are arbitrary. For this example, 0 is taken as stationary, and 10 as the robot's highest speed. The robot turns right for positive nudge values, and left for negative ones, with the magnitude of the nudge value indicating the rate at which the turn is made. Again, 10 is taken as the maximum magnitude.
- Access to the robot's sensor data is assumed available through simple functions, as described in Table 6-4. It is not necessary for sensor data to be transmitted along connections, since there should never be any need to subsume or override it.

[55] For the robot on which this work was implemented, these inputs were combined in a single structure. The two components are kept separate here for simplicity.

Table 6-4

Sensor	Description
leftSense( )	This returns a high value when there is an object close to the robot on the left. The units are arbitrary. In this example, 50 or greater is taken as high.
rightSense( )	This returns a high value when there is an object close to the robot on the right.
forwardSense( )	This returns a high value when there is an object in front of the robot.
reverseSense( )	This is high when there is an object to the rear of the robot.
closestSense( )	This is high if there is an object near the robot in any direction. It is derived by taking the maximum of all the other sensors.

Each of the behaviours will be implemented in turn, and then their interaction will be analysed in terms of priorities. The state machines of the behaviours were made as simple as possible so as not to distract attention from the main focus of the example, and could in many cases be made more efficient and robust.

6.5.1 The "MoveForward" Behaviour

This behaviour is responsible for starting the robot moving forward if the operator approaches it from behind, and stopping it if it meets an obstacle or if the operator approaches it from the front[56] (see Figure 6-8). A simple state machine to achieve this behaviour is described in Table 6-5.

Table 6-5

State	***Action***
Wait	The robot remains stationary, waiting for the operator to approach it.
Move	The robot moves forward, watching out for anything in front of it.

[56] It is important that the robot does not need to behave differently when it meets an obstacle compared to when it meets the operator, since the robot cannot distinguish between different objects obstructing its sensors.

The behaviour has a single output that drives the speed input of the Motor behaviour. For the sake of this example, it is also given an input called "power" that chooses what fraction of its highest speed the robot should advance at when it is in motion.

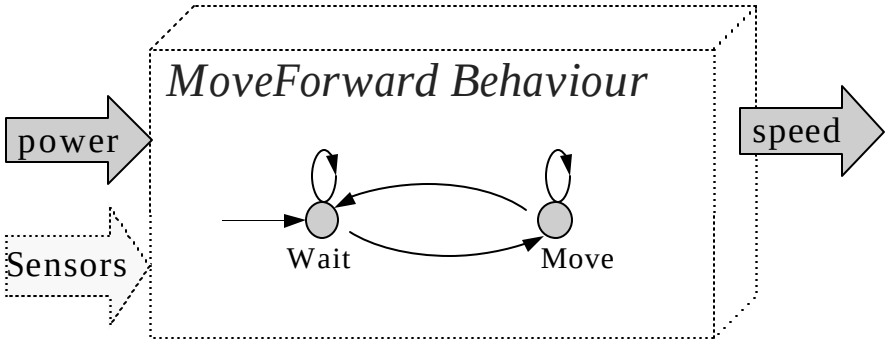


Figure 6-8: The MoveForward Behaviour

The code for such a behaviour in Zac Script would be as follows:-

```
BEHAVIOUR Push
{
      INPUT ( float, power, NULL );       // "power" input
      OUTPUT ( int, speed, Motor.speed ); // "speed" output

      @Wait                   // "Wait" state
            speed.Set ( 0 );  // No forward movement in this state
            DEFAULT NEXT @;   // State loops by default
            // Robot starts moving if there is nothing in front of
            // it, and it detects something on its rear sensor
            if ( reverseSense() >= 50 && forwardSense() < 50 )
                  NEXT MoveForward;       // Robot starts moving

      @Move                   // "Move" state
            // Robot moves at desired fraction of maximum speed
            speed.Set ( 10 * power.Value() );
            DEFAULT NEXT @;   // State loops by default
            // Robot stops moving if it detects something in
            // front of it
            if ( forwardSense() >= 50 )
                  NEXT Wait;  // Robot reverts to staying still
};
```

The "power" input and the "speed" output are specified in the syntax discussed in Section 6.3.3. The "power" input carries floating point data (a fraction), and is left unattached. The "speed" output carries an integer speed value, and is attached to the corresponding "speed" input of the Motor behaviour assumed to be present.

The state machine is specified as described in Section 6.3.1. There are two states, "Wait" and "Move". The Wait state sets the robot's speed to zero through the behaviour's output to the Motor behaviour. It then loops until there is a significant proximity reading at the rear of the robot, and no significant proximity reading to its front, at which point the state machine transitions to "Move". "Move" sets the speed of the robot to the desired fraction of its maximum value, and keeps the robot advancing until an obstacle is detected in front of the robot.

6.5.2 The "Turn" Behaviour

The "Turn" behaviour is even simpler than "MoveForward", because it is purely reactive - that is, it requires no state information. It simply checks the robot's sensors, and turns depending on whether it detects something to the left or the right (see Figure 6-9). It has an output connection to the "nudge" input of the Motor behaviour to control the rate at which the robot turns. For the sake of this example, it takes a "power" input connection to determine whether it is attracted or repelled by objects. The input is positive if it is attracted to turn towards objects it detects, and negative if it is repelled. The magnitude of the input indicates at what fraction of the maximum rate the robot should turn.

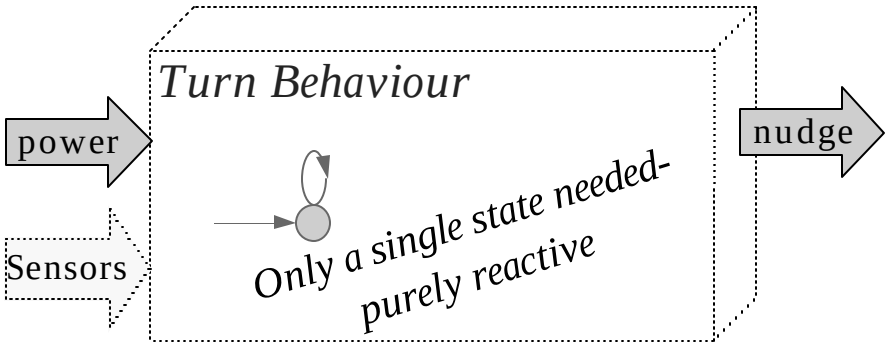


Figure 6-9: The Turn Behaviour

Reactive behaviours are implemented as a single looping state as follows:-

```
BEHAVIOUR Turn
{
      INPUT ( int, power, NULL );         // "power" input
      OUTPUT ( int, nudge, Motor.nudge ); // "nudge" output
      int direction;                      // a local variable

      @@    // Anonymous state- no point naming it, with
            // only one state being present

            // Robot should turn if it senses something
            // to the left or the right
            if ( leftSense() >= 50 || rightSense() >= 50 )
            {
                  // Choose correct direction to turn
                  if ( leftSense() > rightSense() )
                        direction = -1;   // Counterclockwise turn
                  else
                        direction = +1;   // Clockwise turn
                  // Actually turn the robot
                  nudge.Set ( 10*direction*power );
            }
            else nudge.Set ( 0 );
            NEXT @;           // Keep looping on same state
};
```

The robot repeatedly checks its left and right sensors, and if they are above a certain threshold, it turns either towards or away from the direction of the closest object.

6.5.3 The "Push" Behaviour

This behaviour combines the two behaviours developed above, MoveForward and Turn, into a single behaviour that implements the desired functionality of the wheelbarrow robot as described earlier (see Section 6.5). For the sake of this example, it also adds some extra functionality rather than just being a simple combination of MoveForward and Turn. If the robot has been moving for over a certain threshold time without anything being sensed to its

rear, the Push behaviour will stop the robot moving. This is to prevent the robot from "running away" too far from its operator.

A simple state machine to exhibit this behaviour is described in Table 6-6.

Table 6-6

State	***Action***	***Sponsorship to..***
StayStill	The robot is held completely stationary because the operator has not approached the robot recently.	None
Operate	The operator has approached recently, so turning and moving forward is allowed.	MoveForward, Turn

The behaviour has two outputs, one to each of the two behaviours it uses (see Figure 6-10). The output to the MoveForward behaviour chooses the speed at which the robot moves, and the output to the Turn behaviour sets the rate at which it can change direction[57]. These connections also act as channels along which the Push behaviour can sponsor MoveForward and Turn to operate on its behalf. The cut-off time for when the robot should stop moving is set by an input to the behaviour.

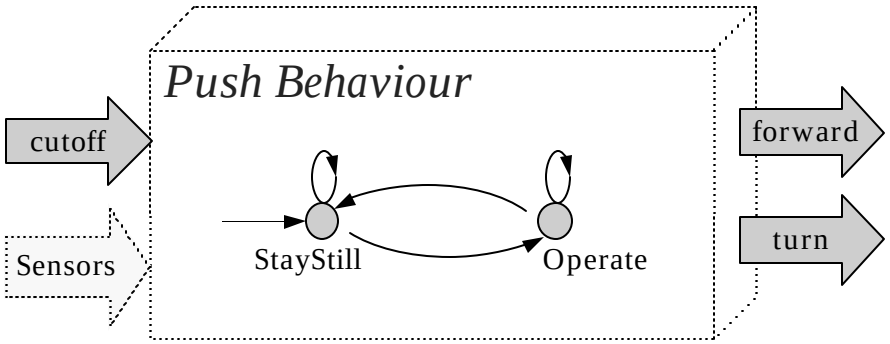


Figure 6-10: The Push Behaviour

The code to implement this behaviour would be along the lines of the following:-

```
BEHAVIOUR Push
{
      // Input specifying how long the robot may move on its own
      INPUT ( int, cutoffTime, NULL );
      // Output to "power" input of MoveForward behaviour
      OUTPUT ( int, forward, MoveForward.power );
```

[57] It also ensures that the robot turns *away* from the direction the operator approaches from by making the output to the Turn behaviour negative.

```
      // Output to "power" input of Turn behaviour
      OUTPUT ( int, turn, Turn.power );

      @StayStill
            // "forward" chooses what fraction of max speed the robot
            // will move at when going forward- here 100%
            forward.Set ( 1 );
            // "turn" chooses what fraction of max speed the robot
            // will turn at when going left or right- here 50%
            // Sign indicates that robot turns away from operator
            turn.Set ( -0.5 );
            // No priority carried on the "forward" output
            forward.SetRelPriority ( 0 );
            // No priority carried on the "turn" output
            turn.SetRelPriority ( 0 );
            DEFAULT NEXT @;   // State loops by default
            // If operator approaches from behind, turning and
            // moving forward are enabled in the Operate state
            if ( reverseSense() >= 50 ) NEXT Operate;

      @Operate
            // Full priority of behaviour now carried by "forward"
            forward.SetRelPriority ( 1 );
            // Full priority of behaviour also carried by "turn"
            turn.SetRelPriority ( 1 );
            DEFAULT NEXT @;   // State loops by default
            // Robot must stop after specified cut-off time
            if ( StateTimeSec() >= cutoffTime )
                  NEXT StayStill;
            // Every time the robot is "pushed" by the operator, it
            // can start timing again from zero
            if ( reverseSense() >= 50 || leftSense() >= 50
                        || rightSense() >= 50 )
                  ResetStateTimer();
};
```

Some new functions are used here. The "SetRelPriority( )" function sets the fraction of a behaviour's priority an output connection may carry on to whatever it is attached to. "StateTimeSec( )" returns the time in seconds for which a given state has been executing

continuously. "ResetStateTimer( )" sets the timer that tracks this interval back to zero. These are inherited from the base class of all behaviours (see Appendix C3 for a full list).

This behaviour has two states, one in which it allows the behaviours it uses to function without any interference, and one where it effectively "turns them off". In the "Operate" state, the behaviour's two outputs are set to carry the full priority of the behaviour itself. Hence if the Push behaviour is at a high priority, the MoveForward and Turn behaviours will also have the same high priority. However, in the "StayStill" state, the outputs of the behaviour are made to carry none of the priority of the "Push" behaviour. Hence MoveForward and Turn will have no source of priority at all. When the priority of a behaviour falls to zero, then outputs from that behaviour are ignored- so the robot will not move forward or turn.

6.5.4 The "ImprovedPush" Behaviour

As a simple example of the use of Lateral's sponsorship system, an additional behaviour that extends on the "Push" behaviour is given here. The Push behaviour stops the robot's motion entirely- it does not allow the robot to turn when it is not moving forward. It might be desirable to change this. If the robot were allowed to turn while stationary, then the operator could simply approach the robot along the direction he/she wishes the robot to move, and the robot would turn appropriately and then move forward. For example, if the operator approaches from the left, the robot will start turning right- and continue to do that until the operator is in front of its rear sensor rather than the left sensor, at which point the MoveForward behaviour will start the robot advancing. An analogous argument applies if the operator approaches from the right. Of course, if the operator approaches from the front, the idea breaks down- but it is sufficient for this example. This behaviour can be implemented very simply. A suitable state machine for it is shown in Figure 6-11, and will be explained in a moment.

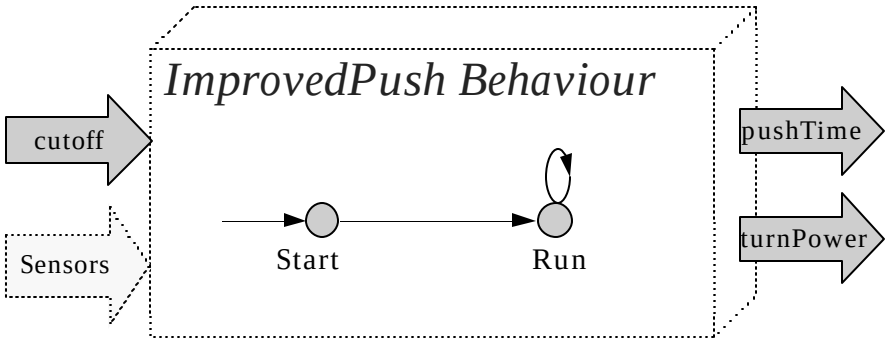


Figure 6-11: The ImprovedPush Behaviour

The behaviour takes the same "cutoffTime" time input as the "Push" behaviour, and simply passes it along on an output to "Push". It also has an output to the "Turn" behaviour. Code to implement this behaviour would be along the lines of the following:-

```
BEHAVIOUR ImprovedPush
{
      INPUT ( int, cutoffTime, NULL );
      OUTPUT ( int, pushTime, Push.cutoffTime );
      OUTPUT ( int, turnPower, Turn.power );

      @Start
            // Cutoff time passed on to "Push"
            pushTime.Set ( cutoffTime );
            // Turning speed output set to 75% repulsive
            turnPower.Set ( -0.75 );
            // Full priority of behaviour carried to "Push"
            pushTime.SetRelPriority ( 1 );
            // Half priority of behaviour carried to "Turn"
            turnPower.SetRelPriority ( 0.5 );
            NEXT;             // Goes to "Run" state
      @Run
            DEFAULT NEXT @;   // Loops indefinitely
};
```

The behaviour is quite simple. It passes on its full priority to Push, and half its priority to Turn. While Push is sponsoring MoveForward and Turn at full priority, this behaviour's connection

to Turn will be ignored[58]. But when Push stops sponsoring those behaviours, the connection to Turn from ImprovedPush will provide that behaviour with an alternate source of priority, so it will remain in action.

All the connections between the Wheelbarrow behaviours are shown in Figure 6-12. Note that, as the above discussion suggests, Push and ImprovedPush compete for control of Turn through its "power" input.

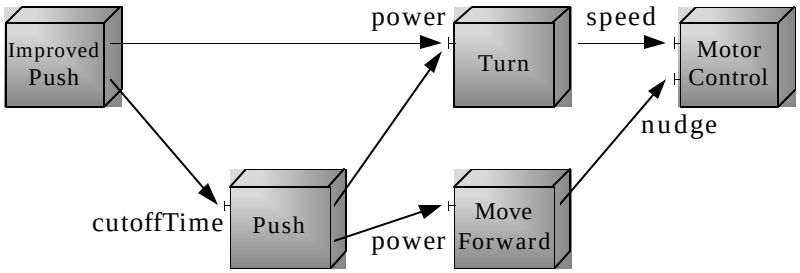


Figure 6-12: Behaviours for the "Wheelbarrow" example

Figures 6-13 and 6-14 illustrate the operation of ImprovedPush in terms of priority planes. The connection from ImprovedPush to Push carries the ImprovedPush behaviour's full priority. When Push is in its "Operate" state, it passes that priority on to both MoveForward and Turn. When this happens, ImprovedPush fails to control Turn because it assigns it less sponsorship. ImprovedPush could of course choose to lower its sponsorship to Push and increase its sponsorship to Turn if it was important that it gain control of Turn- but in this case, it does not need to.

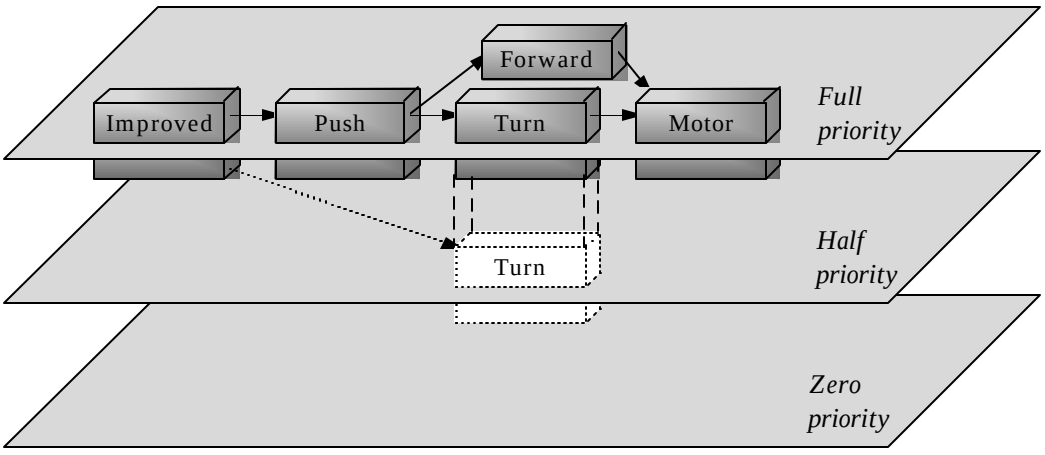


Figure 6-13: Priorities while Push behaviour is sponsoring Turn and MoveForward

[58] The robot will turn at the rate specified by the input from Push (-0.5) rather than the rate specified by

When Push is in its "StayStill" state, it stops sponsoring MoveForward and Turn. In the absence of ImprovedPush, these behaviours would now fall to zero priority and no longer have any effect on the robot. The MoveForward behaviour will in fact do so. However, the connection from ImprovedPush to Turn now becomes an alternative source of priority for Turn, so it remains active when the robot is at a stop, as desired. Turn is now being controlled by ImprovedPush rather than Push[59].

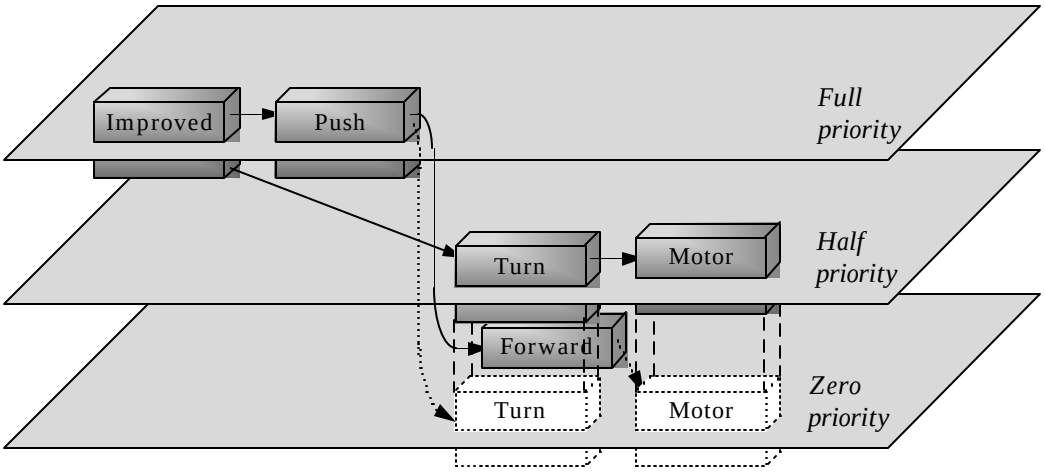


Figure 6-14: Priorities when Push behaviour has stopped sponsoring Turn and MoveForward

This concludes the extended example of the use of Zac Script for specifying behaviours and connections. The set of behaviours described in Chapter 5 were implemented using Zac Script in a manner completely analogous to these much simpler "wheelbarrow" behaviours.

6.6 Limitations

Zac Script proved a useful tool when the work described in this thesis was being implemented, particularly because it allowed the mapping between the Lateral constructs and their C++ equivalents to be altered without having to modify control system source code that had been already written. This pragmatic advantage was the main reason it was developed. Its use as a general-purpose tool for encoding Lateral constructs was considered important, but of

ImprovedPush (-0.75).

[59] The robot will turn at the rate specified by the input from ImprovedPush (-0.75) rather than the rate specified by Push (-0.5).

secondary consideration, since the Lateral architecture is very much exploratory in nature and doubtless its constructs themselves would evolve and change considerably in any future work[60]. Hence it was unreasonable to develop Zac Script to a level where the effort put into its development exceeded the time it saved in easing the maintenance of source code. Therefore a number of limitations of Zac Script in its current form were considered acceptable for the sake of ease of implementation. The most important is that, since Zac Script is not a complete language in itself, some knowledge of how it is being translated to C++ is needed before it can be completely understood. The semantics of Zac Script are also inelegant in places because of difficulties translating to C++. For example, input and output connections are translated to *public* member variables by the translator so that they can be attached to from other behaviours- but this also means they could be accessed in other undesirable ways from these behaviours. Neither of these difficulties detract from the use to which Zac Script was put- maintaining source code- but *would* detract from its use as a general-purpose tool for encoding Lateral constructs.

6.7 Summary

This chapter introduced "Zac Script", a set of extensions to the C++ programming language that make it easier to express Lateral constructs in a program. With Zac Script, control code can be written in a form that includes special syntax for behaviours and connections, and then be translated automatically to C++ for compilation. The full motivation for introducing these extensions was given, with the most compelling reasons being that they make it easier to alter the source code at a later stage, and they allow the mapping between Lateral constructs and their C++ translations to be changed without having to modify pre-existing control system source code. The most important extensions provided are for behaviours and connections, and these were described in some detail- both in terms of their use and how they were translated to C++. Then the complete syntax of Zac Script was examined formally. An extended example of the use of Zac Script was given, to demonstrate how behaviours such as the ones described in Chapter 5 can actually be implemented. Finally, the limitations of Zac Script were

[60] Also, practically speaking, it is currently rare for a robot architecture to be used in more than a handful of projects.

discussed, concluding that it was useful for simplifying the maintenance of source code, but not suitable in its current form as a general-purpose tool for encoding Lateral constructs.

7. Implementation

The work presented in this thesis was implemented in both "embodied" and simulated form, using a miniature "Khepera" robot [[28]] and its simulator. This chapter examines how the robot's control system was structured, and how the Lateral architecture and the sentry application were interfaced to the robot.

The chapter starts with an overview of the strategy taken in designing the robot's control system. A discussion is given of the differing functionality of the physical and simulated robot, and how those differences can be reconciled. The control system is then examined systematically in terms of the interfaces between the modules within it. These interfaces are arranged to minimise the dependencies between modules. Finally, supervision tools that interact with and monitor the control system, but are not part of it, are presented.

7.1 Overview

For the purposes of implementation, the robot's control system was divided into three separate sections:-

- **The robot kernel**- This is concerned with interfacing to the robot.
- **The Lateral runtime module**- This provides support for the Lateral robot architecture.
- **The "user" control system**- This section of the control system implements the specific behaviours of the robot for a particular application (such as the sentry behaviours described in Chapter 5).

