A fuzzy controller prototyping shell, and its application in a sonar-sensing mobile robot control system

Pushkar Piggott
University of Wollongong
A Fuzzy Controller Prototyping Shell, and its Application in a Sonar-Sensing Mobile Robot Control System.

A thesis submitted in fulfilment of the requirements for the award of the degree

Doctor of Philosophy

from

University of Wollongong

by

Pushkar Piggott

Bachelor of Informatics with Honours
(First Class)

Intelligent Robotics Research Laboratory

1996
Acknowledgements

A thesis tends to prey on time: free time, dinner time, bed time or any time. This thesis was no exception, and so my first acknowledgement is to my family, of the times they have had to carry on without me.

The main purpose of these acknowledgements, though, is to be a vehicle for my gratitude to my supervisor; to be an Acknowledgements Gratitude Vehicle, and I believe this AGV's navigation system is up to its task. Dr. Phillip McKerrow provided invaluable help and guidance.

I would also like to thank Dr. Alex Zelinsky, Prof. Greg Doherty, Les Ohlbach and Michael Millway; for pertinent comments and much else, sane council, support above and beyond, and the revelation that the duck pond holds the ultimate solution, respectively.
Published Papers from this Thesis


- Pushkar K. Piggott and Phillip J. McKerrow. Controller Development with a Rapid Prototyping Shell. Accepted for publication with alterations by *Australian Journal of Intelligent Information Systems*. 
Abstract

This thesis presents the results of the investigation, implementation and testing of a tool to support rapid-prototyping development of high performance fuzzy controllers for mobile robotics. Fuzzy inference is a strong candidate for the development of robot control systems because it represents the continuous real-time inputs and outputs necessary for control as convenient textual rules. This thesis shows how the inflexibility of traditional fuzzy inference development methods can be overcome by using a free-form rule-base similar to an expert system. A rapid-prototyping approach can then be taken to controller development.

The investigation identifies four issues that distinguish controller development from traditional applications of expert system shells. The tool is unique in incorporating methods that deal with all four issues. It has a novel architecture as a control relationship prototyping environment, which is linkable to a control system that defines the variables and the control loop.

The shell syntax has some novel features. It allows control variables to be defined as vectors that can considerably reduce the complexity of the rule set, and in some cases simplify the inference computation. It also permits a group of fuzzy subsets to be defined collectively over a subrange of the variable, and this can reduce errors.

The tool is first tested in a control simulation, first to duplicate an existing truck backing-up controller, and then to develop a new controller. The power of the incremental rapid-prototyping approach is demonstrated through the development of a simple but effective rule set.

The tool is then tested for its suitability as a research tool for mobile robotics. I propose a complete robot navigation system, and then focus on the implemen-
tation of the lowest level of control using the shell. The shell is used to make
the initial wall identification, and to track a wall. Two instances of the shell
are integrated into the control system, and perform well within the tight time
constraints. The incremental rapid-prototyping approach again produces very
simple rule sets.
## Contents

1 Introduction ................................................................. 3
   1.1 Context ........................................................................ 3
   1.2 Motivation .................................................................... 3
   1.3 Objectives .................................................................... 5
   1.4 Hardware and Development Platforms ......................... 5
   1.5 Thesis Overview .......................................................... 6

2 Controller Design .......................................................... 11
   2.1 Mathematical Modelling & Expert Systems .................... 12
   2.2 Knowledge Representation .......................................... 13
   2.3 Control Surface Continuity ......................................... 15
   2.4 Fuzzy Inference .......................................................... 18
   2.5 Shell Script vs. Fuzzy Associative Matrix ..................... 24
   2.6 Time Constraints ......................................................... 26
   2.7 Symbol Grounding ....................................................... 28
   2.8 Data Structure ............................................................ 30
   2.9 Rapid Prototyping ....................................................... 31
   2.10 Fuzzy Qualitative Modelling ...................................... 32
   2.11 Summary ................................................................. 33

3 The Shell: Architecture & Interfaces ................................. 35
   3.1 The Structure of a Controller Development Tool ............ 36
   3.2 The Shell's Architecture & Program Interface ................ 38
   3.3 The Script Syntax ........................................................ 42
      3.3.1 The Variable Definition ......................................... 42
CONTENTS

3.3.2 Single Subset Specification .................................................. 44
3.3.3 The Representation of Output Subsets .............................. 45
3.3.4 Specification of a Subset Cover ......................................... 47
3.3.5 The Rule Definition .............................................................. 49
3.3.6 Vector Variables .................................................................... 50

3.4 Summary ....................................................................................... 52

4 The Shell: Implementation .............................................................. 55

4.1 Fido: The Top Level Object ........................................................... 55
4.2 Script: The Lexical Analyzer ........................................................... 58
4.3 Rules: The Fuzzy Rules ................................................................. 61
  4.3.1 Common Sub-Expression Elimination................................. 63
  4.3.2 Encoding Conditional Expressions ....................................... 65
4.4 Vars: The Fuzzy Variables.............................................................. 67
  4.4.1 Individual Subset Shapes ..................................................... 67
  4.4.2 Subset Covers....................................................................... 68
  4.4.3 Fuzzification.......................................................................... 73
4.5 Summary ........................................................................................ 75

5 A Simulation Trial ............................................................................. 77

5.1 Kosko's Solution ............................................................................. 78
5.2 A Prototyped Solution.................................................................... 81
5.3 A Comparison of Performance ........................................................ 85
5.4 Summary ........................................................................................ 91

6 Shell Supported Sonar-Based Wall Following .................................. 93

6.1 A Mobile Robot Navigation System Outline ............................... 94
6.2 The Use of Sonar for Mobile Robotics ........................................ 96
  6.2.1 The Sonar Beam ................................................................. 97
  6.2.2 The Echo ............................................................................. 98
  6.2.3 Detection ............................................................................. 99
  6.2.4 Angular Resolution ............................................................ 99
  6.2.5 Sensing Time ...................................................................... 99
## List of Figures

2.1 Mathematical and boolean rule-based models of a sigmoid curve. 15

2.2 Mathematical, boolean and fuzzy rule-based models of a sigmoid curve. 16

2.3 Fuzzy subsets representing 3, [3…4] and about 3. 19

2.4 The label subsets used for *Stimulus*. 19

2.5 Fuzzy inference using correlation-minimum correlation-product. 19

2.6 Centre of gravity defuzzification. 20

2.7 A truck represented by a main axis and a steering axis. 22

2.8 Control variables for a truck backing up to a dock. 22

2.9 A method of fuzzy inference applied to a truck backing up to a dock. 23

2.10 Matrix representation of a rule set. 24

2.11 Textual representation of a rule set. 25

2.12 A comparison of rule-based symbolic and control systems. 28

2.13 The parts of a fuzzy control system and stages of system identification. 32

3.1 The program interface. 39

3.2 An example of the use of the program interface. 40

3.3 The initial script of control variable stubs generated by the shell. 40

3.4 The script *steer.scr* developed to control a truck backing up. 41

3.5 The script syntax for variables. 42

3.6 Trapezoidal shapes and how to specify them. 44

3.7 *Steer* defined with singleton subsets of unit weight. 46

3.8 *Bearing* defined using single subsets and a semi-regular cover. 47
XVI

LIST OF FIGURES

3.9 Semi-regular covers. ................................................. 48
3.10 Fully regular covers. ................................................. 48
3.11 The script syntax for rules. ........................................ 50

4.1 The structure of the tool’s higher level objects. .............. 56
4.2 Syntax for symbolic integer constants and debugging key words. 56
4.3 Fido’s top level of parsing. ........................................... 58
4.4 Script’s lexical analysis, part 1. ................................... 59
4.5 Script’s lexical analysis, part 2. ................................... 60
4.6 Recursive parsing of rule conditional expressions: the first term. 62
4.7 Recursive parsing of rule conditional expressions: the second term. 63
4.8 The extraction of common complex expression trees from three rules. ................................................. 64
4.9 The evaluation order of common complex expression trees. .... 64
4.10 Abstract stack machine operations. ............................... 65
4.11 Stack register usage minimization. ............................... 67
4.12 Parsing an individual subset shape. ............................... 68
4.13 Parsing a subset cover. ............................................. 69
4.14 The parts of fully regular cover subsets. ......................... 70
4.15 Parsing a semi-regular cover ShapeList. ......................... 71
4.16 Fuzzification of a value with respect to a fuzzy subset .......... 74
4.17 Defuzzification of an output variable. ............................ 75

5.1 A truck represented by a main axis and a steering axis. ........ 78
5.2 Kosko’s control variables. ............................................ 78
5.3 Kosko’s matrix rule set. ............................................. 79
5.4 The first two rows of Kosko’s matrix as script rules. .......... 79
5.5 The truck’s path using a boolean rule set. ....................... 80
5.6 The truck’s path using Kosko’s rule set. ......................... 80
5.7 Control variables for a truck backing up to a dock. ........... 82
5.8 The subsets for Turn. ............................................... 82
5.9 A simple first-cut rule set. ........................................ 83
LIST OF FIGURES

5.10 The truck’s path using the first-cut rule set. 83
5.11 The subsets for Bearing and Turn in the second cut. 84
5.12 A second cut rule set. 85
5.13 The truck’s path using the second-cut rule set. 85
5.14 The subsets for Bearing in the final cut. 86
5.15 The final rule-set for truck backing up. 86
5.16 The final rule-set as a matrix. 87
5.17 The truck’s path using the final rule set. 87
5.18 A comparison of paths. 87
5.19 The path length differences. 88
5.20 A comparison of regions of path length difference. 88
5.21 A comparison of final x offset. 89
5.22 A comparison of regions of final x offset error. 89
5.23 A comparison of final orientation in degrees. 90
5.24 A comparison of regions of final orientation error. 90

6.1 The shape of the beam spread of a Polaroid sensor. 97
6.2 Reflection of the main lobe from a plane surface. 98
6.3 The sensors divide the circumference of the robot into 22.5° sectors. 102
6.4 The wall identification script. 103
6.5 The robot’s track as it identifies a wall sector. 104
6.6 Wall identification data values. 105
6.7 The wall tracking script. 106
6.8 The range bands and a path to approach the wall. 107
6.9 The turn angle is the difference between the current and desired wall sectors. 108
6.10 The wall approach algorithm. 110
6.11 The robot’s actual approach path. 111
6.12 The wall following algorithm. 114
6.13 The robot’s path round the play pen. 115
6.14 The robot’s reverse path round the play pen. 116
6.15 The robot’s path around a wall end. 117
LIST OF FIGURES
Glossary

Atomic variable: An atomic variable is a variable that takes a single scalar value (see also vector variable).

Cover: A cover is a series of overlapping fuzzy subsets that cover a subrange of a fuzzy variable (see also regular cover and semi-regular cover).

Fuzzy subset: see fuzzy variable.

Fuzzy variable: (Section 2.4) A fuzzy variable is represented by a graph mapping the range of the variable on the x axis to fuzzy membership (μ) in the y axis. A fuzzy subset or label has a convex shape defined on the graph. The graph maps between a value in the range of the variable and a corresponding membership value for each subset. The membership value of a subset is the y coordinate of the point on its outline indexed by the x value. It may be zero.

Grounding code: (Section 2.7) Grounding code grounds a variable represented by a script symbol to input or output control signals that represent properties or actions in the world (see also symbol grounding).

Intermediate variable: (Section 2.6) An intermediate variable is a consequent of one rule and a condition of another. It represents state in the otherwise stateless rule set by contributing to an inference a value obtained from the previous inference.

Label: see fuzzy variable.

Regular cover: (Section 3.3.4) A regular cover is a group of equal sized fuzzy
subsets defined collectively to provide a complete cover over a subrange of a variable.

**Semi-regular cover:** (Section 3.3.4) A *semi-regular cover* is a group of fuzzy subsets defined collectively to provide a complete cover over a subrange of a variable. The cover is semi-regular because the overlap between subsets is regular but the subsets themselves can be of differing size.

**Symbol grounding:** (Section 2.7) *Symbol grounding* is the process of grounding a symbolic variable to values that correspond to properties or actions in the real world.

**Variable subset pair:** (Section 3.3.6, Section 4.3) A *variable subset pair* is a textual pairing of a variable name with the name of one of its subsets to represent the value of that subset in the scope of that variable.

**Vector variable:** (Section 3.3.6) A *vector variable* is a variable that takes a vector of distinct scalar values. (see also *atomic variable*)
Chapter 1

Introduction

1.1 Context

This work was carried out at the University of Wollongong Intelligent Robotics Research Laboratory. Our research focusses on air-born sonar sensing, sensor fusion, and autonomous mobile robotics. In view of the recent success of fuzzy control, we decided to experiment with fuzzy controllers for mobile robot navigation. My preliminary research revealed that fuzzy inference is indeed an ideal candidate for a central role in robot navigation system development. It also revealed that available tools to support the development of fuzzy controllers are not well suited to the rapid-prototyping approach effective for developing such controllers. The work then focussed on the construction of an appropriate tool, and the testing of the tool in simulation and with a robot.

1.2 Motivation

Mobile robotics is currently a rapidly developing field where success depends on a close and appropriate mapping of a robot's sensor inputs to output actions. Brooks (e.g., [Bro91]) has successfully shown that, in its moment to moment operation, a robot cannot afford extensive reasoning about its environment but must act on the basis of its immediate inputs. Its responses are then fast and based on the actual current state of its environment.
Although these are a robot's primary requirements, the need for memory and reasoning remains if the robot is to avoid becoming deadlocked through competing behaviours [ZK95]. It is becoming increasingly clear (e.g., [Con92]) that a robot requires both a lower level architecture supporting close coupling of input and output for reactivity, and a higher level architecture that operates on a longer time-scale to coordinate its behaviour. The challenge is to provide an integrated architecture in which the lower level can present a coherent model of its operation and a clean interface to the upper level.

Fuzzy inference is a prime candidate for a major role in such an architecture. It has proved an excellent tool for coupling control outputs to inputs with a minimum of intermediate processing [Sel90]. It uses rules for this which, unlike boolean rules, give a smooth and continuous control curve. The rules use linguistic terms that make the control relationships clear to the developer and easy to maintain. Thus, it appears ideally suited to the task of implementing low level behaviours while presenting a clear model for manipulation by higher levels of control.

Fuzzy inference also promises to be effective at a higher level, as a way of arbitrating between behaviours [HP91]. Promising results are reported by Maeda, Tanabe and Yuta [MTY92], and Li [Li94]. The fuzzy development tool developed for this thesis is demonstrated here only for low level control, but it would be equally applicable to the development of such arbitration between behaviours.

Fuzzy logic replaces the formal development methods of mathematical modelling with a more experimental approach through the development of rules made up of linguistic terms [Sel90]. Rapid-prototyping is an ideal method for this kind of development [GB95]. A rapid-prototyping tool is highly interactive, and needs to be flexible and easy to use. The traditional matrix implementation for fuzzy rules is quite inflexible and restrictive, but these limitations can be overcome by using a free-form rule-base similar to an expert system. In other words, the tool should be a controller development shell.

On the other hand, performance is a critical issue at the behaviour level, and a controller development tool must meet tight performance constraints on its
products. Embedded microprocessors are one possible target implementation for the final controller. I therefore propose a two stage process in which the controller is first prototyped using the shell, and then compiled down to a form suitable for high performance applications if required. Only the first prototyping stage is addressed in this thesis.

1.3 Objectives

The motivations outlined in the previous section led to the following objectives for this project:

- to create a tool that would:
  - support fuzzy rule-based controller development,
  - support rapid prototyping of controllers,
  - integrate easily into robot control systems,
  - generate controllers that operate in real-time,
  - generate controllers suitable for implementation on embedded microprocessors;

- to demonstrate the tool initially in a simulated control application;

- to test the tool in an autonomous mobile robot control system that would:
  - be part of a credible proposed complete navigation system,
  - use fuzzy control for low-level (within behaviour) control of the robot.

1.4 Hardware and Development Platforms

The lab is equipped with a Transitions Research Labmate mobile platform [TRC, Eva90] with two independently driven wheels and two casters. It is 800 mm by 800 mm, and 280 mm high. The Labmate is fitted with a Transitions Research Proximity Sensing Subsystem, which uses a ring of Polaroid sonar sensors.
Henceforth the complete assembly will be referred to as the robot, and the sensing system as the sonar ring.

The lab's preferred development language changed from Modula-2 to C++ during the project, and an initial Modula-2 prototype was ported to C++. The lab is amply supplied with Apple Macintosh personal computers, but after initial work on that platform using both Modula-2 and C++ in the Macintosh Programmer's Workshop, and Semantec C++, I reverted to the more familiar and less encumbered Borland C++ development environment on an IBM compatible 386 PC. I developed complete object-oriented C++ support for communication with the robot base and sensors. This included interrupt-driven serial port handlers, as these are not part of the DOS operating system. Over 10,000 lines of PC C++ code, excluding comments, were developed and used in this project.

1.5 Thesis Overview

The next chapter, Chapter 2, outlines the issues in controller design and development. Traditionally, mathematical modelling has been used but it has problems both in development and in execution. Some processes that require control are not well defined, and appropriate models cannot be developed for them. Some are too complex for control functions to be computed in real-time. Rule-based controllers can be model-free, and thus applicable where a model cannot be derived. Linguistic rules are easier to understand and maintain than equations; they are also simpler to compute, and often more robust.

Controllers differ from traditional symbolic applications of rule-based systems, however. I have identified four significant differences, continuous control surface shape, real-time response, symbol grounding and data structure. Control surfaces are continuous, unlike the boolean domain. I explain the use of fuzzy inference to handle this in Section 2.3. A controller must respond in real-time, and I therefore limit inference to a single step at each time step. Limited multi-step inference is still possible, however, using intermediate variables (Section 2.6) to transfer a value from one step to the next. The symbols used in the rules must be
grounded to control signals and this strongly affects the architecture of the shell (Section 2.7). Finally, the control variables may have structure that can be exploited by the controller development shell (Section 2.8). One achievement of this project is the development of a rule-based controller development environment that handles all of these issues.

The techniques for model-free controller development can be roughly divided into knowledge acquisition, rapid prototyping and machine learning. Rapid prototyping, which is supported by the shell, is increasingly popular [GB95], and is the appropriate development method for validating and tuning the researcher's initial intuitions (Section 2.9). Sugeno & Yasukawa [SY93] characterize fuzzy controller development as qualitative modelling, and distinguish a sequential process in which the identification of control variables comes before the development of fuzzy relationships (Section 2.10). This justifies the shell's architecture, which pre-supposes control variables defined and grounded in a control program. The shell is integrated into this program to support the development of the fuzzy relationships between the variables.

Chapter 3 describes the architecture and interfaces of the shell. Section 3.1 compares the shell with existing fuzzy controller development systems, which fall into one of two categories: extended traditional expert systems, and controller development tools. The former are highly complex and slow in execution, using production systems for inference, and the latter have architectures unsuited to rapid-prototyping development. The shell developed in this project is unique in that it has a flexible expert system like rule authoring environment suitable for rapid prototyping, and the resulting controllers have high performance.

The shell has two interfaces, one for the control system programmer, and one for the fuzzy relationship script author. The program interface, described in Section 3.2, is the C++ class interface that mediates the integration of the shell into the control system. It is kept small and simple, and provides only for the declaration of control variables to the shell, the redirection of fuzzy variable definitions to new data locations, and the compilation and execution of the fuzzy relationships.
CHAPTER 1. INTRODUCTION

The script interface (Section 3.3) consists of a syntax for defining fuzzy relationships, which is used by the script author to define the fuzzy relations. It supports the representation of the input and output variables that have been declared from the control system, and of any intermediate variables. Each variable is followed by the definitions of its subsets. These can be specified individually, but the syntax also allows them to be defined as regular or semi-regular covers over a subrange (Section 3.3.4). By defining subsets as covers, the script author expresses the unity of a set of subsets, and reduces the potential for error inherent in the redundancy of a set of individual specifications.

The syntax allows expressions of arbitrary complexity in the conditional part of a rule (Section 3.3.5). These are made up of primitive terms, which consist of a variable with a subset, combined with conjunctions and negations. The conclusion of a rule is a list of variables with subset terms, with optional negations. The syntax that allows variables to be declared as vectors is described in Section 3.3.6. The rules make optimal use of the various possible combinations of atomic and vector variables.

Chapter 4 describes the internal structure and algorithms of the shell program. The top level object, Fido (Section 4.1), presents the program interface to the control program, handles the initialization of the lexical analyzer and the top level of parsing, and implements some debugging script commands. The lexical analyzer, Script (Section 4.2), allows the rest of the program to view the script as a stream of words. The Rules part, described in Section 4.3, compiles conditional parts of the rules to programs for an abstract stack machine. It also implements common sub-expression elimination to allow the script author to consider repeated complex combinations of input terms as higher-level variables. Vars (Section 4.4) compiles atomic and vector variables and their subsets. It converts subset covers into collections of normal subsets. It also strips out subsets and variables if they are not used so that they are not processed during fuzzification and defuzzification. The program developer can specify which numeric types are to be supported by setting conditional compilation switches and re-compiling the shell.
1.5. THESIS OVERVIEW

Chapter 5 discusses a trial of the shell to develop a controller for a simulated control problem. The shell demonstrates its flexibility by integrating with the simulator as easily as with a real control system. I use a simple truck backing-up problem from Kosko [Kos92b] that allows me to compare the shell with existing fuzzy methods by using it to implement a traditional matrix rule set in Section 5.1. I also demonstrate the complete controller development process by identifying an alternative pair of input variables in Section 5.2. The rule set is then developed incrementally by extending a very simple rule set identified as an example in chapter 2. This incremental approach using rapid prototyping results in a rule set that is significantly simpler than Kosko’s, demonstrating the power of the chosen approach to controller development.

