A Transputer-Based Inferencing System Using Fuzzy Logic Concepts - Design and Implementation

by

Richard Scott Bowyer

Thesis submitted for the degree of

Master of Engineering Science

The University of Adelaide

Faculty of Engineering

Department of Electrical and Electronic Engineering

October, 1996

To my loving wife,

Judy

TABLE OF CONTENTS

ABSTRACT

There has been considerable interest in engineering applications of fuzzy logic to data processing by a fuzzy logic inference engine. This research explores the application of the INMOS Transputer to the implementation of a data processing system that uses fuzzy logic concepts, and details the design and implementation of the system. The research addresses the important area of knowledge encapsulation and representation, and the method by which knowledge is pre-processed into a form suitable for evaluation by the fuzzy logic inference engine.

This thesis begins by examining the concepts of fuzzy and explores how they can be applied in a parallel processing domain. This includes the representation of fuzzy rules, and how the rule-base is pre-processed. Some additional aspects are examined and new operators proposed.

An algorithmic structure is developed for the expert system. The inference engine for the system uses fuzzy logic principles, and parallel processing algorithms for inferencing are developed. The processing is performed on the INMOS Transputer, and the inferencing algorithm is realised in the Occam 2 programming language.

The software package that has evolved from this research is collectively called TransFuzien. This software, together with the associated hardware, has been developed to perform rule-based fuzzy logic inferencing. TransFuzien comprises two parts. The first part is written in C++, and runs under Microsoft Windows on a IBM Personal Computer (PC), and the second part, which is written in Occam 2, runs on a Transputer target system. The former part performs the data pre-processing necessary to configure the inference engine that executes on the Transputer target system.

TransFuzien's graphical user interface is described, together with various features of the system, which includes the ability to select various inferencing strategies via the graphical user interface.

Aspects of the processing performance are addressed, and key issues are identified. Specialise electronic hardware has been developed to facilitate data exchange between the expert system and peripheral systems. Finally, a number of case studies are presented that apply the expert system developed herein. The results from these studies show that the objectives of this research have been achieved.

DECLARATION

This thesis contains no material which has been accepted for the award of any other degree or diploma in any University or other tertiary institution. To the best of the author's knowledge and belief, this thesis contains no material that has been previously published or written by another person, except where due reference has been made in the text.

Signature _____

Date 31-10-96

NAME: **COURSE:** M.ENG.SC.....

I give consent to this copy of my thesis, when deposited in the University Libraries, being available for photocopying and loan.

SIGNATURE: ..... **DATE:** ..12-5-97.....

ACKNOWLEDGMENTS

I would like to thank my supervisor Mr. Mike Liebelt (Senior Lecturer) of The University of Adelaide, Department of Electrical and Electronic Engineering, for his supervision, valuable guidance, and encouragement throughout this work. My thanks are also extended to Associate Professor Doug Pucknell for his supervision in the early stages of this work.

I wish to thank Mr. Des Lamb, former Head of the Information and Signal Processing Group, in the Electronic Warfare Division (EWD), Defence Science and Technology Organisation (DSTO). Des has shown a great interest in my work, and has been a constant source of support and encouragement throughout my studies.

I would like to thank Dr. Andrew Kulessa of the DSTO, for many long and enjoyable discussions about this work, and for acting as a sounding board for some of my ideas, and giving me valuable comments on my work. I also thank Andrew for helping in the review process of this thesis and providing comment on my writing style and the technical content.

My thanks are extended also to Malcolm Brown (Head of the Information and Signal Processing Group, EWD) who provided encouragement throughout my work.

I am very grateful for the support of the Electronic Warfare Division, Defence Science and Technology Organisation, Salisbury, for making available computing resources and the Transputer hardware and development system that was used in this work.

Finally, I wish to express my sincere thanks to my family and friends for their support and prayers. In particular I want to thank my wife Judy, for her love, understanding and continued support throughout the time I have been working on this thesis, especially as I spent many long hours working in our study at home.