This partitioning of the control system was useful to do because it made the implementation re-usable. If the work were to be implemented on a different robot at a later date, only the kernel would need to be modified- assuming the new robot could provide at least the same functionality as the current one. Similarly, if the behaviours of the robot were changed, only the "user control system" section would require modification.

This chapter will discuss how the various sections of the control system interact with each other. The Lateral architecture and the sentry application have already been described in chapters 3 and 5 respectively, so only the robot kernel need be examined in detail here. Because both a physical and simulated version of the robot were used, it was useful to

structure the kernel carefully so that the minimum amount of code had to be rewritten for the two versions of the robot. An abstraction called the "Common Robot" was developed to embody the functionality both versions of the robot have in common, and to abstract away from their differences. Two versions of a module implementing the common robot abstraction were developed, one for each version of the robot. Both versions provided the same interfaces to the rest of the system, and hence any dependence on which form of the Khepera robot was in use was isolated to this module.

The services implemented by the common robot were kept as simple as possible, while still allowing complete control of the robot. This minimised the amount of code that had to be written specially for each version of the robot. To provide more sophisticated control of the robot, a "Logical Robot" module was introduced. This enhanced the services of the common robot module by, for example, implementing obstacle avoidance "reflexes". Only one version of this module was needed, since it was implemented entirely from the services provided by the common robot module. The common robot and the logical robot modules together form the robot's kernel through which the rest of the control system interacts with the robot.

This choice of modules leads to the system decomposition shown in Figure 7-1. This decomposition is elaborated on later in this chapter, in Section 7.3 (page 213). In particular, the exact nature of the interfaces between the modules will be examined.

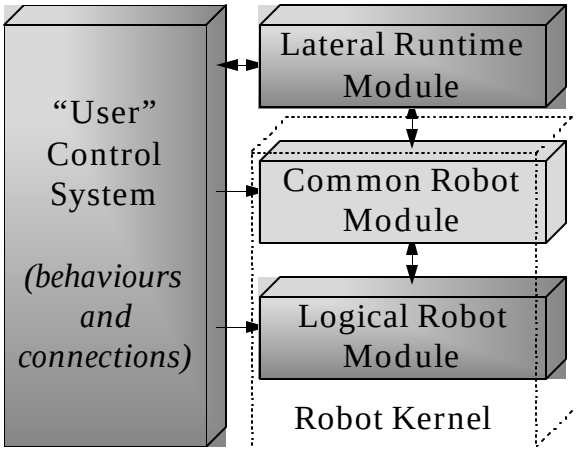


Figure 7-1: Outline system decomposition

The nature of the physical and simulated robots, and the purpose of the common and logical robot modules are clarified in the sections that follow.

7.2 Interfacing with the robot

The following sections describe the nature of the physical and simulated versions of the Khepera robot in terms of their functionality and how they differ. The ideas behind the "Common Robot" and "Logical Robot" modules are then presented and motivated.

7.2.1 The physical robot

The work in this thesis was implemented on a Khepera robot [[28]]. Khepera is a miniature wheeled robot weighing about 70g, with a diameter of 55mm and a height of just 30mm. The robot's physical appearance is shown in Figure 7-2.

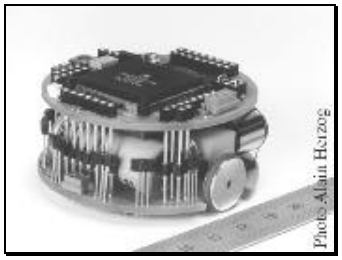


Figure 7-2: A Khepera robot (diagram from Khepera documentation)

Khepera is designed specifically as a vehicle for academic research. Its specifications are given in Appendix A1. Relevant details are included here.

Motors

Khepera sits on two wheels, each of which is moved by an independent motor. The motors have a resolution of 600 pulses per revolution of the wheel, corresponding to 12 pulses per millimetre advancement of the robot. Incremental encoders on each motor axis give feedback on the wheels' positions (24 pulses per revolution). The motors are driven by a PID controller implemented as software on the main processor of the robot. The controller can be used to either maintain a target speed or reach a target distance[61].

Sensors

Khepera can sense the presence of nearby objects and the light level in various directions around its body. Eight sensor units are distributed around the circumference of the robot as shown in Figure 7-3.

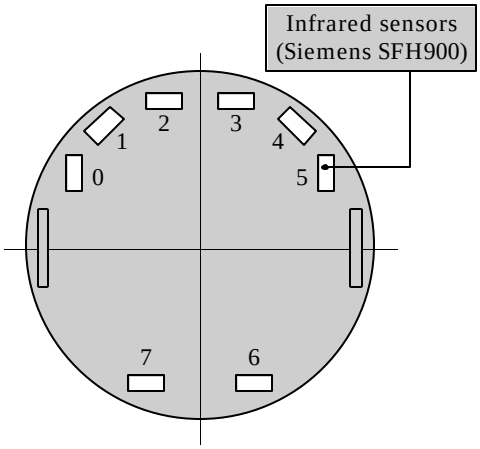


Figure 7-3: Sensors present on Khepera

These sensor units embed an infra-red light emitter and receiver pair. This combination allows both the ambient light hitting a sensor and the light reflected from obstacles to be measured. Some of the detailed properties of these sensors were examined in Section 5.2.3 (page 129) for the purposes of edge following[62].

General specifications

- **Power** - The robot has an onboard battery that allows 30-40 minutes of autonomous action. It can also be powered via the serial connection.
- **Communications** - The robot may communicate with a base PC via an RS232 serial line. This is not necessary for operation of the robot, but it is useful for monitoring its status and modifying its behaviour for experimental purposes.
- **Computing power** - Khepera uses the Motorola 68331 processor. It has 128kB of RAM, and 256kB of ROM, implementing a rudimentary BIOS with a simple multi-tasking operating system.
- **Programming** - Khepera was programmed using GNU CC, a freely available C/C++ compiler (see Appendix A3) configured as a cross-compiler. To download programs to Khepera, they were linked with special support libraries, and the object code was converted into a suitable format for transmission using a utility supplied with the robot.

---

[61] This distance is expressed in terms of revolutions of the individual motors, not the position of the robot. See Section 7.5.2 (page 225).

[62] Because the edge following algorithm depends so heavily on the exact nature of the sensors, it would need to be updated if the robot were modified - unlike most of the other behaviours of the robot.

7.2.2 The simulated robot

It was considered very important that the work presented in this thesis should be implemented in embodied form (see Section 2.4.7, page 22), and this is what was done. However, to speed development time, a simulator for the robot was also used. If a control algorithm fails to work correctly on a simulator it will almost certainly fail on the physical robot, given that the real environment is much more challenging than the simulated one. Hence simulation is a convenient test of an algorithm's feasibility, and this was the use made of Khepera's simulator. If an algorithm failed on the simulator, there was no point going to the trouble of downloading it to the robot. However, success on a simulator is no guarantee of success on the physical robot, so frequent validations were carried out as a "reality check" on the results with the simulator, to ensure the work did not start to rely on the idealisations present in the simulated environment.

A freeware simulator for Khepera is available from the university where Khepera and an associated range of robots were initially developed [[28]]. Its visual appearance is shown in Figure 7-4.

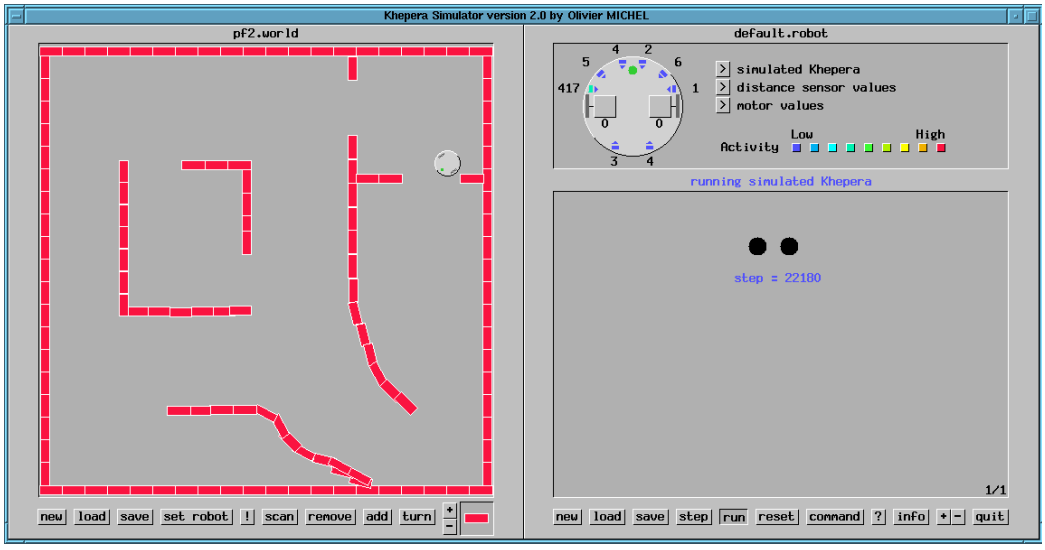


Figure 7-4: The simulator for Khepera

The simulator display is split into a number of panels:-

- One panel shows a plan view of the simulated Khepera robot in a virtual environment which it can sense and move through in a similar way to a real environment. This environment can be controlled to set up different test cases for the robot.
- A second panel shows the readings of Khepera's simulated sensors, and the activity of its motors.
- An auxiliary panel is used for displaying arbitrary user-defined data.

For added realism, random noise is introduced to the sensors.

- Proximity readings are modified by $\pm 10\%$ of their amplitude.
- Ambient light readings are modified by $\pm 5\%$ of their amplitude.

The next section compares the simulated robot with the physical one, and develops a "common denominator" abstraction for interfacing to either.

7.2.3 The common robot

The physical and simulated Khepera robot diverge in a number of respects. The purpose of the common robot abstraction is to hide these differences from the rest of the control system. The major differences between the two versions of the robot are as follows:-

- The physical robot has no intrinsic sense of its position, whereas the simulated robot, by its nature, has a completely accurate knowledge of its position.
- The robots have totally different "operating system" support- the simulated robot is on a PC, whereas the physical robot has a much slower processor and very limited memory capacity.
- The real and simulated sensors and motors have different characteristics. The simulated versions do a reasonable job of capturing the "essence" of their physical counterparts, but not their detailed responses.

The only difference that is difficult to deal with is the physical robot's lack of any knowledge of its position. In general this is one of the great obstacles in moving from a simulated robot to a real one- the robot can no longer do more than guess at its position and the precise nature of its surroundings, rather than knowing them with complete accuracy. However, although an

absolutely accurate sense of position is not possible for an autonomous robot[63], it *is* possible to construct a reasonable position estimate by integrating the robot's motion. Note however that, as discussed in Section 4.4 (page 95), unavoidable errors in readings from the motors mean that integrating the robot's motion to get its position is an approximation, and one that degrades with time. The common robot abstraction supplies a position estimate which for the physical robot is implemented through integration, and for the simulated robot is achieved by supplying either the absolute robot position, or an intentionally degraded position measurement.

In the case of the Khepera robot, the left and right motors can be driven independently at different speeds. The correlation between the feedback the motors give on their activity and the resultant overall movement of the robot is by no means a direct one. The relationship is derived later, in Section 7.4.1 (page 217).

7.2.4 The logical robot

The common robot module implements the minimum functionality possible while still allowing full control of the robot. To provide more sophisticated control of the robot, and to make it a better vehicle for research, a "Logical Robot" module is built on top of the common robot that enhances its services in the following ways:-

- The robot is given some "common sense" so it will not break itself if the higher-level control system fails and generates spurious motor commands. This is achieved by providing a separate "logical" motor interface implementing basic obstacle avoidance.
- Virtual or "logical" sensors are implemented that combine information from the robot's physical sensors to provide indicators as to which directions the robot is free to move in. This is useful in the implementation of many behaviours. These logical sensors are entirely concerned with proximity, since this is the primary sense of the robot, and the one needed for all motion algorithms.

These enhancements will now be described in greater detail.

[63] Unless the robot is "helped" by supplying a reference "beacon signal", or giving it access to the Global Positioning System- both of which detract from the ability of the robot to act in a totally autonomous manner.

The logical motors

Enough intelligence is embedded in the logical robot module so that the robot will not attempt to drive into an obstacle. Only "passive" avoidance is implemented- the robot will not actively move away from an obstacle, but will refrain from moving dangerously close to it. Active avoidance would render the robot useless, because since it has no long range sensors, its only useful information about the environment is extracted when it is in the close proximity of objects. To facilitate passive avoidance, motor commands are given in two components, one of which is unalterable (to control the overall path the robot is following) and one of which is opportunistic (indicating "nudges" to make the robot deviate from its current path). Normally the two components are simply added. However, when the robot has a close encounter with an obstacle, the unalterable component of motion is blocked, but the opportunistic component may still be carried out. The details of this system are described in Section 7.5.2 (page 225). It is a simple idea, but very important given that the robot spends most of its time in the proximity of obstacles, and close encounters are quite common.

The logical proximity sensors

The suite of logical or derived sensors shown in Table 7-1 proved to be more useful for control purposes than the raw sensors present on the Khepera robot.

Table 7-1

Logical Sensor	**Semantics**
leftSense	This is a sensor guaranteed to be low if the robot can move left without immediately bumping into something, and high otherwise.
rightSense	This is a sensor guaranteed to be low if the robot can move right without immediately bumping into something, and high otherwise.
forwardSense	This is a sensor guaranteed to be low if the robot can move forward without immediately bumping into something, and high otherwise.
reverseSense	This is a sensor guaranteed to be low if the robot can move backwards without immediately bumping into something, and high otherwise.
closestSense	This gives the closest obstacle to the robot under normal operation.

Section 7.5.1 (page 223) will describe how these sensors may be generated for Khepera.

This concludes the discussion of how the control system interacts with the robot's sensors and motors through the robot kernel. The entire decomposition of the control system will now be examined systematically. The implementation of the common and logical robot modules will be presented in detail in Sections 7.4 and 7.5 (pages 215 on).

7.3 System decomposition

Now that the general strategy for interfacing with the physical and simulated robot has been presented, it is useful to examine where those interfaces fit in with the overall control system of the robot before discussing their detailed implementation. The outline decomposition given at the start of this chapter showed the modules into which the robot's control system is divided. Figure 7-5 shows how the modules interact with each other.

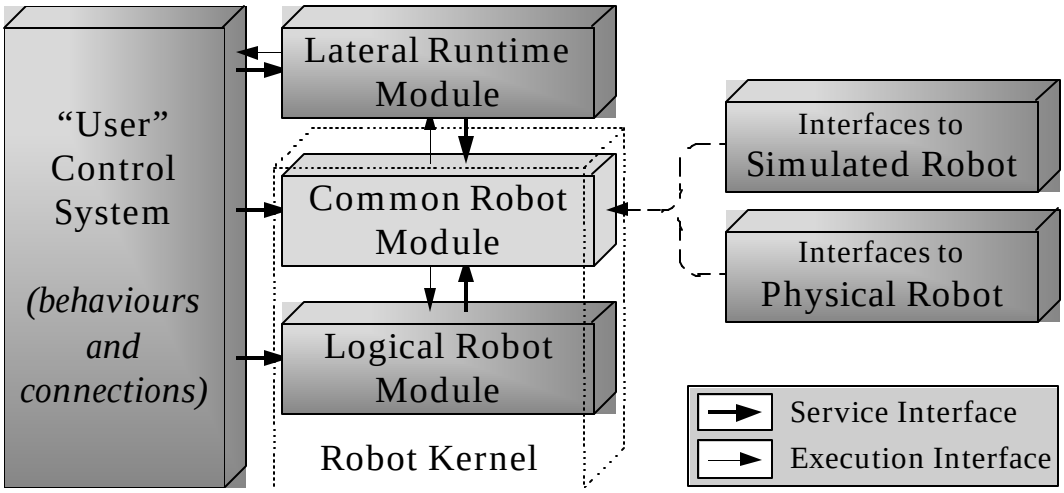


Figure 7-5: System decomposition

The system decomposition has the following features:-

- The interfaces between the user control system and the robot kernel modules are one-way. The user control system can use services provided by the kernel, but not vice versa. This makes the kernel reusable for different applications.
- The common robot, since it embodies the physical/simulated robot, is the actual executing agent in the system. Both the Lateral runtime module and the logical robot module need to execute, so one-way interfaces to these modules are present to allow the common robot to "drive" them.

- The Lateral runtime module and the logical robot module are allowed access to services of the physical/simulated robot that are not made directly available to the user control system. This is to shield the user control system from the details of the robot's "operating system".
- The interface between the lateral runtime and the user control system is necessarily more intricate than the other interfaces present. This is because there is no strict dividing line between an architecture and the control system built from it. The Zac Translator described in Chapter 6 manages the structural dependencies between these two modules.
- In addition to the interfaces shown, an extra interface is needed to allow external monitoring and supervisory tools to interact with the robot. These tools are GUI-based and implemented on a PC. They communicate with the robot through a serial connection, since this is the only way to interact with the physical robot.

All these considerations lead to the more detailed system decomposition shown in Figure 7-6.

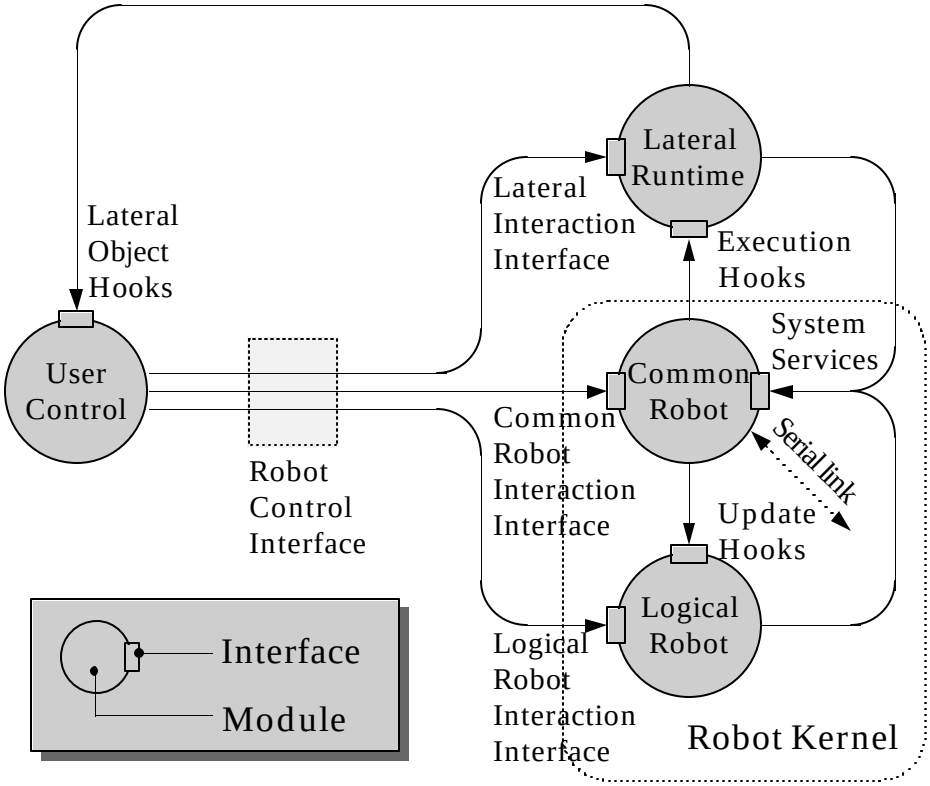


Figure 7-6: Detailed system decomposition

The diagram is arranged to emphasise that- from the point of view of the user control module - the different ways in which the robot may be controlled appear as a single interface. This

interface is in fact split among the Lateral runtime module, the common robot module, and the logical robot module, but this division is not visible to the user control module.

The different interfaces serve the following functions:-

- The "Common Robot Interaction Interface" and the "Logical Robot Interaction Interface" provide the means for the user control system to monitor or control the robot's status through the robot kernel.
- The "Execution Hooks" and "Update Hooks" interfaces allow the common robot module to drive the Lateral runtime module and the logical robot module. The common robot is not concerned with what tasks these modules perform, it just invokes them to give them the opportunity to execute and update their status. Hence these are very simple interfaces.
- The "System Services" interface provide "operating system"-type control of the robot that is not made accessible to the user control system.
- The "Lateral Interaction Interface" embodies all the services that the Lateral runtime module gives the user control system, and the "Lateral Object Hooks" interface gives the Lateral runtime module the means to drive the user control system. These hooks are generated by the Zac Translator (see Chapter 6).

Each of the modules will now be considered in turn, from the point of view of the interfaces they supply and the interfaces they use.

7.4 The common robot module

This implements the common robot abstraction described in section 7.2.3 (page 210). It hides the differences between the physical Khepera robot and its simulator, allowing the rest of the control system to be portable across both. Two versions of this module were written, one for the physical robot and one for its simulator, and the rest of the modules described in this chapter are identical for both robots.

The physical and simulated robots are made indistinguishable at two levels:-

- **Internal interfacing** - By ensuring that both versions of the common robot module implement identical interfaces, no dependencies on the version of the robot in use are propagated to the rest of the control system.

- **External interfacing** - Both versions of the module were written to accept a serial stream of input and generate a serial stream of output. This is important for communication with the external visualisation and supervision tools (see sections 7.8.1, page 229 and 7.8.2, page 230 for a description of these tools). In the physical robot, the stream is channelled across a serial port. In the simulated robot, the stream is channelled through a pipe between the simulator and the processes monitoring it. The two channels are accessed in an identical manner by the rest of the control system.

In accordance with the system decomposition given in section 7.3 (page 213), the common robot module supplies the following interfaces:-

- **System Services** - this gives access to "operating-system" level capabilities of the robot needed by the Lateral runtime module and by the logical robot module. This includes control of task-switching, timing services, and access to the robot's sensors and motors.
- **Common Robot Interaction Interface** - this gives access to all the capabilities of the robot that are complete in themselves, and not managed by other modules. This includes odometry support and serial communications.

The module makes use of the following interfaces:-

- **Execution Hooks** - this allows the robot to drive the Lateral runtime module, which in turn drives the user control system. Note that the common robot module need have no knowledge of the actual control system "application" that is implemented in the user control system. This makes it portable across applications.
- **Update Hooks** - this allows the robot to drive the logical robot module, which maintains logical motors and sensors. The common robot module need have no knowledge of the particular motor and sensor models that are being maintained, so these could be modified without affecting this module.

The different implementations of this module for the physical and simulated robots are now presented. These are the only remaining sections in this chapter specific to either robot - the remainder apply to both.

7.4.1 Physical robot version of the common robot module

The control system of the physical Khepera robot interacts with the robot's hardware through a built-in BIOS implemented in the robot's ROM. The common robot module uses the services of the BIOS as follows:-

- Some of the BIOS services are suitable for use by the rest of the control system almost as they stand, and correspond closely to similar functionality in the simulated robot. A "service shell" was constructed that provided a means to access such aspects of the BIOS.
- A set of concurrently executing tasks[64] were implemented that interfaced with the BIOS to manage odometry, monitoring the robot's serial connection, interfacing to the motors and sensors, and executing the rest of the control system, as follows:-
  - ⇒ **Sensor thread** - this periodically updates the robot's sensor readings and checks for collisions or impending collisions, and passes control to the logical robot module so that it can update the logical sensors (see Section 7.5.1, page 223).
  - ⇒ **Motor thread** - this periodically passes control to the Logical Robot module so that it can control the motor setpoints dynamically according to the setting of the logical motors and the state of the environment (see Section 7.5.2, page 225).
  - ⇒ **Odometry thread** - this periodically updates the robot's best guess at its position, and the direction it is facing in, from knowledge of the motion of the robot since its last update.
  - ⇒ **Communications thread** - this checks for commands coming across the serial line and notifies the rest of the system when a command has been received.
  - ⇒ **Control thread** - this periodically passes control to the Lateral component of the control system so that the "user" control system will be executed.

No *direct* access to the BIOS by the rest of the control system was allowed, as shown in Figure 7-7, since this would prevent the control system from running on the simulated robot, which does not have the same BIOS[65].

---

[64] The Khepera robot is reasonably sophisticated, allowing a total of 14 concurrent user processes to run. This would not be enough for the sentry application, so the "pseudo-concurrency" provided by Zac Script was used. This capability did however make the robot kernel easier to write.