Chapter 6 describes an extended example application of the shell in a sonar-ring based autonomous mobile robot control system. To constitute a realistic example, this control system had to be a substantial project in itself. Its successful achievement vindicates the use of the fuzzy controller development shell for robotics. Fuzzy inference proved simple and effective, and, with the help of the shell, very easy to use.

A part of a robot control system must be justified in terms of the feasibility of the omitted parts that it assumes. I therefore first give a rough outline in Section 6.1 of the proposed autonomous navigation system of which the implemented system would be a part. In Section 6.2 I give brief overviews of sonar sensing for robotics, and of sonar-based navigation research. I describe the way I use the sonar-ring to provide fast and accurate data for continuous control of the robot’s path.

The control system is built of a number of integrated mechanisms, which are described in the following sections. I present the fundamental model of the sonar ring as a single virtual rotatable sensor in Section 6.3. Section 6.4 describes the method of fuzzy wall identification used for the initial determination of which sensor faces a wall. I overcome the very limited angular resolution of the sonar sensor by using the range difference between successive returns as input to a fuzzy controller that allows the robot to track the wall (Section 6.5). The robot
approaches and aligns with the wall by tracking it with successive sensors around the ring (Section 6.6).

With these mechanisms the robot can perform the basic functions of identifying, approaching and tracking a wall (Section 6.7). By switching attention to a forward facing sensor, the robot can use its existing abilities to also negotiate concave corners. Convex corners require, in addition, the ability to turn and re-establish sonar contact with the corner after it has been passed, and to maintain an appropriate angle of turn until the next wall is found. Section 6.8 describes the use of the complete system to allow the robot to circumnavigate a room of quite complex shape and with imperfect surfaces. I finish up in Section 6.9 by contrasting the system with three from recent literature that have some similarity to it.
Chapter 2

Controller Design

This chapter sets the scene by discussing the methods and issues involved in the design and development of controllers. Section 2.1 outlines the problems of traditional controller development using mathematical modelling, and introduces the benefits of rule-based techniques as an alternative. Section 2.2 briefly reviews knowledge representation languages, which are used by rule-based systems. The subsequent four sections deal with the significant differences between controller development and the more traditional, symbolic, applications of rule-based techniques:

Control surface continuity. Section 2.3 notes that a control surface should be continuous but boolean rules yield discrete values. Section 2.4 introduces fuzzy inference as a solution to the problem. Fuzzy control has traditionally used a matrix representation for rule sets, but in Section 2.5 I advocate the more flexible expert system shell approach.

Time constraints. The time constraints for a controller are much tighter than for an expert system interacting with a user, and Section 2.6 clarifies the problem and proposes a solution.

Grounding variables to signals. Unlike an expert system shell, a controller development shell must ground its symbols to control signals, and Section 2.7 introduces the architecture developed for this thesis, which supports this.
Data structure. A final difference between a controller and an expert system is the level of useful structure in the control data. It is discussed in Section 2.8.

A major motivation for this work is the need to support rapid prototyping of controller designs, and Section 2.9 puts the case for rapid prototyping. The shell's basic architecture is justified by Sugeno & Yasukawa's analysis of fuzzy qualitative modelling, outlined in Section 2.10.

### 2.1 Mathematical Modelling & Expert Systems

Conventional controller design requires the derivation of a mathematical model of the system to be controlled. Generic analytical models that are adequate to characterize some processes are available, but most processes require a special purpose model tailored specifically for them, and some cannot be modelled [McK83]. A mathematical model must be based on a detailed knowledge of all the variables, and obtaining this is time consuming, or indeed impossible for complex systems. The model may be derived by fitting curves to data logged during the execution of the process, but this is also difficult. The linear regression and series approximation techniques used are complex and often lead to equations that are too complex to calculate in real-time. These equations are simplifiable only if an underlying structure can be identified but this requires considerable mathematical insight, and further data logging is then required to re-validate the simplified model.

A control surface based on equations is built from curves specified by functions. Sudden arbitrary changes of curvature are difficult to model and may lead to system instability. In practice, mathematical models are often inflexible, and have to be adapted on-line to deal with parameter variations not captured by the equations.

Systems that are not amenable to mathematical analysis are sometimes modelled with look-up tables. These large, multi-dimensional arrays of output values are looked up using inputs as index values to obtain an output at each time step.
Such tables can be derived directly from logged data, but they also are inflexible. They can be invalidated by minor changes in operating conditions or machine setup. Either representation, equations or tables, is difficult to understand or manipulate.

Rule-based systems can be model-free. The rule set may be inferred from a model, derived from rules-of-thumb, or developed through prototyping, but by not representing the model explicitly the rule set can be simpler to compute and more robust. Rules are also more readily comprehensible than equations, and may conform more closely to the way experts represent knowledge to themselves, simplifying both knowledge acquisition and knowledge-base maintenance.

Rule-based system design is simplified by the separation of the representation of knowledge from the mechanism of inference. A shell creates this separation by providing an inference engine and a translator for a knowledge representation language based on rules. It thus allows the system developer to ignore the implementation details, and to concentrate on the structure of the knowledge to be represented. This technique has become highly developed with expert systems and database query languages, and it can be used for controllers also. There are significant differences between controller design and the traditional symbol-based applications of rule-based systems, however. These differences create major problems and opportunities, and are discussed in the following sections.

2.2 Knowledge Representation

A rule-based system uses a knowledge base to store the rules in a form suitable for execution. A knowledge representation language is a language used to represent knowledge formally in a knowledge base. There are numerous such languages available, for example Fensel & van Harmelen [FvH94] compare eight languages that formalize the influential KADS\textsuperscript{1} [SWB93] method for building knowledge-based systems, and van Harmelen et al. [vHdMMT92] compare a further eight

\textsuperscript{1}KADS is a general framework for the development of knowledge-based systems that centres round an implementation-independent conceptual model of the relevant problem solving expertise. It has been used by a large number of academic and commercial groups.
languages that are not KADS-based.

There is a trade-off between the expressiveness of a representation language and its computability [LB85], and different applications demand different trades. At one extreme it can be useful simply to represent knowledge formally without any execution capability. Such a specification provides a consistent record of the available knowledge. There is then a premium on expressiveness, and other concerns, such as knowledge re-usability, may also need to be considered. A controller development tool is at the opposite extreme where execution speed is paramount, and expressiveness must be strictly curtailed in the interests of execution efficiency.

The majority of work in knowledge representation is between the extremes, but with less emphasis on execution speed. A traditional rule-based system channels its inputs and outputs to a human user, and it can therefore afford a number of seconds to respond. This loose time-frame, and ever increasing hardware speeds, allow the focus to remain on expressiveness. After all, knowledge can be very complex to model. For example, KADS distinguishes four different layers of knowledge within the model of expertise and proposes different primitives for each.

A variety of modelling primitives have been proposed to capture different types of knowledge. Frames [Min85] are widely used. These are data structures, similar to program objects, that hold data values and code segments to operate on that data. The data can be atomic or list values, including other frames.

Of the eight languages compared by Fensel & van Harmelen only five are executable, while all but one are either Turing complete or more than Turing complete in expressiveness. Three use frames. Of the eight compared by van Harmelen at al., five are executable while a further three can generate prototype implementations. All but two are equal or more expressive than first order predicate calculus.

Frames and this level of expressiveness are not appropriate for developing controllers. We can make do with simple relationships from combinations of inputs to outputs. We focus instead on control surface continuity, time contrains
2.3 Control Surface Continuity

The arbitrary changes of curvature that cause such difficulties for mathematical modelling are not a problem for traditional rule-based systems because boolean rules are independent. The consequence of a rule is not constrained by the consequences of rules with similar antecedents, and a control surface defined by rules is therefore very malleable. Unfortunately, the complete independence of rules and the grounding of variables, as described in the following sections. A useful level of data aggregation is described in Section 2.8.

Figure 2.1: Mathematical and boolean rule-based models of a sigmoid curve.
Figure 2.2: Mathematical, boolean and fuzzy rule-based models of a sigmoid curve.

brings its own problems.

In the boolean domain there are no half measures, and a rule applies either fully or not at all. This is appropriate for the discontinuous realm of symbol manipulation where the gulf between two symbols is absolute. Control, in contrast, is concerned with relating continuous domains, and equation-based control surfaces are continuous landscapes. Boolean rules can take subranges as their antecedents and yield discrete values as their consequents, but the resulting control curve is a tessellation of plateaux, each bounded by sheer cliffs. As the match between such a surface and a given continuous curve is improved, the number of
rules increases very rapidly towards infinity.

Figure 2.1 shows a simple example. Consider modelling a sigmoid curve, which might represent the relationship between stimulus and response in a biological system. (In such a simple example a mathematical model based on the equation, \( \frac{1}{1 + e^{-x}} \), of the sigmoid function is easily identified, and the equation is relatively easy to compute. In realistic examples this would not be the case, however.) The boolean rules take input subranges as conditional terms (e.g., \([-5 ... 2]\)), and yield distinct output values as consequents, but the results are poor. The addition of further rules would improve the match to the curve, but would reduce both performance and comprehensibility.

The traditional expert system is not limited to inference, however. It can also implement functions [Llo87], and these can be used to generate a continuous output curve. If the consequent of each rule is a function that computes a control output from the current inputs, then the rules partition the space, and map a distinct function, generating continuous output, to each partition.

In a similar vein, Nerode and Kohn [NK93] propose a method in which a digital control automaton is used to alter the control law of a continuous plant controller. In this way a well-established logical system with a state semantics interpretation can be used to describe the evolution of the system while the continuous control surface is preserved.

These approaches are preferable to mathematical modelling because they break down a single complex mathematical model into simpler patch models, one associated with each rule or state. They are also preferable to discrete logic because they provide a continuous output over most of the control space, and because they generalize the rules from a number of point mappings to a complete mapping of the space.

They have two problems, however. First, the mapping is discontinuous at the transitions between the patches. When the control point moves from the domain of one rule to that of another, a new control law is applied and the switch causes a jump discontinuity. Second, only a small amount of the system knowledge is represented in the logic. Much of it remains in the patch equations.
Chapter 2. Controller Design

The alternative we shall consider is based on fuzzy logic, a form of multi-valued logic originated by Zadeh [Zad65]. Kosko [Kos92b] provides a thorough coverage. Fuzzy logic has been associated with issues as diverse and emotive as Zen Buddhism [Kos93] and contemporary intellectual degeneration [LS94]. I do not contribute to that debate here, but make use of the simple and effective mechanism of fuzzy inference.

The output of a fuzzy inference system is the result of combining the outputs of a number of rules firing with different strengths. The ranges of adjacent rules overlap so there is no discontinuity between rules. Fuzzy inference uses a simple graph-based technique to represent the variation in the strength of a rule over its range. Rather than divide the system knowledge into a large number of separate patch equations, fuzzy inference provides a unified mechanism for smoothing between rules. Instead of opaque mathematical notation, it uses a clear graphical representation for the lower level knowledge.

Figure 2.2 shows the effect of using just four fuzzy rules for the sigmoid problem. Each rule defines a point in the control space, but the rules are not fully independent. The fuzzy inference mechanism interpolates between the rules to provide smooth transitions. The interpolation depends on the simultaneous partial firing of multiple rules, and at the extremities of the curve, where only a single rule applies, the curve is a horizontal line identical to a boolean output.

### 2.4 Fuzzy Inference

Fuzzy inference is an approximate technique, and controller performance has been found to be robust in the face of quite substantial alterations to the inference mechanism [PK92]. A rigorous mathematical exposition is therefore out of place here as it would lead to unnecessary complexity and a false impression of precision. Such treatment can be found in the literature (e.g., [Kos92b]), if required. In this section I focus instead on the simplicity of the technique.

Fuzzy inference generates a continuous output curve by applying multiple rules that relate a set of inputs to an output, and then interpolating between
2.4. FUZZY INFERENCE

Figure 2.3: Fuzzy subsets representing 3, [3 .. .4] and about 3.

Figure 2.4: The label subsets used for Stimulus.

Figure 2.5: Fuzzy inference using correlation-minimum correlation-product.

their consequents. Some details of the inference mechanism permit variation, but the method described below is representative. It is the method used by the shell developed for this thesis.

Each variable is represented by a fuzzy membership graph with the variable’s range on the x axis, and membership (μ) on the y axis (Figure 2.3). A fuzzy subset maps a subrange of the variable to membership values. Any convex curve can be used, but simple trapezoids (as shown) are common in real-time applications. The leftmost subset in Figure 2.3 is an important special case, called a singleton. It represents a crisp (non-fuzzy) value.

Fuzzy subsets on a variable are sometimes called fuzzy labels because each is
associated with a text label that describes it. Figure 2.4 shows the trapezoidal subset shapes and associated labels of the input subsets used to generate the fuzzy sigmoid approximation. A fuzzy subset’s shape characterizes the extent to which each value in the variable’s range belongs to that subset. A fuzzy subset in a fuzzy rule activates the rule according to the degree of membership of the current value of the variable.

The process of converting the value of an input variable to membership values of all its subsets is called *fuzzification*. The input value is the result of an observation, and therefore approximate. Strictly speaking, it should be represented by a fuzzy subset with a width and shape dependent on the uncertainty. Fuzzification would then consist of finding the intersection of this with each label subset. In practice, the fuzziness of the label subsets subsumes the input imprecision however, and fuzzification simply finds the intersection of the crisp input singleton with each label subset.

Figure 2.5 shows the rule *if Stimulus is Negative then Response is Small* given an activation of 0.75 by a crisp input stimulus value of −0.5. The rule modifies the weight of its consequent subset (*Response is Small*) accordingly. Two widely used methods (correlation-minimum and correlation-product) of modifying the output subset weight are shown.

Fuzzy inference interpolates between the rule defined points on the control curve through the relationships between the membership values of the input subsets, and the relationships between the weights of the output subsets on the other. In a simple case, like the fuzzy sigmoid approximation, where an output depends on only one input variable, the interpolated slope of the output is
generated directly from the overlap of the sloping portions of adjacent subsets as the corresponding rules apply simultaneously but with differing weight. For example, for an input of $-0.5$, we have seen that the rule if Stimulus is Negative then Response is Small is given an activation of 0.75, and the rule if Stimulus is Positive then Response is Large is given an activation of 0.25. Interpolation is achieved by defuzzification, which takes the centre of gravity (COG, a standard technique, e.g., [Bro86]) of the subsets on the output variable to obtain a crisp output. In this case the result is 3.7, as illustrated in Figure 2.6.

It is important to appreciate that the flatness of the regions at the extremities of the curve in Figure 2.2 ($-5$ to $-3$ and $3$ to $5$) is the result of only a single subset applying, and is not caused by the flatness of the peaks of the input subsets VeryNegative and VeryPositive over those ranges. These subsets could have any non-zero value over these ranges and the result would be the same because the corresponding rules are un-opposed there and their output subset COGs therefore define the output value irrespective of their actual weights. In these circumstances then, for clarity, a subset should have a membership value of either $0\mu$ or $1\mu$ in any part that does not overlap with a neighbour.

The situation is more complex and less explicit, unfortunately, in the more useful case of inference from multiple input variables. The fractional membership values over a region of a subset that is not overlapping may then be significant because the rule that depends on it may be competing with another rule based on a different input variable. This relationship is not visible in the single input variable graph but is implicit in the relationship between two or more graphs and the rules that relate them.

As a more realistic example, consider a truck backing up to unload at the mid-point of a dock. This scenario will be studied in depth in Chapter 5. For the purposes of control, the truck can be abstracted to a main axis representing its orientation and wheel-base, and a steering axis to represent the steering angle (Figure 2.7). Figure 2.8 shows the assignment of the control variables to the truck and to its orientation to the docking point. The mid-point of the dock, which is the target, is at a given Bearing from the rear of the truck. The axis of the truck
has a *Turn* angle with respect to the *Bearing*. The trailing front wheels are given an angle *Steer* to control the truck’s path.

The simple fuzzy controller illustrated in Figure 2.9 backs the truck up to the centre of the dock but does not align it. Each variable has subsets labelled *Left* and *Right*, with the shapes shown. The scope of a label is restricted to its variable, and thus *Bearing is Right* and *Turn is Right* are distinct. At each time step, the crisp value of each input variable is matched against its subsets to obtain a membership value for each subset. For example, Figure 2.9 shows that a bearing of $-8^\circ$ gives $0.4\mu$ for *Bearing is Left*, and $0.1\mu$ for *Bearing is Right*, and that a turn of $-18^\circ$ gives $0.8\mu$ for *Turn is Left*, and $0.2\mu$ for *Turn is Right*. 
Rule 1: if Bearing is Left and Turn is Left then Steer is Left
Rule 2: if Bearing is Right and Turn is Left then Steer is Left
Rule 3: if Bearing is Left and Turn is Right then Steer is Right
Rule 4: if Bearing is Right and Turn is Right then Steer is Right

Figure 2.9: A method of fuzzy inference applied to a truck backing up to a dock.

Conditional terms are combined using $\min$ for and, and $\max$ for or to determine the overall activation level of a rule. Thus Rule 1, for example, has an activation of the minimum of the $\mu$ values for Bearing is Left and Turn is Left. This is $\min(0.4, 0.8)$, or 0.4. The activation level scales the rule’s contribution to the output, but each output subset takes on the value of only the most active rule that implies it. This is effectively disjunction of rules with the same consequent. For example, the output value of the subset Steer is Left is the maximum of the activations of Rules 1 and 2, in this case 0.4.

Finally, the centre of gravity of all the subsets of an output variable provides its crisp output value ($\Delta$ in Figure 2.6). Using the correlation-product method, an output subset is simply a weight at an offset that contributes to the centre of gravity, and it is therefore represented here as a vertical line.

There are two extra points to note from this example. The first is that even if the consequent value of each rule corresponds to only one of the conditional
terms, the other terms can have an influence. If these were boolean rules, Bearing would be irrelevant because the value of Steer corresponds to the value of Turn. With fuzzy rules, all input terms can have an influence. In this example, the small absolute value of the bearing limits the effect of Rules 1 and 2 from the 0.8 of Turn is Left to the 0.4 of Bearing is Left.

The second point to note is that, as stated above, a fractional membership value is significant even where it does not overlap with another subset. The fractional membership value of Bearing is Left is significant even where it does not overlap with Bearing is Right. If the bearing were $-12^\circ$, for example, Bearing is Right would be 0.0, but Bearing is Left would be 0.6, and would still limit the influence of Rule 1 against Rule 2. This is in contrast to the fuzzy sigmoid approximation where there was only a single input term, and if subsets were not overlapping, whichever had a non-zero value took full control of the output.

### 2.5 Shell Script vs. Fuzzy Associative Matrix

Much of the work in fuzzy control has used a matrix representation for fuzzy rule-sets (FAMs [Kos92b]). This arrangement is similar to a look-up table but the result is obtained by what is effectively a parameterized smoothing over a small neighbourhood of table entries. It leads to a compact representation of the rule-set in a form suitable for direct VLSI implementation.

Figure 2.10 shows an example of a fuzzy rule matrix representation of a controller that aligns the truck while backing it up to the dock. This is the final rule-set developed in Chapter 5. Each dimension of the table represents an input variable, and each heading represents a fuzzy subset. Each cell represents a rule.

<table>
<thead>
<tr>
<th>Turn</th>
<th>Bearing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Left</td>
</tr>
<tr>
<td>Left</td>
<td>Left</td>
</tr>
<tr>
<td>Zero</td>
<td>Left</td>
</tr>
<tr>
<td>Right</td>
<td>CentreRight</td>
</tr>
</tbody>
</table>
if Turn is Left and Bearing is not Right then Steer is Left
if Turn is Left and Bearing is Right then Steer is CentreLeft

if Turn is Zero and Bearing is Left then Steer is Left
if Turn is Zero and Bearing is CentreLeft then Steer is CentreLeft
if Turn is Zero and Bearing is CentreRight then Steer is CentreRight
if Turn is Zero and Bearing is Right then Steer is Right

if Turn is Right and Bearing is Left then Steer is CentreRight
if Turn is Right and Bearing is not Left then Steer is Right

Figure 2.11: Textual representation of a rule set.

For example, the top left corner represents the case where Bearing is Left and Turn is Left. The cell contains the label Left, and thus the cell defines the rule if Bearing is Left and Turn is Left then Steer Left.

This matrix representation has a number of draw-backs that are avoided by the more flexible textual format typical of traditional expert systems, which is illustrated in Figure 2.11:

**Exponential rule-set growth:** Each cell represents a rule, and the matrix as a whole represents the exhaustive conjunctive combination of the subsets. Each variable in the rule-set adds a dimension to the matrix, and thus, if there are \( n \) variables with \( m \) subsets each, the matrix is required to contain \( O(m^n) \) rules, and the rule-set expands exponentially with the number of input variables. A textual rule-set is not so constrained, and the rule-script author can take the attitude that a variable is a resource to be exploited when required and ignored with impunity, not something which must be consulted by every rule.

**Redundancy:** If regions of the control space are not visited then the corresponding rules are redundant. The matrix is forced to represent them, but redundant textual rules may be simply omitted.

**No disjunction:** Figure 2.10 shows a typical level of duplication of cell values. The matrix format cannot take advantage of this redundancy either because it does not permit disjunction. Figure 2.11 shows how a textual rule-set can
use disjunction to reduce the number of rules.

**Unsuited to incremental development:** My purpose is to develop support for rapid prototyping of controller rule-sets. It must be possible to introduce terms incrementally, adding new subsets and variables where required, and retaining them only if they demonstrate improved performance. The matrix format is inflexible, and unsuited to such experimental development work.

For these reasons I have taken the expert system approach, and developed a shell that stores knowledge as discrete rules with arbitrarily complex conditional expressions, and any number of consequents.