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

i	Interval counter
j	Interval counter
k	Interval counter
α	Degree of Fulfillment of a fuzzy rule
x	Input data vector
$\mu(k)$	Membership function
A	Agreement Matrix
A^c	Compensated Agreement Matrix
C	Contradiction factor
P	Number of Resultant membership functions per Final membership function
$Z[k]$	Discrete representation of Final Membership Function
γ	Fusion adjustment factor
Inf	Fuzzy Information Measure
MAX	Maximum fuzzy operator
MIN	Minimum fuzzy operator
$N(x)$	Fuzzy negation of x
N_p	Number of processors
N_r	Number of fuzzy rules in the rulebase
$t(x,y)$	Fuzzy t-norm of x and y
$t^*(x,y)$	Fuzzy t-conorm of x and y
T_{comms}	Communications time between processes
T_{defuz}	Time to perform FMF defuzzification
T_{fusion}	Time to perform RMF fusion
T_p	Time to evaluate a single fuzzy rule
T_{proc}	Total Processing time per output

ABBREVIATIONS

AI	Artificial Intelligence
BCF	Bootable Code File
COG	Center of Gravity defuzzification
DBM	Data Base Manager
DOF	Degree of Fulfillment
EPLD	Electrically Programmable Logic Device
FAM	Fuzzy Associative Memory
FIE	Fuzzy Inference Engine
FMF	Final Membership Function
GUI	Graphical User Interface
KBM	Knowledge Base Manager
MF	Membership Function
MIMO	Multiple Input - Multiple Output
MISO	Multiple Input - Single Output
MOM	Mean of Maxima defuzzification
mS	milli-second
PC	Personal Computer
PED	Process Event Diagram
REN	Rule Evaluation Node
RMF	Resultant Membership Function
TDS	Transputer Development System
TIM	Transputer Interface Module
TRAM	Transputer Module
WRMF	Weighted Resultant Membership Function

PUBLICATIONS

1. Bowyer R.S., "Implementation of a Parallel Fuzzy Logic Controller", ATOUG-4 The Transputer in Australasia, IOS Press, Amsterdam, pp. 13-18, Sept. 1991.

2. Bowyer R. S., "TransFuzien - A Transputer Based Fuzzy Logic Inference Engine", IEEE ANZIIS-95, Proceedings of the Australian and New Zealand Conference on Intelligent Information Systems, pp. 140-145, Nov. 1995.

Chapter I

INTRODUCTION

1.1 Motivation

Artificial intelligence (AI) systems [1, 2], and in particular rule-based expert systems, are being applied at an ever increasing rate to the solution of information processing problems. One area of AI which is gaining in popularity is Fuzzy Logic, the basic theory of which was first described by Zadeh [3]. Fuzzy logic can be applied to the solution of a broad range of computational problems. There are many accounts in the literature which describe uses for fuzzy logic, ranging from an expert system for medical diagnosis such as MYCIN, CADIAC, and SPERIL-II [4, 5], to the control of an electric passenger train in Japan [6].

The motivation for this research is to investigate fuzzy reasoning, and examine the issues of knowledge representation, fuzzy inferencing strategies, and methods by which parallel processing techniques can be applied to this area. A fuzzy logic rule-based expert system is developed in this thesis to facilitate the investigation of these issues. It is a Transputer-based fuzzy logic inferencing engine, implemented as a data processing workstation. The computer workstation is a familiar concept, enabling a user to interact with a computer system via, in most cases, a Graphical User Interface (GUI), operating under the Microsoft Windows[1] operating system.

The user interface is a very important component of this and any expert system [7]. The system should be simple to operate so the user can concentrate on transferring knowledge to the knowledge base, and not worry about the details of the mechanism that under-lies that transfer. A suite of software is developed that supports a number of the present interpretations of fuzzy inferencing, thus providing a flexible and instructive environment for fuzzy processing. It provides all the functionality to perform rule-based fuzzy logic inferencing.

The hardware for this system comprises a PC, an Inmos B008 Transputer motherboard [8], fitted with two T800 TRAM modules, and the hardware interface. Figure 1.1 shows how the various system components fit together, whilst Figure 1.2 shows the top level structure for a rule-based inferencing system.