[65] The implemented set of tasks deviates slightly from that documented here for reasons of efficiency, but there are no significant differences.

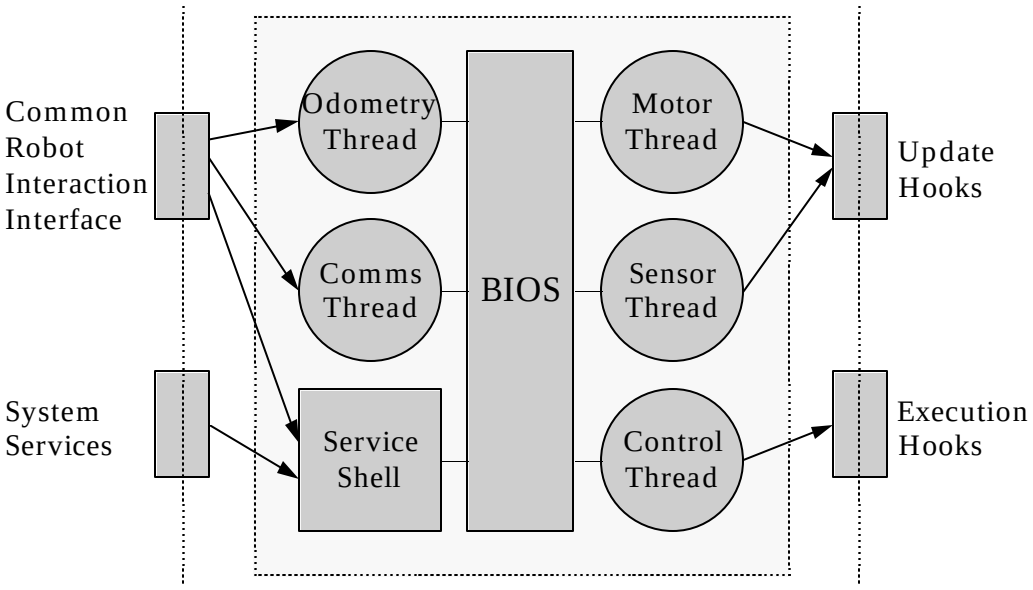


Figure 7-7: Implementation of the common robot module on the physical robot

The physical Khepera robot has no built-in sense of position- no odometry data is available. Such a sense is necessary for cartography (see Chapter 4). The odometry thread attempts to track the robot's position and direction by integrating the motion of the robot. This is an approximate estimate only, and it is not possible to generate an absolutely accurate sense of position from it. The rest of this section discusses how this tracking of the robot's motion is achieved.

The robot can move in three ways:-

- In a straight line, in the forward or reverse direction. This occurs when the robot drives the left and right motors at the same speed in the same sense (forward or reverse).
- Turning in-place (i.e. the robot turns while keeping its centre at the same point). This occurs when the robot drives the left and right motors at the same speed in opposite senses.
- Differential turning, where the robot drives the motors at different speeds.

Ideally, an example of the path the robot would trace out should look something along the lines of Figure 7-8.

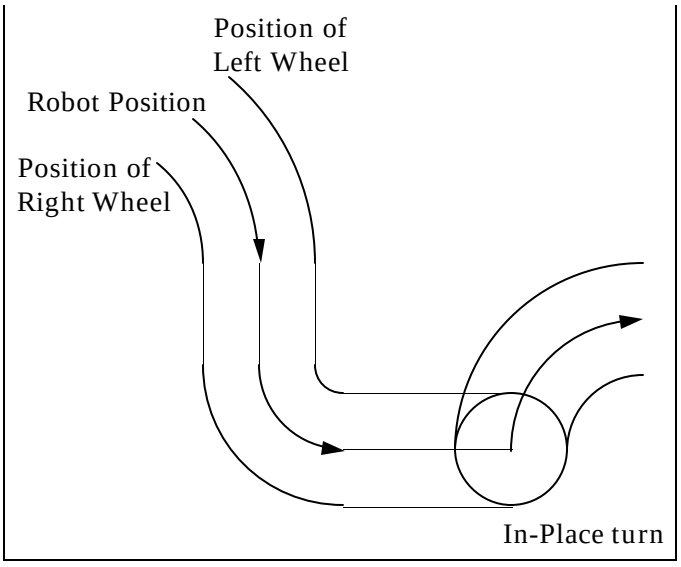
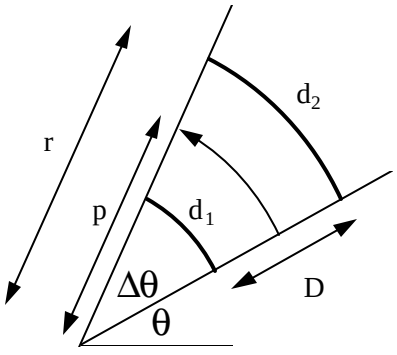


Figure 7-8: Example path of robot

However, the actual path of the robot will be somewhat different, because the robot cannot change between speeds instantaneously, but will have a period of acceleration. This results in quite a complex path, particularly when the motors are being driven at different speeds. There may also be slippage in the wheels. Because of these considerations, odometry is maintained based on feedback from the motor rather than on the setpoints to which the motors are commanded. The general strategy is to measure the distance the wheels have moved at frequent enough intervals for the speed of each motor between each sample to be approximately constant over the sampled range (for the Khepera robot, measuring the wheel positions every time either wheel moved by about one tenth of the robot's diameter was found to be adequate). Under these circumstances, the wheels will trace two arcs of a circle in the period between measurements, as shown in Figure 7-9.



d_1	distance travelled by left wheel
d_2	distance travelled by right wheel
D	diameter of robot
p	distance of robot from centre of turn
r	distance of far wheel from centre
θ	angle before turn
$\Delta\theta$	change in angle after turn

Figure 7-9: The robot's motion between motor measurements

From the diagram, the length of the arc that the right wheel moves through is:-

$$d_2 = r\Delta\theta$$

Similarly, the length of the arc that the left wheel moves through is:-

$$d_1 = (r - D)\Delta\theta = r\Delta\theta - D\Delta\theta = d_2 - D\Delta\theta$$

Since these lengths are known- they are given by the feedback from the motors- the angle the robot turns can be calculated as:-

$$\Delta\theta = \frac{(d_2 - d_1)}{D}$$

To find the change in position of the robot after the turn, the robot's position before the turn can be rewritten as:-

$$(p\cos q, p\sin q)$$

Where p is the distance from the centre of the turn to the centre of the robot:-

$$p = \frac{d_1}{\Delta\theta} + \frac{D}{2}$$

The robot is at angle q initially, and then is rotated an extra Δq degrees. Therefore the robot's position is mapped to:-

$$(p\cos q, p\sin q) \Rightarrow (p\cos(q + \Delta q), p\sin(q + \Delta q))$$

So the changes in position are:-

$$\Delta x = p[\cos(\theta + \Delta\theta) - \cos\theta]$$

$$\Delta y = p[\sin(\theta + \Delta\theta) - \sin\theta]$$

And the overall distance the robot travels is:-

$$\Delta s = p\Delta\theta = \frac{(d_1 + d_2)}{2}$$

At this point all the information needed to update the robot's odometry is known. Note that two sets of cosine and sine values are needed for each update, but one set can be stored for the next update so that only one cosine and sine calculation need be done during each update. Provision has to be made to prevent the possibility of a divide-by-zero occurring. This would happen if Δq is zero. Such a situation occurs when the robot is moving in a straight line, so it's a degenerate case and is straightforward to deal with.

For the Khepera robot, the motor positions were checked 10 times per second, and if either motor position changed by a distance equivalent to about one tenth of the robot's diameter,

the odometry information was updated using the method described above. This arrangement was found experimentally to work well.

7.4.2 Simulated robot version of the common robot module

This version of the common robot module is more straightforward to build than the version for the physical robot because:-

- Odometry is not a problem in this case, since the simulated robot "knows" exactly where it is within the simulated environment- otherwise the values for the simulated sensors could not be calculated. In fact, to make the simulation more realistic, the robot's position sense may be deliberately degraded.
- The simulator operates on a PC, which has a faster and more powerful processor than the physical robot. Hence there is no need for careful speed optimisations and the use of concurrent tasks to satisfy real-time constraints- one thread of control is more than adequate.

Although the services provided by the simulated robot are similar in nature to those provided by the real robot, the programming interface is quite different. The relevant interfaces to the simulator are given in Appendix A5. Khepera's simulator comes complete except for a single module, called the simulator "User" module, which needs to be provided by the user of the simulator to specify the actual behaviour of the robot (see Figure 7-10). Mapping the simulator onto the common robot interfaces was done by building a suitable simulator "User" module implementing those interfaces. Serial I/O was directed to a pipe for connection to the Visualisation and Supervision tools.

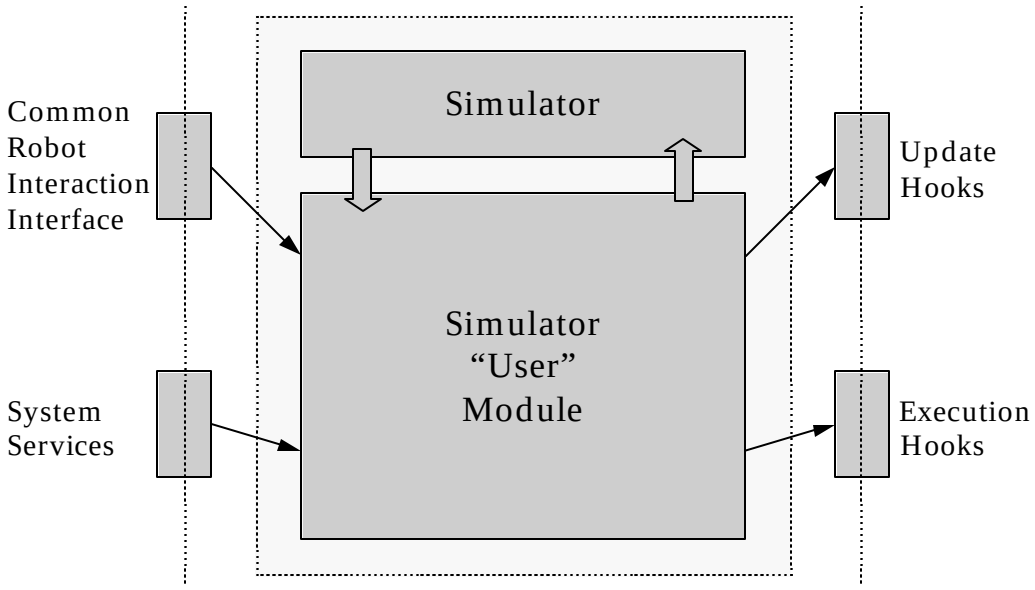


Figure 7-10: Implementation of the common robot module on the simulated robot

The simulator was left almost entirely unchanged except for one useful addition: it was extended so that if the mouse was clicked within the simulator, the robot would sense a circular obstacle centred at the position of the mouse. This meant that the robot's response to moving obstacles could be tested. The simulator as it stood only permitted static obstacles.

7.5 The logical robot module

The ideas behind the logical robot module were introduced in section 7.2.4 (page 211). This module enriches the interaction interface supplied by the Common Robot module by providing support for logical motor and sensor models that simplify controlling the robot, and make it more robust. The implementation is split into two components, as follows:-

- An "Action" partition updates the motor and sensor models at frequent intervals.
- An "Interaction" partition allows the status of the motor and sensor models to be queried and manipulated.

The partition decouples the part of the system monitoring the motor and sensors from the part of the system manipulating them, removing unnecessary timing dependencies between the two that would otherwise be present. The organisation of the logical robot module is shown in Figure 7-11.

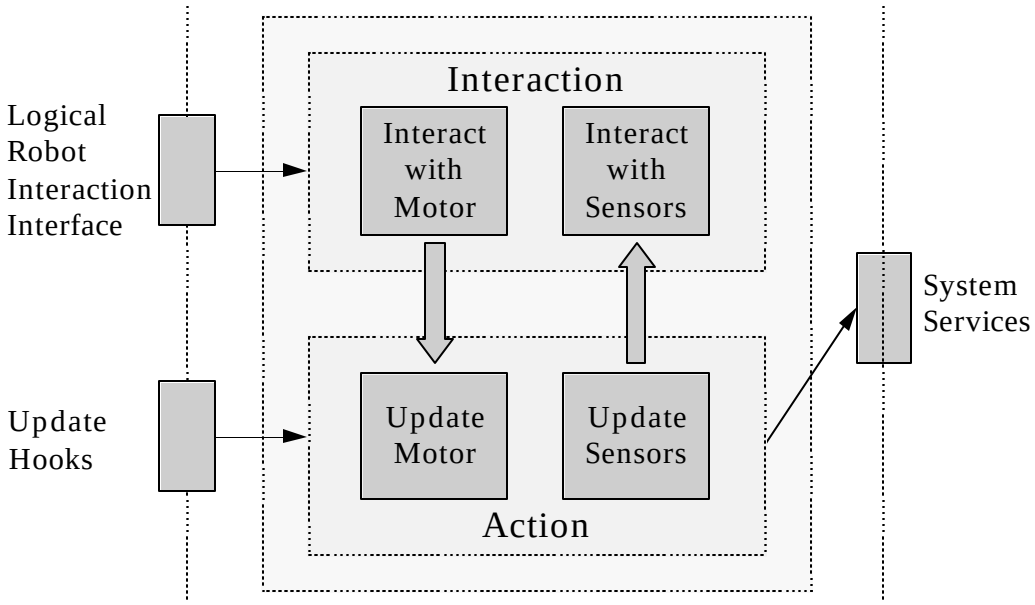


Figure 7-11: Implementation of the logical robot module

The action and interaction partitions operate on the sensor and motor model maintained by the logical robot module. The following sections describe how these models are generated.

7.5.1 Sensor model

The implementation of the logical proximity sensors was introduced in section 7.2.4 (page 211). Figure 7-12 shows the correspondence between actual and logical sensors for the Khepera robot.

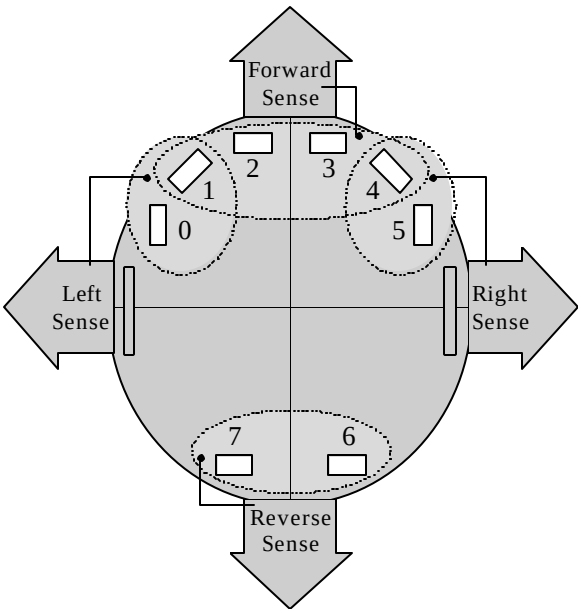


Figure 7-12: Logical proximity sensors

- **forwardSense**- as discussed in section 7.2.4 (page 211), the semantics of this sensor should be such that it is low if the robot can move forward, and high if an obstacle is detected that blocks forward motion. Referring to Figure 7-12, this judgement can be made by combining the two direct-forward sensors and the two diagonal sensors.
- **leftSense**- this should be low if the robot is free to move left. Again referring to Figure 7-12, this judgement can be made by combining the direct-left and the diagonal-left sensors.
- **rightSense**- this should be low if the robot is free to move right, and can be generated by combining the direct-right and the diagonal-right sensors.
- **reverseSense**- this should be low if the robot is free to move in reverse, and can be generated by combining the two direct-reverse sensors. This is not entirely guaranteed to be correct, because the robot has no diagonal sensors in the reverse orientation. Hence it is safer to turn 180° to reverse, as this ensures the robot has maximum sensing ability in the direction of motion.
- **closestSense**- this should be low if the robot is free to move forward, left, or right. It should be high if there is anything to block the robot's motion in any of these directions. This sensor can be generated by combining the other logical sensors.

The logical sensors derived are as accurate as is possible for the actual sensors available on Khepera. One restriction on their use is that they assume the robot is driven "head-first". In other words, for the logical sensors to have the semantics described here, the robot should generally move in the direction of its sensor-rich "head" (the hemisphere with 6 sensors rather than the one with two). If the robot is driven in reverse, then the semantics are no longer valid- since the robot does not have enough sensors in that orientation to make the necessary judgements. The exact implementation of the logical sensors is given in Table 7-2.

Table 7-2

Logical Sensor	Implementation
leftSense	The maximum of the left and left-diagonal sensors.
rightSense	The maximum of the right and right-diagonal sensors.
forwardSense	The maximum of the two forward sensors, or the left- or right-diagonal sensors if either indicate a very close encounter.
reverseSense	The maximum of the two backwards-facing sensors.

Logical Sensor	Implementation
closestSense	The maximum of the left, right and forward logical sensors. The reverse logical sensor is omitted for normal "head-first" operation.

7.5.2 Motion model

The robot's logical motion model simplifies controlling the motors, and provides some intelligence for dealing with "close encounter" situations. This intelligence is important because, as discussed in Section 7.2.4 (page 211), the robot has to operate close to obstacles for its proximity sensors to be of use in characterising its environment, and so it is often in situations where it is close to a collision. When constructing a motion model for the robot, a choice has to be made between two broad types of model that are possible:-

- **Displacement oriented**- where motor commands are expressed in terms of the distance the robot should travel.
- **Speed oriented**- where motor commands are expressed in terms of the speed at which the robot should move.

Both alternatives are supported by the Khepera robot's motor control facilities based around a classic PID controller implementation. The model used in this project was speed-oriented, because such a model is more naturally reactive. Controlling the distance the robot should travel intrinsically involves decisions about the future, while controlling its speed generally only requires consideration of what is happening now- at least in simple behaviours.

To facilitate the handling of close encounters, the speed of the robot was split into two components- an unalterable "major" speed, and an opportunistic "nudge" speed. The major speed specifies the basic motion the robot is making- forward, reverse, turning clockwise or anti-clockwise- and at what rate it is performing that motion. The nudge component is superimposed on this speed for a given duration to divert the robot from this basic motion. At regular intervals the two speed components are combined to generate the appropriate setpoints for the motor speeds. The speeds are capped so the robot does not move excessively fast. Figure 7-13 shows an example of the result of combining the two speed components.

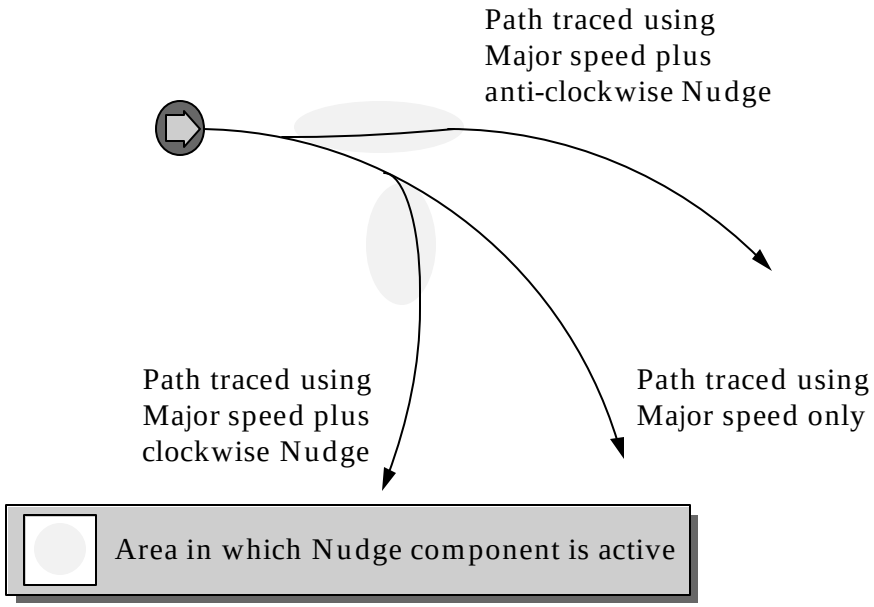


Figure 7-13: Composite paths

If the robot is close to an obstacle, the major motion is blocked by the logical robot module if such motion is likely to lead to a collision. The situations where this could occur are as follows:-

- When the motion is in the reverse direction[66], and there is something immediately behind the robot.
- When the motion is in the forward direction[67]. All such motion is blocked, unless the space directly in front of the robot is free of obstacles, and:-
  - ⇒ When the motion is directly forward, it is allowed if no obstacle is detected by the diagonal sensors either.
  - ⇒ When the motion tends to the right, it is allowed if no obstacle is detected directly to the right or diagonally to the right, and there is no extremely close obstacle diagonally to the left.
  - ⇒ When the motion tends to the left, it is allowed if no obstacle is detected directly to the left or diagonally to the left, and there is no extremely close obstacle diagonally to the right.

[66] A reversing motion is characterised by either both motors being in reverse, or one reversing faster than the other drives forward. Otherwise the motion is a forward motion.

[67] This is any motion other than a reversing motion or an in-place turn.

If the major motion is blocked, only the nudge component of the motion (if any) should be allowed, otherwise a collision is likely. Since the nudge component merely turns the robot, and the robot can turn in-place, it will never be blocked. The length of time the major motion is blocked is used to generate a logical "frustration" sensor so that high-level behaviours can deal with this condition if necessary.

7.6 The Lateral runtime module

This module provides support for behaviours and connections in the user control system. This is where the Lateral architecture developed in Chapter 3 fits in to the actual implementation on the Khepera robot. Behaviours and connections are implemented as described in section 3.6.3 and the "pull cycle" for updating them is as described in section 3.6.2 (page 71). Figure 7-14 shows the organisation of this module.

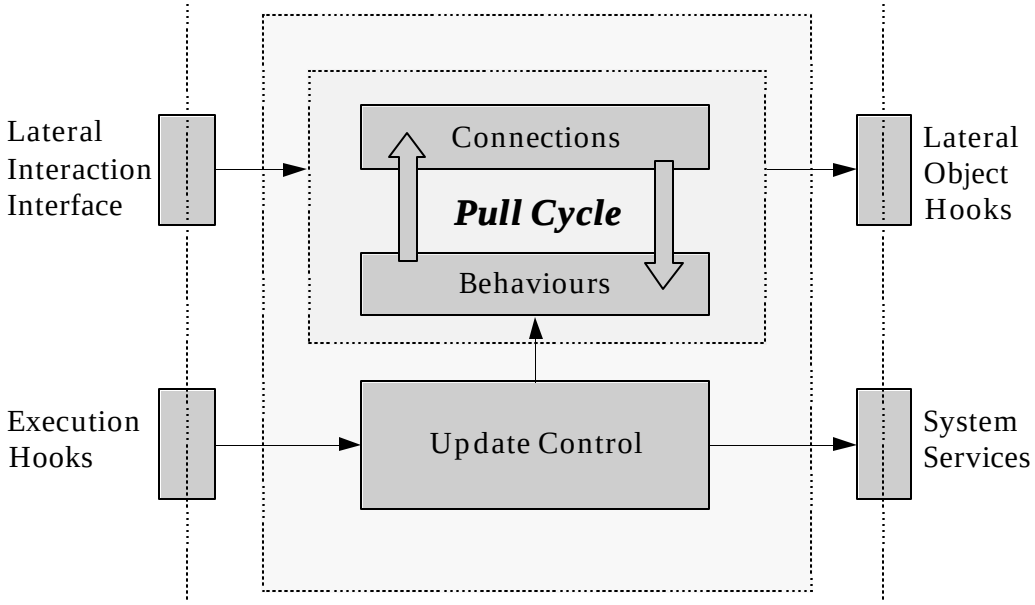


Figure 7-14: The Lateral runtime module

The Lateral runtime module gains access to the specific connections and behaviours running on the robot through the Lateral Object Hooks interface generated by the Zac Translator. Those behaviours in turn can use the services of the Lateral runtime module through the Lateral Interaction Interface. All the interfaces are described in full in Appendix C3.