### 2.6 Time Constraints

The inference engine of an expert system, its *production system*, works towards a goal within a conceptual world defined by the given facts and rules. A control system, in contrast, works towards its goal through the actions of the controlled system in the physical world. To ensure safe operation, it must be responsive to changes in the world, and this means that it must be deterministic and meet hard deadlines. It is both a strength and a weakness of production systems that they guarantee to either track down every consequence of a rule set and a set of facts, or perish in the attempt. This guarantee obligates them to follow chains of inference of unknown length, and if a chain has no end then the system does not terminate. Such behaviour is not appropriate for a control system, which must generate an output signal promptly. It can bear neither the cost of following long chains of inference nor the risk of non-termination.

Time sensitive production systems suitable for complex control are being developed [IGR92], but for controllers we can take a simpler approach. Within the chain of deductions carried out by a production system, the firing of a single rule adds facts to the database, changing its world and forcing the re-application of all the rules. A controller cannot act on a single rule but must generate a coherent response through the application of all its rules. The rules are not chained, however, so this takes constant time and allows the controller to meet hard deadlines.
When the response is acted upon, the controller’s world is changed, and, as in the case of the production system, the rules must be re-applied to determine the response to the new world state. In equating a rule-based controller to a production system then, the application of the controller’s complete rule set maps to the application of a single production rule, not to a complete production system run. Each instance of deducing control outputs from inputs is a single step in a chain of similar deductions, and thus it is entirely appropriate for only a single inference step to occur at each time step.

Fuzzy rules are particularly well suited to simultaneous application because the law of the excluded middle is not an axiom of fuzzy logic. As a result, contradictory rules can fire simultaneously without rendering the outcome trivial. Contradictory conclusions merely apply weight to opposite sides of the output variable; the centre of gravity defuzzification method has no difficulty deriving a sensible outcome in most cases (see [PYL92] for an exception).

Note that by basing action directly on the current state of the inputs, a rule-based controller is reactive and follows Brooks’s injunction [Bro91] to use the world as its own model. The world takes the place of the database, and rules directly apply to world states. In some cases it is difficult or unnecessary to transcribe state changes out into the world, and in these cases intermediate variables [HHNT86] can be used. These obtain their values through inference in one time step, and serve in the conditions of rules in the next time step. They thus allow the system to maintain some internal state distinct from the world state. They propagate the results of inference between time-steps to allow limited multi-step inference chains in a system with only a single inference-step per time-step.

The controller does not avoid the termination problem. Rather it transforms it from a problem of a single decision to a problem of the whole mission. If a given rule set is inadequate to reach the goal in a given world instance then the system will become stuck just as surely as a production system would. The difference made by restricting the inference at each time step is that the system remains responsive to changes in its environment, and both continues to act safely
and takes advantage of any improvement in its situation effected by external influences. For example, a robot acting on a rule set that leads it to a dead-end will remain responsive, and continue to the goal if the blocking obstacle is removed.

Although this arrangement clearly eliminates the possibility of planning, I do not deny its utility. I consider planning a distinct issue that must be addressed separately (as [MC92]). In the example above, if the robot can detect that it is making no progress, then it can initiate planning to identify a new path round the obstacle. A lower level rule set must maintain its active interaction with the environment in the mean time, however. A rule set prototyped with the shell should be able to handle the moment-to-moment real-time responses, but the planner would require a different inference method. The fuzzy controller development shell presented in this thesis has an architecture that allows it to be easily integrated into larger systems, which may include such a planner.

### 2.7 Symbol Grounding

Figure 2.12 compares a rule-based symbolic system to a rule-based control system. The inputs and outputs of a symbolic system are text strings that are interpreted
by a user. The buffering effect of this interpretive stage has for many years cushioned the field of artificial intelligence (AI), and made assessment of the real significance of many of its results very hard. Expert systems are such symbolic systems. Symbol grounding [MS90] is the name given to the grounding of the symbols manipulated in a computer system to objects (or at least perceptions) and actions in the real world. Brooks [Bro90] calls this the physical grounding of a system. Its extreme difficulty has caused a drifting apart of the closely related fields of robotics and AI; the former is forced to deal with it, and the latter is mostly content to ignore it [Bro91]. A recent resurgence in robotics has forced a recognition in AI circles of the central importance of symbol grounding. It is an area in which practical control applications have a large part to play [MS90].

Here we are concerned with the mechanics of interfacing between the symbols used in composing control rules, and the underlying control data. Input and output variable names refer to data values that are derived from or affect the controlled system. At the same time, they refer to conceptual entities central to the control model manipulated in the mind of the system developer. The model is further refined by the subset and intermediate variable names. Taking a simple view, we merely need a way to match names to values. From a deeper perspective, we need a way to create the concepts used by the developer. The variables referred to by the rules need not be raw values taken directly from the hardware because the development of a controller involves the choice of appropriate control variables at an appropriate conceptual level. These variables may not correspond to the actual quantities offered by the hardware. There is scope for abstraction: the creation of virtual inputs and the interpretation of virtual outputs.

The mapping between hardware values and virtual variables may therefore be many-to-many, and require arbitrary computation. For example, a number of sensor inputs (hardware values) may need to be combined to identify a plane wall, which is then described by a distance and an angle (virtual) variable. The grounding translation between variables and actual control signals is therefore most conveniently achieved by code developed in an established programming language. We are left with the question of how to link the shell to this grounding
code. My solution is to use an object-oriented language to develop the grounding
code, and to make the shell available as a program object from an object library
in the same language. Its interface methods allow the grounding code to make
variables available to it, and to call for script compilation and for inference.

2.8 Data Structure

Abstraction over data groupings is a powerful language tool. Language support
for data structure is an area where there is almost no limit to the level of sophis­
tication that can be implemented. Data structuring frames have been mentioned
above, and complex data access syntax has been developed for object-oriented
database query languages [Bee88], for example. Such sophistication bears a cost
in both compilation and execution speed, however, and is not appropriate for
controller prototyping.

The array is perhaps the most primitive level of data structure. It is the only
structure available in the early languages Fortran and Algol, and more recently
in Occam [JG90]. Language support for arrays can simplify the rule-set consider­
ably. It can be supported at the compiler level, and thus have no execution cost.
It is simple enough to also have small impact on compilation time.

A single rule that references an array is equivalent to a set of rules defined one
for each element. For example, if the speed of a robot is to be controlled by the
proximity of obstacles detected by a ring of 16 sonar sensors, then a single pair
of rules defined on a 16 element array can replace 16 pairs of rules defined on the
elements. This is a powerful notational convenience and can also be the basis for
simplification of computation in some cases, as described in the next chapter.

If the number of elements varies at run time, then support for the array
abstraction is a necessity. For example, if a robot system identifies plane wall
segments for use as navigation beacons, then the number of segments currently
detected will vary. A fuzzy inference system can handle this situation if it supports
arrays and is provided with an instance count at each time step. Without this
facility the system could not be used.
2.9 Rapid Prototyping

In Section 2.1 I argued that rules are a better medium than equations for knowledge representation in controller modelling. The question of how to approach the modelling task remains. By abandoning mathematical modelling I have turned my back on a considerable body of well established formal development methods. The alternatives can be divided roughly into three types, knowledge acquisition, rapid prototyping and machine learning. The traditional form of knowledge acquisition is not relevant to our research, where pre-existent expert knowledge is not available. We must rely instead on the general experience of the researcher to identify effective control variables and potentially useful rules. Rapid prototyping is the appropriate method whereby these initial guesses can be tested and adjusted, and I therefore chose to focus on the support of rapid prototyping in the development of the shell.

Interest in rapid prototyping is growing. Gordon & Bieman [GB95] surveyed 39 software projects that used it in a wide variety of applications, and found a high level of success and satisfaction. Rapid prototyping has been applied successfully to the development of fuzzy control systems (e.g., [Sel90, Lin93]).

Machine learning has also been used successfully with fuzzy control systems (e.g., [KV93, Bir93]), with robots (e.g., [MC92, VdV93]) and with fuzzy control in robotics (e.g., [KB95]). This thesis focuses on prototyping for fuzzy controller development, however, and learning is outside its scope. It will not be considered further.

The architecture of the fuzzy controller development support shell developed for this thesis assumes two phases of development, a prototyping phase, in which control variables and fuzzy relationships between them are developed, and an optional implementation phase. The implementation phase involves the compiling down of the prototype into an independent fuzzy controller that can be linked to the control system without the intervention of the shell. Implementation of support for that phase would be quite straightforward, and this thesis is concerned only with the prototyping phase.
Figure 2.13: The parts of a fuzzy control system and stages of system identification.

2.10 Fuzzy Qualitative Modelling

Sugeno & Yasukawa [SY93] propose a fuzzy-logic-based approach to qualitative modelling that provides us with a model of the fuzzy controller development process. They consider modelling a system as a black box where only the input and output characteristics are known. While this is the most difficult form of modelling, it is the most appropriate for our purposes because we may be concerned with systems that are too complex for mathematical modelling, and we may have to rely on sample data points, existing expertise, or common sense and trial and error.

Black box modelling involves identifying a model system that has the same input and output characteristics as the modelled system. Since nothing is known of the actual mechanism, the designer must propose one that exhibits the same behaviour. Sugeno & Yasukawa divide model identification into two sequential stages: Structure Identification and Parameter Identification, and further divide Structure Identification into Stages I and II. Structure Identification Stage I consists of identification of the control variables. Input to output relationships are identified in Stage II, and Parameter Identification tunes the parameters (vertex locations) of the fuzzy subsets.

The following chapters describe a fuzzy controller development shell whose architecture exploits the sequence of the stages to accelerate prototyping. The identification of the control variables in Structure Identification Stage I forms an
initial stage in which standard object-oriented software design techniques are used to develop the grounding code that defines the input and output variables. Thus, the first stage encapsulates the controlled system, and provides an abstract, and relatively static, platform for the shell. The shell supports Structure Identification Stage II, the identification of input to output relationships with rules, and Parameter Identification, the adjustment of the subset trapezoids. The overall structure is illustrated in Figure 2.13.

2.11 Summary

This chapter has dealt with the following issues:

- Rule-based control can be simpler, more efficient, more robust, and easier to understand and maintain than mathematical modelling.

- Traditional matrix-based fuzzy controllers restrict the design process and cannot exploit redundancy in the rule set. Textual rule sets, similar to expert systems, can be used instead, and they do not have these drawbacks.

- Control differs from traditional symbolic applications of rule-based systems in the following four ways:
  
  - Inputs and outputs are continuous. Fuzzy inference can be used to handle this, and the restrictive matrix representation for fuzzy rule-sets is not necessary.

  - Time constraints are much tighter, requiring the use of single-step inference at each time step.

  - The symbols used in the rules must be grounded to real input and output data.

  - The data may be structured, and this structure can be exploited to simplify the rule set.

- Turning from mathematical modelling to model-free rule-based control has removed the applicability of many well established development methods.
Rapid prototyping is the alternative of choice.

- Fuzzy qualitative modelling suggests an architecture in which the control variables are identified first, and defined by grounding code. The fuzzy relationships are prototyped with the shell in a subsequent developmental stage.

On the basis of this analysis, I chose to develop a textual rule-based tool for the development of fuzzy controllers. It exploits Sugeno & Yasukawa’s model to separate out the preliminary variable definition stage and allow rapid prototyping of the fuzzy relations.
Chapter 3

The Shell: Architecture & Interfaces

This chapter deals with the requirements for a fuzzy controller development tool, and the novel design of the tool developed for this thesis. I call the tool Fido. Section 3.1 discusses the architectures of other fuzzy controller development tools. These fall into two classes: expert systems with fuzzy extensions, and fuzzy controller specification compilers. The former can be used for prototyping, but result in systems with slow response. The latter are not suited to rapid prototyping.

Fido has a novel architecture that allows it to be used for the rapid prototyping development of fast response fuzzy controllers. It is a C++ program object that can be linked into a control system written in C++. The developer identifies the control variables, writes C++ grounding code to define them, and incorporates Fido into the control system with a minimum of fuss. The result is a system that allows rapid prototyping of the fuzzy relationships.

The subsequent sections describe the shell’s two interfaces, one for the control system programmer (Section 3.2), and one for the script author (Section 3.3). The program interface has been made as small and simple as possible. It allows control variables to be declared to the shell, and script compilation and inference to be called for. It also permits fuzzy variable definitions to be moved to new data addresses. The script syntax is quite complex, to allow for the definition of fuzzy
variables (Section 3.3.1), individual subsets (Section 3.3.2), and of fuzzy rules (Section 3.3.5). Novel syntax extensions to simplify the script author's task are also described, including facilities to permit the definition of sets of overlapping subsets (Section 3.3.4), and of vector variables (Section 3.3.6).

3.1 The Structure of a Controller Development Tool

A tool that supports the development of fuzzy controllers must allow the developer to define fuzzy relationships between variables, and to ground these variables to actual control signals. Grounding can be achieved with a conventional programming language, and it would be possible to extend such a language to include a syntax for defining fuzzy relationships. However, although the two tasks deal with the same variables, they are quite different conceptually and it is preferable to use separate languages. Thus, a fuzzy controller development system must incorporate two languages, and provide cross-language linking of corresponding variables. I next discuss some of the approaches currently available, and describe how Fido, the shell developed for this thesis, differs from them.

FLOPS [BST86], FEST [WS94] and LIFE FEShell [THS93] are expert system shells that have fuzzy extensions to provide continuous output. They rely on production systems for inference, and the resulting multi-step inferencing leads to substantial delays, as described in Chapter 2, Section 2.6. Wood & Schneider [WS94], for example, report a 4.5 second time step for a FEST application, equivalent to a distance of 200 metres travelled by the controlled helicopter. We require a much tighter control loop to safely control autonomous robots.

Togai InfraLogic's Fuzzy-C Expert System compiles a script-defined fuzzy controller into C code that can then be compiled as part of a control system. This is the model proposed in this thesis for the second (implementation) phase of controller development using the shell. It is a good way to support fuzzy controller development in an appropriate language, but its separate compilation phase makes it unsuitable for rapid prototyping.
3.1. THE STRUCTURE OF A CONTROLLER DEVELOPMENT TOOL

Motorola and Aptronix' Fuzzy Inference Development Environment, *Fide* [Mot92], supports script-defined fuzzy controller development, but it uses four different text file types, each with its own syntax. Fuzzy relationships are defined in *Fuzzy Inference Units* (FIUs), arithmetic functions to normalize input and output values are defined in *Fide Operating Units* (FOUs), grounding is achieved in C code files, and *Fide Execution Units* (FEUs) define the linkage between the other files and the C code. *Fide* has its own linker, called the *Composer*, to link the various object files together to create a system. The linking step must be repeated whenever a fuzzy relationship is altered, and *Fide* is therefore unsuitable for rapid prototyping.

*Fide* is the closest to our needs, but it is over-elaborate and slow to compile. This is because it does not reflect the structure inherent in the system development task that was identified by Sugeno & Yasukawa in their model of fuzzy design as black box modelling. *Fide* does not impose a priority between development of the grounding code (Structure Identification Stage I), and of the fuzzy relationships (Structure Identification Stage II and Parameter Identification). An effective rapid-prototyping fuzzy controller development tool should prioritize the two phases of development, and thus both clarify the development task and accelerate the update-compile-test prototyping cycle.

I have taken advantage of the additional problem structure to develop a new fuzzy controller development support tool that is suitable for the rapid-prototyping of fuzzy relationships. The tool is a shell in which fuzzy relationship scripts can be authored. It is itself a code module written in the programming language used to develop the grounding code, and it is integrated into that code by the standard linking process. This arrangement obviates the need for multiple definition languages.

Following Sugeno & Yasukawa's model, the controller developer first develops the grounding code that creates the control variables from the input and output signals. This code may be part of a larger control system. It is unlikely to require much alteration during the subsequent Structure Identification Stage II and Parameter Identification. The normal code linking mechanism is used to
link the shell with the grounding code to form a complete prototyping controller that reads, compiles and executes a script file of fuzzy variable, subset and rule definitions. Compilation of this script does not require re-linking because it is carried out by the program, which was written with speed of script compilation in mind. The update-compile-test loop is therefore fast, and appropriate for rapid prototyping.

This new tool differs from existing tools in that it is suitable for rapid prototyping development of fuzzy relationships for short time-step controllers. The grounding code, which defines the control variables, is written first and linked with the shell to create a scripting environment in which the fuzzy relationships between the defined variables can be modified with a minimum of time overhead.

3.2 The Shell’s Architecture & Program Interface

The shell is a module encapsulated as a program object, a Fuzzy Inference Development Object (Fido). It is written in C++ to conform with our laboratory standard. C++ is a good choice for this application because it both provides the necessary object support and is very suitable for writing low-level grounding code. It has been commended as an excellent language for robot control systems because of its support for object-oriented design, and clean handling of initialization, termination and exceptions [Cox88, CG89].

The grounding code and the shell are parts of a complete control system written in a single base programming language. In its simplest form, this system consists of an initialization section that initializes the controlled system and compiles the script, and an operating loop that generates control outputs from its inputs in discrete time steps. Figure 3.1 describes the methods that the control system uses to manage the shell. infer is called in the operating loop, the rest are used for initialization.

Figure 3.2 shows an example of the simplest form of shell-supported control system. We assume the object Truck grounds the control variables by imple-
variableIn & variableOut pass the character-string name, typed data address, and arity of a variable to the shell. Optionally, the bounds of its range can also be specified to constrain the script.

compile creates, or reads and compiles a script file.

infer performs one complete step of fuzzification, inference and defuzzification.

moveVarIn & moveVarOut take a fuzzy variable definition identifier, as returned by variableIn and variableOut, and a typed data address, and make the fuzzy definition apply to the data indicated.

Figure 3.1: The program interface.

menting a simulation of the truck backing up problem or interfacing to a real truck. variableIn and variableOut are called during initialization to declare the control variables to the shell. As an example, the arity and range limits are set explicitly for Bearing. The ranges of the other variables are unbounded, and their arities default to 1. The script author may tighten the bounds on a control variable, but its other features, including its name, are defined here by the control system programmer.

In this case, the script is to be compiled from the file named steer.scr. When the control system is run for the first time, this file does not exist. When compile cannot find the file, it creates a new one with that name, writes the control variable declarations (provided through variableIn and variableOut) to it, and causes the program to exit. The resulting initial file of control variable stubs is shown in Figure 3.3.

The script author can then extend the script to a complete fuzzy controller specification, and re-run the control system to test it. Now that the file exists, compile compiles it and, if it is error free, allows the control system to execute using the specified fuzzy relationships. Figure 3.4 shows the final version of the script, the syntax for the subset declarations is explained in the following sections. At each iteration the operating loop transforms the input signals to the input variables, calls infer to update the output variables according to the fuzzy rules, and transforms the output variables to the output signals.
int main()
{
    Truck truck;
    Fido shell;
    float bearing, turn, steer;
    int arity = 1;
    const float Bound[2] = {-90.0, 90.0};
    shell.variableIn( "Bearing", &bearing, &arity, Bound );
    shell.variableIn( "Turn", &turn );
    shell.variableOut( "Steer", &steer );
    shell.compile("steer");
    while( ! truck.atdock() ) {
        bearing = truck.bearing();
        turn = truck.turn();
        shell.infer();
        truck.move( steer );
    }
    return 0;
}

Figure 3.2: An example of the use of the program interface.

Bearing is input float from -90 to 90
Turn is input float
Steer is output float

Figure 3.3: The initial script of control variable stubs generated by the shell.

A shell-supported control system is not limited to this simple usage, however. A complex control system may require multiple sets of fuzzy relationships, applicable in different situations. The control system discussed in Chapter 6, for example, makes use of two instances of the shell. A control system may also use the fuzzy output variables of one script as input to higher-level fuzzy relationships defined by another script.

If the variable is large, an array of float for example, the programmer may prefer to wait until it is used before allocating memory for it, perhaps on the stack. The interface allows for this with the moveVar methods. A variable method can be passed a typed null data pointer, and the fuzzy definition identifier it returns stored and passed to moveVar along with the true data pointer when it becomes
3.2. THE SHELL’S ARCHITECTURE & PROGRAM INTERFACE

Bearing is input float
[-90 -90 Left -48 -8 CentreLeft -4 4 CentreRight 8 48 Right 90 90]

Turn is input float
[-180 -180 Left -30 0 Zero 30 Right 180 180]

Steer is output float
.Left is -20
.CentreLeft is -2
.CentreRight is 2
.Right is 20

if Turn is Left and Bearing is not Right  then Steer is Left
if Turn is Left and Bearing is Right  then Steer is CentreLeft
if Turn is Zero and Bearing is Left  then Steer is Left
if Turn is Zero and Bearing is CentreLeft  then Steer is CentreLeft
if Turn is Zero and Bearing is CentreRight  then Steer is CentreRight
if Turn is Zero and Bearing is Right  then Steer is Right
if Turn is Right and Bearing is Left  then Steer is CentreRight
if Turn is Right and Bearing is not Left  then Steer is Right

Figure 3.4: The script steer.scr developed to control a truck backing up.

available.

The shell is a substantial body of code, and adds considerably to the overall size of the control system. The bulk of the shell is the compiler, which is only used during initialization. This suggests that the compiler might be split off as a separate program, and launched by the compile method when required. The communication between the compile program and the rest of the system would extend the compile time, however. Also, the whole shell can be removed from the control system once prototyping development is complete and the proposed final implementation phase has compiled the script to independent C code. The shell has therefore been left as a single unit.