[1] Microsoft Corporation

The potential benefits of applying Transputers to this works are to be examined, and include;

1. Simplifying the process of mapping algorithms to hardware.
2. Using multiple processors enables multiple connections between the processing system and the external world via the serial links.
3. Processing performance as defined as the time taken to complete specific tasks, is reduced according to the number of processors assigned to a task. This will be explored in Chapter 5.

The software comprises the graphical user interface that runs on the PC, and the inferencing software that runs on the Transputers. Each of these components is described in the following sections.

The system that is developed, processes data sets by applying the rule base to each new input data set, and computing the corresponding output(s). Both inputs and outputs may be displayed on the GUI. This is an open loop processing system, as is typical of expert systems that provide some answer to a particular set of input conditions, such as in computer assisted medical diagnosis or weather classification [7, 9].

Closed loop processing is required when an expert system is being used for continuous processing of the input data, such as the for the control of the speed of an electric motor under varying load conditions. A closed loop approach is required for systems that exhibit time varying dynamics, particularly on a time scale that is short compared to the time a human operator would need to effect some appropriate control action. (eg. increase motor current when load increases, to maintain motor speed at some desired set point).

This second mode of operation is a challenge for expert systems, and places considerable responsibility on the programmer (human expert), to construct a reasonable rule base, that describes the dynamic behaviour of the system to be controlled. The objective of this research is to produce an expert system, that fulfills both modes of operation.