7.7 The user control unit

This consists of actual instances of connections and behaviours, implementing a specific application such as the sentry-like behaviour described in Chapter 5. The organisation of this module is shown in Figure 7-15.

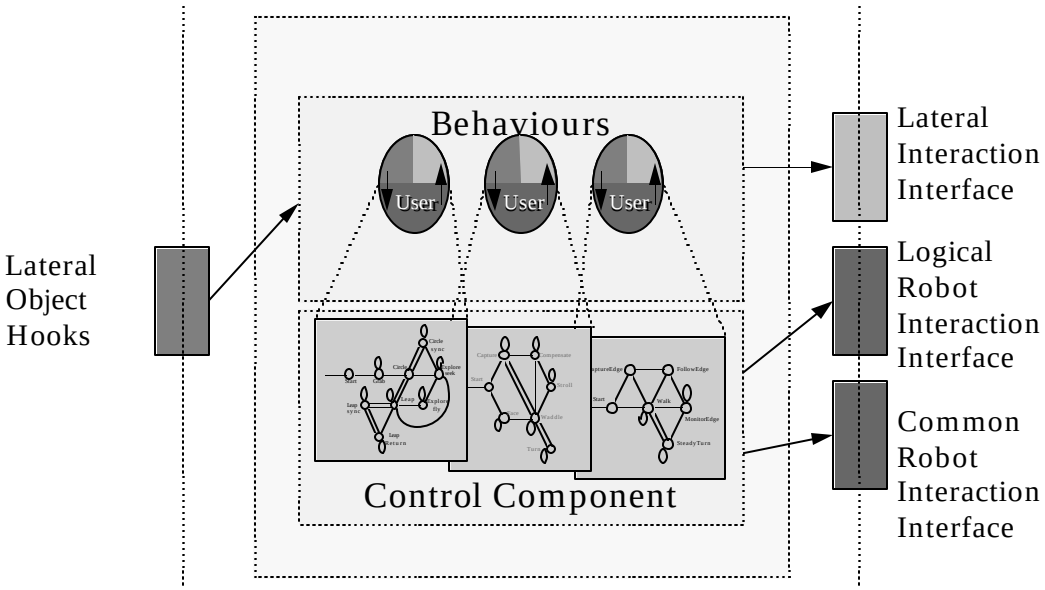


Figure 7-15: User control system module

Each individual behaviour and component can use the services of the Lateral architecture through the Lateral Interaction Interface, and each one of them also supplies an element of the Lateral Object Hooks interface that lets the Lateral runtime module drive it. This interface is simply two global registries with which behaviours and connections must be enrolled so that the Lateral runtime knows of their existence. The Common and Logical Robot Interaction Interfaces are available for controlling the robot through the kernel.

7.8 External tools

Some useful tools external to the robot's control code were implemented: a supervision tool for making it easier for the user to control the robot and monitor its status, and a visualisation tool for displaying the robot's path graphically, showing the points at which significant events happen, and the robot's internal map of its environment. These tools are now described.

7.8.1 Supervision tool

This is a tool constructed for allowing the user to control the robot (whether real or simulated) from the PC, and to monitor its status while it is in operation. The appearance of the tool is shown in Figure 7-16.

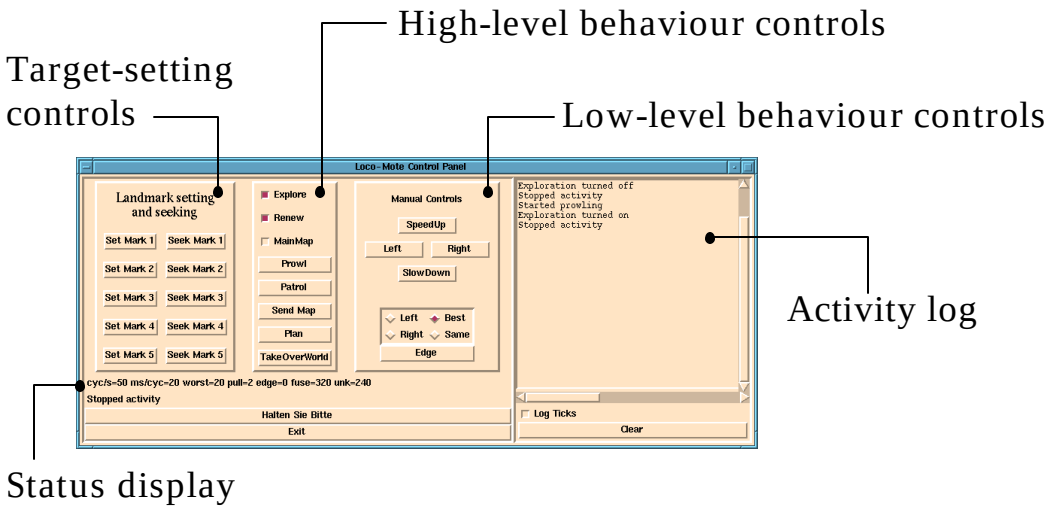


Figure 7-16: Robot supervision tool

The tool has the following visual components:-

- **Status display**- this shows diagnostic performance statistics reported by the robot every second, and the last action the robot reported taking.
- **Activity log**- this records all acknowledgements from the robot, and optionally the performance statistics it generates.
- **Simple behaviour controls**- these allow the user to take direct control of the primitive behaviours of the robot, by initiating edge-following or by setting its speed and direction of motion.
- **Compound behaviour controls**- these give control of higher level behaviours, such as prowling and patrolling. They also allow control of the robot's mapping abilities.
- **Target setting controls**- these allow the user to direct the robot to take note of its current position, and to command it to return to that position at a later stage.

The tool is implemented in Tcl/Tk and Perl. It is designed to expect the robot's output on its standard input, and to send its commands to the robot on its standard output. Hence it can be

connected by pipes to either the real or the simulated robot. The following two section outline what the tool expects to see on its input stream, and what it will generate on its output stream.

Expected input

The tool pays attention only to lines that start with "/ME" or "/TICK", which are the tags the robot uses for acknowledgements and performance statistics respectively.

- "/ME"- a line starting with /ME describes some action the robot took in response to user request. The panel shows this on the status display, and logs it to the activity log.
- "/TICK"- every second the robot generates performance statistics (load, cycle time - longest and average- cycles per second, etc.) and reports them preceded with a /TICK. The panel shows these on the status display, and optionally logs them to the activity log. See the "Reporting behaviour", section 5.4.3 (page 161).

Output generated

Each button on the panel generates output consistent with that expected by the "Proxy behaviour" described in Section 5.4.4 (page 162). The details of the simple codes involved are given in Appendix B4.

7.8.2 Visualisation tool

This tool visualises spatial information about the robot's actions. The robot outputs the following information about its current status and activity:-

- Its estimate of its position. The position is tagged to indicate whether an obstacle or edge is sensed near it.
- The position of new markers being placed. These are the unit used by Khepera to build its dynamic map of its surroundings (see Chapter 4).
- The position of markers that have been moved or removed.
- Any position at which the robot detects a landmark.
- Any target which the robot is currently trying to approach.

This data is visualised by filtering it, logging it, and piping it to "gnuplot", a graphing utility. An example of the result of this process is shown in Figure 7-17. The display is updated in real-time using a Perl script.

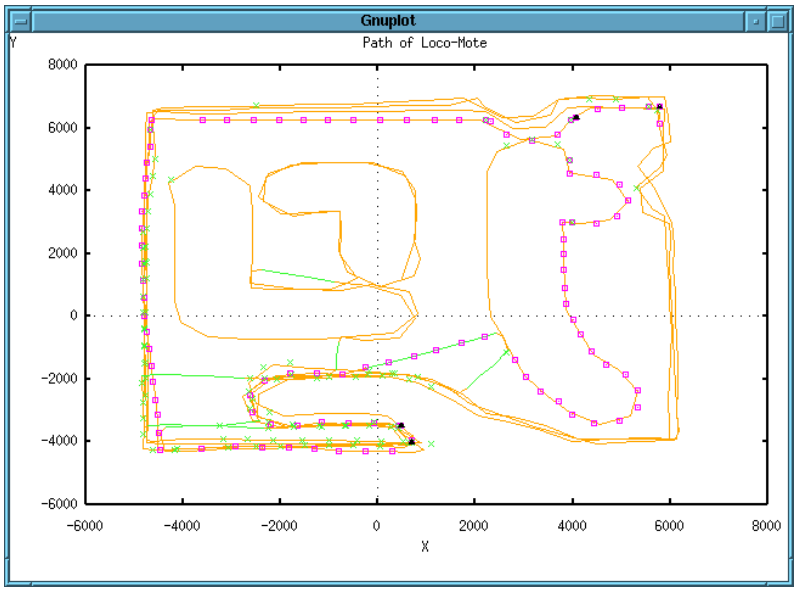


Figure 7-17: The visualisation tool

The robot can be requested to send the current state of its internal map of its surroundings to the PC. The visualisation tool can display this instead of the default trace of the robot's path and the points at which significant events occur, as shown in Figures 7-18 and 7-19.

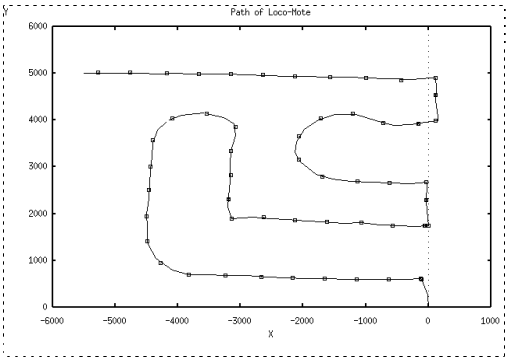


Figure 7-18: Robot's path and significant events

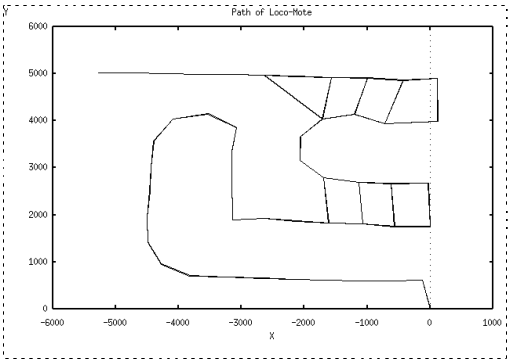


Figure 7-19: Internal map

The visualisation tool is extremely useful for understanding how the robot is interpreting its environment.

7.9 Summary

This chapter provided details of how the work presented in this thesis was implemented on the Khepera robot, in both its physical and simulated form. The decomposition of the robot's control system maximised the portability of the control system across changes of the robot or in its application. This chapter clarified the context within which the Lateral architecture and the sentry application were implemented.

8. Experimental Results

This chapter examines how well the robot's behaviours work, and how robust its cartographic system actually is. Results from the simulated robot and the physical robot are both presented and compared. Comparisons are also made with results in the literature.

8.1 Simulated Robot

Results derived from the use of the simulated robot are given first, because :-

- The environment the robot is in is a simulated one, and is therefore known and can be superimposed on traces of the robot's motion- aiding comprehension of the results.
- The robot's overall behaviour is more idealised than with the physical robot, so the results are easier to understand. For example, the simulated robot does not need to use landmarks to maintain its sense of position.

8.1.1 Boundary Following

In this test case, the robot demonstrates its ability to smoothly follow the edges of curved, stepped, and erratic boundaries.

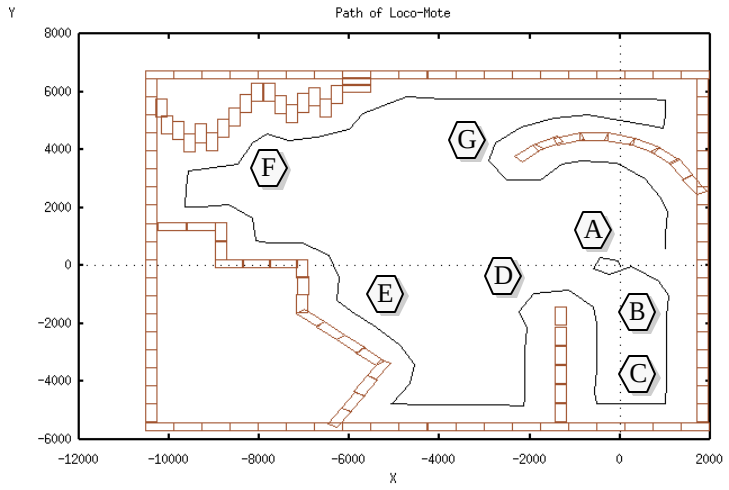


Figure 8-1: Edge following

Figure 8-1 shows the performance of the robot's edge following behaviour under different conditions.

- The robot is not initially close to a boundary (it starts at A in the diagram), so it performs a

spiralling search for one.

- It finds an edge at B and starts following it. The edge is straight, so the path the robot follows quickly settles into a straight line too.
- At C the robot meets a concave corner, turns, and continues to follow the edge as before.
- Around D the robot navigates around a 180° convex corner. There is a little bump in the robot's path at D caused by the robot finding that itself losing sensor contact with the boundary due to the sharp turn and starting an edge recovery procedure.
- Around E the robot follows a series of edges at different angles, its own path changing gracefully to match.
- At F the robot meets a very erratic boundary, and follows it smoothly. This is partly a consequence of how the sensors work, and partly due to the averaging processes within the edge following algorithm.

Before and after G the robot follows a convex and concave curving edge just as easily as a straight edge.

An even more erratic boundary is shown in Figure 8-2, and as can be seen the robot has no difficulty with it.

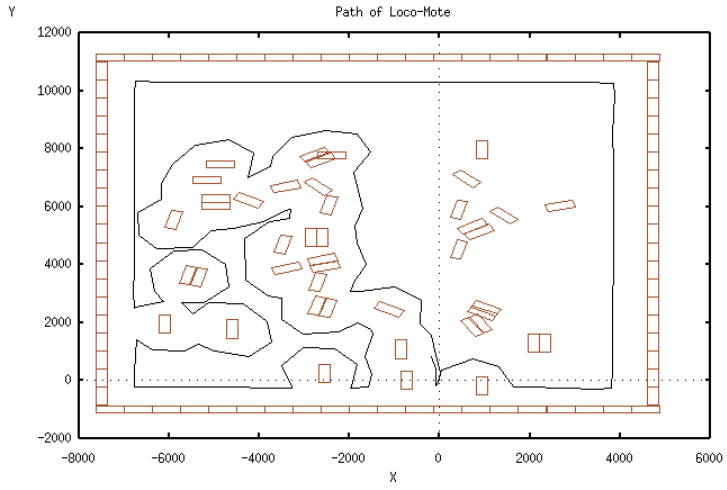


Figure 8-2: Following an ill-defined edge

8.1.2 Target seeking

8.1.2.1 Direct approach and boundary following

In this test case, the robot negotiates a one-sided concave obstacle without the use of map search.

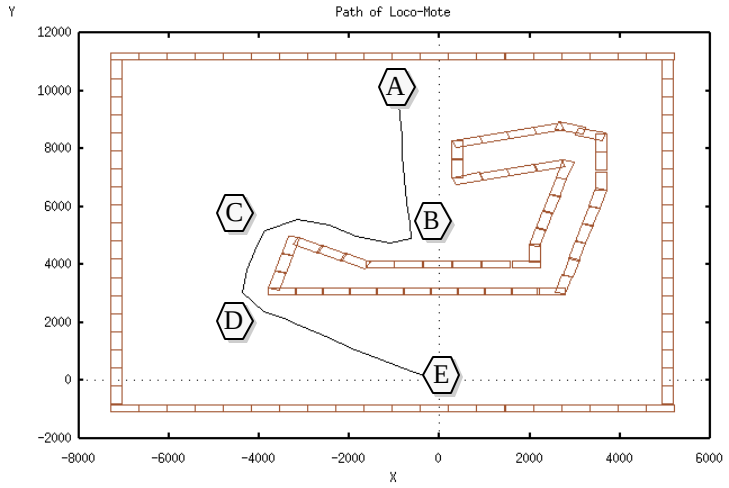


Figure 8-3: Negotiating an obstacle

Figure 8-3 shows the performance of the robot's edge target seeking behaviour with map search disabled.

- The robot starts at A and is trying to reach E. Initially there is no obstacle in the way so it moves directly towards E.
- At B it meets an obstacle and can no longer directly approach its target. The robot therefore seeks to get around the obstacle by following its boundary. It can follow it either left or right; the decision is essentially random (when the robot hits a boundary roughly at right angles to it), and in this case the robot turns to its right and follows the boundary in that direction.
- At around C the robot would turn back if no progress was being made towards its goal. Since it finds that the boundary turns here and allows it to approach its goal, it stays going in the same direction for a while longer.
- When the robot reaches D it finds it is again free to move directly towards the target, and so stops following the obstacle boundary and heads towards E.

There are no further problems; the robot reaches its target without incident.

8.1.2.2 Boundary search

In this test case, the robot negotiates the same obstacle as in Section 8.1.2.1, but using a different technique.

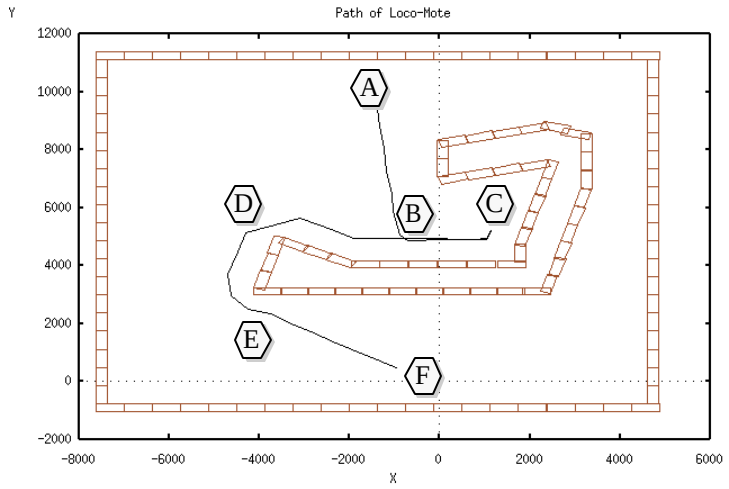


Figure 8-4: Negotiating an obstacle after different guess

The robot here works in the same test environment as the last one, under the same conditions, but with a different guess when it meets an obstacle (see Figure 8-4).

- The robot starts at A, moves directly towards its target, and meets an obstacle at B as before, but instead of following the obstacle's boundary to its right, it turns towards its left. There is no local way of knowing which is the better way to turn, so it might turn either way (and does in fact do so in successive runs).
- At C, the robot has not yet had success in getting closer to its target, so it turns back to try the other direction along the boundary.

As before, it meets with success at D, and can approach its target once again when it reaches E.

8.1.2.3 Extended boundary search

In this test case, the robot uses physical search and its familiarity virtual sensor to reach a target at the opposite side of a bowl-shaped obstacle.

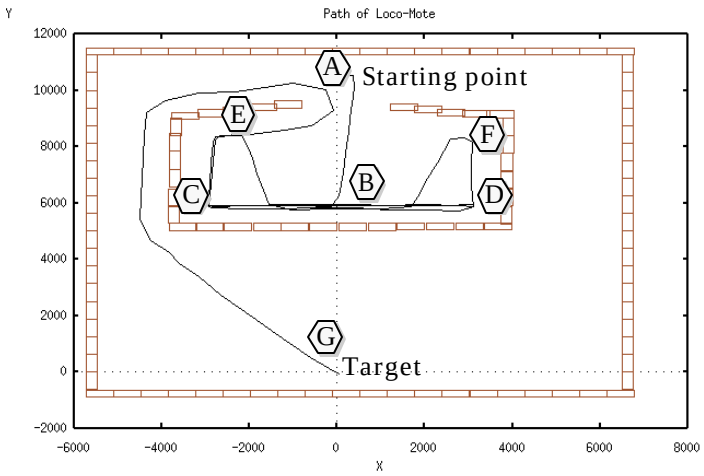


Figure 8-5: Use of familiarity in seeking target

Figure 8-5 shows the performance of the robot when seeking a target leads to it becoming trapped, using physical search only, with map search turned off. It behaves as follows:-

- The robot moves from A towards the target at G until it hits an obstacle at B.
- It turns to follow the boundary to the left (it chooses the direction it has to turn least to start edge following). No opening is found, and at C the robot turns back to try going the opposite direction.
- The robot has no more success to the right either, and turns back again at D.
- Since the area up to C is now familiar, the robot will proceed a further distance on from C. At E it gives up again and turns back.
- Similarly, the robot continues on past D, but gives up at F and turns back.

Finally, the robot proceeds past C and E, all this area being now familiar, and rounds the lip of the cave and escapes. From there, reaching the target at G is straightforward.

8.1.2.4 Map search

In this test case, the robot demonstrates the use of a background map search while seeking a target at the opposite side of a bowl-shaped obstacle.

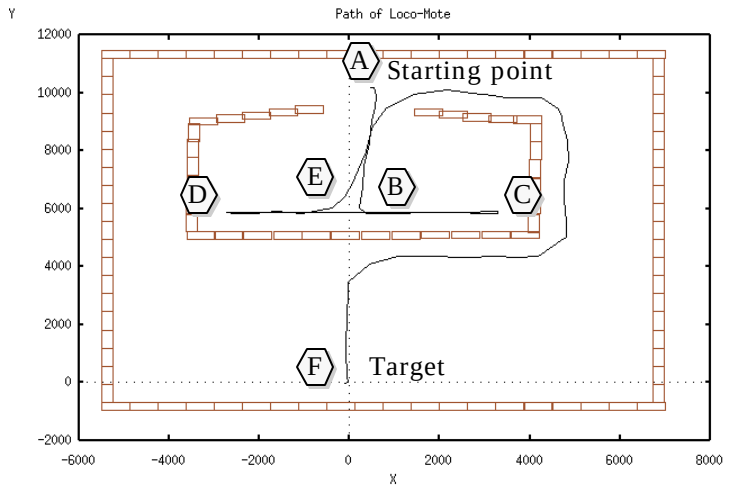


Figure 8-6: Use of map search in seeking target

Figure 8-6 shows the performance of the robot using both physical search and map search to escape from a hollow obstacle. It behaves as follows:-

- The robot starts a background map search for the target. This has no immediate effect on its behaviour.
- The robot moves from A towards the target at F until it hits an obstacle at B.
- It turns to follow the boundary to the right. No opening is found, and at C the robot turns back to try going the opposite direction.
- The robot has no more success to the right either, and turns back again at D.

At E, the background map search reaches the robot's current position, and the robot switches from physical search to following the path to the target found. Note that the path will be through areas the robot has already travelled through, so it will not take shortcuts. However the path will be the shortest path possible through familiar territory, so if the robot has had the chance to make a good exploration, the path it chooses will also be good.

8.1.3 Prowling

In this test case, the robot demonstrates the prowling behaviour within a maze with four disjoint boundaries.

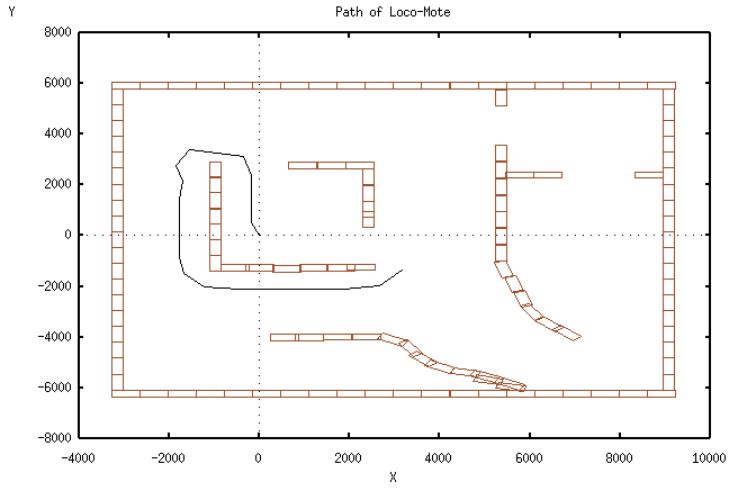


Figure 8-7: Early stages of prowling

The robot starts off by finding any obstacle, then following its boundary to the full extent (or until the robot's confusion level becomes too high). In Figure 8-7 the robot has found its first obstacle and is following its boundary.