This section has shown how the novel architecture of the shell makes it very easy to incorporate into a control program to create a prototyping system that allows fuzzy relationships to be defined, and refined empirically. Because the shell is written in the same language, it integrates with the control system in the
VarDefine ::= VarDeclare { SubSpec | SubCover }*

VarDeclare ::= VarName ['is'] [Arity] [IODir] [Type] [Range]
IODir ::= 'input' | 'output' | 'intermediate'
Type ::= 'char' | 'int' | 'long' | 'float' | 'double'
Range ::= ['from'] Bound ['to'] Bound

SubSpec ::= '.' SubName ['is'] ( Shape | COG )
Shape ::= Vertex ['to'] Vertex ['to'] Vertex ['to'] Vertex
COG ::= Vertex ['weight' Weight]

SubCover ::= ShapeList [Width]
SubCover ::= NameList [Width] [Width]
ShapeList ::= ['[' Vertex Vertex { SubName Vertex [Vertex] }+ ']
NameList ::= ['[' Vertex [Vertex] { SubName }+ Vertex [Vertex] ']
Width ::= Interval | ( IntervalPC '%' )

VarName and SubName are character strings.
Arity is an integer.
Bound, Vertex, Weight and Interval are values of type Type.
IntervalPC is floating point.
Arity and Weight are greater than 0 and default to 1.
Width is greater than or equal to 0, and defaults to 0.
COG is only allowed in output variable subsets.
Vertex is within Range (if present).

Figure 3.5: The script syntax for variables.

normal program linking step, and the programmer can access it within the normal language environment. After identifying the control variables, the programmer writes the grounding code that defines them (represented in Figure 3.2 by the object Truck), and declares them to the shell. He or she then calls compile to compile the script, and writes a control loop to access the input values, perform the inference, and emit the output values. These two parts, grounding code and control loop, can comprise the entire program, or be just a part of a more complex system.

3.3 The Script Syntax

3.3.1 The Variable Definition
A script consists of variable and rule definitions. They can be in any order, but a rule cannot refer to a variable that has not yet been defined. Figure 3.5 presents the script language syntax for variable definitions, it is explained in this and the following sections. I borrow from C convention and distinguish a variable's declaration from its definition, but use the terms definition and specification synonymously.

A variable definition consists of the declaration and zero or more trailing subset specifications (Figure 3.4). Subset names need to be distinct only within the scope of the enclosing variable. A variable declaration comprises all the arguments to variableIn or variableOut, except the data address: the variable's name, arity, type, and optionally, the boundaries of its range. It also includes the variable's IO direction. The declarations of input and output variables are thus provided by the grounding code but intermediate variables are declared by the script author.

The arity of a variable defaults to 1, as in Figure 3.3. Variables whose arity does not equal 1 are called vector variables. Their behaviour is described in Section 3.3.6. In the case of variables with dynamic arity, the given arity value is the maximum. IO direction declaration by the script author is optional because the shell assumes any variable not already declared for input or output by the grounding code is intermediate. The type declaration defaults to int. The bounds can be specified by the grounding code and/or the script author. Script author defined bounds must be within any program defined bounds, and subset vertices must be within any bounds given.

The use of linguistic terms is one of fuzzy control's prime claims to simplicity and ease of use. In practice, these terms must be tied numerically to subset shapes, however, and the potential for variety of shape is a major loop-hole for complexity to be re-introduced. The standardization and simplification of subset shapes is therefore an important area of research. Some work suggests Gaussian curves and other complex shapes (e.g., [ARH92, Rus92]), but practical real-time systems (e.g., [Ned92]) commonly use trapezoids. Trapezoids lead to very simple computations, and do not degrade performance appreciably because of the
approximate nature of fuzzy inference.

It is a firm tenet of this thesis that subset shape complexity should be minimized to reduce the complexity of specification and computation. The following sections outline simplifications of subset specification and representation that are implemented in the shell to reduce the potential for confusion and error in fuzzy relation definition.

The use of four values to describe a trapezoid is well known (e.g., [Mot92]), and in the following sections I first describe an extension of this to allow for simpler shapes to be specified with fewer values. I then propose a simplified representation and syntax for output subset shapes that reduces the potential for mismatch between the author's intentions and the actual behaviour of the controller. Rather than focus on individual subsets, it allows specification of a higher level abstraction, a regular series of overlapping subsets that I call a subset cover. In this way it increases conceptual and representational simplicity, and significantly reduces redundancy and the concomitant potential for error.

3.3.2 Single Subset Specification

A single subset specification (SubSpec in Figure 3.5) begins with a fullstop, followed by its name and trapezoid specification. Figure 3.6 gives some examples. The connectives *is* and *to* are optional. A subset trapezoid's base lies on the
3.3. THE SCRIPT SYNTAX

\(x\) axis of the membership graph and the shape of the subset can therefore be specified by the \(x\) offsets of its four vertices (e.g., Trapezium in Figure 3.6). The offsets are in the units of the \(x\) axis, which are the units of the variable. I call a vertex on the axis a toe, and a vertex at the top a crest. The toes define a subset's span, and the crests define its peak.

A triangle, a rectangle and a line are degenerate trapezoids having identical offsets for some adjacent vertices. A triangle's two crest offsets are equal, a rectangle consists of two pairs of equal offsets, and a line is four equal values. To specify all these shapes by four values is both tedious and error prone, so the redundancy is avoided by specifying a triangle with three offsets, a rectangle with two, and a line with one. A rectangle represents a crisp range, and a line is a singleton representing a crisp value. This notation requires that wedges—trapezoids with just one vertical side that are often used at the range boundary—still be specified by three or four values, even though the two representing the vertical side are equal (e.g., Wedgel in Figure 3.6).

3.3.3 The Representation of Output Subsets

I followed common practice (e.g., [Kos92b, PYL92]) on page 19, and represented the output subsets as trapezoids. This representation is consistent with the trapezoidal representation of input subsets but it is complex, and I suggest that it can be misleading. Subsets are often scaled in breadth according to their distance from the centre of the domain, with broader ones towards the extremities. If this arrangement is used for output subsets then the COG calculation will give the larger subsets at the extremities the advantage of both leverage and weight, an effect that may not be part of the script author's mental model of the mechanism. The potential for confusion is avoided if correlation-product inference is used, and shape information omitted from the output subsets. They can then be represented explicitly as weight/offset pairs, where the weight takes the place of the subset's area, and the offset is its COG.

The shell syntax therefore allows the option of defining an output subset as a weight/offset pair, and if it is specified as a trapezoid then the weight and offset
are computed at compile time from its area and centre of gravity. An output subset can be represented graphically as an extended singleton—a singleton whose height is in the range $[0..\infty)$. Similar simplifications are suggested in [Miz92, Sib92]. The weight is in the units of area of the subset which, for practical purposes, are equivalent to the units of the variable. It is optional, and if omitted defaults to 1 to clarify the specification of output variables with equal sized subsets (e.g., Figure 3.7).

Figure 3.7:\textit{Steer} defined with singleton subsets of unit weight.

Clearly, a singleton omits the shape information necessary for the correlation-minimum adjustment (Figure 2.5, page 19). By exploiting the extra information, that adjustment method causes a change to a high activation level to have less effect on total subset area and weight than an identical change to a low activation. In doing so it makes more complex the representational model that the designer must bear in mind. Our aim should be to maximize intuitive and computational simplicity without sacrificing functionality, and the shell does this by representing consequent terms as extended singletons, and using the correlation-product method of adjustment.
3.3. THE SCRIPT SYNTAX

Bearing is input float from $-90$ to $90$

. Left is $-90 -90 -48 -8$
. CentreLeft is $-48 -8 -4 4$
. CentreRight is $-4 8 4 8$
. Right is $8 40 90 90$

[ $-90 -90$ Left $-48 -8$ CentreLeft $-4 4$ CentreRight $8 48$ Right $90 90$ ]

Figure 3.8: Bearing defined using single subsets and a semi-regular cover.

3.3.4 Specification of a Subset Cover

The syntax described so far permits the specification of individual subsets, but a fuzzy subset does not operate in isolation (as explained in Chapter 2, Section 2.4 on pp. 20–23). A variable's subsets are usually intended to provide a complete cover over its range. There are instances in the literature of subset covers with irregular distributions and irregular regions of fractional membership that do not overlap (e.g., [SHL92][Kos92b, chap. 9]), but commonly subsets and the extent of overlap are regular (e.g., [Bir93, Kos92a, PK92]) and often fractional memberships only occur at overlap (e.g., [KK92, PK92, Ned92]).

A cover of subsets with regular overlap between neighbours is hard to represent and maintain with a collection of individual subset specifications. The simple relationships between the vertex offsets of successive subsets are obscured by subset textual independence, and the specification has more flexibility, and therefore complexity, than is required. I therefore define a multiple subset cover to consist of a series of subsets with regular overlap, and the shell incorporates a special syntax for specifying them.

As an example, Figure 3.8 shows the subsets of Bearing defined singly and as
a cover. Careful inspection reveals an error in the definition of Right. The second vertex should be 48 (not 40) to match the toe of CentreRight. Such errors are easy to make and hard to spot in the arrays of numbers required for multiple individual subset definitions. The subset cover syntax makes mismatches impossible because a single number represents both vertices.

I divide covers into semi- and fully regular types. A semi-regular cover, like Figure 3.8, has individually defined subsets, but the absolute overlap, or percentage overlap, between neighbours is the same for all (Figure 3.9). A fully regular cover consists of equal sized subsets with, optionally, half size wedges at either end. Each non-wedge subset in a regular cover is identical and symmetrical, overlap between neighbours is identical but not necessarily complete, and an end wedge subset is a standard subset cut in half (Figure 3.10).
3.3. THE SCRIPT SYNTAX

A subset cover specification (*SubCover* in Figure 3.5) is enclosed in square brackets, and optionally followed by one or two quantities. The subsets of a semi-regular cover are specified explicitly in a *shape list*, but the toe and crest of neighbouring subsets are initially assumed to match, and a single value is specified for both (Figure 3.10a). The left-most subset in a shape list is represented by two vertices (toe then crest) followed by its name and one or two more vertices (crest then toe). The next subset assumes the last two vertices of the previous subset to be its first two vertices, so its name comes right after the last toe vertex of the previous subset. Thus, a subset's name always comes immediately after the offset of its first crest. The trailing vertices of the rightmost subset are followed by the closing bracket. As with the single subset syntax, the vertical side of a wedge requires a pair of duplicate values.

If the script author wishes to reduce the overlap of adjacent subsets, he or she specifies a toe retraction quantity following the closing square bracket (Figure 3.10b). The retraction quantity can be either a fixed value in the variable’s units, or an integer or floating point number followed by % that specifies the percentage of each gulf between the crests that does not overlap.

The shapes of a fully regular cover are calculated automatically, and only its bounds and the names are supplied in a *name list* (Figure 3.5). As with wedges in a shape list, a terminal wedge is specified in a name list by duplicating the first or last bound (Figure 3.10). A name list is followed by a value representing the proportion of each subset width that is at a membership of 1 (the peak width). This value is optional and defaults to 0 to give triangular subsets, but it must be included if a toe retraction value is also appended.

### 3.3.5 The Rule Definition

A rule consists of the word *if* followed by a conditional expression, *then*, and a conclusion part (Figure 3.11). The condition part is a straightforward expression tree of input terms conjoined by *and* and *or*. The conjunctions are right associative. A *VarSubPair* is a variable name followed by the name of one of its subsets, with optional intervening *is*. A term can be negated by inserting *not* between
Figure 3.11: The script syntax for rules.

```plaintext
RuleDefine ::= 'if' Condition ( 'then' | 'then1' ) Consequents

Condition ::= VarSubPair
Condition ::= '(' Condition ')' Condition
Condition ::= 'not' Condition
Condition ::= Condition ( 'and' | 'and1' | 'or' | 'or1' ) Condition

Consequents ::= [ 'not' ] VarSubPair [ 'and' Consequents ]

VarSubPair ::= VarName [ ['is' 'not'] SubName ]
```

the names, after the *is* (if present). Any conditional expression can be preceded by *not* to negate it, but only the first term will be negated unless the expression is enclosed in brackets. The conclusion part is one or more terms conjoined by *and*. Negated output terms are assigned the fuzzy negation of the rule's value. This is useful to allow a single rule to influence two subsets symmetrical about a central target control point.

The special subset names *True* and *False* can be omitted from terms in the condition part as they are deduced from the context. A term with no subset specified is assumed to refer to the subset *True*, a negated one to the subset *False*. If *True* is assumed, and only *False* defined, then *not True* is assumed. The inverse works for the assumption of *False*.

### 3.3.6 Vector Variables

I outlined the need for array support in Section 2.8 of the last chapter. I call variables with arity greater than 1, *vector variables* (VV$s$). They require careful handling in rules.

Each subset of a vector variable is an array of membership values, one for each element. A *vector term* is either a VV/subset pair, or two vector terms of the same arity joined by a conjunction. A *scalar term* is anything else: a scalar variable/subset pair, the conjunction of two scalar terms or the conjunction of two terms of differing arity. A composite vector term can also be forced scalar by suffixing the conjunction with '1'.

In fuzzy inference, if a number of rules affect the same output subset then only the rule with the highest activation is significant. Thus, when a vector term is converted to a scalar term, the element that will give the rule its highest activation is selected as its value. This will be either the maximum or the minimum element, depending on the negation status of the term. If the term is not negated (or is negated an even number of times in the expression tree) then the maximum element is chosen. If the term is negated an odd number of times then the minimum element is chosen.

Conjunctive operations within a vector term in the condition part of a rule are performed element by element. If the whole condition part of a rule is a vector term, and the consequent terms also have that arity, then the inference too is performed element by element. The rule becomes effectively a family of rules relating corresponding elements of condition and consequent terms. If the arity of a consequent term differs then the condition is made scalar, and the consequent's elements are treated as distinct terms. The condition part can be forced scalar by suffixing the then with '1'.

Consider the example of a mobile robot with a ring of 16 sensors, and two independently controlled wheels. The rule: \textit{if Sonar Return is Close then Wheel is Slow} would be useful to reduce speed near obstacles. The arities of SonarReturn and Slow are different, so the vector rule is evaluated by taking the maximum of the SonarReturn is Close array of memberships (i.e., the minimum range), and using it to activate both elements of the output subset array, Wheel is Slow. This vector rule is equivalent to 32 scalar rules, one relating each sensor to each wheel. All 32 scalar rules would have to be evaluated at each time step, even though only the two with the highest value for SonarReturn is Close would actually be used to limit the speed of the wheels. Thus this use of a vector variable has a significant effect on the efficiency of rule evaluation.

If an infra-red unit is associated with each sonar sensor, then the (boolean) rule \textit{if IRDetect then Sonar is On} will fire a sonar only if its infra-red unit registers an obstacle. The arities of IRDetect and Sonar are the same, so this single rule is implemented as 16 separate rules. Each relates one infra-red unit to its associated
sonar sensor. In this case every scalar rule would be significant, and the vector rule achieves no economy in implementation. It does simplify the script a great deal, however.

### 3.4 Summary

This chapter has dealt with the external view of the fuzzy controller development tool, Fido. It has covered the following:

- **Other tools described in the literature.** Fido differs from these in that it is suitable for incremental rapid prototyping development of short time-step controllers.

- **The novel architecture of the tool as a C++ program object.** This makes it easy to incorporate into diverse control systems.

- **The control-system programmer's interface,** which provides methods for declaring control variables to the shell, and for initiating compilation and inference.

- **The controller-script author's interface,** including syntax for:
  
  - variable declarations,
  
  - individual subset specifications, and
  
  - rule definitions,

  and novel extensions for:

  - regular and semi-regular subset cover specifications,
  
  - negated consequents,

  - default *True* and *False* subset names, and

  - vector variables.

The tool is novel in that it exploits Sugeno & Yasukawa's model of qualitative modelling to support a rapid prototyping environment for short time-step controllers using textual rules. The separate initial control variable definition stage
is facilitated by making the tool a program object that links to the variables using the normal program linking step. The tool's support for subset covers and vector variables is also new.

The next chapter deals with its internal view: the implementation.
Chapter 4

The Shell: Implementation

This chapter outlines the internal structure of the tool. The implementation of a fuzzy control shell as a program object is novel, and the tool also supports the subset cover notations described in the last chapter and incorporates optimization techniques that have not previously been applied to fuzzy development tools.

Figure 4.1 shows the relationship between the higher level classes in the tool, and the following sections describe their implementation. Section 4.1 describes the top level object, Fido, that supports the program interface, and delegates calls to the objects Vars and Rules andDefs. Section 4.2 describes the lexical analyzer object, Script, that allows the rest of the code to access the script as a stream of tokens. Section 4.3 describes the implementation of the rules, including common sub-expression elimination (Section 4.3.1) and encoding of condition parts as abstract stack machine processes (Section 4.3.2), and Section 4.4 describes the implementation of the variables, including parsing of single subset shapes and subset covers. The other classes are used by Vars and Rules, and are described in the relevant sections.

4.1 Fido: The Top Level Object

Fido implements the interface that is available to the rest of the control system. This interface consists of methods (described in Chapter 3, Figure 3.1) for input and output variable declaration (variableIn & variableOut), input and
Figure 4.1: The structure of the tool’s higher level objects.

\[
\text{Show} \quad ::= \text{'show'} \ ["" \text{DOSFileName} \ ""]
\]

\[
\text{Monitor} \quad ::= \text{'monitor'} \ ["" \text{skip} \] \text{Integer} \ ["" \text{DOSFileName} \ ""]
\]

\[
\text{Enumeration} \quad ::= \text{'enum'} \ \{ \text{DecList} \}
\]

\[
\text{DecList} \quad ::= \text{DecTerm} \ [\ ', \ ', \text{DecTerm}]
\]

\[
\text{DecTerm} \quad ::= \text{Symbol}
\]

\[
\text{DecTerm} \quad ::= \text{Symbol} \ ' = ' \text{Integer}
\]

where:

- \text{DOSFileName} is a string of 8 or fewer characters,
- \text{Integer} is an integer, and
- \text{Symbol} is a string of characters.

Figure 4.2: Syntax for symbolic integer constants and debugging key words.

output variable moving (\text{moveVarIn} \& \text{moveVarOut}), compilation (\text{compile}) and inference (\text{infer}). Calls to the member functions for variable declaration and movement are passed on to \text{Vars} with little alteration, and \text{Fido::infer} merely calls \text{Vars::fuzzify}, \text{Rules::infer} and \text{Vars::defuzzify}. \text{Fido::compile} has a significant top-level role in parsing the script file that is described in this section.

The shell supports optional fuzzy definition verification and inference monitoring. These aids are requested using the key words \text{show} and \text{monitor} in the script. The syntax is shown in Figure 4.2. The relevant data are written to files with the suffixes \text{sho} and \text{mon} respectively. By default the script name is used as
the first part of their names, but an alternative can be supplied in quotes after the key word. An optional integer after monitor specifies the number of initial steps to be skipped and not monitored.

Definition verification takes the form of a de-compilation of the definitions back into script form, and listings of the sub-expression index and rule condition evaluation programs. This is performed after parsing, whereas monitoring occurs while the controller is running. At each time step the values of the input variables and the membership values of their subsets are listed, then the activations of the rules, and then the membership values of the output variable subsets and the values of the output variables themselves. I found monitoring invaluable for debugging both the inference mechanism itself and controllers.

Defs provides Fido with support for the definition of symbolic names for vector variable indices, in a style similar to the C enumerated type (Figure 4.2). The symbols are given incremental values starting from zero, unless they are explicitly assigned a value, in which case subsequent symbols increment from the new value.

Figure 4.3 shows the top-level parsing loop implemented in Fido. Before entering the loop, the given name for the script file is checked and the user prompted for one if it has not been supplied (1-2). The script file is then opened if it exists (3). If it does not exist, it is created, the control variable declarations written to it, and the program exits (4-5).

Within the loop (7-13), control is passed to the appropriate object for further parsing according to the type of the token read by the lexical analyser, Script. The program exits with an error message if a misplaced token is found (14). After the loop, the results of the compilation are printed out in script file format, if required (16-17). The script object is no longer necessary and it is deleted (18). Redundant variables are also stripped (19), and the rule parse trees are encoded as abstract stack machine programs (20). The programs are appended to the show file, if present (21-22).
int Fido::compile( const char *scriptFileNames )
{
    if( scriptFileNames==0 )
        get file name from user;
    if( named script file not found ) {
        create file and write variable declarations;
        return 0;
    }
    for( each token taken from Script ) {
        switch( token->type ) {
            case Script::Word::Label: compile variable; break;
            case If: compile rule; break;
            case Enum: compile enumeration; break;
            case Monitor: set to monitor; break;
            case Show: set to show; break;
            default: report misplaced token and exit;
        }
    }
    if( set to show )
        write result of compilation;
    delete script;
    strip unused variables and subsets;
    encode rules as programs;
    if( set to show )
        write programs;
    return 1;
}

Figure 4.3: Fido's top level of parsing.

4.2 Script: The Lexical Analyzer

Script is a lexical analyzer that converts the script file to a stream of tokens. An instance of the class Token has an accessible type, and has a numeric or string value, if appropriate. It also records its file name, line and position, and can be called on to report an error at its location, exiting if necessary. The file name is used to distinguish script instances in systems that use multiple instances of the shell.

The type of a token records whether it represents the end of file, a user-defined symbol (label), or a number, or else which of the reserved symbols it represents. Script is not tied to the shell, but can be initialized with a string
4.2. SCRIPT: THE LEXICAL ANALYZER

```cpp
Script::Token* Script::get()
{
    Strip white-space & comments, maintaining a line count;
    token = new Token(fileName,lineCout,++tokenCount);
    if( end of script file ) {
        token->type= Token::Eof;
        return token;
    }
    for(;;) {
        get next character;
        switch( character type ) {
        case Alphabetic:
            if( token->type== new token )
                token->type= Token::Label;
            store character in symbol table;
            break;
        case Punctuation:
            if( token->type==Token::Label )
                if( sign ) goto End; // Mid-token?
                store in symbol table; // Add digit
            else
                get next char;
            goto End; // End token
        case DigitOrSign:
            if( token->type==Token::Label ) { // Mid-token?
                if( sign ) goto End; // End token
                store in symbol table; // Add digit
            } else
                read number;
        case Space: case Eol: case Eof: goto End;
        }
    }
}
```

Figure 4.4: Script's lexical analysis, part 1.

of reserved symbols and a string of punctuation characters for any language. Reserved status is a semantic issue, and punctuation status a syntactic issue. Each of the reserved symbols is assigned a unique positive value which becomes the token type, while the other token types are negative. Punctuation characters are characters that form tokens by themselves, irrespective of white-space, and a punctuation character may or may not be a reserved symbol.