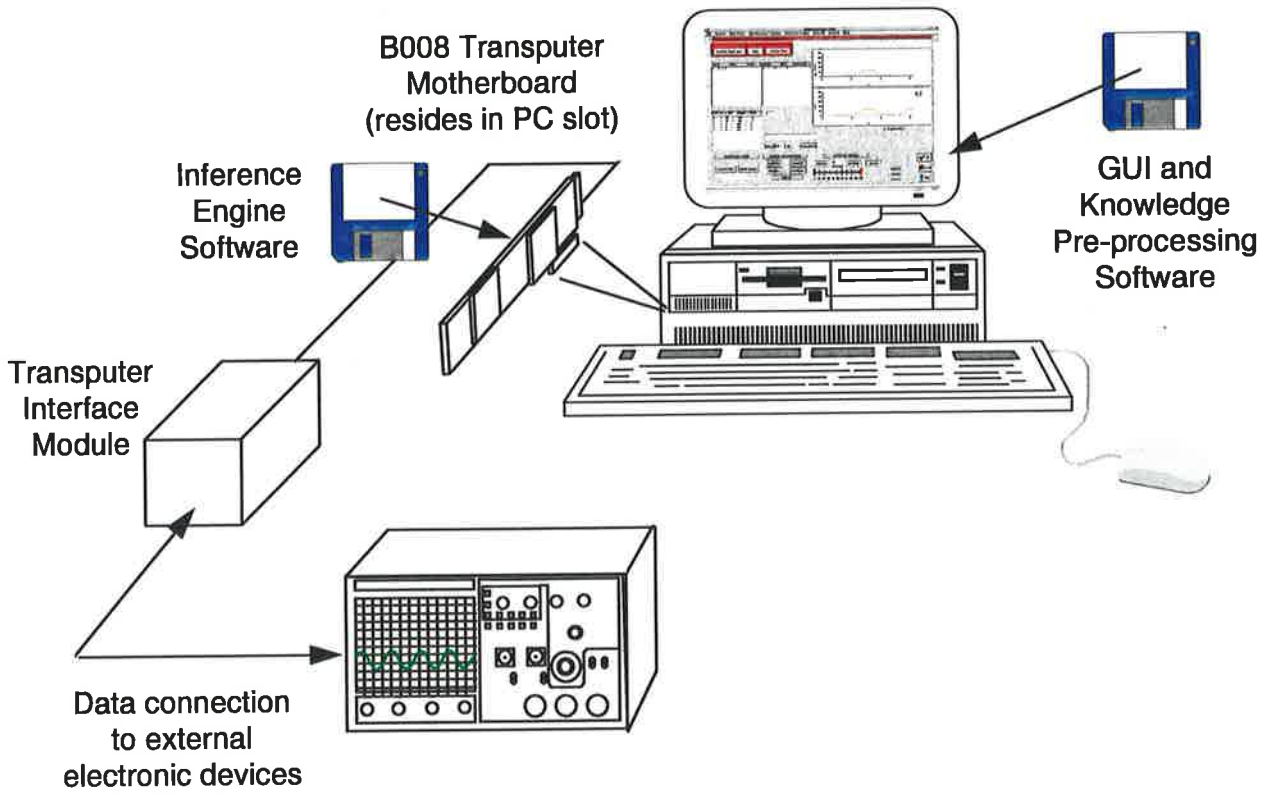


Figure 1.1: The components of the fuzzy inferencing system comprise the host computer with the graphical user interface and knowledge pre-processing program, the Inmos Transputer sub-system and the inferencing software, and the Transputer Interface Module that connects to external electronic hardware.