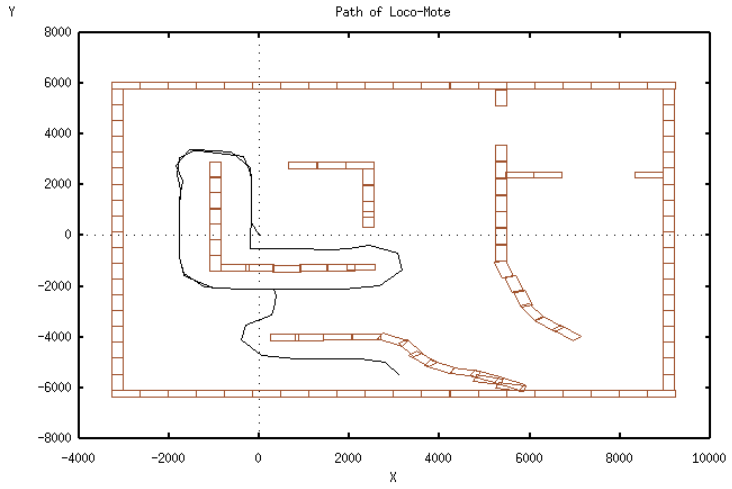


Figure 8-8: Exploration component of prowling

When the robot finds it has explored a boundary completely, it will find a vector in a direction it has not yet explored, and follow that vector. In Figure 8-8, the robot is shown after it has performed its first such exploration. The vector has in this case lead to another obstacle.

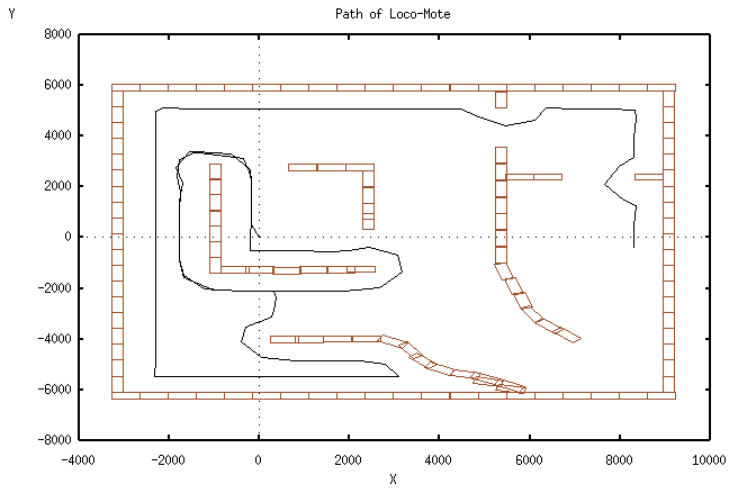


Figure 8-9: Exploring multiple boundaries

The robot can detect if the obstacle is a new one, and if so it will explore its boundary as before. In Figure 8-9 the robot is following the new boundary.

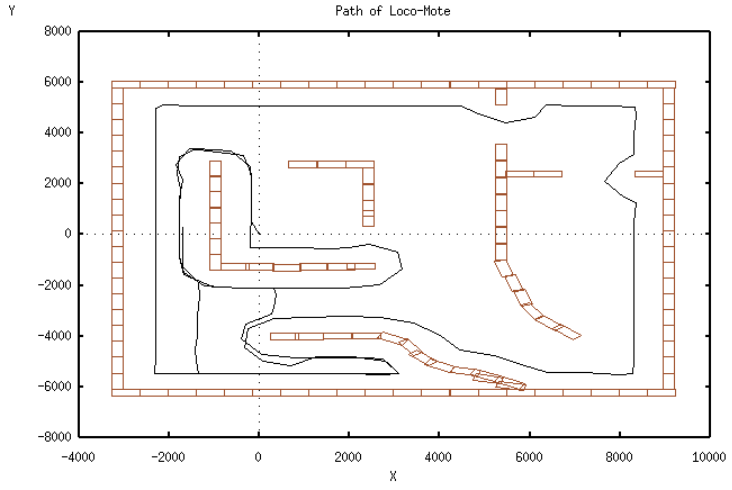


Figure 8-10: Further exploration

In Figure 8-10, the robot has completed exploration of the boundary. The robot then picks a new vector to explore. The vector in this case leads to an obstacle it has already explored (this is quite acceptable, the vector exploration is targeted at exploring unfamiliar spaces rather than necessarily finding new boundaries).

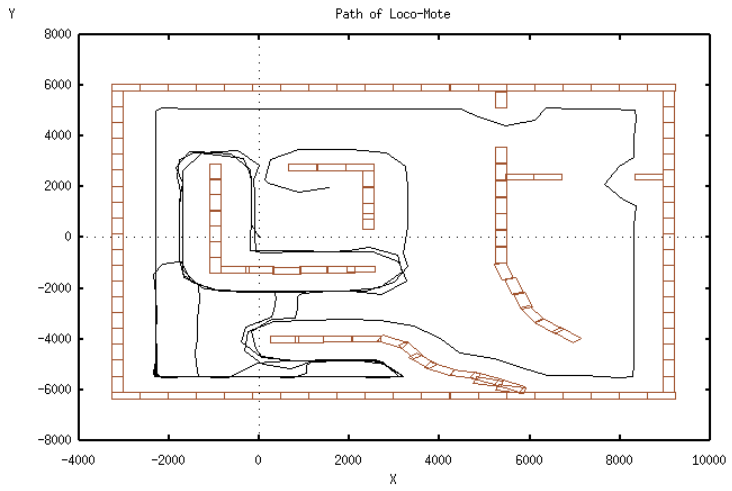


Figure 8-11: Exploration combined with patrolling

The robot follows the obstacle's boundary for a while until it spots a likely vector to explore, and repeats this cycle until a new obstacle is found, or the robot decides some part of the territory it has explored needs re-exploration ("patrolling") because it hasn't checked it for a while. As shown in Figure 8-11, the robot explores the area in the lower left, patrols the first obstacle again, then finds another obstacle to explore.

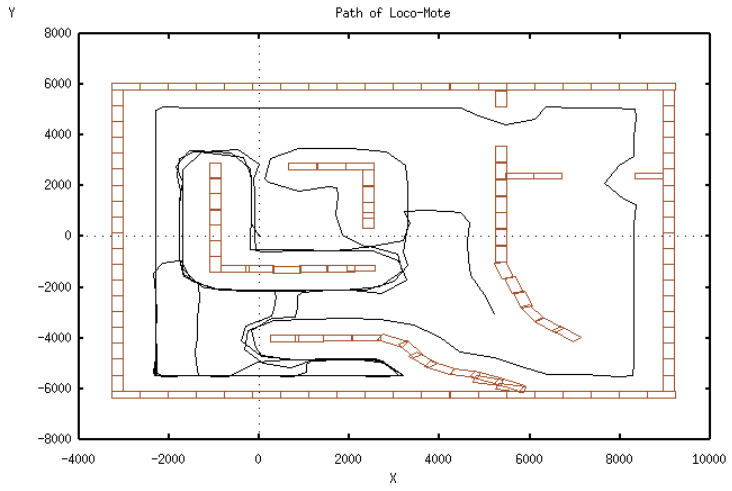


Figure 8-12: Continuation of prowling cycle

This cycle of vector exploration, boundary exploration, and patrolling continues. In Figure 8-12, the robot is shown as it discovers the last obstacle in the test environment.

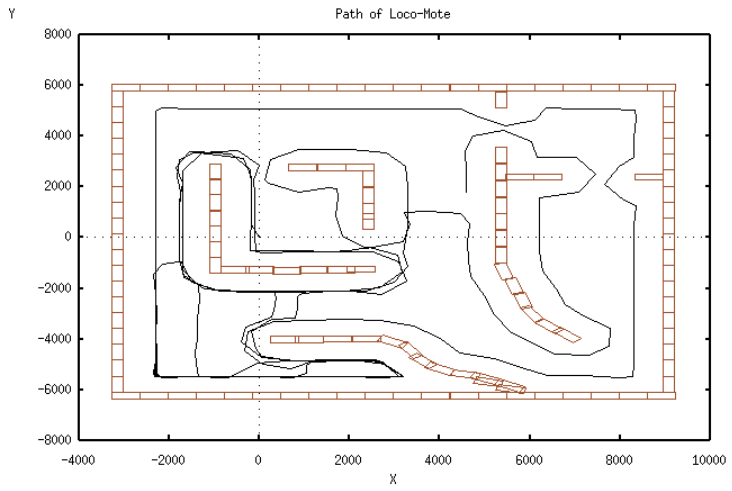


Figure 8-13: Eventual result of prowling

At this point the robot has explored all the boundaries in the test environment. From here on, the robot will patrol what it has already explored, checking to see if the environment changes, as shown in Figure 8-13.

8.2 Physical Robot

8.2.1 Boundary following

In this test case, the physical robot exhibits boundary following in a maze.

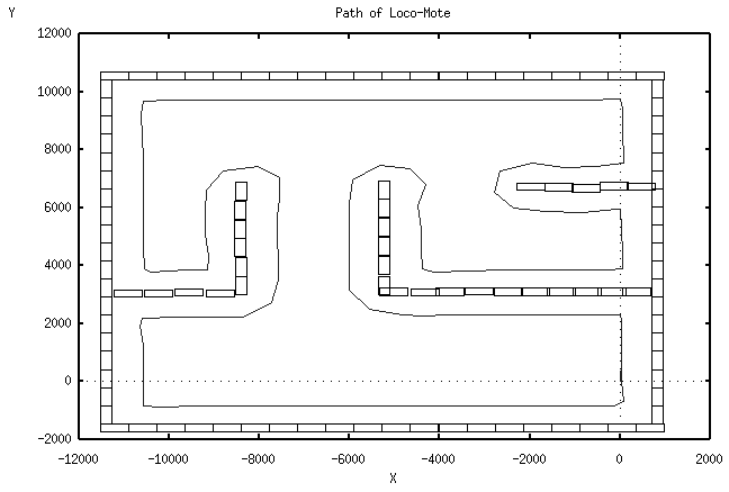


Figure 8-14: Shape of standard maze

Many of the experiments on the physical robot were carried out in a standard maze, as shown in Figure 8-14. When the visualisation tool (from which all these snapshots are taken) is run

with the simulator, it automatically merges a sketch showing the shape of the robot's environment on to the graph of the robot's position. This is not possible for the physical robot, since by definition it has no global knowledge of its environment, and its coordinate system is only guaranteed locally (so global graphs may not be consistent). So for reference purposes, the approximate shape of the maze is given here, with the path that the simulated robot follows when edge following its virtual analogue.

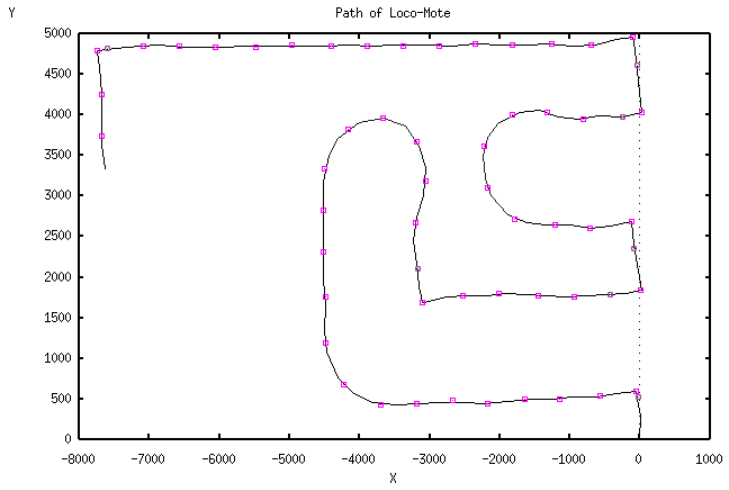


Figure 8-15: Path followed by physical robot while edge following

Figure 8-15 shows the shape the robot traces out while edge following the standard maze. Note that it smoothly navigates around both convex and concave turns, and follows edges accurately. The robot's odometry is quite accurate during edge following because its motions are smooth.

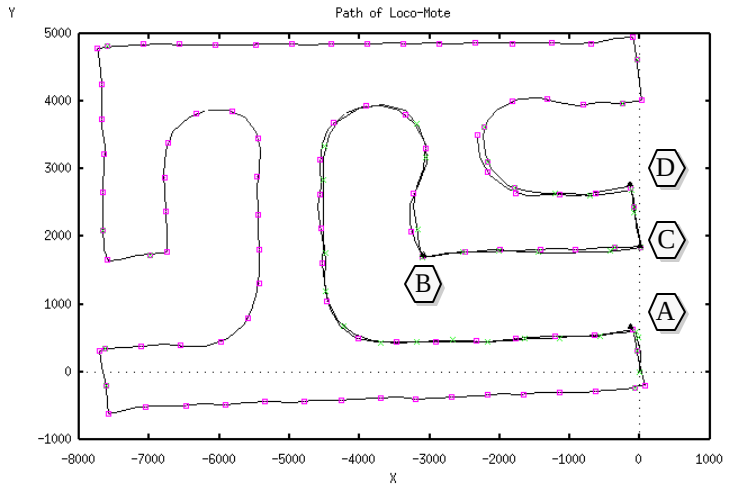


Figure 8-16: Physical robot maintaining consistent map

In Figure 8-16, the robot has fully circumnavigated the maze. At concave corners, such as A, B, C, and D, the robot can compensate for accumulating error in its odometry by comparing its best-guess position now with what it was last time it passed a similar corner in a similar

position. Compensation also occurs along straight edges, such as between B and C. As a result, the robot's path is kept consistent.

8.2.2 Use of landmarks

8.2.2.1 Corner landmarks

In this test case, the robot follows the boundary of the standard maze, and uses corners as landmarks at which to calibrate its position.

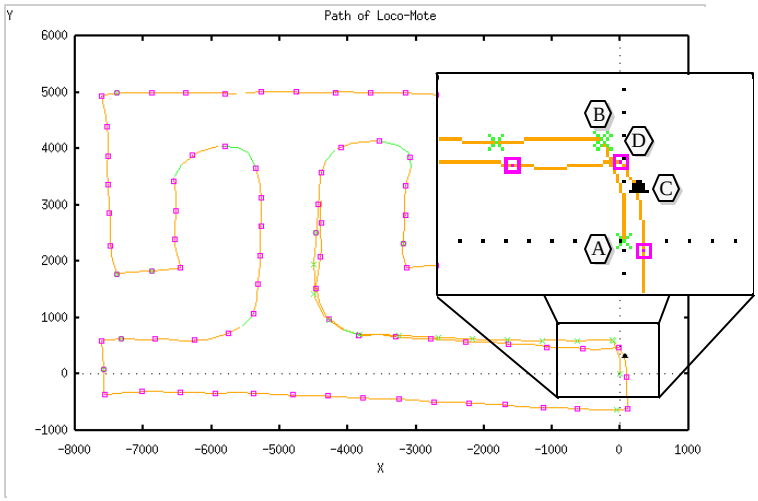


Figure 8-17: Use of corners

Figure 8-17 shows a magnified view of the operation of the robot at a corner.

- The robot starts at position A, and follows the boundary beside it.
- A corner is detected at position B.
- The robot continues to edge-follow, and eventually comes back to the same area.

When it meets the corner this time round, its best guess at its position is C. It seems to have changed location. The robot cannot tell if this is due to a change in the environment, or to accumulating errors in its odometry (as it is in this case). Therefore it averages the apparent position of the corner and its previous position to get a new best guess, D.

8.2.2.2 Edge landmarks

In this test case, the robot follows the boundary of the standard maze, and uses edges as landmarks at which to calibrate its position.

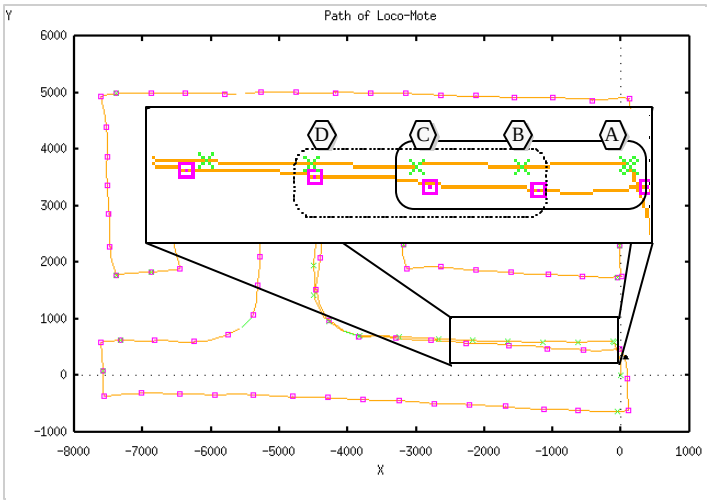


Figure 8-18: Use of edges

In the area shown in Figure 8-18, the robot is following a boundary that it has met before, but which appears to be at a different position because of accumulated error in its odometry.

- The robot had previously placed markers along the edge at A, B, C, etc.
- As it follows the boundary a second time, it places new markers overriding the out of date markers, as shown.

When it has laid markers at A, B, and C, the robot detects that these markers are in a straight line beside an older set of markers that were also in a straight line in a direction close to the current one, and somewhat displaced. This information is used in an averaging process where the best-guess angle and position of the robot is adjusted, as shown.

Experimentally, the physical robot's sense of position tends to be most accurate if the robot makes smooth motions as occur in edge following. Under these conditions, position correction using corners and edges can maintain a consistent sense of position within the robot. Hence in the prowling behaviour it was, pragmatically, important that the robot spent a good deal of its time edge following, since this is the only time the robot can extract useful information from its environment for any purpose, not just position correction.

8.2.2.3 Ablation study

In this test case, the utility of the use of landmarks is demonstrated by removing them. The robot follows the boundary of the standard maze with landmark detection disabled.

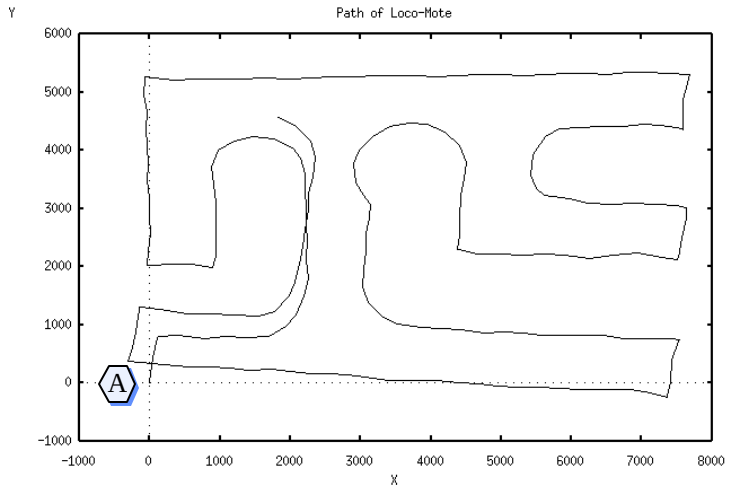


Figure 8-19: Position drift

In Figure 8-19 a trace of the robot's best-guess of its position over time is shown while edge-following a closed boundary. Note that when it reaches its starting point around A, error has accumulated in its position sense and it is starting to deviate significantly from the robot's true position.

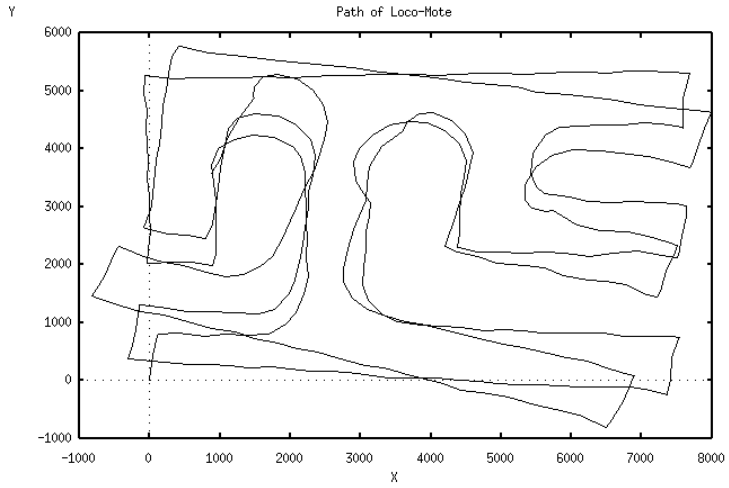


Figure 8-20: Aliasing of topological features

The error in the robot's estimate of its position continues to accumulate as it moves. In Figure 8-20, the robot is circling the boundary for the third time. Topological features have started to overlap, and the robot's map is useless- worse than useless, misleading.

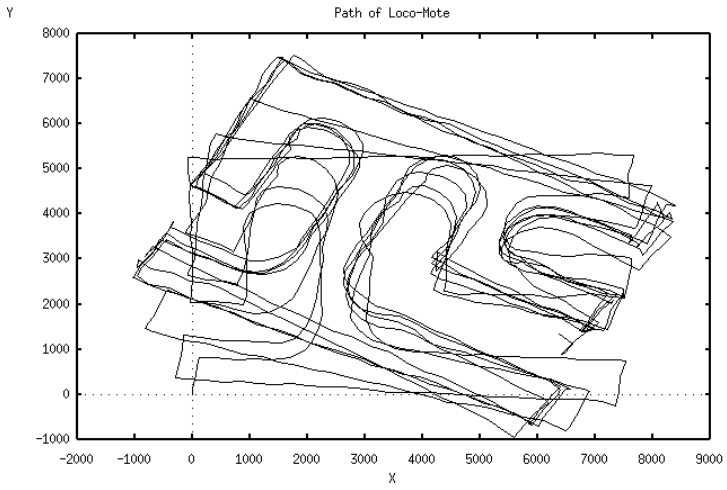


Figure 8-21: Recovery when landmarks reintroduced

The robot was left to continue circling for a while, and then the use of landmarks was turned back on. The trace in Figure 8-21 shows that the robot eventually started to develop a consistent map of the boundary it was tracing. This is seen where a cluster of lines make a seemingly thicker line, showing that the robot started to trace the boundary in a consistent manner.

8.2.3 Target seeking

8.2.3.1 Map search

In this test case, the physical robot escapes from a cave-shaped obstacle using a background map search.

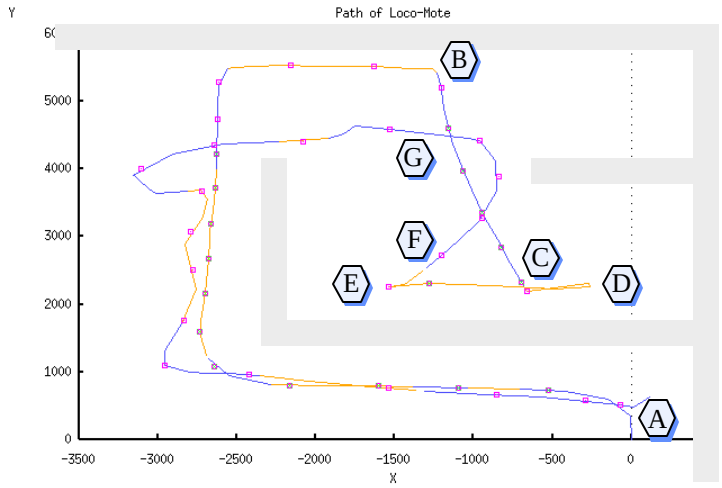


Figure 8-22: Target seeking in the physical robot with map search enabled

Figure 8-22 shows the performance of the robot using both physical search and map search to escape from an obstacle. The robot started at A, then was directed to B by user control. Then it was asked to find A. (Note that the approximate shape of the maze has been superimposed by hand on the trace).

- The robot moved from B towards the target until it hit an obstacle at C.
- It turned to follow the boundary to its left for a bit, with no immediate success, so it turned back to check the other direction.
- The robot had no immediate success to its right either, so it turned back again at E.