Script uses a symbol table to hold the string values (symbols) for tokens. The symbol table implements the distinction between reserved symbols and user
defined labels, by maintaining a separate index for each. A reserved symbol’s index value is its type.

Figure 4.4 and Figure 4.5 show the function, Script::get, that retrieves the next token. Script::get ignores white-space and comments, and keeps a record of the current line and token number in the new token (1-2). It determines whether the file has finished, and if so it quits, returning an end of file token (3-5). Otherwise, it enters the loop to convert a series of characters, terminated by white-space, punctuation or end of file, into a token (7-30).

If the new character is alphabetic then it is added to the current label, and, if it is the first character of the current token then the token is marked as a label (10-13). If the new character is punctuation then the current token is terminated, or, if it is the first character of a new token then this token becomes the punctuation character (15-21). If the new character is a digit it is added to the current label. A sign terminates the current token, and a sign or an initial digit start a number (22-27). The details of number parsing are not shown. White-space or end of file end the current token (29).
4.3 RULES: THE FUZZY RULES

After the loop, the last character is put back (33), and the token as a whole is processed (34-44). If it has been marked as a label then it is looked up to see if it is one of the reserved symbols, and its type set accordingly (35-40). If it is a number then its calculated value is assigned to it (42-43).

4.3 Rules: The Fuzzy Rules

Rules is a list of rules. A call to Rules::compile creates a new rule and compiles it from the script. A call to Rules::infer cycles through all the rules, applying each.

Compilation of a rule requires the compilation of the conditional expression and the consequent list. The consequent list is simply compiled into a list of references to output subset membership values. The conditional expression of a rule is first parsed to a conditional expression tree (CETree). Then, after the script has all been read, the expression trees are encoded into programs for an abstract stack machine. Evaluation of an expression is subsequently achieved by running the program on the stack machine.

The class Process is the implementation of the abstract stack machine, a conceptual computing device. It has a stack where working values are stored, and a vocabulary of operations that can be applied to values at the top of the stack. A stack machine is the ideal choice for the evaluation of the conditional expressions because they are recursive tree structures and a stack machine is a suitable target for a recursive compiler. A stack machine is also hardware independent but very easy to implement, even on an embedded microprocessor.

Figure 4.6 and Figure 4.7 show the top level of parsing for conditional expression trees, the recursive function getTerm. The data type generated, CETree (4), is a recursive structure. It is a node that may represent a variable/subset leaf, or an intermediate negation, conjunction, or reduction of a vector term to a scalar. Intermediate nodes contain pointers to their children.

In getTerm, any negations are counted first, modulo 1, to obtain the negation status of the term (1-3). Then, if a left parenthesis is found, the sub-expression
CETree* Rule::getTerm( int neg_status )
{
    int not= 0;
    while( 'not' read )
        not= !not;
    CETree *nl;
    switch( token->tokenType ){
        case Fido::LeftParenthesis:
            skip left parenthesis;
            nl= getTerm(neg_status'not);
            check for right parenthesis;
            break;
        case Script::Word::Label:
            parse variable/subset pair;
            not'' = negation within pair;
            nl= new parsed pair;
            break;
        default:
            report variable name omitted;
            } 
    if( not )
        insert negation node;

Figure 4.6: Recursive parsing of rule conditional expressions: the first term.

is parsed recursively (6-9). The negation status (neg_status) is passed down the tree (8) so that, if a vector term requires reduction to a scalar, then the appropriate operator (max or min) can be used (as described in Chapter 3, Section 3.3.6). If a left parenthesis is not found, then a variable name is expected and the variable subset pair is parsed (12-15). If negations occur within the pair (e.g., Bearing is not West) then these are added, modulo 1, to any current negation (14), and if the sum of the negations is 1 then a negation node is inserted above the node representing the term (21-22). Such internal nots affect only the leaf node, and are not added to the negation status.

A term may be followed by a conjunction and a second term. Figure 4.7 shows the parsing of this second part. First the conjunction is identified (24-28), and if none is present then the first term is returned (29). The second term is parsed recursively (31). If the conjunction had a 1 suffix then both terms are reduced to scalar (32-33). If the two arities differ then either term that is a vector is reduced
4.3. Rules: The Fuzzy Rules

```c
23 Process::Op::Code conj;
24 switch( token->tokenType ) {
25     case Fido::And: conj= Process::Op::And; break;
26     case Fido::And1: conj= Process::Op::And1; break;
27     case Fido::Or: conj= Process::Op::Or; break;
28     case Fido::Or1: conj= Process::Op::Or1; break;
29     default: return n1; // No 2nd term
30 }
31 CETree *n2= getTerm(neg_status); // Parse 2nd term
32 if( conjunction followed by 1 )
33     reduce n1 & n2 to scalar;
34 if( n1->arity!=n2->arity ) {
35     set arities of n1 & n2 to 1;
36     report non-uniform arity;
37 }
38     link nodes with a conjunctive node;
39     return conjunctive node;
}
```

Figure 4.7: Recursive parsing of rule conditional expressions: the second term.

to a scalar, and a warning is issued (34-36). Finally, the appropriate conjunctive node is inserted to link the two terms, and is returned (38-39).

4.3.1 Common Sub-Expression Elimination

The final encoding of the expression trees is held over until after the script is fully processed, to allow for common sub-expression identification using an expression index. The shell eliminates common sub-expressions to allow the script author to make multiple references to complex combinations of terms at no more cost than a single reference. Thus these common complex expressions are effectively author-defined higher-level variables. This technique is very suitable for the free-form rules used by this shell. It has not previously been applied by fuzzy inference development tools because of the traditional emphasis on matrix representation.

A conditional expression tree that consists of more than a single subset load is called a complex expression tree (CET). Figure 4.8 shows a set of three rules, and the table lists all the complex expressions present. During parsing, a newly parsed CET is looked-up in an index of previously parsed CETs to see if is
CHAPTER 4. THE SHELL: IMPLEMENTATION

Rules:
Rule1: if ( (A and B) or (C and D) ) then ...
Rule2: if ( (A and B) or (C and D) ) or E then ...
Rule3: if (A and B) or E then ...

<table>
<thead>
<tr>
<th>Complex Expression Tree Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex Expression</td>
</tr>
<tr>
<td>A and B</td>
</tr>
<tr>
<td>C and D</td>
</tr>
<tr>
<td>(A and B) or E</td>
</tr>
<tr>
<td>(A and B) or (C and D)</td>
</tr>
<tr>
<td>(A and B) or (C and D) or E</td>
</tr>
</tbody>
</table>

Figure 4.8: The extraction of common complex expression trees from three rules.

Programs:
CCET1: load A; push; load B; and;
CCET2: load CCET1; push; load C; push; load D; and; or;
Rule1: load CCET2;
Rule2: load CCET2; push; load E; or;
Rule3: load CCET1; push; load E; or;

Figure 4.9: The evaluation order of common complex expression trees.

already present. If no match is found, the CET is added to the index, and the same treatment applied recursively to its complex sub-trees. Differing orderings of terms in an expression can be equivalent, but no attempt is made to identify duplicates by re-ordering the tree. It is left to the script author to ensure that repeated expressions have the same order.

Each CET in the index has an associated instance count. If a newly parsed CET is found to match one already in the index then it is not added. Instead, it is deleted and replaced by a reference to the CET in the index. The instance count for that CET is incremented. A CET with an instance count greater than one is called a common complex expression tree (CCET). The table in Figure 4.8 represents the complete CET index derived from the three rules.

After parsing is complete, the CETs are encoded into stack machine programs (Figure 4.9). The CCETs in the index are encoded first, and placed in a library.
### Description

<table>
<thead>
<tr>
<th>Description</th>
<th>Node</th>
<th>Stack before</th>
<th>Stack after</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable/subset pair:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eg. <em>Turn is Left</em></td>
<td></td>
<td>-</td>
<td>0.4</td>
</tr>
<tr>
<td>load <em>Turn is Left</em>;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negation:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eg. <em>not A</em></td>
<td></td>
<td>(A)</td>
<td>1 - (A)</td>
</tr>
<tr>
<td>...; <em>not</em></td>
<td></td>
<td>not</td>
<td></td>
</tr>
<tr>
<td>Conjunction:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eg. <em>A and B</em></td>
<td></td>
<td>(B)</td>
<td>min((A,B))</td>
</tr>
<tr>
<td>...; <em>push</em>; ...; <em>and</em></td>
<td></td>
<td>(A)</td>
<td></td>
</tr>
<tr>
<td>Reduce:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(invisible)</td>
<td></td>
<td>(A_{n-1})</td>
<td></td>
</tr>
<tr>
<td>...; <em>max</em>; or</td>
<td></td>
<td>min</td>
<td></td>
</tr>
<tr>
<td>...; <em>min</em></td>
<td></td>
<td>(A_0)</td>
<td>min((A))</td>
</tr>
</tbody>
</table>

Figure 4.10: Abstract stack machine operations.

The other CETs in the index are ignored. Then the expressions representing entire rule conditions are encoded. Each reference to a CCET is replaced by a load of the evaluation result of the CCET program in the library. The expressions in the library are evaluated in order, shortest first, and the expressions representing entire conditions are evaluated last. When evaluated in this order, common sub-trees are always evaluated first, and their results ready for use by the expressions that include them. As an example, Figure 4.9 shows, in order, the programs derived from a rule-set via a CET index. The syntax is explained in the next section.

### 4.3.2 Encoding Conditional Expressions

An abstract stack machine consists of a stack and a repertoire of operations applicable to it. An operation can load a value into the top register, alter the
top register, push a new register on top of the stack, or pop the top register off
the stack. In the Process class used in Fido, expressions from the conditional
expression tree are encoded as the following operations (Figure 4.10):

A variable/subset pair, such as Bearing is Left, is a leaf node on the expres-
sion tree. It is encoded as load, which loads the top register with the
designated subset's current membership value.

Negation is an intermediate node in the expression tree. It is encoded as not,
which subtracts the value of the top register from 1, and leaves the result
in the top register.

A conjunction is a branching intermediate node in the tree. It is encoded as a
push after evaluation of the first term, to provide a new register to evaluate
the second, and and or or after the evaluation of the second term. and
applies min, and or applies max to the two top registers. Both then pop
the top-most register off the stack, and leave the result in the new top.

Reduction is the substitution of the single most significant element (maximum
or minimum) for a whole vector. It is required when the arities of two vector
terms do not match, or the operator has a 1 suffix. It is an intermediate node
on the tree, and is encoded as min or max. These perform the specified
operation on the vector on the top of the stack, and release all but one of
its registers, leaving the result in that register.

The efficiency of the evaluation of boolean rules can be improved by short-
circuiting—evaluation is terminated when further terms will not effect the out-
come. This makes the order of terms in an expression significant. Fuzzy terms
have values intermediate between 0 and 1, so all terms must be considered in
the evaluation. Execution order is therefore not significant, and branches of the
expression tree can be swapped, as shown in Figure 4.11, to keep the deeper
tree on the left, and thus minimize the size of the stack. The space economy
thus provided is not significant at the prototyping stage but would be valuable
if the prototype were compiled for an embedded microprocessor. Again, the use
4.4. VARS: The Fuzzy Variables

CCET2: C1 or (C and D)  [where C1 is the result of CCET1, (A and B)]

\[
\begin{align*}
\text{or} & \quad \text{C1; push; C; push; D; and; or; C1+(C-D)} \\
\text{and} & \quad \text{C1} \\
\text{C and} & \quad \text{C} \\
\text{D} & \quad \text{D} \\
\end{align*}
\]

Figure 4.11: Stack register usage minimization.

of this simple optimization in the tool is novel because the emphasis on a matrix representation has previously prevented its application to fuzzy rule sets.

4.4 Vars: The Fuzzy Variables

Vars manages the variables. Vars::variable creates a new input or output variable during program control variable declaration. Vars::compile retrieves an input or output variable that matches the script declaration, or creates a new intermediate variable, and calls Var::compile, to compile the subsets from the script. When the parsing is finished, Vars splits the variables into three lists, input, output and intermediate, ready for processing, and strips out those variables and subsets that are not referenced by any rule so that they are not processed during inference.

4.4.1 Individual Subset Shapes

The shape of an individual subset is compiled at the level of the Subset class, as shown in Figure 4.12. The vertex values are first read into the subset's trapezoid data structure. Each vertex is checked against the bounds, if given (1-2). This checking is performed on all vertices, and will not be mentioned again. The
void Subset::readShape( Script &script, Fido::I0Dir iodir,
                      Nums *bounds )
{
    read vertices from script into trapezoid, checking each against
    the bounds, and assigning the number read to n_vertices;
    switch( n_vertices ) {
        case Line:
            trapezoid[1] = trapezoid[0];
            break;
        case Rectangle:
            trapezoid[1] = trapezoid[0];
            break;
        case Triangle:
            trapezoid[3] = trapezoid[2];
            trapezoid[2] = trapezoid[1];
            break;
    }
    if( script.next.tokenType==Fido::Weight ) {
        if( iodir==Fido::In )
            report weighting of an input subset and exit;
        if( n_vertices>1 )
            report weighted multi-vertex subset and exit;
        weight = script.next.value;
    }
}

Figure 4.12: Parsing an individual subset shape.

number of vertices is counted and used to determine the shape represented (3). The values read are spread symmetrically through the trapezoid data structure so that fuzzification will be achieved correctly (4-14). The last section of the code (17-22) checks for a weight value, and, if one is given, uses it to replace the default output subset weight of one.

4.4.2 Subset Covers

Subset covers are compiled at the level of the Var class, as shown in Figure 4.13. Var::readCover is called when an opening square bracket is found. It reads the leading one or two vertices and the first subset name, and creates the first subset
void Var::readCover( Script &script, Num::Type type )
{
    Script::Token *vertex[2];
    vertex[0] = script.token;
    if( script.next.type==Script::Token::Number )
        vertex[1] = script.token;
    else
        vertex[1] = vertex[0];
    add a new subset named by the current token;
    switch( script.next.type ) {
    case Script::Token::Label:
        readNames(script,vertex,type);
        break;
    case Script::Token::Number:
        if( vertex[0]!=vertex[1] )
            readShapes(script,vertex,type);
        else
            report 'second initial subset point expected' and exit;
        break;
    default:
        report 'subset name or vertex expected' and exit;
    }
    if( script.next.type==Script::Token::Number )
        adjust subsets according to retraction value;
}

Figure 4.13: Parsing a subset cover.

readCover then determines whether a fully regular cover, (represented by a NameList (Figure 3.5, Chapter 3)), or a semi-regular cover (represented by a ShapeList) follows, and calls the appropriate function to parse it (8-19). A NameList has no interspersed vertices, and is detected if another subset name (Label) immediately follows the first (9). A ShapeList is a list of subset names with interspersed vertex values, and is detected if a vertex (Number) follows the first subset name (12). If a ShapeList was detected, then the leading vertices are checked to confirm that there were two of them because a ShapeList cannot begin with a single vertex (13-16). Finally, after the list has been parsed, the trailing optional subset toe retraction value is read and used to adjust the toes of all the subsets (22-23).
CHAPTER 4. THE SHELL: IMPLEMENTATION

Figure 4.14: The parts of fully regular cover subsets.

**Fully Regular Covers**

A fully regular cover is represented by a *NameList*. A *NameList* is parsed by reading the subset names, creating a subset for each, and dividing the cover range regularly between them. Figure 4.14 shows the different types of regular-cover subset, and their parts. A *bounding* subset lies against each of the bounds of the cover, and *intermediate* subsets lie between. A is a *sloped* bounding subset, specified by the single bound vertex, 0. It is identical to an internal subset, such as B. C is a *vertical* bounding subset, specified by the pair of bound vertices, 42.5 42.5. Every subset has *sloping* parts (marked a), but a vertical bounding subset has only one. If the subsets have flat peaks, as shown, then each has a *peak* part (marked b). A vertical bounding subset is exactly half an internal subset, and has a half-sized peak part (marked c).

A regular cover has a uniform spacing between the subsets across the range, and the same inter-subset spacing separates the sloped bounding subsets from the range boundary. The total number of these spaces is therefore the number of subsets plus one. Vertical-sided bounding subsets alter this arrangement because the ‘centre’ of a vertical bounding subset is on the bound (e.g., C). The number of spaces is therefore reduced by one for each vertical bounding subset. The space count is calculated as follows,

$$|spaces| = |subsets| + 1 - |vertical\ bounding\ subsets|$$

and for the cover shown in Figure 4.14, for example, the number of spaces is 3 +
int Var::readShapes(Script &script, Script::Token **t,
    Num::Type type )
{
    int first= 0, n_subsets= 1;
    Subset *s= subsets.top;
    for(;;) {
        check t[first] <= t[!first];
        s->trapezoid[0]= t[first]->value;
        s->trapezoid[1]= t[!first]->value;
        delete t[first];
        t[first]= next script value;
        if( token.type==Script::Token::Number ) {
            check t[!first] <= t[first];
            delete t[!first];
            t[!first]= next script value;
        } else
            first= !first;
        s->trapezoid[2]= t[first]->value;
        s->trapezoid[3]= t[!first]->value;
        if( token.type!=Script::Token::Label )
            ++n_subsets;
        s= new Subset(token,type);
        subsets << s;
    }
    check t[first] <= t[!first];
    check for closing bracket;
    return n_subsets;
}

Figure 4.15: Parsing a semi-regular cover ShapeList.

1 — 1, or 3.

The spacing is calculated roughly by dividing the range by the number of spaces. If the peak width value is given as a percentage then the actual peak width is calculated as a percentage of this approximate spacing.

In general, the spacing is made up of the peak width, \( b \) and the slope width, \( a \). In the case of sloped bounding subsets, however, the spacing consists of the slope width and only half the peak width (e.g., 0 to the centre of \( A \) at 12.5 in Figure 4.14). The global slope width must be increased slightly to make up for the shortage of half a peak width for each sloped bounding subset. The slope
width calculation is therefore as follows,

\[
slope\_width = \frac{range + peak\_width/2 \times |sloped\_bounding\_subsets|}{|spaces|} - peak\_width
\]

and for the cover shown in Figure 4.14 the slope width is \((42.5 + (5/2) \times 1)/3 - 5\), or 10.

There is an extra wrinkle in the computation for integer variables: the division of the range by the spaces may leave a remainder. The cover is made symmetrical by dividing the remainder between the two bounding subsets.

**Semi-Regular Covers**

A semi-regular cover is represented by a *ShapeList*. A *ShapeList* is parsed by the function `Var::readShapes`, shown in Figure 4.15. On entry to this function, `readCover` has already read the two leading vertex value tokens and the first subset name, and created the first subset. The vertex value tokens are available to `readShapes` as `t[0]` and `t[1]`, and the new named subset as `subset.top`. The main parsing loop (3-22) is entered with `first` indexing the first vertex value token, and `s` referencing the subset (1-2). The parsing state is as follows,

\[
[ t_1 \ p_1 \ S \ |\ldots \ t[\text{first}]=t_1, \ t[!\text{first}]=p_1
\]

where `t_1` is the first toe vertex token in the script, `p_1` is the first peak vertex token, `S` is the subset name, `'|` is the limit of reading, and `'|\ldots` is unread script.

The main task of the loop is to correctly handle the two types of shape, trapezium and triangle. For a trapezium, the subset name is followed by two values representing the second peak and the second toe. For a triangle, the subset name is followed by a single value representing the second toe. The second peak must be copied from the first peak.

The loop first checks that the two values are in non-descending order, and stores them in the first toe and peak slots of the trapezoid data structure (4-6). `t[\text{first}]` is now finished with and is replaced by the next value from the script (7-8). The state is now:

\[
[ t_1 \ p_1 \ S \ x |\ldots t[\text{first}]=x, \ t[!\text{first}]=p_1
\]
4.4. VARS: THE FUZZY VARIABLES

If the next value is another vertex then $t[\text{first}]$ is checked against the old $t[\text{!first}]$ for non-descending order, and $t[\text{!first}]$ is replaced by the new value (9-12). The state becomes:

$$[ t_1 \ p \ S \ p_2 \ t_2 \ ... \ \ t[\text{first}]=p_2, \ t[\text{!first}]=t_2$$

If there is no second value, then $t[\text{first}]$ is holding $t_2$ and $t[\text{!first}]$ still holds $p_1$, which must also act as $p_2$. The state becomes:

$$[ t_1 \ p \ S \ t_2 \ ... \ \ t[\text{first}]=t_2, \ t[\text{!first}]=p$$

The tokens are the wrong way round, and this is fixed by negating $\text{first}$ to reverse them (14).

The two token values are then assigned to the second peak and second toe slots of the trapezoid data structure (15-16). If the following token is not a name, then the loop is exited (17-18). Otherwise, the subset count is incremented and a new subset created and pushed on the stack (19-21), and the loop repeats.

After the loop, the last two vertices, which would normally be checked at the start of the loop, are checked for non-descending order (23). Finally, the closing bracket is checked for (24), before the number of subsets created is returned (25).

4.4.3 Fuzzification

$\text{Vars}::\text{fuzzify}$ fuzzifies the input variables and zeroes the output subsets, and the output variables are defuzzified by $\text{Vars}::\text{defuzzify}$. Currently, intermediate variables do not take part in fuzzification and defuzzification, so the same subsets are used in both conditions and consequences. An intermediate variable has space assigned for two sets of subset values (one set for input and one for output) and access is reversed at each time-step to present the previous output as input for the new inference. The alternative of fuzzifying and defuzzifying intermediate variables has not been implemented because although it is more flexible, it is more computationally complex and has not yet been found necessary.