At F, the background map search succeeded and the robot found a possible path to the target in its map. At F it switched from physical search to using the results of the map search, by retracing the path it had been taken from A to B. Note that the robot can still take local short cuts such as at G while following a path, and if a better path is returned by the ongoing background map search, it will switch to that.

8.2.3.2 Physical search

In this test case, the physical robot escapes from a cave-shaped obstacle using physical search alone.

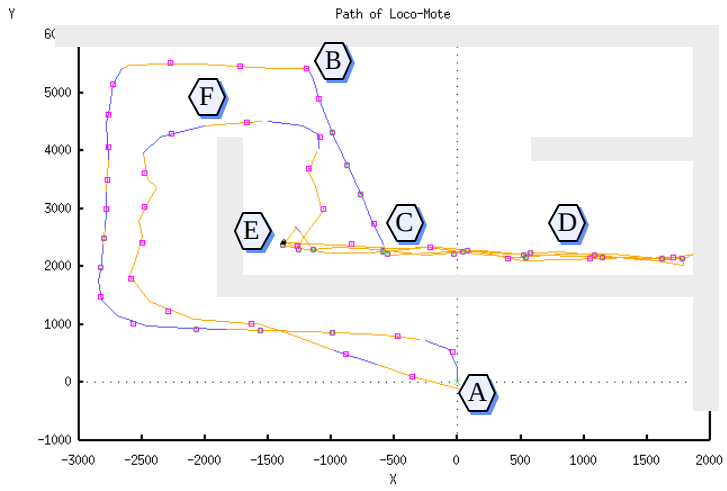


Figure 8-23: Target seeking in physical robot using physical search only

Figure 8-23 shows the performance of the robot using physical search only to escape from an obstacle. As before, the robot started at A, then was directed to B by user control, and then was asked to find A.

- The robot moved from B towards the target as before, until it hit an obstacle at C.
- The robot searched out towards D, then turned back to check the other direction.
- The robot had no immediate success towards E either, so it turned back towards D.
- This continued with the robot extending its search further out towards D and E until it escaped the obstacle at F.
- Since the target could not yet be directly approached, the robot continued to follow the boundary until it turned and the robot could go straight towards the target.

8.2.4 Prowling

In this test case, the robot explores the standard maze using the prowling behaviour.

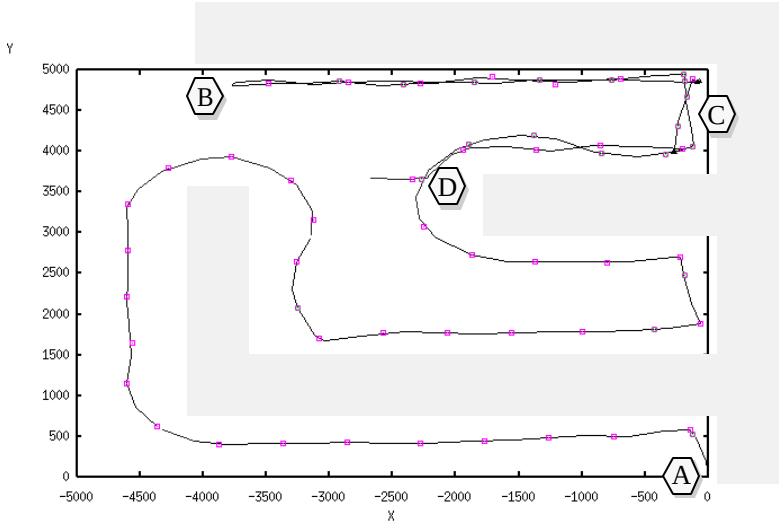


Figure 8-24: Early stages of prowling

Figure 8-24 shows the robot prowling in the standard maze. It started by finding an obstacle (at A) and attempting to trace its boundary. In this case, the boundary is very long, and at B the robot turns back because its sense of confusion has risen above a threshold. This prevents it from wandering too far and losing track of itself. At C, the robot finds and recognises corner landmarks, regains confidence, and its confusion reduces. At the first likely looking point, D, it sets out to explore again.

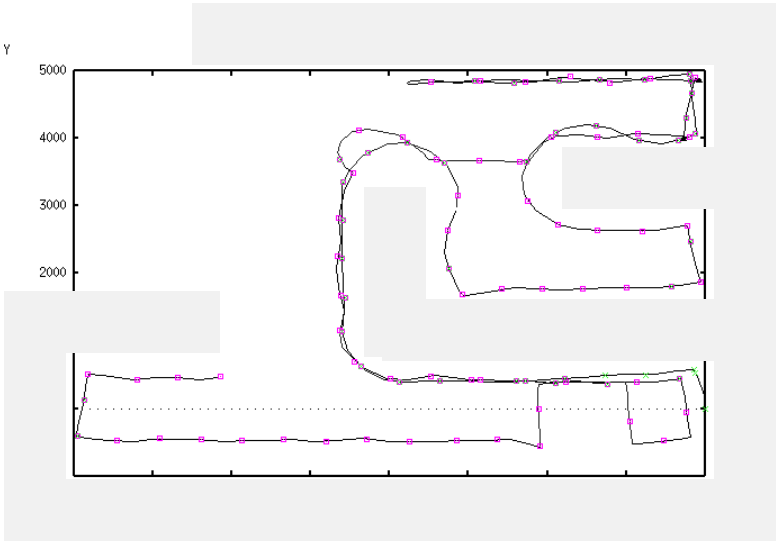


Figure 8-25: Further exploration

In Figure 8-25 the robot continues with its exploration, which because of the shape of the maze tends to lead to somewhere it has already been. Its territory is extended more by edge following, as shown, than by vector exploration.

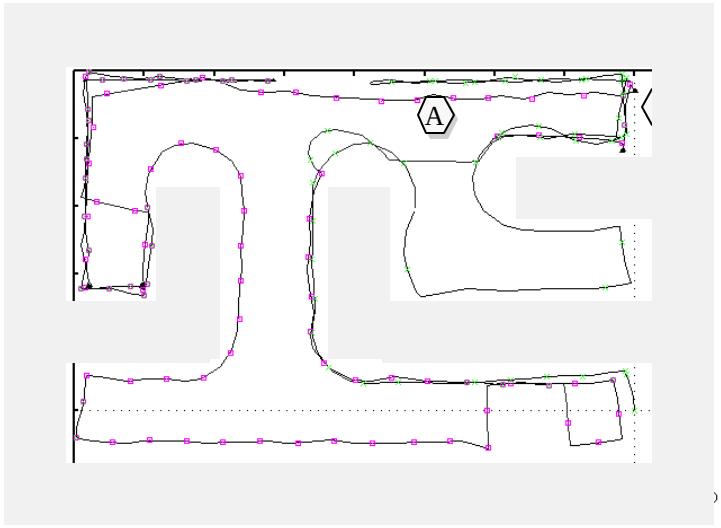


Figure 8-26: Combination of exploration and patrolling

At the stage shown in Figure 8-26 the robot has explored most of the maze. In area A, the robot was experiencing increasing errors in its position, which were rectified at B. The map, overall, retains its consistency despite errors occurring in any part of it. So long as errors are not too violent, feedback from the rest of the map will correct any damaged section of it.

8.3 Comparative results

Making meaningful comparisons between autonomous robots is a thorny issue. A first step in performing comparisons in any branch of science is generally to try to eliminate environmental influences that could interfere with the subject under scrutiny. The problem in robotics is that a robot designed to be autonomous is *nothing* without its environment- the whole significance of what such a robot does is bound up in how it is affected by environmental influences, and how it responds to them [[11]]. Trying to test an autonomous robot in isolation for this reason makes little sense, and simulated approaches are also weak, as explained in Section 2.4.7, page 22. The most satisfactory solution is the use of standard environments with equally standard "bench-mark" tasks for the robot to perform. No such standard bench-marks currently exist for autonomous robots that can be implemented by a research group to examine their own robot or to replicate the results of another group. The closest to them are robot "Olympics" where robots compete either directly against each other (by trying to physically disassemble each other, for example), or indirectly by performing a given task in a given environment. This last is a form of bench-mark, except it cannot be performed at will in a laboratory to- for example - validate someone else's claims for their robot. A bench-mark that could be applied in this fashion is difficult to imagine anyway, given the huge variation that will exist between any two robots in terms of physical attributes, sophistication of processor(s), etc. It is hard to find compare like with like when there seem to be no two "likes" to be found. While these issues do make it difficult to compare in detail the particular robot application built in this project with that of other projects, the problem is not so severe. This is because the project is less concerned with getting a robot to do something that has never been done before than it is with showing that behaviour that would normally be expected to require vision or sonar sensors can in fact be achieved with much less. So it is sufficient to compare the operation of the robot in this project with the operation of other robots in the literature with more sophisticated sensing equipment, and show that it can successfully complete qualitatively similar tasks. In this way, abstract arguments of about the relative merits of robots with different sets of features can be avoided.

The rest of this chapter compares robots with long-range sensors performing various tasks with the Khepera robot performing similar tasks. The tasks are not chosen very systematically, since they are limited to what is reported in the literature for the different robots.

8.3.1 Comparison with the ACBARR system

This robot architecture was described in Section 2.5.3 (page 31). The results presented for it are from a simulator- the system was to be implemented on GEORGE, a robot with several ultrasonic sensors around its body for long-distance obstacle detection. The *schemas* used in the architecture requires a good estimate of the position of all obstacles to be found from sensor data and/or pre-entered data, so it was chosen as a good contrast to the robot in this thesis.

8.3.1.1 Case 1 - Canyon

First we compare the robots trying to reach a target, but with a "canyon" in the way. Note that only an approximation to the original test-case can be replicated, but it is sufficient for comparing the qualitative nature of the robot's behaviour.

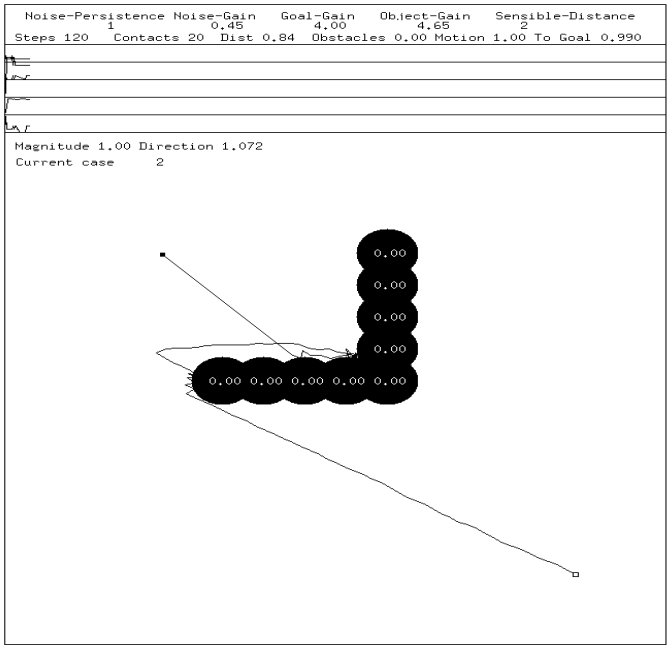


Figure 8-27: GEORGE in a closed canyon

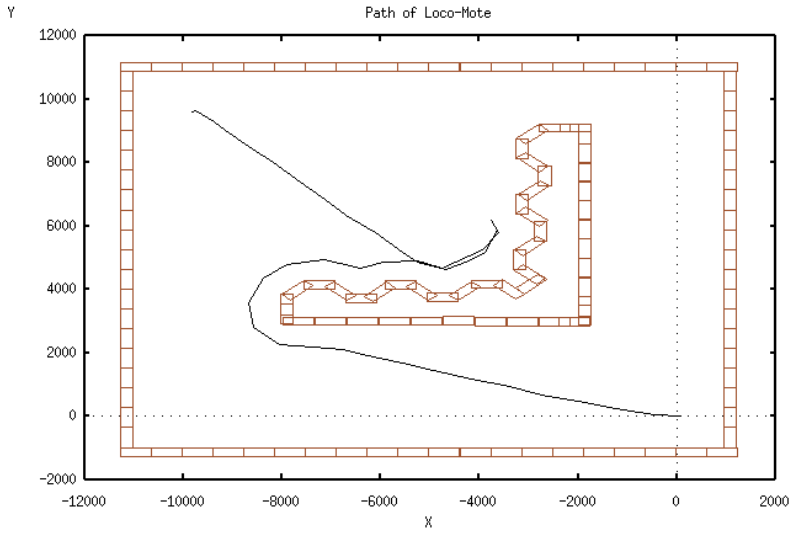


Figure 8-28: Khepera in a closed canyon

Figures 8-27 and 8-28 show that both the robots perform in a similar way. The diagram appears to show that Khepera escapes the canyon along a much smoother path, but conclusions such as this cannot be backed up without access to more data than is publicly available about GEORGE.

8.3.1.2 Case 2- Opened Canyon

In Figure 8-29 a gap is placed in the canyon, and the robots find their way through the centre.

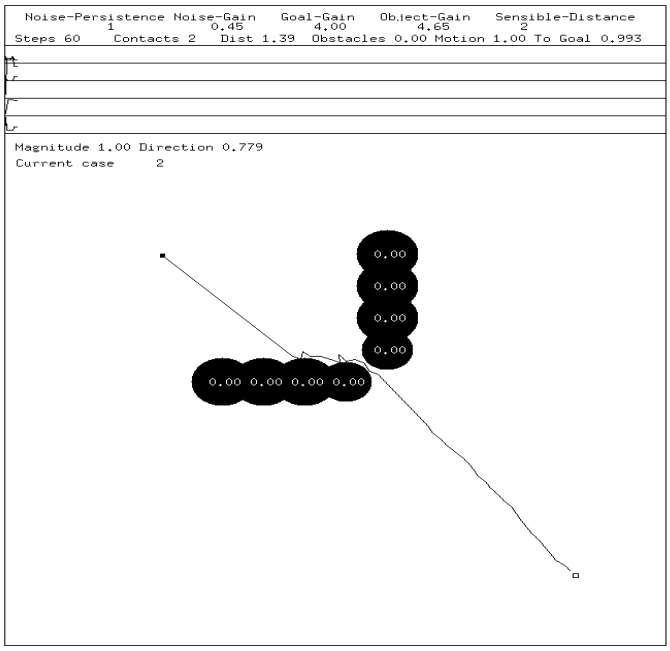


Figure 8-29: GEORGE at an opened canyon

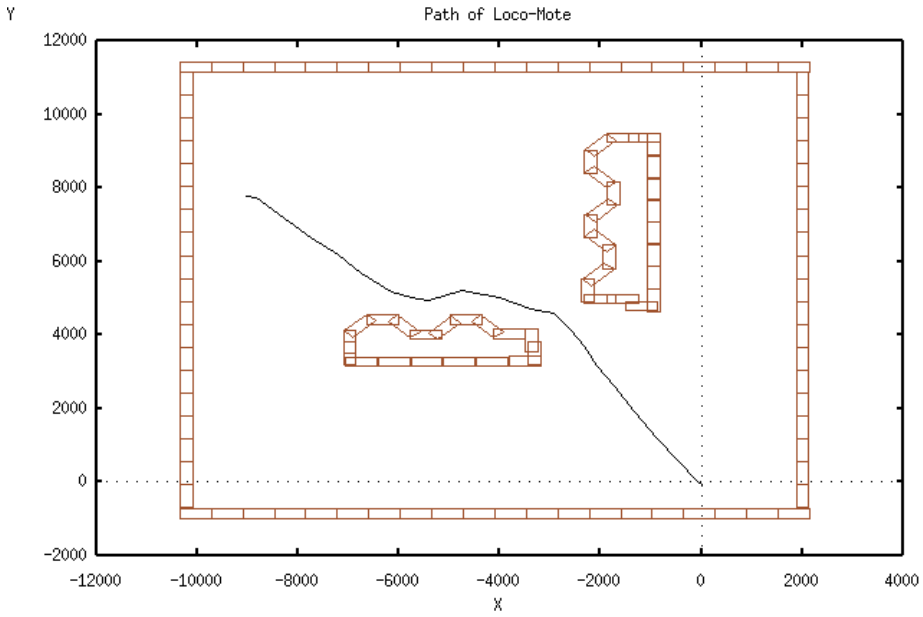


Figure 8-30: Khepera at an open canyon

The trace in Figure 8-30 shows that neither Khepera nor GEORGE have any trouble getting through gaps like this.

8.3.1.3 Case 3- Simple Wall

In Figures 8-31 and 8-32, the robots navigate their way around a blocking wall.

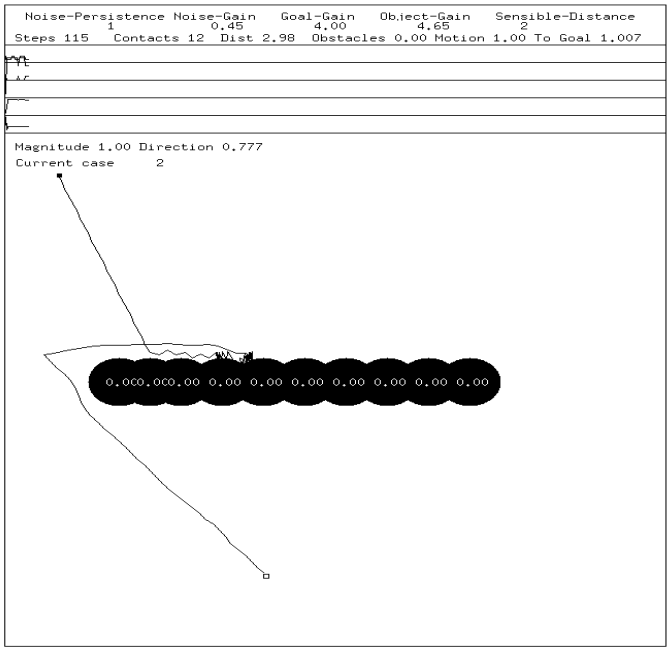


Figure 8-31: GEORGE at a wall

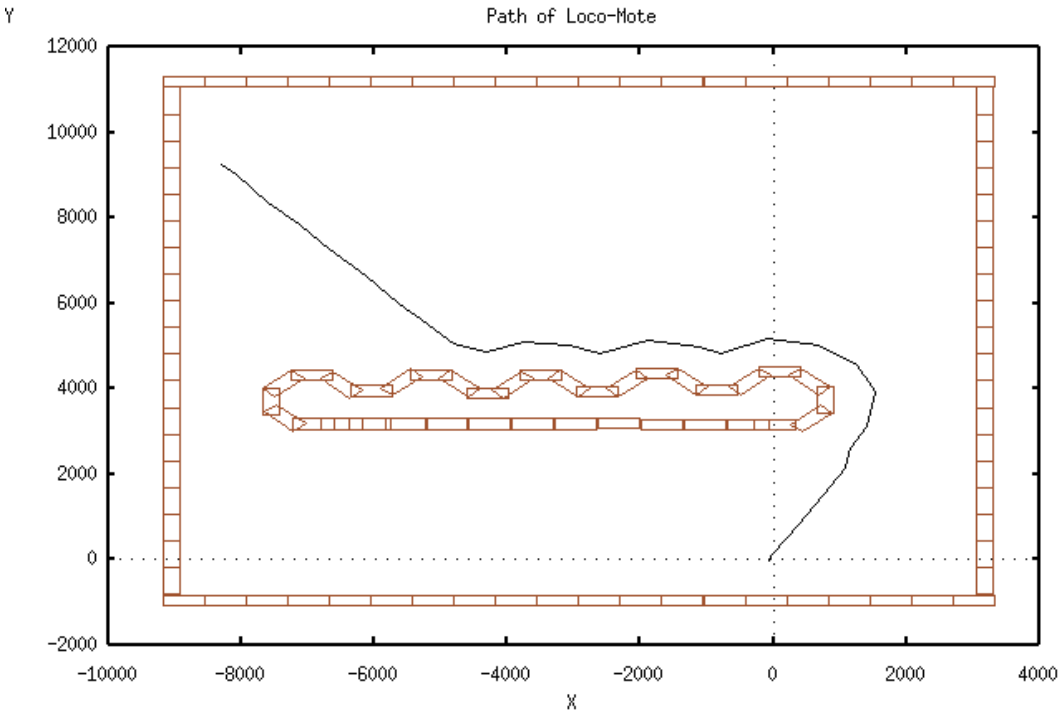


Figure 8-32: Khepera at a wall

The traces show again that Khepera behaves just as well as the robot with ultrasonic sensors. Note that the robots chose different paths around the wall. In different runs of Khepera for different start and goal positions, it would go in different directions around the wall. If the wall seemed to be taking it away from its target for too long, Khepera would turn back and try the other way for a while. Theoretically, a robot with long-range sensors could make a better guess than Khepera could at which way to move along the wall to get to its target quickest, because it can simply look and see which appears shorter. In practice, Khepera's strategies work very well even without the ability to do this, as described earlier in this chapter.

8.3.2 Comparison with Scarecrow

This next robot has 16 ultrasonic range sensors. It is shown in Figure 8-33 moving towards a goal through a fairly cluttered environment. Notice how the Khepera robot in Figure 8-34 works equally well, even though it has no access to long range sensors.

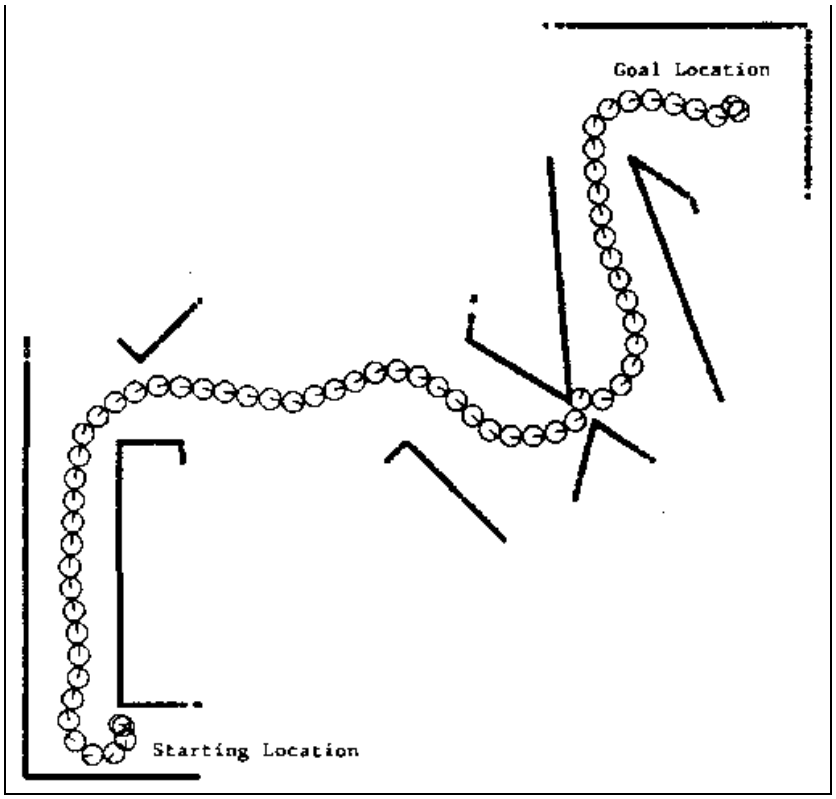


Figure 8-33: Scarecrow moving through a room

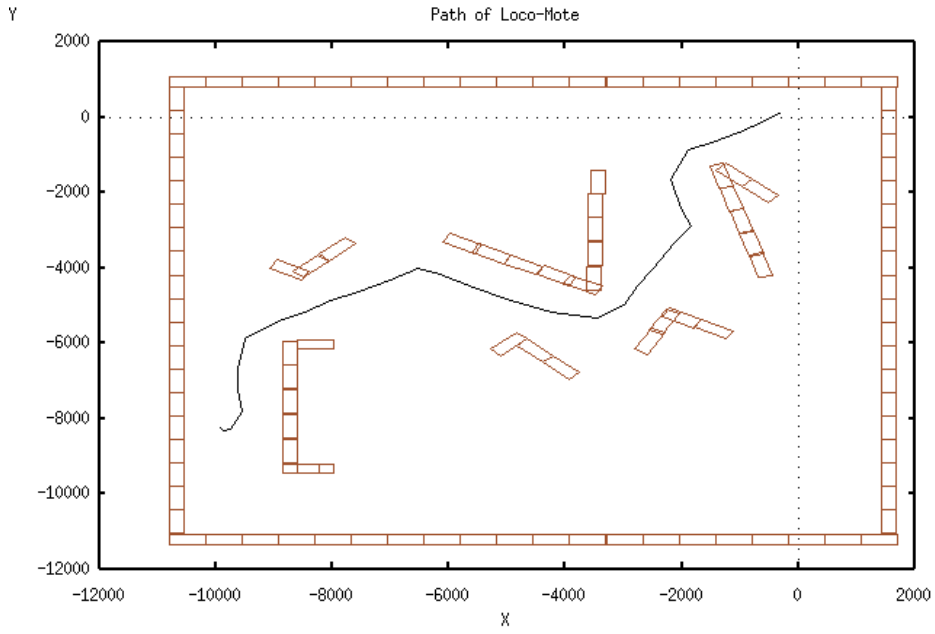


Figure 8-34: Khepera moving through a room

8.3.3 Comparison with Robbie

The robot with which Khepera is compared in this section, "Robbie", has stereo vision cameras. The robot moves through a cluttered environment made of discrete small obstacles rather than walls. The robot is more complex physically than Khepera is- it is a four wheeled vehicle rather than a robot with a uniform circular cross section like Khepera.