Subset membership ($\text{Mship}$) is a fixed point type, implemented as unsigned integer. The shell supports variables of type $\text{char}$, $\text{int}$, $\text{long}$, $\text{float}$ and $\text{double}$. $\text{Num}$ and $\text{Nums}$ implement individual values and arrays of these types, and the
Mship fuzzify( const Type v, const Type *trap )
{
    if( v<trap[0] || v>trap[3] )
        return 0;
    else if( v<trap[1] )
        return slope(v-trap[0],trap[1]-trap[0]);
    else if( v<=trap[2] )
        return Mship::One;
    else
        return slope(trap[3]-v,trap[3]-trap[2]);
}

Mship slope( Type offset, Type range )
{
    if( offset==0 ) return 0;
    return offset *Mship::One /range;
}

Figure 4.16: Fuzzification of a value with respect to a fuzzy subset.

comparisons, operations and assignments between them that are used during compilation. For efficiency reasons, Var::fuzzify and Var::defuzzify contain separate routines to process each type. Shell-wide support for each type is enabled through conditional compilation, and can easily be turned off individually to eliminate redundant code. Extra code is also eliminated if only a single type is supported. Thus, the shell’s size can be tailored according to the number of different types it supports. Support for unsigned types could also be added relatively easily.

Figure 4.16 shows the fuzzification functions. These match the input value against a subset and return its membership. The commonest case, a variable that does not match the subsets trapezoid at all, is dealt with first (1-2). The value is then classed according to which part of the subset it matches, left slope, peak or right slope (3-8). A membership of one is returned for values that match the peak (5-6), and the slopes are handled by the slope function using the offset of the value from the toe, and the slope range (4, 8).

A vertex that matches a toe exactly is given a membership of zero (9). The
void Var::defuzzify()
{
    enum{ Weight, Leverage };  
    Type accumulated[2]= { 0, 0 };  
    for( each subset s ) {  
        Type activation= s->trapezoid[Subset::Weight]
        *s->mshipOut;  
        accumulated[Weight]+= activation;  
        accumulated[Leverage]+= activation  
        *s->trapezoid[Subset::COG];  
    }  
    if(accumulated[Weight]!=0)
    val= accumulated[Leverage] /accumulated[Weight];
}

Figure 4.17: Defuzzification of an output variable.

membership of other values is calculated from the slope gradient. The multiplication by MShip::One could cause the value to overflow before the division is performed, and in the implementation a more capacious type is therefore introduced to hold the multiplication result.

Figure 4.17 shows the defuzzification functions. The loop accumulates the total weight and leverages of the activated subsets (3-8). These are then used to calculate the output value (10-11).

4.5 Summary

This chapter has covered the following parts of the tool:

Fido is the top level object. It provides the program interface, file access and the top level of parsing control, and debugging aids.

Script presents the script to the rest of the program as a stream of Tokens, each Token having a location for error reporting, a type, and a value if appropriate.

Rules compiles rules, including common sub-expression elimination and encoding of condition parts to abstract stack machine processes, and executes
inference.

`Vars` compiles variables of the required types, strips out subsets and variables that are not used for inference, and performs fuzzification and defuzzification inference processing.

The tool works very flexibly with a control system by integrating into it at the link phase, and then providing an environment in which fuzzy relationships can be prototyped. Prototyping has a rapid turn-around because shell parsing of the script file is fast.

The operation of the tool is demonstrated in the following chapters.
Chapter 5

A Simulation Trial

This chapter demonstrates the use of the shell for controller development. It first shows how free-form rules can be used to re-encode an existing matrix rule-set, and then describes the complete prototyping development of a controller rule-set.

The controlled system is a simulation of a truck backing up to a dock. The shell demonstrates its flexibility by integrating easily with the simulator, as shown in Figure 3.2, even though it was not intended for use with simulations. Rather than ground shell terms to sensors and actions, the grounding code itself simulates the problem.

More complex truck control problems, which include trailers, have been studied with mathematical modelling [SV95], but for this trial I adopt a scenario that is used by Kosko [Kos92b] to demonstrate matrix-based fuzzy control. It consists of backing a four-wheel rigid-body truck up to a position perpendicular to, and against, the middle of a dock.

The choice of this example problem allows a useful comparison with prior fuzzy methods, and demonstrates the ease with which existing matrix-based controllers can be re-implemented with the shell (Section 5.1). I then identify an alternative pair of input control variables to illustrate the prototyping process in the development of a new controller (Section 5.2). This simulation example does not illustrate time limitations or the exploitation of data structure, but these are demonstrated with the robot controller in the next chapter.
5.1 Kosko’s Solution

The control task is to back a truck up to the middle of a dock that forms one side of a rectangular yard. The truck can start from any location and orientation in the yard. The control model for the truck is shown in Figure 5.1, and Figure 5.2 shows the yard. The truck is 5 units long and has a steering lock of $-25^\circ$ to $+25^\circ$. The yard is 40 units deep, and $-50$ to $+50$ units across, with a docking point in the middle at the bottom.

Figure 5.2 also shows Kosko’s choice of control variables. The truck axes are not to scale. Kosko [Kos92b] takes the orientation of the truck ($\theta$), and its lateral displacement ($x$) from the docking point as control inputs, and gives the steering
5.1. KOSKO’S SOLUTION

Figure 5.3: Kosko’s matrix rule set.

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\theta & x & LE & LC & CE & RC & RI \\
\hline
LB & NB^1 & NB^8 & NM^{15} & NM^{22} & NS^{28} & \\
LU & NB^2 & NB^9 & NM^{16} & NS^{23} & PS^{30} & \\
LV & NB^3 & NM^{10} & NS^{17} & PS^{24} & PM^{31} & \\
VE & NM^4 & NM^{11} & ZE^{18} & PM^{25} & PM^{32} & \\
RV & NM^5 & NS^{12} & PS^{19} & PM^{26} & PB^{33} & \\
RU & NS^6 & PS^{13} & PM^{20} & PB^{27} & PB^{34} & \\
RB & PS^7 & PM^{14} & PM^{21} & PB^{28} & PB^{35} & \\
\hline
\end{array}
\]

Figure 5.4: The first two rows of Kosko’s matrix as script rules.

\[
\begin{align*}
\text{if } (\theta \text{ is LB or } \theta \text{ is LU}) \text{ and } (x \text{ is LE or } x \text{ is LC}) & \text{ then } \phi \text{ is NB} \\
\text{if } \theta \text{ is LB and } (x \text{ is CE or } x \text{ is RC}) & \text{ then } \phi \text{ is NM} \\
\text{if } \theta \text{ is LB and } x \text{ is RI} & \text{ then } \phi \text{ is NS} \\
\text{if } \theta \text{ is LU and } x \text{ is CE} & \text{ then } \phi \text{ is NS} \\
\text{if } \theta \text{ is LU and } x \text{ is RC} & \text{ then } \phi \text{ is PS} \\
\end{align*}
\]

angle (\(\phi\)) as output. He develops the matrix-based fuzzy controller shown in Figure 5.3.

A shell script for the controller is created by simply writing the subset vertices from Kosko’s graphs into control subset definitions, and converting the matrix representation of the rule-set into script rules. Rectangular groups of adjacent rules with the same output can be combined using \textit{or}. For example, in the rules for the first two rows of the matrix (shown in Figure 5.4) the upper left block of four rules returning \textit{NB} are represented by the single first script rule.

The subsequent figures show the truck under the control of various controllers. Each figure shows the truck starting from the three points and orientations \(A: (-40, 30, 100^\circ)\), \(B: (-40, 20, 100^\circ)\) and \(C: (0, 20, 180^\circ)\). The first two points were
chosen to illustrate the effect of varying the distance from the dock, and the third to show the effect of starting facing away from the dock.

Figure 5.5 illustrates the limitations of boolean rules. It shows the paths of the truck under the control of a script implementation of Kosko’s controller, but using boolean rather than fuzzy rules. The paths appear adequate initially. The limitations of boolean rules reveal themselves near the end point, when it is necessary to tightly control the truck and draw it to the target. The truck over-shoots the goal, and knees form in the path near the bottom where control switches crudely from one rule to another, trying to bring the truck back to the docking point. Also, close inspection of the upper straight regions of the paths reveals instability there. Successive points step slightly from side to side because the steering flips back and forth at each time-step between left and right full-lock as control switches between opposing rules.
5.2. A PROTOTYPED SOLUTION

Figure 5.6 shows the paths of the truck under the control of the scripted version of Kosko's fuzzy rule set. Kosko does not use the perpendicular distance from the dock as an input, so the truck follows the same path irrespective of its closeness to the dock. Hence the paths from A and B are mostly parallel.

Tracing the paths from A and B, the truck initially turns in a semi-circle to adjust its heading. The control point passes through all the rules in the left column of Figure 5.3 from top to bottom, as the orientation ($\theta$) swings from Left Big towards Right Big, while $x$ remains Left. Finally, the control point is caught between rule 6 and rule 7 because the steering angle ($\phi$) crosses zero to become positive if $\theta$ becomes Right Big. The orientation is thus held between Right Upper and Right Big while $x$ is Left, and the truck heads slightly away from the dock to give itself room to align.

As $x$ decreases to Left Centre, the control point is pushed up to the steering angle zero crossing between rule 12 and rule 13. The truck curves and heads towards the dock as the orientation changes to between Right Upper and Right Vertical. The path from A finally straightens up as the control point passes on to the central rule 18 as $x$ and $\theta$ become Vertical. The path from B demonstrates the limitation imposed by not taking the distance from the dock into account. The truck is too close to the dock, it arrives before $\theta$ becomes Vertical, and does not align properly.

The truck arrives from A and C very slightly to the right of the docking point. This reflects the use of an integer for $x$. The final mis-placement is around 0.4, and is too small to register with the rules.

5.2 A Prototyped Solution

In this section I identify a different pair of input variables for the truck backing-up problem, and use them as the basis for an example derivation of a fuzzy controller from scratch, using the shell. I identify these particular variables in the belief that control inputs should be based on the controlled system's subjective perspective. This perspective is usually more easily obtained from a real-world system, and it
can be the source of substantial economies in problem representation [PS94].

From the truck driver’s perspective, the bearing of the target is easier to perceive than the absolute lateral position of the truck, and I therefore use it as a control input (Figure 5.7). For the same reason, I replace the other input, absolute orientation, with the angle ($\text{Turn}$) of the vehicle’s heading relative to the bearing. The reliance on controller-centric variables increases the number of self-relative subset names over absolute variable names. In this case all the subsets are named in terms of left and right, rather than east and west for example.

The shell is more flexible than the traditional matrix representation, and allows us to develop the rule-set incrementally, adding subsets and rules only as required. We start with the tiny first-cut rule set of Figure 5.9, with the subsets shown in Figure 5.8. It gives the behaviour shown in Figure 5.10. In path A, the controller can initially see the target through the right window ($\text{Turn is Right}$), and so the wheel must be turned to the right ($\text{Steer is Right}$) to align the truck with the bearing. This makes the truck turn and follows the bearing to the
5.2. A PROTOTYPED SOLUTION

Turn is input float
[-180 -180 Left -30 30 Right 180 180]

Steer is output float
.Left is -20
.Right is 20

if Turn is Left then Steer is Left
if Turn is Right then Steer is Right

Figure 5.9: A simple first-cut rule set.

Figure 5.10: The truck’s path using the first-cut rule set.

docking point. It does not align the truck with the dock, however.

To align with the dock, the truck needs to head for the mid-line some distance away from the dock, and then get lined up and approach. To do this it must initially steer across the bearing. The rule set shown in Figure 5.12 achieves this. While the target is to the left of the truck, the control point is held in the zero crossing between the second and third rules, and thus Turn is kept Right, in the crossover region between 0° and 30° in Figure 5.11. This causes the truck to steer clear of the dock, as shown in Figure 5.13, until it approaches the middle and the effect is cancelled by an opposite influence from the Bearing is Left rules.

Unlike Kosko’s controller, this controller is sensitive to the closeness of the truck to the dock because the input variable Bearing is affected by such closeness. The sensitivity can be seen in the difference between the shapes of the paths from A and B. The path from A approaches the dock more quickly because it starts from farther away. This is a sign that we have the beginnings of a superior
controller, but it still has some obvious limitations. The path from $C$ is no good, and although the paths from $A$ and $B$ are better than for the first cut controller, the truck still does become correctly aligned.

Figure 5.15 shows the rule set for the finished controller, and Figure 5.16 shows it as a matrix. An extra pair of subsets have been added to $Bearing$ (Figure 5.14), and a corresponding pair of weaker subsets have been added to $Steer$. These allow closer control of the truck as it approaches the docking point.

Figure 5.17 shows the successful paths. As with Kosko's rule set, the truck initially re-orients itself by traversing the left-hand column of rules from top to bottom, to be held finally between the lower two rules. About two thirds of the way along its path, the lower two rules from the second column cut in, and as they get stronger they turn the truck down towards the docking point. Finally, the control point moves away from the first column and towards the third, reducing the rate of turn as the truck docks. Once again, the truck arrives to the right of the docking point by an amount too small to register with the rules.
5.3. A COMPARISON OF PERFORMANCE

Bearing is input float
\([-90 \text{ Left } -10 \ 10 \text{ Right } 90 \ 90]\)

Turn is input float
\([-180 \text{ Left } -30 \ 0 \text{ Zero } 30 \text{ Right } 180 \ 180]\)

Steer is output float
.Left is \(-20\)
.Right is \(20\)

if Bearing is Left and Turn is Left then Steer is Left
if Bearing is Left and Turn is Zero then Steer is Left
if Bearing is Left and Turn is Right then Steer is Right

if Bearing is Right and Turn is Left then Steer is Left
if Bearing is Right and Turn is Zero then Steer is Right
if Bearing is Right and Turn is Right then Steer is Right

Figure 5.12: A second cut rule set.

Figure 5.13: The truck’s path using the second-cut rule set.

5.3 A Comparison of Performance

The final figures compare multiple paths controlled by Kosko’s rule set and by the prototyped rule set. Only the left side of the yard is shown because the results are symmetrical about the centre line. The initial orientation of the truck is zero (facing the dock) in all cases because the preliminary orienting turn demonstrated in the previous figures holds no further interest.

Paths starting from a matrix of points are compared individually. The matrix points are spaced 6 units apart in each dimension, with the closest points 4
Figure 5.14: The subsets for Bearing in the final cut.

\[
\begin{align*}
\text{Bearing} & \text{ is input float} \\
& \begin{bmatrix} -90 & -90 & -48 & -8 & -4 & 4 & 8 & 48 & 8 & 90 & 90 \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
\text{Turn} & \text{ is input float} \\
& \begin{bmatrix} -180 & -180 & -30 & 0 & 30 & 180 & 180 \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
\text{Steer} & \text{ is output float} \\
\text{Left} & \text{ is } -20 \\
\text{CentreLeft} & \text{ is } -2 \\
\text{CentreRight} & \text{ is } 2 \\
\text{Right} & \text{ is } 20 \\
\text{if } \text{Turn is Left } \text{and Bearing is not Right} & \text{ then Steer is Left} \\
\text{if } \text{Turn is Left } \text{and Bearing is Right} & \text{ then Steer is CentreLeft} \\
\text{if } \text{Turn is Zero } \text{and Bearing is Left} & \text{ then Steer is Left} \\
\text{if } \text{Turn is Zero } \text{and Bearing is CentreLeft} & \text{ then Steer is CentreLeft} \\
\text{if } \text{Turn is Zero } \text{and Bearing is CentreRight} & \text{ then Steer is CentreRight} \\
\text{if } \text{Turn is Zero } \text{and Bearing is Right} & \text{ then Steer is Right} \\
\text{if } \text{Turn is Right } \text{and Bearing is Left} & \text{ then Steer is CentreRight} \\
\text{if } \text{Turn is Right } \text{and Bearing is not Left} & \text{ then Steer is Right}
\end{align*}
\]

Figure 5.15: The final rule-set for truck backing up.

units from the dock. For higher resolution, comparisons are also made between regions of contiguous starting points from which the truck arrives at the dock with particular values for parameters of interest.

Figure 5.18 shows the paths taken from the matrix points, and Figure 5.19 shows the differences in length between corresponding paths (Prototyped minus Kosko). Kosko's paths are shorter (positive values) in the lower three rows, and in a few cases in the middle of the fourth row. However, these paths do not reach
5.3. A COMPARISON OF PERFORMANCE

<table>
<thead>
<tr>
<th>Turn</th>
<th>Bearing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Left</td>
</tr>
<tr>
<td>Left</td>
<td>Left</td>
</tr>
<tr>
<td>Zero</td>
<td>Left</td>
</tr>
<tr>
<td>Right</td>
<td>CentreRight</td>
</tr>
</tbody>
</table>

Figure 5.16: The final rule-set as a matrix.

Figure 5.17: The truck’s path using the final rule set.

Figure 5.18: A comparison of paths.

the docking point and should be ignored.

There is one case at \((-18, 28)\) where Kosko’s controller generates a valid path shorter by 1. Figure 5.20 gives a more complete picture. It shows every starting point leading to a given range of path length difference. Points from which Kosko’s controller does not reach the docking point are omitted. The left yard shows the small, bounded region from which Kosko’s controller generates shorter
paths. For the rest, the prototyped controller produces paths of equal or shorter length, as shown to the right.

Figure 5.21 shows the final $x$ offsets of the truck when it reaches the dock from the matrix points, and Figure 5.22 shows the regions in which the truck reaches the dock with $x$ offset off by 1, by 2 or 3, and by more than 3. In the following discussion the right-most paths that start opposite the docking point are ignored as both controllers have no difficulty maintaining the heading until the dock is reached.

The poor performance caused by the insensitivity of Kosko's controller to distance from the dock is amply demonstrated in these figures. Kosko's paths are substantially inaccurate in final $x$ offset and orientation from most starting points.

### Table

<table>
<thead>
<tr>
<th>Offset</th>
<th>Kosko</th>
<th>Prototyped</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>•</td>
<td>-</td>
</tr>
</tbody>
</table>

### Diagram

![Diagram showing path length differences and regions of path length difference.](image-url)
right out to 20 units from the dock. The way this happens has been described above and can be seen in Figure 5.18.

The prototyped controller performs much better. If the truck is too close to the dock (< 5 units) then it cannot turn in time and hits the dock with a large x offset. If the truck is fairly close to the dock (< 13 units) then it overshoots the docking point by 1 due to the tightness of the final turn. The turn is tightened, and the error exacerbated, if the initial x offset is also small (< 21 units).

Finally, Figure 5.23 shows the final orientations of the truck when it reaches the dock from the matrix points, and Figure 5.24 shows the regions in which the truck reaches the dock with orientation off by 3° . . . 9°, 10° . . . 29°, and ≥ 30°.
They show that apart from the limitation described above, Kosko’s controller is generally more accurate in final orientation than the prototyped controller. In the region above 20, Kosko’s controller makes few orientation errors.

This can be explained in terms of the operation of the controllers. Kosko’s controller brings the truck to the centre-line, minimizing $x$ offset and orientation. Any remaining orientation error is ironed out during the final approach to the dock. The prototyped controller brings the truck to the docking point in a continuous curve. The orientation changes up to the last moment and some error may remain on arrival.

By choosing appropriate control variables we have developed a controller that
is sensitive to the distance of the truck from the dock, and is therefore more often successful in generating a path to the docking point. It also generates shorter paths with better final $x$ offset. There has been some trade-off, however, and the final orientation is not generally as accurate. By prototyping with the shell to build up the controller from a simple rule-set, adding new subsets and rules only as necessary, we have developed a controller that is considerably less complex. This is evident from a comparison of the matrices of Figure 5.3 and Figure 5.16.

5.4 Summary

The simulation example used in this chapter has demonstrated the following:

- The ease with which a traditional matrix-based fuzzy controller can be translated into a shell script.

- The ineffectiveness of plain boolean rules for control.

- The importance to the effectiveness of the controller of the choice of control variables.

- The effectiveness of the incremental rapid prototyping approach to controller development.

This simulation trial has proved the utility of the shell as a platform for rapid prototyping development of fuzzy controllers. The next chapter demonstrates its use with a real robot.
Chapter 6

Shell Supported Sonar-Based Wall Following

This chapter describes an autonomous mobile robot navigation system and the part that I implemented as an example of the usage of the shell in a real-world robot control system. It demonstrates not only that fuzzy rule-based control is effective for robotics, but also that with the help of the shell it is very easy to use.

The shell allowed me to develop a fuzzy controller to control the robot's path relative to a wall directly through a single sensor. This direct servoing from a sensor accords with Brooks' view that sensing and action should be closely tied [Bro86]. It is very effective, and the use of the shell makes it extremely simple to implement. The focus on a single sensor for servo control led to a novel approach to the sonar ring, using it as a virtual rotating sensor, switchable between 16 gross orientations, and with fine adjustment achieved through turning the base.

The next section (Section 6.1) describes a complete proposed autonomous mobile robot navigation system, and where the implemented part fits in the whole. Section 6.2 is a brief overview of sonar sensing and some sonar-based navigation research. Section 6.3 deals with the usage of the sonar ring as a virtual rotating sensor. Section 6.4 describes the fuzzy wall identifier, and Section 6.5 the fuzzy controller for wall tracking. Both were developed using the shell. They are integrated into a system for identifying, approaching and tracking walls in
Section 6.6. The system is extended in Section 6.7 to follow walls round both concave and convex corners, and the results shown in Section 6.8. Section 6.9 compares the system with three reported in the literature.

6.1 A Mobile Robot Navigation System Outline

The problem of autonomous navigation of a mobile robot can be divided into five interrelated parts:

**Localization** The robot should be able to discover and keep track of its current position and orientation, in order to navigate from, or build, a map in which goals, obstacles and paths can be marked. Dead reckoning cannot be relied on for this [LDWC90].

**Mapping** The robot should be able to build a map for itself by exploring its environment, and to use it to locate itself, and paths and goals.

**Path finding** The robot should be able to find on the map one or more paths to a given goal from its current location.

**Control** The robot should be able to follow a chosen path to the goal from its current location. This includes avoiding obstacles.

**User Interaction** The robot should be able to communicate with a user so that goals can be established or changed, and guidance given.

Necessary compromises made in one part of a navigation system have consequences for the entire design [Bro91]. Although the independent development of parts is an essential feature of both research and software design, the specifications for all the parts and how they will work together must therefore be explicit and practicable or the whole effort may be worthless. Brooks [Bro91] has criticised artificial intelligence (AI) for researching robot reasoning and planning in isolation from the practical business of operating in the world. Current robotics research can also be censured, however, for omitting one or more of the above abilities. For example, the recent popularity of the *animat* model (e.g., [MW91])
6.1. A MOBILE ROBOT NAVIGATION SYSTEM OUTLINE

has produced many designs without allowance for user interaction, and hence for the essential setting of goals.

While condemning AI's assumptions about the applicability of its abstract mechanisms to real robots, Brooks' work has provided some justification for making assumptions about the feasibility of grafting on the higher-level capabilities to practical robots that operate successfully in the real world. For this strictly constrained demonstration project, then, I shall briefly outline a rough overall architecture, and then proceed with the implementation of only its lowest level as an example application of the fuzzy controller prototyping shell.

The map is the fundamental data structure for robot navigation, but map building through exploration reveals a deadlock between geometric mapping and localization. The geometric information used to build the map is only as accurate as the robot's localization, and, in the face of the cumulative errors of dead reckoning, localization must itself rely on some form of map. While many approaches to mapping attempt to generate explicit geometric maps from sensor input (e.g., Elfes [Elf87], Zelinsky [Zel91] and McKerrow [McK93] use dead reckoning, Leonard [LDWC90] uses Kalman filters to minimize uncertainty on ad hoc beacons), and then derive the topological information required for path planning from the geometry (e.g., [Zel91]), Kuipers & Byun [KB91] present a method whereby the topology of widely accessible features and identifiable pathways is discovered first through exploration, and geometric properties are added later, as required. This sequence breaks the deadlock between localization and map building, and omits the topology-from-geometry step.

I propose a system in which walls and their junctions act as the pathways and features. For a robot that can follow walls, a wall is a localizing beacon, a map landmark, a path segment, and the basis for communication with a user. It is a localizing beacon because the robot can identify it and control its position relative to it. It is a landmark because it has a length which can be measured and used, in conjunction with its connectivity to other walls, to uniquely identify it on a map. A wall is a path segment because it can be followed to move towards the goal, and finally, it is the basis for communication because it is a feature of
both the robot’s and the user’s perceived world.

I therefore propose the following methods to implement the five parts of the navigation problem:

**Localization** The robot identifies its current wall on the map and keeps track of its position relative to it.

**Mapping** The robot builds a map of the walls that it has experienced, and can re-locate itself relative to the map by following and matching walls.

**Path finding** The robot plans a path made up of wall segments and short trans­sits through free space to short-cut across doorways or corridors.

**Control** The robot follows its chosen path by following walls and making short forays across free space.

**User Interaction** The user draws an initial out-of-scale sketch map of the wall topology, marking the robot’s current position, and its goal. The robot plans a path from this sketch, and updates the dimensions of the walls as it goes. It would also be useful for the user to be able to utter commands such as “Stop!” and *ad hoc* suggestions such as “Not that doorway” [Cha91].

Obstacle avoidance is the most obvious weakness in the above proposal. The robot’s perception of a wall as a continuous entity is interrupted by an obstacle’s short-term presence and its own obstacle avoidance behaviour.

For this thesis, only the lowest level capability is implemented. This consists of identifying a wall, approaching and tracking along it, and following it around corners. The fuzzy controller shell is particularly useful for this low-level control, and is used to implement the identification and tracking abilities. It might also have a role in arbitrating between the wall following behaviour and other behaviours such as obstacle avoidance in a complete implementation.

6.2 The Use of Sonar for Mobile Robotics

Sonar sensors have been popular in robotics for ten years (e.g., [Mil84]), and interest in the ring configuration is nearly as old (e.g., [Wal87]). Sonar sensors
6.2. THE USE OF SONAR FOR MOBILE ROBOTICS

Figure 6.1: The shape of the beam spread of a Polaroid sensor.

are simple, cheap and robust, and they generate a modest volume of useful data. We shall consider the popular Polaroid sensor used in the experimental work for this thesis; it is typical, and widely used. Descriptions are numerous in the literature (e.g., [KS87, MH90, Leo90]), and I give only a brief outline here to reveal the limitations of sonar sensing that must be addressed by any sensing strategy. I then note some of the major contributions to sonar-based navigation.

6.2.1 The Sonar Beam

The sensor emits an ultrasonic chirp—a short pulse of ultrasound—of approximately constant amplitude and frequency. The time till the first echo returns (the time of flight) is used to calculate the distance to an obstacle. The chirp is emitted as a beam taking the form of an axial main lobe with small symmetric side lobes (Figure 6.1). It is tempting to approximate this to a single conical beam (e.g., [Dru85]), but in some circumstances the side lobes must also be taken into account. The main lobe has a characteristic angle of divergence, the beam angle, that depends on the frequency and radius of the sensor. For the Polaroid sensor it is measured from the axis to the angle at which the amplitude has dropped to −30 db, about 12.5°. An object within this cone may—or may not—return an
echo to the sensor, as may objects within the angular ranges of the side lobes.

### 6.2.2 The Echo

The strongest echoes are the result of specular reflection from plane surfaces, and indoor environments are amply supplied with these in the form of walls. The angle between a plane surface and the sensor is critical, however, because the beam may be reflected away from the sensor, and the echo returned via some farther obstacle, if at all. Thus, a sonar sensor used indoors is like a torch in a hall of mirrors [Bro85]. A concave corner, for example, cannot be distinguished from a plane wall by a single sensor because the beam returns via reflections from both walls [KD86].

For the echo to return directly, the normal to the plane must be within the beam angle or one of the side lobes. Figure 6.2 shows specular reflection of the extremities of the main lobe (dotted lines) and the range measured (solid line). In a, the sensor is orthogonal to the wall, and in b, it is at an angle smaller than the beam angle. In c, the sensor is at an angle larger than the beam angle, and will obtain an echo only if some farther obstacle returns it.

The echo is not returned along the line of sight of the sensor unless it is orthogonal to the wall. Specular reflection has equal angles of incidence and reflection, and, for an echo to return to the source, these angles must be 90°. Thus, for pure specular reflection from a plane, the range returned is the distance to the point on the plane from which the normal passes through the sensor [Leo90].
6.2. THE USE OF SONAR FOR MOBILE ROBOTICS

(Figure 6.2b).

An echo is also returned from features of small cross-section such as the apex of a corner (edge) and curved obstacles, but these returns are relatively weak. Kuc & Siegel [KS87] show that the return from an edge is a cylindrical wave-front and decays with the square of the range.

6.2.3 Detection

The gain of the echo detection amplifier is increased with time to compensate for the attenuation due to the solid angle divergence of the beam. An echo is recognized if it exceeds a threshold value, and the maximum range of a sensor is reached when the gain becomes so high that noise may exceed the threshold. An echo, such as that from an edge, that has been very strongly attenuated will not be adequately compensated, and will only be detectable at close range.

Whereas a strong echo exceeds the threshold with its first cycle, a weak echo commonly achieves this only after a number of cycles. Weak echoes, such as those from edges or planes at a large angle, therefore yield erroneously extended range readings [KS87].

6.2.4 Angular Resolution

The sonar sensor’s coarse angular resolution is the key to its economy of data. Any perceptible object within twice the beam angle returns an echo, and only the first is recorded. This makes sonar appear ideal for obstacle avoidance, although its blindness to plane walls at certain angles undermines its utility [Kuc90, TS94]. The down-side to its insensitivity to angle is that target bearing is one of the most useful pieces of information.

6.2.5 Sensing Time

The time a sensor waits before declaring that no obstacle is perceptible can be set by the user. A sonar chirp travels at about 343.6 m/s. in air at 20°C, and tens of milliseconds are needed for detection over a reasonable range in an indoor
environment. Moreover, a single sensor reading is of little use, and a number of readings are usually interpreted together. Chirps interfere with each other, so sensors must be fired separately, although Borenstein & Koren [BK92] have developed a method allowing up to four sensors to be fired simultaneously, and Hanebeck & Schmidt [HS94] a method for interleaving readings. A simpler alternative approach taken by Ando & Yuta [AY95] extends the principle of using only the first echo to the entire sensor set. They fire multiple sensors simultaneously and ignore all but the first return. This approach makes the sensors blind to almost all the objects in its environment but is very suitable when that is the only object of interest, as is the case with obstacle avoidance and wall following.

6.2.6 Sensor Fusion

One way to deal with the shortcomings of sonar is to use it in combination with other sensing media. Flynn [Fly88] proposed fusing the input from sonar and infra-red sensors to minimize the limitations of both. Maeyama et al. [MOY94] built a specialized tree detecting sensor for mobile robot navigation from a sonar sensor and a video camera. Akbarally & Kleeman [AK95] are developing a more general-purpose sensor based on a sonar array and a video camera.

6.2.7 Sonar-Based Navigation

Navigation based on sonar sensing generally requires the representation of the uncertainty inherent in the data. The research can be divided into two groups: grid-based, and geometric methods.

**Grids-Based Methods** Moravec & Elfes [ME86, Elf87] developed a method of environment mapping using sonar returns to create a grid-based map made up of occupancy probabilities. Borenstein & Koren [BK91] developed a fast extension of this method suitable for in-motion obstacle avoidance. Zelinsky [Zel91] also uses sonar to create grid-based maps. Grids are excellent for path planning (e.g., [Zel91, ZY93]), and Scheding et al. [SNP95] and Oriolo et al. [OVU95] use certainty grids for path planning. Grids are not well suited to robot localization
6.3. A NOVEL USAGE OF THE SONAR RING

however [DW95a], and much work has focussed on geometric models.

Geometric Models  Drumheller [Dru87] developed an algorithm to match the
return set from a complete sweep of a sonar ring to a given environment map
stored as line segments. Kuc & Siegel [KS87] developed an accurate geometric
model of the sonar sensing process, and a simulation based on this model gave
results very similar to those generated by a real sensor. They also identified and
characterized sonar-specific environmental features: plane surface, edges, and
concave corners. In our own lab., McKerrow [McK93] built on Kuc & Siegel’s
work and used sonar to generate a map made up of line segments.

Hallam [Hal84] proposed a method of continuous underwater localization using
ad hoc beacons and Kalman filters to minimize sensor data uncertainty. Durrant-
Whyte [LDWC90, DW95b, SSDW95] has promoted the use of Kalman filters
for terrestrial robot localization, often with sonar. Leonard [Leo90] improved
Kuc & Siegel’s geometric model and set of sonar-specific environmental features,
and used the features as beacons for Kalman filter based localization, and map
making. His work laid the foundations for the practical part of this thesis and has
been exploited for reliable navigation in an industrial setting through the OxNav
[SSDW95] project.

6.3  A Novel Usage of the Sonar Ring

To summarize, the sonar sensor is cheap and simple, and economical in the data
generated, but it is slow, and in many situations returns cannot be taken at
face value. For a wall servoing controller I require a fast response to minimize
the control time-step, and consistent returns. This suggests that the number of
sensors in use should be minimized, and that any sensor used for servoing should
be properly oriented so as to obtain a correct reading. Of particular relevance to
wall tracking is the fact that, assuming the specular reflection typical of indoor
surfaces, only one sensor from a ring of 16 will return a reliable return from a wall
[KS87, Leo90]. I therefore developed a model of the sonar ring as a single virtual
rotatable sensor (VRS) with two levels of control. Coarse control is achieved by
The sensors divide the circumference of the robot into 22.5° sectors. switching the active sensor between actual sensors, and fine control by adjusting the orientation of the robot (adjustment turns).

The sensors divide the circumference of the robot into 16 equal sectors of 22.5°, the sector angle. A sector's angle is its angle from the robot's heading. It is used as a name for the sector, as in Figure 6.3. The robot manoeuvres relative to the wall by turning through multiples of the sector angle (manoeuvre turns), while switching between sensors to compensate for the turn and ensure that the active sensor still faces the wall. The limitation of robot manoeuvrability to turns of multiples of the sensor angle makes its movements jerky at times, but does not limit its behaviour in indoor environments dominated by flat walls and near rectangular corners. The VRS approach also does not necessarily waste the resources of all but one sensor, as it can co-exist with other usage of the ring. In our experiments, for example, the sensor at right angles to the tracked wall that faced the up-coming wall ahead was also active. Other sensors could be used as well to improve obstacle avoidance, but at a cost in controller time-step length.

6.4 Fuzzy Wall Identification

I call a sector that contains a perceptible wall a wall sector. To be perceptible, the wall must be not far from orthogonal to the sensor. When the robot first starts, it has no idea which sectors, if any, are wall sectors. Its first task is to identify the most promising wall sector in preparation for tracking. A single return tells nothing about the shape of an obstacle, and adjacent sensors do not detect the same wall, so pairs of sensors in the ring cannot be used either. To identify a
Range is 16 input int
.Near is 0 0 2000

ddRange is 16 input int
.Flat is 0 0 200
.VeryFlat is 0 0 50

Sector is 16 output int
.Useless is 0
.Wall is 500
.StronglyWall is 1000

if Range is Near and ddRange is Flat then Sector is Wall and Sector is not Useless
if ddRange is VeryFlat then Sector is StronglyWall

Figure 6.4: The wall identification script.

wall, the robot must move and compare the returns from a given sensor over time [Zel91, McK93].

Potential wall sectors are identified by moving the robot forward at constant velocity, and firing a set of three complete scans of the sonar ring. A good potential wall sector is one that generates a set of returns that are of short range and are colinear (i.e., lie on a line). Short ranges are preferable both because a nearby wall will require a shorter approach, and because they are unlikely to be the result of multiple reflections. Colinear returns are preferred because it is likely that they come from a continuous plane surface, a wall, and unlikely that they are the result of multiple reflection because such returns tend to be inconsistent. The most colinear set of returns is the set with the smallest second differential:

\[
\frac{(R_3 - R_2)}{T_2} - \frac{(R_2 - R_1)}{T_1},
\]

where the \(R_i\) are range values and the \(T_i\) are intervals between returns.

The shell script for wall identification is shown in Figure 6.4. The two input variables take the range (Range) and the double differential of range (ddRange), and the output variable is Sector. After inference, the element of Sector with the largest value is the most promising wall sector. Note that in this case the script does not define a controller, rather it defines a fuzzy wall identification mechanism that the robot can use whenever it is unsure of the whereabouts of
CHAPTER 6. SHELL SUPPORTED SONAR-BASED WALL FOLLOWING

Figure 6.5: The robot’s track as it identifies a wall sector.

I take advantage of the vector facility of the shell to give each variable 16 elements corresponding to the 16 sensors. The moveVar methods allow the large variable arrays to be allocated on the stack only while they are needed. I initially gave each input variable a single wedge-shaped subset, and the output variable two subsets to define a wide arbitrary output range. The shell’s support for rules with negative as well as positive consequents allowed me to use a single rule with a complementary pair of consequent terms. To boost the influence of pronounced colinearity I subsequently added extra subsets and a rule with a strong effect at small (VFlat) ddRange values.

Figure 6.5 shows the robot’s track as it identifies a wall amongst clutter, and Figure 6.6 tables the data for the sectors. Objects directly ahead and behind tend to appear like walls because there is no lateral movement relative to them. In this case the box (sector 0°) is too far away, and the weak return from its edge too erratic, for it to qualify as a wall, and the returns from the rear are worse.
6.5. A FUZZY CONTROLLER FOR WALL TRACKING

6.5 A Fuzzy Controller for Wall Tracking

It is one of the major limitations of sonar sensing that the return from a single sensor does not provide sufficient information to determine the robot's angle with respect to the wall to better than twice the beam angle. The difference between successive returns does provide this information though, and I use it to control the robot's path.

The adjustment turn affects the perceived wall range however, and it tended to obscure the true change of distance in initial trials. To eliminate this error, I developed a preliminary design in which intervals of turning were interleaved with intervals of moving straight and sensing the range difference. Not surprisingly, the robot's movements were abrupt and jerky until it converged to a relatively straight path parallel to the wall. I subsequently found that the simpler approach of ignoring the adjustment turn error works perfectly well provided the amount of adjustment turn in a time-step is kept small by keeping the time-step small.

This is a case in which the idiosyncrasies of the sonar sensor work in our favour. For specular reflection, the range return represents the distance to that point on the wall from which the wall normal intersects the sensor (Figure 6.2b). Changes in the angle of the sensor affect this range less than the line of sight range.

Figure 6.7 shows the script for wall tracking. The controlled variable \( Adjust \)
Closing is 1 input int
  .VNeg is 32768 to -6001
  [-6000 -6000 Neg Pos 6000 6000]
  .VPos is 6001 to 32767

RangeError is 1 input int
  [-50 -50 Negative Positive 50 50]

Adjust is 1 output char
  .VOut is 25
  .Out is 15
  .SmallOut is 16
  .SmallIn is -16
  .In is -15
  .VIn is -25

if Closing is Neg then Adjust is Out
if Closing is Pos then Adjust is In
if Closing is VNeg then Adjust is VOut
if Closing is VPos then Adjust is VIn

if RangeError is Positive
  then Adjust is SmallIn
if RangeError is Negative
  then Adjust is SmallOut

Figure 6.7: The wall tracking script.

is the rate of adjustment turn of the robot, i.e., the radius of curvature of its path. It is supported directly by the hardware. The primary input variable is the closing velocity, or rate of change of range. The interval between readings is not guaranteed constant but this input value is independent of the interval, and is easily obtained by dividing the range difference by the interval. If the closing velocity is outside the normal control range then the rate of turn is increased abruptly to the maximum (VIn/VOut) beyond which the robot might turn too far in a time step to retain the wall within the sector.

The absolute range is also used as an input in order to coerce the robot towards the intended tracking distance. If the absolute range exceeds its thresholds it is ignored for reasons that will be made clear below.
6.6 Wall Approach

We now have mechanisms for identifying a wall and for tracking one. These must be put together, along with an ability to approach a wall. On each side of the robot I associate each sector with a band of wall ranges for which, if it is the wall sector, the robot will move appropriately towards the tracking distance (Figure 6.8). The forward facing sensors are each associated with a band of ranges larger than the tracking distance, one rearward facing sensor on each side with ranges smaller, and the lateral sensors with a narrow band of ranges around the tracking distance itself. If a wall sector is identified 90° left, for example (Figure 6.9), then the sector on the left whose range includes the current wall distance (22.5° in this case) is selected as the desired wall sector. The required turn is then the angular difference between the current wall sector and the desired wall sector. There is no need to make allowance for the angle of the wall since the wall is constrained to be approximately perpendicular to the wall sensor.

The robot travels at a constant speed, and uses predefined turn durations to turn in multiples of the sector angle. Once it is on an approach path, it turns at intervals through single sector angle manoeuvre turn steps. The initial
CHAPTER 6. SHELL SUPPORTED SONAR-BASED WALL FOLLOWING

Figure 6.9: The turn angle is the difference between the current and desired wall sectors.

The manoeuvre turn may be much larger than a single sector angle, though. The method used to identify the wall sector is relatively slow, and it is incompatible with rotation of the robot because it relies on a linear series of returns. The robot is in a position similar to that of an eye-ball: it can compute an estimate of the trajectory required to switch to the desired orientation, but its sensory mode is not appropriate to monitor its movement in real-time. Having calculated the turn required to orient appropriately to the potential wall, the robot, like an eye-ball, performs a ballistic turn without feedback. The turn is only approximate, and the path is re-adjusted after the turn using the wall tracking controller. If the potential wall fails to behave as expected—whether because it was not a wall or because the turn was excessively inaccurate—then the robot repeats the identification sequence. If the failure was the result of inaccuracy in the turn then the subsequent manoeuvre turn will be smaller and less prone to inaccuracy.

This ballistic method is much more effective than a gradual sensor-mediated turn. For example, the next sensor round the ring could be fired repeatedly during the turn and an attempt made to use its returns to determine when it is orthogonal to the wall and the robot had turned through one sector angle. Sonar
sensing gives discrete returns at finite, and quite large, intervals. Thus, such an approach would require a very slow rate of turn to have any precision, and would therefore require the robot to stop. Sonar gives poor angular resolution as mentioned above, and it would also be very difficult to distinguish the correct wall return from an echo returned by the corner itself. After turning, the robot would have to restart, but would be unable to track the wall until its full velocity was restored. Thus, this approach would be slow and ineffective, and would break the sensing mode, which is based on differencing returns at a fixed velocity.

Further, a sonar system cannot control turn rate. The system designer is forced to predefine some sensible rate of turn that will give a sensor the best opportunity to identify the wall. The best rate of turn to chose is one that puts a sensor approximately orthogonal to the wall so that it is in a position to track the wall effectively. This is the rate chosen by the ballistic approach. It maintains sensing accuracy by preserving robot velocity and sensor orthogonality, and keeps the coupling between sensor and actuator as tight as possible by putting a new sensor into position quickly.

The wall tracking controller is designed only to hold the robot's path parallel to the wall, but its functionality can be extended by subtracting the expected closing velocity (at the current velocity and angle to the wall) from the measured closing velocity. This effectively 'fools' the controller that the robot actually is parallel to the wall. This is the reason why the controller ignores the absolute range error when it exceeds its thresholds: the range error is used to control the absolute range from the wall only when the robot really is parallel.