8.3.3.1 Case 1

The robots are shown here moving towards a goal in Figures 8-35 and 8-36. Khepera acquits itself quite well- working with scattered small objects as easily as it would with walls.

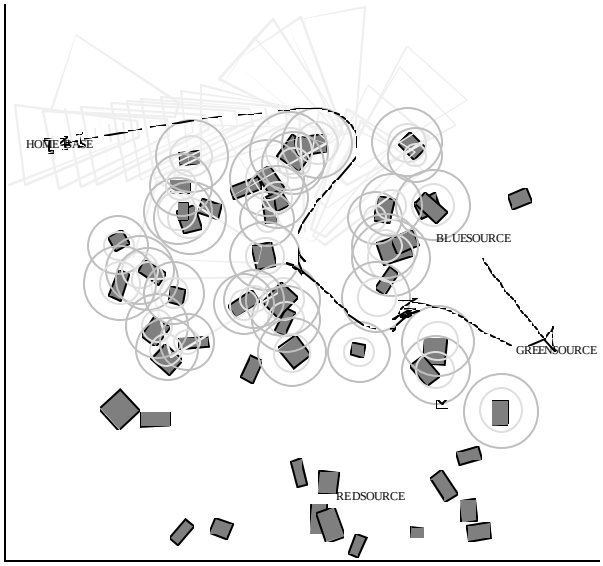


Figure 8-35: Robbie moving towards a target

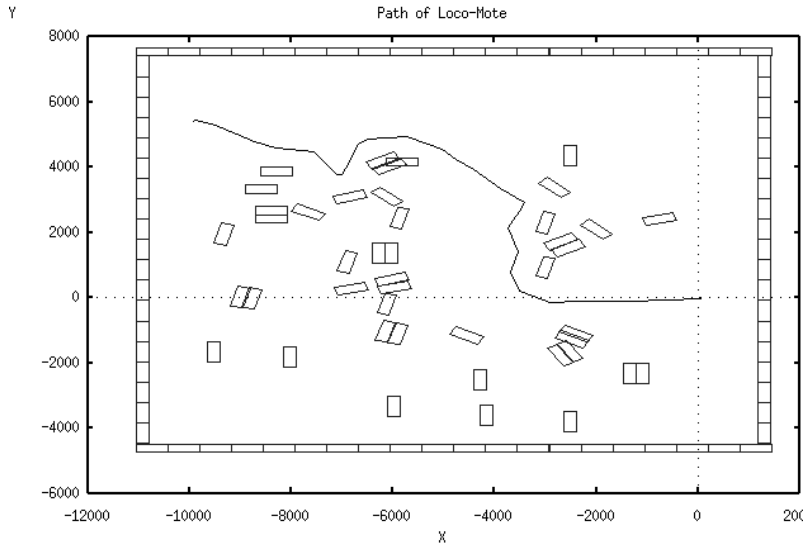


Figure 8-36: Khepera moving towards a target

8.3.3.2 Case 2

In this case, the robots are shown recovering from taking a mistaken navigation decision. At the source, the robot can move in two directions, both of which seem to lead to the target, but one of which is actually blocked from the target. Figure 8-37 shows the action of Robbie.

Figure 8-37: Robbie recovering from a mistake

Two runs of Khepera are shown. For this particular test case, Khepera does not make the mistake that Robbie makes (Figure 8-38). By changing the starting point, Khepera can be made to take the wrong initial choice, and this is shown in Figure 8-39.

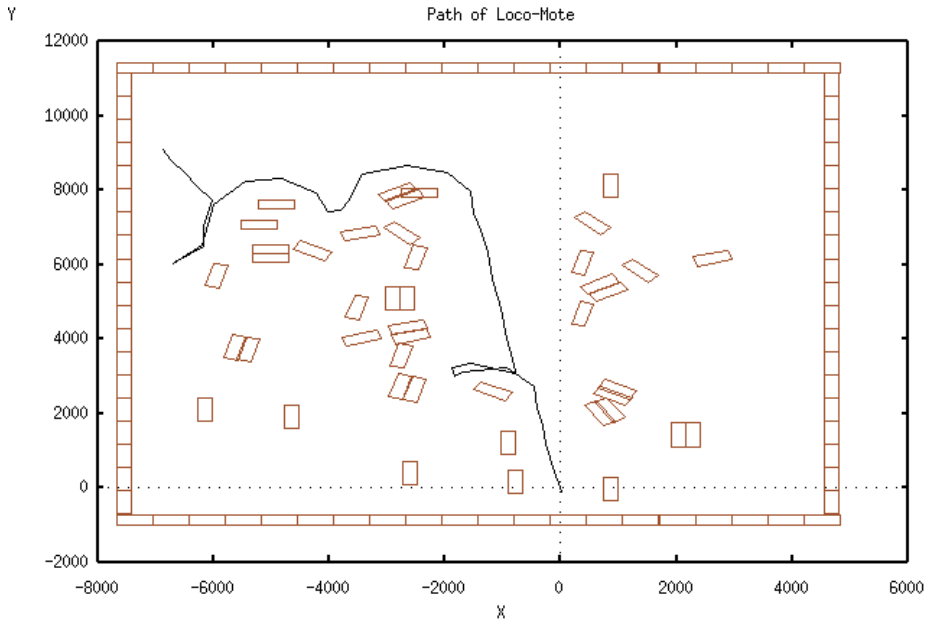


Figure 8-38: Khepera making the right choice

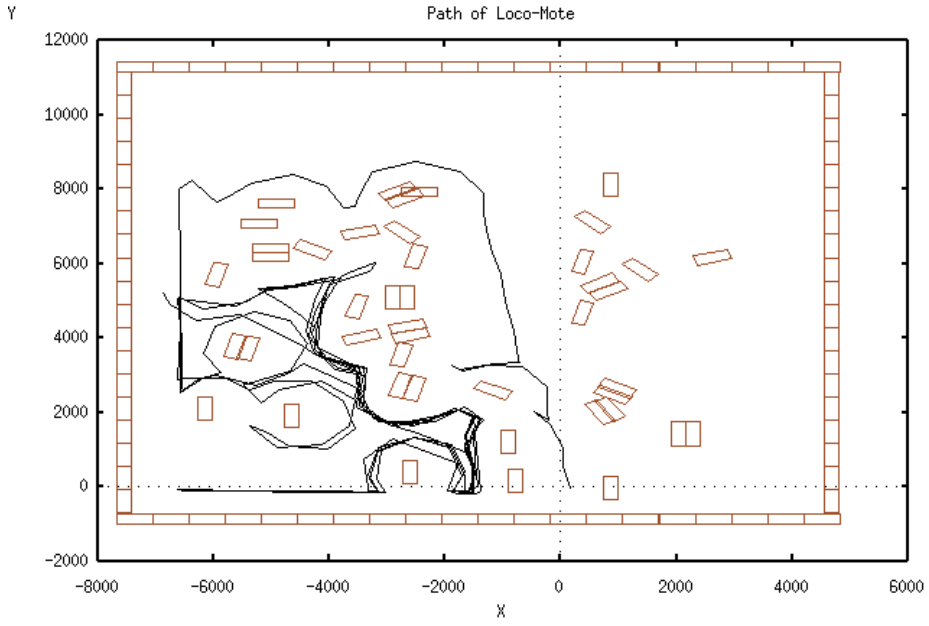


Figure 8-39: Khepera making the wrong choice, and recovering

Khepera does succeed in back-tracking out of the trap. It takes more work than for the robot with vision, because it has to do more searching. Note that it follows the bottom horizontal

wall for a while, in case there might be a gap further on. It gives up when it sees the wall turn back upwards.

As is shown in this trace, Khepera's physical search behaviour when trapped can lead to quite an amount of movement back and forth through the same area as the robot extends its search boundaries in the two directions along the boundary of the obstacle (or in this case, group of obstacles) blocking it. This was a trade-off for the considerable advantage the scheme has of avoiding the possibility of *looping*, where an unforeseen boundary shape leads to the robot entering a search loop from which it cannot escape.

8.4 Summary

This chapter presented results from various test cases applied to the physical and simulated Khepera robot. They served to demonstrate that the cartographic system developed in Chapter 4 was practical, and that the behaviours developed in Chapter 5 functioned as anticipated. The robot was shown to compare favourably with other robots possessing better sensing capability.

9. Conclusions

This chapter discusses how well the work presented in this thesis met its aims, and how it could be built upon and extended in the future.

9.1 Discussion

The discussion of the work is split between the "sentry application" developed and the Lateral robot architecture.

9.1.1 Sentry Application

The sentry-like behaviour developed for the robot in this thesis demonstrated that it is possible to navigate intelligently with proximity sensors only, provided that the following constraints are adhered to:-

- The robot must be able to keep track of its position to a good degree of accuracy. It can do this by summing its motion or other means. The position estimate need only be accurate on a local scale- it is acceptable for accumulating error to occur.
- The robot must avoid being in motion for an extended length of time in an area it is "unfamiliar" with- an area that it has never been in before. In such a circumstance, it has no information with which to compensate for accumulating error.
- The robot must ensure that it frequently passes by recognisable features of its environment that it can use as landmarks to keep its position estimate consistent across time.

If these constraints are met, then it has been demonstrated that the robot can create and maintain a useful internal map of its environment. A *prowling* behaviour has been shown to be practical for a robot with only proximity sensors. The robot can behave as a sentry, exploring its environment, patrolling it exhaustively, and reacting to any changes in it. Although the application is called "sentry duty", the exact same behaviour would be useful for any robot that has to exhaustively and repeatedly traverse a floor-space, such as sweeping, cleaning or polishing robots, or security robots in a gallery, etc. It is also a very useful basic behaviour for

use as a periodic "learning" phase for a robot engaged in any task involving navigation[68] so it can become familiar with its environment in an autonomous manner, and then use the knowledge it gains autonomously to accomplish specific, externally imposed tasks. For example, in this project the robot could be ordered to seek a special location, and it would navigate towards that location using a combination of information from searching its map and performing intelligent "physical search" strategies that allow it get around obstacles, back out of dead-ends, etc.

Chapter 8 showed that the performance of the robot compares favourably with the performance of other robots using more sophisticated sensing equipment. This demonstrated that the cartographic system implemented was in fact successful in extracting relevant information from the environment and presenting it to the control system in a timely fashion, and could maintain a consistent map of the environment over time.

The memory requirements of the cartographic system used grow with the ratio between the area of the region the robot is required to patrol and the area a single marker in the map is taken to represent (see Section 4.3.2, page 93). In this project, the physical robot had limited memory available for mapping (approximately 50kB), enough for a region approximately 400 times its own area (about 20×20 times its dimensions), with markers representing an area of about one third of the robot's body. Features of smaller granularity were found to be negligible. On a larger robot with less constrained memory, mapping a floor of a building would be perfectly feasible, since both storage and processing requirements scale well. The storage requirements increase proportionately to the area and the processing requirements increase only with the logarithm of the area because of the neighbourhood system (see Section 4.3.1, page 88).

9.1.2 Lateral Robot Architecture

The motivation for creating the Lateral architecture was to make behaviour combination simpler than is possible in other behaviour-based architectures. The success of the architecture can be judged from the nature of the system decomposition used in Chapter 5, for example.

[68] Technically "steerage". Sometimes navigation is taken to mean moving through an environment without any a priori knowledge of that environment, and steerage indicates movement through an environment about which the robot has a certain amount of information.

Instead of being forced to place all behaviours in a rigid layered hierarchy, a richer and more flexible hierarchy is possible. Subsumption forces behaviours into a single inheritance tree, where a single behaviour is taken and modified to give another more specialised behaviour [[7]]. Lateral, on the other hand, lets a behaviour make use of other behaviours as tools, without necessarily being a specialisation of any of them unless that is appropriate. The difference is somewhat analogous to IS-A versus HAS-A relationships in software engineering. While a behaviour may be an "enhancement" of one that is already present (IS-A relationship), it may also be a combination of a group of other behaviours (similar to a HAS-A relationship). Lateral will allow either possibility, while Subsumption supports the first possibility exclusively. The reason Lateral is more flexible is largely because it has a dynamic priority system rather than a static one. This means that the effective behaviour hierarchy (determined by the relative priorities of the behaviours) can change at runtime, whereas in Subsumption it is fixed. At any particular instant, a decomposition made using behaviour combination could be collapsed into one that uses behaviour enhancement only, since that is sufficient to represent the relative priorities at that instant. However, when priorities change, the equivalent decomposition using enhancement changes also. Hence Lateral in effect allows the effective decomposition to change as the priorities of behaviours change.

The practical usefulness of this became apparent during design. For example, although the main set of behaviours implemented in this project were concerned with patrolling, prowling, and exploring, when it came to implementing a "Region seeking" behaviour, it could be superimposed on the decomposition already present, reusing behaviours without disturbing the existing decomposition (see Section 5.4.2, page 159). In general it turned out to be very easy to "superimpose" two overlapping behaviour hierarchies, with the architecture handling any conflicts between them transparently.

Another new feature that proved beneficial in Lateral was its use of local priorities in contrast to the global priorities implicit in Subsumption. In Lateral it is not necessary to assign global priority levels to behaviours to resolve conflicts between them. Instead each behaviour assigns priorities to its outputs that are chosen relative to its inputs, and the Lateral system uses these to deduce the effective relative priority of behaviours at runtime. This means that while building a behaviour, the programmer can concentrate on the local view alone, and the architecture

automates the process of using these local decisions to determine the global activity of the system.

Lateral is a superset of Subsumption, so it does not lose any of the features that made that architecture so successful. As described above, at any instant a behaviour decomposition made in Lateral has an equivalent decomposition in Subsumption. There is another more direct link. If connections are at the same level of priority, then arbitration between them is achieved using rules analogous to those in Subsumption- there are equivalents to the "suppressing", "inhibiting", and "defaulting" connections of Subsumption. So if the dynamic priority of all behaviours were set to a single fixed value, all arbitration would be done following the same rules as Subsumption would use.

9.2 Future work

This section describes how the work presented in this thesis could be built upon and extended in the future. There are a number of areas in this project that could be expanded on:-

- The Lateral runtime support could be re-implemented in a distributed processing environment. All the work done in this thesis was geared towards being executable in a simple processing environment without multi-tasking. This fits the theme of working with cheaper, lower-end robots and seeing what can be done with them. However it would also be useful to implement support for the Lateral architecture with real multi-tasking and multiple processors, which is the nature of the control system on more sophisticated robots.
- There are some "magic numbers" chosen for various important parameters of the robot (see Sections 4.4.2 and 5.2.3 in particular, on pages 99 and 129 respectively). It would be useful if the robot could derive these itself to suit its environment, rather than having them pre-set by the programmer. In particular, the granularity of mapping was manually chosen. Future work could allow this parameter to be chosen by the robot itself. In fact there is no need for the granularity to be constant- it could easily be increased in the locality of intricate obstacles and reduces in areas of large empty spaces.
- The robot's behaviours could be improved on almost indefinitely by implementing alternate strategies that the robot can use if the ones it has do not succeed, and adding better

facilities for cognisant failure (see Section 2.5.6, page 34) so that the robot can recognise when it is in trouble and should switch strategy.

- The use of landmarks could be extended somewhat. Currently two types of landmarks are recognised- corners and approximately straight boundary sections. There is no theoretical problem with recognising more complex boundaries- but there is the practical problem of trying to do this in real-time, avoiding computationally expensive algorithms. It would be interesting to see how much more could be done in this area without simply requiring the robot to have a faster processor.

Appendix A

A1. Khepera specifications

Processor	Motorola 68331
RAM	128Kb
ROM	256Kb
Motion	DC motors with incremental encoder
Sensors	8 infra-red proximity and light sensors
Power	Rechargeable Ni-Cd batteries or external
Autonomy	30 minutes
Size	Diameter 55mm, height 30mm
Weight	About 70g

A2. Implementation platform

The Lateral architecture support developed had to work on:-

- a PC running Linux (for the Khepera simulator)
- the Khepera robot with its own mini-operating system
- DOS (for an early use of the system with another robot built within the college)

It was important for this project that it be grounded in a physical robot. At the start of the project the robot available for experimentation was Z80-based with a serial link to a controlling PC running DOS. It was hoped that funding would become available for purchase of a robot capable of autonomous operation, but the nature of this hypothetical robot was not known. It was therefore important that the Lateral support be developed to work in systems that met minimal requirements.

A3. GNU CC Cross-Compiler

To compile C++ code for Khepera, the GNU CC Cross Compiler was used, running under Linux on an x86. To use GNU CC as a cross-compiler, two components are required: the compiler itself, and some associated utilities. The compiler and utilities are publicly available software. Installation is straightforward, and consists of building the compiler for a nominal target of "sun2", which is the appropriate target for Khepera. The cross-compiler cannot be built completely without some additional library and header files tailored specifically to Khepera and its BIOS. These are supplied by the manufacturers of Khepera.

A4. Relevant Khepera BIOS Services

A list of the BIOS services relevant to this project are given in the table below. Their nature is summarised here.

- The serial line is represented as standard input and output to Khepera's control system. No special function calls are needed to interface with the serial connection.
- The robot has a time service, accurate to milliseconds.
- The robot implements a simple multitasking system which can execute at most 15 user processes concurrently using a fixed time quantum task switching scheme. Tasks can be put to sleep for a fixed number of milliseconds and then re-awoken as normal. Task switching can be blocked if required to allow operations that must be guaranteed atomic to be performed safely.
- As described earlier, control of the robot's motors is allowed in terms of speed setpoints (from -10 to +10, fast reverse and fast forward respectively) or distance to rotate. Full control of the parameters of the robot's PID controllers is made available, but is not used in this project.
- Feedback from the motors is in terms of the distance each motor has rotated.
- Two LEDs on the robot body can be controlled during normal operation of the robot. At other times they indicate the progress of a download or error status.

- Proximity and ambient light measurements can be taken from the eight sensors distributed around the circumference of the robot's body. A proximity level of 0 means nothing is perceived near the sensor, while values of around 1000 mean that there is some object close to the sensor. Light readings vary from about 500 in the dark to about 50 directly in front of a light source.

tim_get_ticcount	Returns the number to milliseconds since the last reset
tim_new_task **tim_suspend_task**	Adds a new function to be executed within Khepera's pre-emptive multitasking system Put a task to sleep for a specified period
tim_lock	Temporarily prevents time sharing. Used when entering a critical region.
tim_unlock	Permits time sharing after a tim_lock()
mot_new_speed_1m	Sets a new speed for one of the motors
mot_get_position	Gets the absolute position of one motor
mot_new_position_1m	Sets the absolute position of one motor
sens_get_reflected_value	Gets the reflected value of one infra-red sensor (proximity)
sens_get_ambient_value	Gets the ambient value of one sensor (ambient light)
var_on_led	Turns on an LED
var_off_led	Turns off an LED
var_change_led	Toggles the state of an LED
standard I/O	The serial link acts as standard input/output
bios_reset	Perform a software reset of the robot

A5. Khepera Simulator Programmatic Interface

Motor interface

```
struct Motor
{
  double    X,Y,Alpha;
  short int Value;
```

```
};
```

Sensor interface

```
struct IRSensor
{
  double    X,Y,Alpha;
  short int DistanceValue;
  short int LightValue;
};
```

General interface

```
struct Robot
{
  u_char            State;
  char              Name[16];
  double            X,Y,Alpha;
  double            Diameter;
  struct Motor      Motor[2];
  struct IRSensor   IRSensor[8];
};
```

/* X and Y (millimetres), Alpha (radians) */

A6. Communications with Khepera

Khepera can be operated in a number of communication modes, configured by a set of jumpers. Here is a summary :-

- Some modes configure Khepera to listen for commands and requests over the serial link at various baud rates.
- The robot can alternatively be configured to expect a program download in a defined format across the serial link (again at different baud rates).
- Some auxiliary modes, not specifically concerned with communication, are provided to :-
  - ⇒ Run a hardwired demonstration (based on a Braitenberg vehicle style algorithm)
  - ⇒ Execute some self tests and report their outcome across the serial line
  - ⇒ Execute an application stored in an EPROM.

For this project, Khepera was kept in a command accepting mode rather than a downloading mode, listening for commands at 38400 Baud. This is convenient because in this mode it is possible to issue a command ordering the robot to transition to a download mode without having to change jumpers, while still being able to make it revert to accepting commands by performing a software or hardware reset.