The expected closing velocity, $V^e$, is calculated from the robot's current linear velocity, $V^l$, and the wall sector's angle, $\theta$, thus:

$$V^e = V^l \cos \theta$$

For speed, constants corresponding to the cosines of the sector's angles are stored in a table and the calculation is performed with fixed point arithmetic.

Figure 6.10 shows the wall approach algorithm. The robot gets up to speed and registers that it does not know where the wall is before entering the main loop. Whenever the robot loses track of the wall it uses the wall identification routine
robot.start();
robot.waitUpToSpeed();
adjust := LostTrack;
loop
    if (adjust = LostTrack) then
        wall := robot.identifyWall();
        reorient := robot.checkBand(wall);
        if (reorient ≠ 0) then
            robot.turn(reorient);
            wall+ = reorient;
        adjust := robot.track(wall);
    if (adjust ≠ LostTrack) then
        robot.turnRate(adjust);

Figure 6.10: The wall approach algorithm.

to find a promising wall sector. It then tests the current wall range against the
range bands to determine which sector should be the wall sector, and performs a
manoeuvre turn and updates the wall sector if necessary. The manoeuvre turn is
synchronous—execution awaits completion of the timed turn to ensure that the
wall sector faces the wall. The new wall sensor is then fired to get an initial range
reading.

The robot then tracks the wall. The tracking routine fires the sensor once
only and compares the current range with the previous return. This is usually
the range from the previous tracking cycle, but it may be a return left-over from
wall identification, or the initial sense after a turn. If the range difference is within
limits then the adjustment turn radius is issued to the robot. The adjustment
turn is asynchronous, and execution continues as soon as the command has been
issued.

Figure 6.11 shows a track generated by the robot as it identified, oriented to,
approached and followed a wall at 150 mm/s. The manoeuvre turns can be clearly
seen. The single sector angle turns are a little too strong, and a slight adjustment
back into line can be seen after each. Also visible is the final adjustment to the
exact tracking distance once the robot is parallel to the wall.
6.7 Wall Following

I have established that it is possible to use fuzzy scripts to permit a robot to identify, approach and track a wall. This is not what is generally meant by \textit{wall following}, however, and is not sufficient for the basic behaviour of the proposed navigation system. To follow a wall in the accepted sense, a robot must not be shaken off by corners and doorways, and thus the next step is to extend the control system to handle both walls that appear up ahead and the sudden disappearance of the tracked wall. Neither of these extensions requires the services of the shell, but they are developed for completeness.

6.7.1 Concave Corners

Handling walls that appear ahead, and hence concave corners, is much simpler than might be expected. We already have the entire mechanism for approaching and following a wall identified in any sensor; it remains only to monitor the sensor that faces the up-coming wall, and to switch control input to it as soon as its return crosses the threshold between the $0^\circ$ sector range band and the $22.5^\circ$ sector range band, the \textit{approach threshold} (Figure 6.8). The robot then approaches and follows the new wall in the established manner.

The upcoming wall does not need to be at exactly $90^\circ$ to the current wall, the
same adjustment used to correct ballistic turns can correct for small variations in wall angle. If the angle is too acute the robot will not be able to turn in time, however, and if it is too far from 90° the forward sensor will not be able to see the wall at all. In environments with walls at angles far from 90°, more than one forward facing sensor could be used, though this would slow the rate of sensing considerably. Depending on the approach threshold, there may still be some angles between 90° and 67.5° or 112.5° at which the wall is not perceived. This is a subject for further research.

Looking at Figure 6.8, a new upcoming wall may come within the approach threshold while the robot is still approaching the current wall, monitoring its 67.5° sector for example. Thus, the upcoming wall sector is not necessarily the front sector but the front facing sector that is at 90° to the current wall sector. This happens when the robot travels down a narrow dead-end corridor, for example. The robot never parallels the end wall but breaks off its approach to it to start its approach to the return side-wall.

There remains a question of consistency of the side of the robot that faces the wall. When the upcoming wall sector becomes the wall sector, as the robot turns to align with the new wall, the turn must be such that the wall sector is moved back round the ring towards its old position on the left or on the right. Thus, as the robot approaches a concave corner, the wall sector first jumps 90° forward, and then steps back round the ring towards the same lateral sector.

6.7.2 Convex Corners

Convex corners are more difficult but it would appear to be simple to detect them at least. The wall range should suddenly jump to a much larger value. Unfortunately, in practice, the jump is not always very clean. The wall continues to be detected as long as it remains in the sector, and thus the robot is actually past the corner before it detects it. As the corner reaches the extremity of the sector its return weakens towards an edge echo, and its perceived range may increase (Section 6.2.3). This can trigger a small erratic correction response.

More important is the question of how to respond to the corner, how to nego-
tiate it. The echo from the tracked wall is gone, and the edge echo is somewhere behind. This is a weak echo, but the robot is close and has no difficulty perceiving it. The robot can turn quickly and re-establish contact with it, but, again, it provides insufficient angular information to control the robot's path. I found that the range difference method used to follow walls was not effective in this case, and the robot tended to lose track of the corner.

The solution was to use the *boundary* between perceiving the edge and not perceiving it as the control point. When the tracked wall range suddenly increases, the robot turns sharply and re-establishes contact with the edge. It then stops turning and allows the edge return to be lost again, and then turns to regain it. By repeating this behaviour, it tracks around the corner. The short control-step time makes the switching imperceptible, and this turn is in fact smoother than the wall approach curve.

There are two problems, however. The first is how to end the turn. As the lateral sensor comes closer to orthogonal to the new wall, it takes longer and longer to lose the edge return when it goes straight. A count on the number of consecutive hits without turning is taken and the turn completed when it exceeds a threshold. The second problem is that the robot moves away from the wall as it turns. This is apparent in the tracks shown on the following pages. It is inevitable because of the repeated relaxation of the turn to lose the edge echo. As a result, the robot is farther than the tracking distance from the new wall. This is unfortunate but not serious. The turn is ended by a switch back to the normal approach and track mode. The excess distance is quickly recognized and rectified by the normal approach mechanism.

Figure 6.12 shows the wall following algorithm. The robot now identifies a wall before entering the loop because a sudden jump in wall range is now the trigger for negotiating a convex corner, not for re-identifying the wall. Inside the loop, the robot first checks the ahead-facing sensor's return, and makes it the new active sensor if its range is less than the approach threshold. It then manoeuvres and adjusts as before except that if it has lost track of the wall it commences negotiation of a convex corner.
robot.start();
robot.waitUpToSpeed();
wall := robot.identifyWall();
loop
    if (robot.range(robot.ahead(wall)) < ApproachThreshold) then
        wall := robot.ahead(wall);
        reorient := robot.checkBand(wall);
        if (reorient ≠ 0) then
            robot.turn(reorient);
            wall+ = reorient;
        adjust := robot.track(wall);
        if (adjust = LostTrack) then
            robot.convex();
        else
            robot.turnRate(adjust);

Figure 6.12: The wall following algorithm.

6.8 Results

The control system was tested in a play-pen of about 3 by 3.6 metres. The walls had no special preparation and had many blemishes. The notches between the wall board sections measured about 4 mm across by 2 mm deep. They are significant features for non-orthogonal sonar beams, but invisible to this system. There were also surface-mounted electrical conduits of about 20 mm diameter, which sometimes caused small perturbations. One wall contained a closed doorway with architrave, and another was made of two doors laid on their sides, with a gap of 50 mm or so between them. A convex corner was created by leaning another door against a couple of cardboard boxes in one corner to make the space ‘L’ shaped. The robot travelled at a manoeuvring speed of 150 mm/s and accelerated up to 300 mm/s when ample free-space was perceived ahead.

Figure 6.13 shows the robot’s path through one circuit of the pen. The co-ordinates of each point are derived from the Labmate’s internal dead-reckoning system and become increasingly inaccurate. I have drawn in the approximate positions of the walls relative to their adjacent tracks. The robot starts from the
point marked a, and travels straight up the page to identify a wall. It then turns and approaches the wall, as before. At b the robot aborts that approach and commences its approach on the next wall. It accelerates at c to traverse the straight wall segment at high speed, and decelerates again at d. At e it accelerates, only to find that the wall disappears almost immediately, and it commences a convex turn. The turn ends at f, at which point the robot is too far from the wall due to the relaxation of the turn described above. It adjusts its distance from the wall using the normal approach mechanism until, at g, it starts to approach the next wall. At h it accelerates for the final leg to complete the circuit. The robot was happy to continue around the pen a number of times until I stopped it to prevent
excessive twist in its umbilical cable.

Figure 6.14 shows an anti-clockwise circuit. The stages are much the same, but a longer wall gives the robot the chance to go further at high speed before commencing the convex corner at e.

The method for negotiating convex corners is independent of corner angle, and the robot successfully negotiated corners up to 180° (Figure 6.15). Its handling of concave corners is limited by the angular bounds of the forward facing sensor. Small variations in concave corner angle were handled effectively, larger variations would require the help of the two neighbours of the up-coming wall sensor.
6.9 Comparisons with Other Work

6.9.1 Hanebeck & Schmidt

Hanebeck & Schmidt [HS94] model a ring of 24 sonar sensors as a variety of novel virtual sensors. One of these is a generalization of the Virtual Rotatable Sensor described here. It is a point source sensor that generates a coherent circular wavefront around a large segment of the circumference of the robot by coordinating the emissions of a number of transducers.

Such coordination is not possible with our ring, which is a stock item having only two sets of electronics for the entire ring. This prevents more than two sensors at most being used simultaneously. Also, each set of electronics is associated with a fixed subset of the ring, so the pairings of sensors are also highly constrained.

For simplicity, I restricted the Virtual Rotatable Sensor to a single sensor and, when using another sensor at the same time, did not use it simultaneously. As the sensor pairs are always at 90° it would be possible to divide the ring into
CHAPTER 6. SHELL SUPPORTED SONAR-BASED WALL FOLLOWING

four quadrants with adjacent quadrants controlled by separate electronics. This would allow the current and upcoming wall sensors to be fired simultaneously.

6.9.2 Ando & Yuta

Ando & Yuta’s ring [AY95] has much in common with Hanebeck & Schmidt’s virtual point source sensor. It divides the circumference into sixteen 22.5° segments, like our own, but uses only 12 sensors. It omits two sensors from each side of the rearmost sensor as these are of no use for obstacle avoidance and wall following. The sensors’ beam angle is 50°, so their fields overlap considerably. The sensors are also more independent than ours. They are divided into two sets and each set can be fired all at once. Ando & Yuta are only interested in obstacles within 2 metres of the robot so they allow only 30 ms per firing. This arrangement gives a very fast and somewhat directional sonar buffer around the front of the robot, and Ando & Yuta use it for wall following.

The Yamabico robot has a sophisticated motion control system built-in that allows the specification of straight line and circular arc trajectories in a coordinate system. The robot moves at 300 mm/s and receives a command based on sensing every 30 mm travelled, which specifies a trajectory. This contrasts with my direct control of radius of turn. The Yamabico also stops and spins on the spot rather than turning gradually to negotiate concave corners.

The sonar ring has sufficient sensors with sufficient angular spread to detect walls at almost any angle, but Ando et al. identify situations in which the robot does collide with a wall. This is due to their sensors’ inability to perceive edges even at close range [ATY95], and may be a result of their use of a time adjusted threshold instead of time adjusted gain to compensate for beam spread [OXY95].

Comparing the two systems, both operate autonomously in standard unmodified indoor environments, and both are tricked by certain wall configurations. Ando & Yuta’s has better general obstacle avoidance, but has to stop to negotiate corners. It is also based on more sophisticated hardware.
6.9. COMPARISONS WITH OTHER WORK

6.9.3 Leonard

Leonard [Leo90, LDW91, LCDW92] developed a robot exploration and mapping system. The robot uses a technique called *regions of constant depth* to group and disambiguate the sonar returns, and thus to identify geometric features in the environment. It then uses Kalman filtering to match these with features it has previously identified and marked on a feature map.

A *region of constant depth* (RCD) is a series of adjacent returns that agree to within 1cm. The robot takes a dense scan of its environment (0.588° increments) and accepts only RCDs with an angular size of 10° or more. This guarantees that the returns are generated by the central lobe of the sonar beam, and not by the error-prone weak returns or reflections. As a result, the robot can be confident that the features it detects exist in the environment and are not artifacts of the sensing system, and that they are accurately located.

It takes about 2 minutes to make a complete scan. This is prohibitive for practical purposes but Leonard and Durrant-Whyte propose the use of multiple independent tracking sonars, each focussing on a particular feature, to overcome it. That approach is taken by Stevens, Stevens and Durrant-Whyte [SSDW95], as described in Section 6.9.4. Lee [Lee95] takes a different approach described in Section 6.9.5.

RCDs are used to identify *geometric features* in the environment. Leonard focusses on point and line features. These are matched with previously identified features on a map and used to strengthen the reliability of those features for future matching. The approach can be contrasted with the simple wall tracker described in this thesis. The wall tracker orients the robot to ensure that the sonar returns are generated by the central lobe of the beam. It operates reactively based on the matching of successive returns to extend the hypothesized wall that it is following. The result could be used to generate a map of the walls in a manner similar to that reported by McKerrow [McK93].

Leonard uses Kalman filtering for matching. Like a fuzzy system, a Kalman filter makes an allowance for uncertainty and measurement noise. The fuzzy system described in this thesis uses user-defined subsets that incorporate the
uncertainty. A Kalman filter is more like a self-tuning fuzzy systems (e.g., [Bir93, KB95]), as it adjusts the allowance at each cycle according to the computed uncertainty of the previous result.

6.9.4 Stevens, Stevens & Durrant-Whyte

The OxNav mobile robot system of Stevens et al. [SSDW95] uses four independent rotatable sonar sensors to track previously surveyed sonar features as landmarks for navigation. Each sensor comprises two transducers to allow walls to be distinguished from concave corners [BK90]. By using independently rotatable sensors, OxNav decouples sensor orientation from robot orientation, and thus neither uses robot turns to adjust sensor alignment nor constrains the robot’s manoeuvrability. By focussing each sensor on a feature, OxNav ensures that the return is generated by the central lobe of the beam and avoids the time delays incurred by Leonard in scanning the whole circumference.

The system described in this thesis has some similarity to OxNav. Both systems track features with the main lobe of the sonar beam, and both use specific techniques to deal with uncertainty. OxNav uses Kalman filtering, I use fuzzy inference. Both improve reliability by navigating on the basis of pairs of returns, but OxNav uses a more sophisticated sensor arrangement that allows it to obtain the pair simultaneously.

6.9.5 Lee

In his thesis [Lee95], Lee describes experiments in exploration with a small (300 mm diameter) mobile robot. The model of the sonar sensor Lee uses is relevant to the present work.

The robot uses a single rotatable Polaroid sonar sensor that scans ahead at seven pre-set angles. Thus, it approximates to a ring of seven forward facing sensors. Lee rejects Leonard’s region of constant depth method [Leo90] as too slow. He uses an intelligent interpretation of a sparse set of scans instead. He discards maximum range returns, and groups returns from adjacent scans that agree to within a threshold of 3cm to make up composite readings that include
one or more returns. The angle of a reading is the average angle of the scans that it is composed from, and its angular width is represented by the number of scans included. The result is a method that retains much of the benefits of the region of constant depth approach but is 40 times faster.

Lee requires the processing of returns into readings because the robot uses them to build maps of free-space and features. Readings from consecutive time steps are compared explicitly by Kalman filter based algorithms to identify walls and other features that are recorded on the map. The wall tracker described here is a simpler reactive system that groups successive returns implicitly by assuming that it is following a wall. As long as the returns remains trackable their source constitutes a wall. If the range varies significantly then a bend or convex corner is assumed and appropriate action is taken.

### 6.9.6 Pin & Watanabe

Pin & Watanabe [PW94] present the results of robot control experiments using fuzzy inference hardware and a novel fuzzy behaviourist approach. They consider the use of output subset weight to effect an arbitration between rules or behaviours to be an important element of their approach. Two rules with different inputs variables that affect the same output variable may conflict. Their priority is defined in terms of output weight. If conflict occurs then one outweighs and suppresses the other, just as if they were derived from the same input variable.

Pin & Watanabe describe experiments using the fuzzy behaviourist approach for reactive navigation (i.e. no maps were used or generated). They use an omnidirectional platform fitted with a ring of 24 sonar sensors. The forward facing sensors are divided into three groups covering 75° each, and these are used for the control inputs. The fuzzy inputs are orientation to the goal and object proximity, the outputs are speed and direction.

Pin & Watanabe have developed a number of effective rule-sets for this experimental set up. They use behaviour suppression very effectively to avoid the standard potential field dead-lock situations. The robot will carry on following a barrier past the point that is closest to the goal, and it can follow and then ex-
tricate itself from a dead-end corridor. They note that the system is not immune
to more complex limit cycles, as memory would be required to identify these.

Pin & Watanabe describe one rule-set in detail in the paper. They distinguish
two input types (goal orientation and obstacle proximity) and two output vari­
ables (speed control and turn control). Obstacle proximity has three variables,
proximity ahead, to the left and to the right. Goal orientation has one. Proximity
ahead is related only to speed, and the left and right proximities are related only
to turn. Goal orientation has groups of rules relating it to each output variable.
Consequent weight is used to arbitrate between conflicting responses.

The relationships are very simple. For example, the most complex relate
obstacle proximity on a given side to turn control, and consist of four subsets
(dangerously close, very near, far and very far) with a rule each. The output
subsets decrease in weight so that a response to dangerously close will tend to
out-weigh responses from other inputs.

Pin & Watanabe's controller is similar to the wall tracker in that it uses few
rules (1–4) for each variable. It use four input and two output variables whereas
the wall tracker uses two input and one output variable. Both controllers use
simple output subset weight to arbitrate between the conclusions from different
input variables.

Pin & Watanabe's experiments demonstrate fuzzy reactive navigation, and
corroborate the experimental results of this thesis by showing the effectiveness of
fuzzy inference for such low-level navigation tasks. They note that the definition
of the membership functions is one of the major challenges when implementing a
fuzzy set based approach. Unfortunately, they do not describe the tools they use
to test and refine their designs. The shell described in this thesis would be ideal,
and could be adapted to generate the outputs necessary to configure the VLSI
fuzzy inferencing boards.
6.10 Summary

This has been a long chapter dealing with a substantial piece of experimental research making use of the fuzzy controller shell. It has comprised the following:

- A rough outline of the complete autonomous navigation system of which the implementation described would be part.

- An overview of the principles and limitations of sonar sensing and trends in sonar-based navigation research.

- My novel usage of the sonar ring as a virtual rotatable sensor with fine control achieved by adjusting the orientation of the robot, and coarse control by switching between sensors.

- A fuzzy sonar-based wall identifier.

- A fuzzy controller for sonar-based wall tracking.

- The integration of the above, with a mechanism for wall approach, into a wall finding and tracking system.

- The extension of the above to allow wall following around concave and convex corners.

- Annotated tracks showing that the robot can find its way round a very imperfect room.

- A comparison of the final system with similar systems in the literature.

A single sonar return is a very unreliable datum. This project demonstrated the efficacy of directing the beam at a feature and of using successive pairs of sonar returns for reliable wall tracking.

The shell demonstrated its merits by being easy to integrate into the control system and by making the fuzzy controller and identifier among the simplest parts of the system to develop. Multiple instances of the shell were used, and it demonstrated its ability to perform within tight time constraints. Again, the iterative rapid-prototyping approach led to the development of very simple controllers.
Chapter 7

Conclusion

This project has met the majority of its objectives. A novel tool was created that supports rapid-prototyping development of real-time rule-based controllers in an easy-to-use shell environment. The tool was also tested, and demonstrated its effectiveness.

The optional compilation of working controllers into a form suitable for embedded microprocessors has not been implemented. PC-based controller execution was appropriate for the available hardware, and the controllers were found to execute fast enough on this platform. Compilation is a separate stage whose omission does not affect the rest of the development. Also, it does not promise any significant research issues.

The project involved research in a number of areas, and led to some novel developments. The architecture of the tool as a program object with shell and program interfaces is new. It makes real-time controllers simple to develop and easy to integrate into larger control systems. The shell interface allows a controller to be defined in a script. The program interface allows an application program to initiate fast compilation of scripts, to ground shell symbols to control signals, and to incorporate script-defined inferences into a control loop. Thus the tool readily integrates into a control system, and provides that system with a rapid-prototyping controller development environment. The controllers use fuzzy inference to handle continuous inputs and to generate continuous outputs in real-time.
The tool handles structure in the control-data through vector variables whose arity can be dynamic. This is a novel concept in rule-based systems. These variables permit the tool to be used in situations that could not be addressed with atomic variables. Vector variables can make the control scripts considerably simpler, and allow the tool to improve performance in some cases. The shell syntax also allows fuzzy subsets to be defined in groups that cover subranges of a control variable. This new format reduces redundancy, and thus the potential for error.

The tool was first tested in a control simulation. This trial compared the tool with traditional fuzzy methods exemplified by Kosko’s truck backing up controller. The shell can be used to duplicate an existing fuzzy control matrix, but the trial showed that the rapid-prototyping environment allows a developer to take an incremental approach, which results in very simple but effective controllers.

The tool was then tested in the development of part of a complete autonomous mobile robot navigation system. I chose to implement the lowest level because it embodies the basic real-world competence that makes a robot architecture credible. It also requires real-time control using continuous inputs and outputs, and these are the rare strengths of fuzzy inference.

In this example the tool was used to develop a fuzzy controller that implements a wall following behaviour. The behaviour is based on sonar sensing, which is slow, but the fuzzy controller minimizes the control loop delay by using only a single sensor. The example shows both that fuzzy rule-based control is a very useful technique for robotics and that the tool makes it very easy to use.

The complete navigation system outline and lower level behaviour implementation beg for the further development of higher level control using fuzzy behaviour arbitration. This is an exciting direction for further work.
Bibliography


BIBLIOGRAPHY