Mode	**Purpose**
0	Demonstration mode- Khepera executes a Braitenberg vehicle algorithm
1	Khepera listens for commands on the serial link, expecting 9600 Baud
2	As 1, but Khepera expects 19200 Baud
3	As 1, but Khepera expects 38400 Baud
4	User application mode- starts an application stored in an EPROM if present
5	Khepera expects a program to be downloaded to it over the serial link at 9600 Baud (in "S format")
6	As 5, but Khepera expects 38400 Baud
7	Test mode. Performs a number of tests, and reports their results on the serial link at 9600 Baud

Appendix B

B1. Edge following behaviour

State	Next State	Condition
Start (start edge-following gracefully)	Capture	No edge close
	Face	Edge close
Face (orient robot with edge)	Face	Robot not oriented correctly
	Waddle	Robot oriented to edge
Waddle (follow a smooth edge)	Waddle	Edge straight or turning smoothly
	Turn	Edge turned sharply concave
	Capture	Edge turned sharply convex
Turn (turn a concave corner)	Turn	Way forward is not clear
	Waddle	Way forward is clear
Capture (recover edge if lost)	Capture	Still seeking edge
	Waddle	Edge detected
	Compensate	Edge lost
Compensate (undo capture gesture)	Compensate	Still compensating
	Waddle	Edge detected
	Stroll	Compensation complete
Stroll (move forward to nearest edge)	Stroll	No edge found
	Waddle	Edge found

B2. Location seeking behaviour

State	Next State	Condition
Start (start seeking gracefully)	CaptureEdge	Seek used edge-following recently
	Walk	History does not suggest edge-following as appropriate
CaptureEdge (evaluate edge as obstacle)	FollowEdge	Edge suitable for following
	Walk	Edge not an obstruction
Walk (move towards target)	Walk	No obstruction, roughly on target
	FollowEdge	Obstruction
	SteadyTurn	Facing away from target
FollowEdge(start edge-follow)	MonitorEdge	(Transition always occurs)
SteadyTurn	SteadyTurn	Facing away from target
	Walk	Approximately facing the target
MonitorEdge	MonitorEdge	Edge worth following
	Walk	Edge following inappropriate
	SteadyTurn	Turn required to resume walking

B3. Prowling behaviour

State	Next State	Condition
Start (start prowling gracefully)	Grab	(Transition always occurs)
Grab (start circling gracefully)	Grab	No edge close
	Circle	Edge close
Circle (patrol a boundary)	Circle	Complete circle not yet made
	CircleSync	In unfamiliar territory, and confused
	ExploreSeek	In familiar, well explored territory
	Leap	Complete circle made of boundary
CircleSync (recover from confusion)	CircleSync	Still confused
	Circle	No longer confused
ExploreSeek (explore a boundary)	ExploreSeek	In familiar area, or not confused
	CircleSync	In unfamiliar territory, and confused
	ExploreFly	Exploration opportunity spotted
	Leap	Complete circle made of boundary
ExploreFly (explore away from a boundary)	ExploreFly	Nothing near, and not confused
	Leap	Confused
	Circle	Boundary found
Leap (move to another boundary)	Leap	Not yet at another boundary
	Circle	At another boundary
	LeapSync	Confused
LeapSync (recover from confusion)	LeapSync	Still confused
	Leap	No longer confused
	LeapReturn	Away from familiar territory
LeapReturn (recover from loss of familiarity)	LeapReturn	Still confused and in unfamiliar area
	Leap	Confusion not excessive
	LeapSync	Territory more familiar

B4. Proxy behaviour

Command	Code	Description
Prowl	t	Puts robot in prowling behaviour, a composite behaviour built from all the other autonomous behaviours of the robot. See Prowl Algorithm.
Patrol	x	Makes the robot patrol the area it has explored or passed through- it will repeatedly move through every reachable part of that area again and again. See Patrol Algorithm.
Edge	g	Makes the robot follow the edge of any nearby obstacle (or it there is none, it'll apply a search gesture to find one).
SetMark	=	Sets the current location of the robot as a landmark for returning to later.
SeekMark	?	Seeks a landmark set earlier
Explore Renew	e n	Turns on exploration mode Turns on dynamic environment mapping
SendMap	d	Transmit the internal best-guess map to the PC
Think	@	Makes the robot plan heavily for an interval (which makes its normal behaviour sluggish).
Conquer	!	Requests that robot take over the world (service not yet implemented)
Manual	m	Orders the robot to follow user motion requests
Halt	[space]	Puts the robot into an idle behaviour

B5. Detailed Map Maintenance

This appendix presents an exhaustive description of how the robot can implement the "neighbourhood" system described in Section 4.3.1 (page 88), and how it can keep its markers up to date so that as a group they reflect its best knowledge of the environment. Every cycle, one marker from each of the neighbourhoods is updated. Updating a marker requires the robot to do the following:-

- Sort the marker into the correct neighbourhood- Since the robot will have moved since the last time the marker was updated, it will have become closer or further away. Therefore the neighbourhood it is kept in may no longer be appropriate. This is checked, and the marker placed in another neighbourhood if necessary.
- Update the marker's goal seeking fields. This advances any search activity the robot might be working on in the background.
- If the marker is "immediate" or "near", update the marker's connectivity data in accordance with the ideas outlined in Section 4.5.4 (page 115). The marker's reach time should be spread into any markers that have been noted as reachable from it. If the marker is "near" it should be compared with the currently most "immediate" marker for reachability, and if they seem reachable from each other, update their links to reflect that relationship. This allows connectivity information to be reconstructed through normal running of the sorting process. Note that the reconstruction is completed as the robot moves *away* from a marker, not while it is at it.
- If the marker is the oldest one seen so far, take note of it.
- If the marker is in the immediate neighbourhood, and is on an edge, update the marker's pass count, since the marker is close enough to consider the robot as passing through it.
- If the marker is outside the local and immediate neighbourhoods, and has been tagged to be killed, or its pass count exceeds the maximum limit, destroy the marker since it is potentially inaccurate.

The robot needs to keep track of certain markers in the "immediate" or "near" neighbourhoods:-

- If the marker is tagged as a corner, then take note of it. The robot is always aware of the nearest recorded corner so that if its motion indicates that it is currently at a corner, it can

compare against the recorded corner. If it can be confident the current and recorded corner represent the same real-world object, then it can use that to make corrections to its position estimate.

- Calculate the marker's desirability as a target for local motion of the robot, in terms of how much the robot would need to turn to move towards it, and when the robot last passed it. If it is the best seen so far, note that.
- Check how close to the search goal the marker is. If it is the best seen so far, note that.

Appendix C

This appendix documents the services supplied by the interfaces of the units of the decomposition described in Chapter 7. Since the Lateral Runtime Unit embodies behaviours and connections, the services listed for it are those available for Behaviour and Connection constructs in Zac Script.

C1. Common Robot Unit

C1.1. Interaction Interface

The common robot interaction interface can supply the following information:-

- Odometry data
  - ⇒ Best guess at current position
  - ⇒ Best guess at direction robot is facing
  - ⇒ Best guess at total distance robot has travelled
- Serial communications
  - ⇒ Flag indicating if a command has been received across the serial line
  - ⇒ The last command received, if any
- Collision flag, set if proximity readings indicate a possible impending or actual collision

The interface can be used to do the following:-

- Control the state of the robot's LEDs
- Control the serial interface
  - ⇒ Issue a response across the link (this can also be done by writing to standard output)
  - ⇒ Mark a command that was received across the link as read
  - ⇒ Choose whether the kernel should wait for commands to be acknowledged as read before accepting other commands. By default commands are stored for one scan cycle so the control system has the opportunity to view them, and then discarded.
- Factor an offset into the robot's best guess at its position or direction, to allow input from a more informed component of the control system

Notice that sensor data (other than a gross collision detection flag) and motor control functionality is absent. Interaction interfaces to these are provided at a higher level in the Logical Robot Unit.

xTrack	Best guess at current position, x coordinate
yTrack	Best guess at current position, y coordinate
angleTrack	Best guess at direction robot is facing (degrees)
angleRaw	Best guess at direction robot is facing (radians)
displacementTrack	Best guess at distance robot has travelled
IsCollision	Checks for any proximity reading indicating a possible collision
SetDisplay	Sets the state of one of the robot's LEDs
GetInput IsInput	Reads the last command issued to the robot across the serial link, if any Checks if a command has been issued to the robot across the serial link
AcceptInput	Marks the last command read as processed
MaintainInput	Chooses whether commands should be discarded automatically, or if an acknowledgement should be waited for
SendOutput	Sends a response across the serial link
AdjustPosition	Modifies the best guess at the robot's current coordinates and orientation

C1.2. System Hooks

This interface provides the following information:-

- The time since the kernel was last reset in milliseconds
- Readings from the ambient light and proximity sensors

It provides the following control functionality:-

- Fixing setpoints for the motor speeds
- Disabling and re-enabling task switching, to allow operations that must be executed atomically to be performed.

ServerLock	Disables task switching so an operation can be guaranteed atomic
ServerUnlock	Re-enables task switching after an atomic operation
ServerGetTick	Gets the time in milliseconds
ServerSetMotor	Fixes the setpoints for the motor speeds
ServerGetProx	Reads from the proximity sensors
ServerGetLight	Reads from the light sensors

C2. Logical Robot Unit

C2.1. Interaction Interface

This interface provides the following information:-

- The logical sensors- left, right, forward, reverse, and closest logical proximity sensors
- The unprocessed sensor readings can also be read directly- left reverse, direct left, left diagonal, left forward, right forward, right diagonal, right direct and right reverse.
- A frustration sensor from the logical motor driver warns when the robot is unable to make any progress at all in the direction its motors have been commanded to drive.

It provides the following control functionality:-

- The basic speeds for the logical motors can be set.
- The modifying speeds for the logical motors can also be set. The two sets of speeds are used as described earlier by the logical motor driver to generate actual speeds that will diverge from the logical speeds in the presence of obstacles.

leftSense	This is a sensor guaranteed to be low if the robot can move left without immediately bumping into something, and high otherwise.
rightSense	This is a sensor guaranteed to be low if the robot can move right without immediately bumping into something, and high otherwise.
forwardSense	This is a sensor guaranteed to be low if the robot can move forward without immediately bumping into something, and high otherwise.
reverseSense	This is a sensor guaranteed to be low if the robot could move backwards without immediately bumping into something, and high otherwise.
closestSense	This combines the left, right and forward sensors to give the closest obstacle to the robot under normal operation (turning for reverse).
GetSense	Reads from a specific proximity sensor
IsFrustrated	Checks if the robot is unable to proceed in the direction its motors have been commanded to drive
SetMotor	Sets the basic motor speeds to request (actual speeds may be different if there is an obstacle)
SetNudge	Sets the modifying motor speeds to request, for superimposing a turn on top of the basic motor speeds

C2.2. Update Hooks

This interface provides the following control functionality:-

- A manager for the logical motors that combines the requested basic motor speeds and modifying ("nudge") speeds, and adjusts the physical motor setpoints to reflect this as closely as possible, intelligently dealing with obstacle conditions
- A manager for the logical sensors that combines the appropriate physical sensor readings to maintain the semantics of the logical sensors (left, right, forward, reverse, closest)

UpdateMotors	Combines the requested basic motor speeds and nudge speeds, and adjusts the motor setpoints to reflect this as closely as possible, intelligently dealing with obstacle conditions

UpdateSensors	Maintains the values of the logical sensors (leftSense, rightSense, forwardSense, reverseSense, closestSense)

C3. Lateral Runtime Unit

C3.1. Interaction Interface, Static Components

The interaction interface to Lateral is divided into three components for convenience- static, behaviour, and connection components.

Static components of the Lateral runtime are accessible from anywhere within the control system. Services include:-

- A trigger activated on initialisation of the lateral system.
- A pair of triggers that indicate when the system should enter shutdown phase and when it should stop completely.
- A timer triggered every second for driving slow periodic events.

AutoStart	Trigger to start behaviours on initialisation of the lateral system
SystemActive	Flag indicating whether system should continue
SystemRetain	Flag indicating whether system can be halted. Used in combination with SystemActive to allow a shutdown phase.
TickerSecond	The number of seconds the system has been active. Signal can be used to trigger a behaviour every second.

C3.2. Interaction Interface, Behaviour Components

Within behaviours, the following status information is available:-

- Whether it is active, enabled, sleeping.
- The current priority level of the behaviour, and whether that priority has been fixed or is being selected dynamically by the Lateral system.
- Current state of the behaviour's state machine
- Timing information in milliseconds, seconds, minutes, or other units for :-
  - ⇒ The time since the system was initialised (also available globally)
  - ⇒ The time since the current behaviour started

- ⇒ The time since the current state was entered from a different state.
- ⇒ The accumulated time the current behaviour has been disabled over its lifetime.
- ⇒ The time since the current behaviour was last disabled.

- ☐ The name of the behaviour (useful for diagnostics)

Control functionality within behaviours:-

- ☐ Priority level can be set, or marked to be controlled by Lateral's default priority selection algorithm
- ☐ The behaviour's state machine can be manipulated by
  - ⇒ Making it transition to an arbitrary state.
  - ⇒ Resetting it to its starting state.
  - ⇒ Zeroing the timer for the current state.

IsActive	Checks if behaviour is active or inactive
IsEnabled	Checks if behaviour is enabled or disabled
IsRunning	Checks if behaviour is running (active and enabled)
GetRelPriority	Returns the priority of the behaviour
SetRelPriority	Sets the priority of the behaviour
UnSetPriority	Leaves the priority free to be chosen by Lateral
GetName	Returns the name of the behaviour
GetLine	Returns the current state of the behaviour
GoStart	Makes the behaviour transition to its first state
ResetStateTimer	Starts the timer for the current state from zero
GetSleeping	Checks if the behaviour has been put to sleep because no other behaviour is using it
IsSolid	Checks if the behaviour is been run at a user-selected priority (rather than allowing Lateral to choose its priority dynamically)

TT	Time in milliseconds. TT(system) is the time since the system was initialised (also available globally) TT(process) is the time since the current behaviour started. TT(state) is the time since the current state was entered from a different state. TT(delay) is the total time the current behaviour has been disabled. TT(quantum) is the total time the current behaviour has been enabled.
TS	Time in seconds. As for TT. For measuring the time in seconds, alternate functions such as StateTimeSec, SystemTimeSec, etc. are also available.
TM	Time in minutes. As for TT.
TF	Time in user-defined units (fractions). As for TT.
NEXT	Transition to a new state

C3.3. Interaction Interface, Connection Components

Information:-

- Information on source
  - ⇒ A flag indicating whether the connection has an active source
  - ⇒ A flag indicating whether the connection is a direct output from a behaviour
  - ⇒ The last source from which the connection received a message
- Information on target
  - ⇒ A flag indicating whether the connection has control of a target to pass on messages to.
  - ⇒ A flag indicating whether the current contents of the connection have been passed on.
- Status information
  - ⇒ Priority level of the connection (for competing with other connections).
  - ⇒ Priority factor of the connection (determining the fraction of its priority the source behaviour of the connection chain is willing to supply a target behaviour).
  - ⇒ A flag indicating whether the connection is active or inactive.

- Content information
  - ⇒ The content of the connection.
  - ⇒ A flag indicating if the contents of the connection have been altered.

The following operations are allowed on Connection objects:-

- Set the priority of the connection.
- Set the priority factor of the connection.
- Activate or deactivate the connection.
- Set the content of the connection.

IsOutput	Checks if the connection is a direct output of a behaviour
SetActive	Controls whether the connection is active or inactive
GetActive	Checks whether the connection is active or inactive
SetRelPriority	Sets the priority level of the connection for competing with others
GetRelPriority	Gets the priority level of the connection when competing
SetOutPriority	Sets the priority fraction the connection supplies to a behaviour it supplies if it gains control of that behaviour
GetOutPriority	Gets the priority fraction the connection supplies to a behaviour
IsFulfilled	Checks if the contents of a connection have been passed on
IsInPower	Checks if the contents of a connection are being passed on
GetLastSource	Gets the last source the connection read from
IsControlled	Checks if the connection has a source to read from
Set	Sets the contents of a connection
Get/Value	Gets the contents of a connection
Delta	Checks if the contents of a connection have been changed

C3.4. Execution Hooks

This interface provides no information services, just control functionality:-

- Execute one cycle of the lateral system.
- Hand over control entirely to the lateral system to run autonomously.
- Initialise and deinitialise the lateral system.

ZAC_Execute ZAC_Drive	Runs one cycle of the lateral system. Hands over control to the lateral system to run autonomously.
ZAC_Initialise	Initialises the lateral system- calls all start up functions in the behaviours.
ZAC_Deinitialise	Shuts down the lateral system. The system can be re-initialised if desired.

C4. User Control Unit

C4.1. Lateral Object Hooks

Each behaviour can implement one or all of the following:-

- A global start-up hook, called on initialisation of the Lateral system.
- A local start-up hook, called when a behaviour becomes active.
- A hook called whenever a behaviour is put to sleep.
- A hook implementing a state machine for the behaviour.

Generating this interface is simplified by use of the Zac Translator (Chapter 6). Note that only a few hooks other than the state machine hook are required, since most conditions except the ones enumerated can be caught in the state machine's control section.

STARTUP	Code executed for global initialisation of the behaviour
LOCAL_STARTUP	Code executed for initialisation of the behaviour whenever it becomes active
SLEEP	Code executed whenever the behaviour is made to sleep

ZAC_Run	State machine is translated into a function of this name (performed automatically)

References

[1] Proceedings of First International Symposium on Robotics Research, MIT, 1983, MIT Press, 1984

[2] Brady, M., Hu, H. Software and Hardware Architecture of a Mobile Robot for Manufacturing, Proceedings of the AAAI Spring Symposium "Lessons Learned from Implemented Software Architectures for Physical Agents", AAAI Technical Report SS-95-02, Menlo Park: AAAI Press, pp. 35-43, 1995.

[3] Brooks, R.A. 1991. Intelligence without Reason. Proceedings of the 1991 International Joint Conference on Artificial Intelligence: 569-595

[4] Albus, J.S.: A Reference Model Architecture for Intelligent Systems Design, An Introduction to Intelligent and Autonomous Control, pp. 57-64, Kluwer Academic Publishers, 1992.

[5] Miller, D.P. 1995. Experience looking into niches, Proceedings of the AAAI Spring Symposium "Lessons Learned from Implemented Software Architectures for Physical Agents", AAAI Technical Report SS-95-02, Menlo Park: AAAI Press, pp. 141-145, 1995.

[6] Gat, E.: Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots, Proceedings of the 10th National Conference on Artificial Intelligence, pp. 809-815, MIT Press, July 1992.

[7] Brooks, R.A. 1986. A Robust Layered Control System for a Mobile Robot. IEEE Journal of Robotics and Automation. RA-2: 14-23

[8] Mataric M.J.: Behavior-Based Control: Examples from Navigation, Learning, and Group Behavior, JETAI Journal of Experimental and Theoretical Artificial Intelligence, special issue on Software Architectures for Physical Agents, Vol. 9, Nos. 2-3, Hexmoor, Horswill, spec.iss. on Software Architectures for Physical Agents, Vol. 9, Nos. 2-3, 1997.

[9] Brooks R.A.: A Robot That Walks: Emergent Behaviors from a Carefully Evolved Network, AI-Laboratory, Massachusetts Institute of Technology, Cambridge, MA, AI-Memo 1091, 1989.

[10] Brooks, R.A.: Elephants Don't Play Chess. Robotics and Autonomous Systems 6 3-15, 1990

[11] Brooks R.A.: Achieving Artificial Intelligence through Building Robots, AI-Laboratory, Massachusetts Institute of Technology, Cambridge, MA, AI-Memo 899, 1986.

[12] J. Bryson, A. Smaill, and G. Wiggins. The reactive accompanist: Applying Subsumption architecture to software design. Research Paper 606, Dept. of Artificial Intelligence, Edinburgh, 1992.

[13] Sahota M.K.: Reactive Deliberation: An Architecture for Real-Time Intelligent Control in Dynamic Environments, in Proceedings of the Twelfth National Conference on Artificial Intelligence, AAAI Press/MIT Press, Cambridge, MA, pp.1303-1311, 1994.

[14] Hexmoor, H. Smarts are in the architecture!, Proceedings of the AAAI Spring Symposium "Lessons Learned from Implemented Software Architectures for Physical Agents", AAAI Technical Report SS-95-02, Menlo Park: AAAI Press, pp. 116-122, 1995.

[15] Moorman, K., Ram, A.: A Case-Based Approach to Reactive Control for Autonomous Robots, AAAI Fall Symposium on AI for Real-World Autonomous Robots, Cambridge, MA, October 1992

[16] Maes P.: Situated Agents Can Have Goals, Designing Autonomous Agents, pp. 49-70, The MIT Press: Cambridge, MA, 1990.

[17] Firby, R. J.: Task Networks for Controlling Continuous Processes, Proceedings of the Second International Conference on AI Planning Systems, Chicago IL, June 1994.

[18] Gat, E. ALFA: A Language for Programming Reactive Robot Control Systems, Proceedings of the 1991 IEEE International Conference on Robotics and Automation, Sacramento, California, April 1991.

[19] Agre P.E., Chapman D.: What are plans for?, Maes P. (ed.) New Architectures for Autonomous Agents: Task-level Decomposition and Emergent Functionality. MIT Press, Cambridge, Massachusetts, 1990.

[20] Rosca, J. P., Riopka, T.: A Constraint-Based Control Architecture for Acting and Reasoning in Autonomous Robots, Proceedings of the AAAI Spring Symposium "Lessons Learned from Implemented Software Architectures for Physical Agents", AAAI Technical Report SS-95-02, Menlo Park: AAAI Press, pp. 159-166, 1995.

[21] Brooks R.A.: The Behavior Language; Users' Guide, Massachusetts Institute of Technology, AI Laboratory, A.I. Memo 1227, 1990.

[22] Arkin R.C.: Integrating Behavioural, Perceptual, and World Knowledge in Reactive Navigation, Designing Autonomous Agents, pp. 105-122, The MIT Press: Cambridge, MA, 1990.

[23] Payton, D. W.: Internalized plans : a representation for action resources, Journal of robotics and autonomous systems(1&2), June 1990, Vol. 6, pp. 89-104, 1990.

[24] K-Team SA, Khepera® User Manual

[25] Franzi, E. Khepera BIOS 4.00 Reference Manual

[26] Flanagan, C., Toal, D., Strunz, B. 1995. Subsumption Control of a Mobile Robot. Moscardini, A. O., Smith, P. (eds.), Proceedings Polymodel 16, Applications of Artificial Intelligence, Sunderland, UK, pp. 150-158, 1995

[27] Ferrell, C. 1993. Robust agent control of an autonomous robot with many sensors and actuators. MIT AI Laboratory Technical Report 1443

[28] Mondada, F., Franzi, E., and Ienne, P. 1993. Mobile robot miniaturisation: a tool for investigation in control algorithms. Proceedings of the 3rd International Symposium on experimental robotics, Kyoto, Japan, October 28-30 1993, Springer Verlag, London, 1994: 501-513

[29] Bryson, J.: The Reactive Accompanist, Masters thesis, University of Edinburgh Department of AI, 1995

[30] Brooks R.A., Stein L.A.: Building Brains for Bodies, Autonomous Robots, 1, pp. 7-25, 1994.

[31] Martin, F. & co.: Interactive C User's Guide, Newton Research Labs, http://www.newtonlabs.com/ic/ic_1.html

[32] Gat, E.: On the role of stored internal state in the control of autonomous mobile robots, AI Magazine, 64-73, Spring 1993, 14(1), 1993.

[33] Arkin, R. C. Just What is a Robot Architecture Anyway? Turing Equivalency versus Organizing Principles, Proceedings of the AAAI Spring Symposium "Lessons Learned from Implemented Software Architectures for Physical Agents", AAAI Technical Report SS-95-02, Menlo Park: AAAI Press, pp. 7-10, 1995.

[34] Michel, O. Khepera Simulator Version 2.0 User Manual

[35] Colombetti, M., Dorigo, M. Training Agents to Perform Sequential Behavior. Adaptive Behavior, MIT Press, 2 (3), 1994

[36] Dorigo M.: ALECSYS and the AutonoMouse: Learning to Control a Real Robot by Distributed Classifier Systems, Machine Learning, 19(3), 1995.

[37] Ram, A., Santamaría, J. C. Continuous Case-Based Reasoning, AAAI Workshop on Case-Based Reasoning, Washington DC, July 1993. 1993.

[38] Ram, A., Arkin, R., Boone, G., Pearce, M.: Using Genetic Algorithms to Learn Reactive Control Parameters for Autonomous Robotic Navigation, Adaptive Behavior, 2(3):277-305, 1994

[39] Kaelbing L.P., Rosenschein S.J.: Action and Planning in Embedded Agents, Robotics and Autonomous Systems, Vol. 6, No. 1; also in Designing Autonomous Agents, The MIT Press, 1991

[40] Malcolm C., Smithers T.: Symbol Grounding via a Hybrid Architecture in an Autonomous Assembly System, Robotics and Autonomous systems(1&2), June 1990, Vol. 6, pp. 123-144, 1990.

[41] Anderson, T. L., Donath M. Animal Behavior as a Paradigm for Developing Robot Autonomy, Designing Autonomous Agents, pp. 145-168, The MIT Press: Cambridge, MA, 1990.

[42] Beer, R. D., Chiel, H. J., Sterling, L. S. A Biological Perspective on Autonomous Agent Design, Designing Autonomous Agents, pp. 169-185, The MIT Press: Cambridge, MA, 1990.

[43] Minsky M.: The Society of Mind, Simon & Schuster, New York, 1985.

[44] Franklin, S., Graesser, A. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996

[45] Song, H., Franklin, S., Negatu, A. SUMPY: A Fuzzy Software Agent, Proceedings of the 5th ISCA International Conference on Intelligent Systems, Reno, Nevada, June 1996.

[46] McLurkin, J. Antware, MIT Artificial Intelligence Lab web-pages, http://www.ai.mit.edu/projects/ants/antware.html (no formal reference available)

[47] Selfridge, O.G. "Pandemonium: A Paradigm for Learning," in D.V. Blake and A.M. Uttley (eds.), Proceedings of the Symposium on Mechanisation of Thought Processes (National Physical Laboratory, Teddington, England; London: H.M. Stationary Office, 1959), pp. 237-50.

[48] Kortenkamp, D., Weymouth, T.: Topological mapping for mobile robots using a combination of sonar and vision sensing, Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), July, 1994.