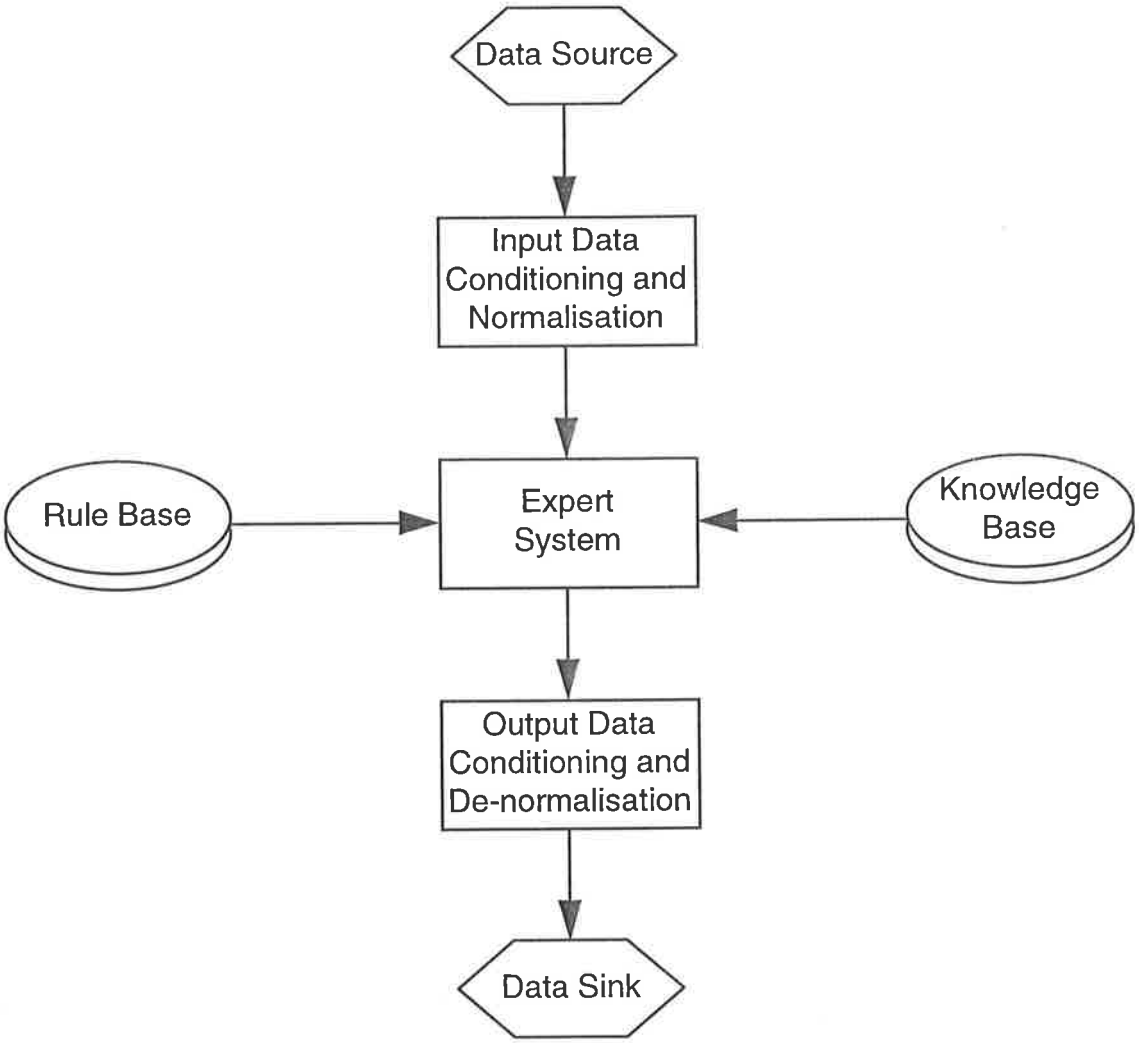


Figure 1.2: An expert system comprises a rule base where the rules are stored, and a knowledge base which stores information about the inferencing strategy. The input and output signals will require some normalisation processing.

1.2 Review of the literature

There are a number of key concepts that are explored and developed in this thesis, which are as follows;

1. Fuzzy logic processing,
2. Development of additional fuzzy operations for t-norm, t-conorm and fusion,
3. Development of an inferencing algorithmic structure,
4. Implementation of the inferencing algorithm,
5. Use of the Inmos Transputer to perform the inference processing,
6. Development of an expert system workstation, featuring a graphical user interface.
7. Software which supports a range of fuzzy operators.
8. Design and development of specialised hardware to assist in system integration.

Systems which use fuzzy logic in their processing can be divided into three broad categories. The first category is that of commercial products which provide a user interface (usually on a PC), and sometimes graphical in presentation. Typically, these development systems implement fuzzy logic rules on a particular secondary platform, allowing a user to enter rules and generate C source code for a micro-controller. Motorola have such a system for their range of micro-controllers [10].

The second category is the commercial product which develops configuration code for dedicated fuzzy processing chips. An example is the fuzzy correlator and fuzzy microprocessor from Infra-Logix [11]. Some other examples are Seimens' Fuzzy Logic Co-processor chip, the SAE 81C99[12], and the VY86C500 12-bit Fuzzy Computational Accelerator from VLSI Technology Inc.[13].

With these specialised components, the number of fuzzy inferences per second is very high, but there are usually limitations to the number and syntax of rules, and the representation of membership functions. These limitations are understandable as the devices have a fixed architecture.

The third category is very common in industry, and represents the many different types of embedded fuzzy process controllers. Typically, these are systems which run on common micro-processor platforms, where the fuzzy algorithms are hard-coded into the software. The fuzzy algorithms are written in languages such as C, Pascal, and even assembler. This makes modifications more difficult, as the code must be changed, recompiled, and then linked with the rest of the software system. Look-up tables are also often used with standard micro-controllers.

Each of these categories have their advantages. This study, however, adopts a flexible approach to the fuzzy inferencing problem, by implementing a generic fuzzy algorithm, which can be customised to suit the user's requirements, via the rule base and a choice of inferencing strategies.

Another common theme in the literature is the need for some form of accelerator or specialised computing architecture, to improve the processing performance of fuzzy systems implemented on a computer platform. The literature contains several references on the parallel implementation of fuzzy inference processing. Some of these systems propose the use of optical processing systems [14] to realise the parallel computation, but these lack the flexibility of the proposed system. In a paper by Linkens and Hasnain [15], the authors describe fuzzy logic implemented on the Transputer. The fuzzy rules are directly coded in Occam, and fuzzy control in parallel, as a means to speed-up the processing, is examined by the authors. The approach taken in this thesis differs from this approach, in that the rules are not coded in Occam.

There are many references in the literature to 'expert system' shells and AI toolkits [5]. There are key papers [16, 17] which describe systems and ideas which combine one or more of the features which are listed above. This work investigates methods which make an expert system more flexible. These papers are listed below and comments made regarding their content and relevance to this current work.

In work published by Marian S. Stachowicz [18], a hardware accelerator for fuzzy inferencing is described. The inferencing and defuzzification techniques are fixed. The universe of discourse is limited to 25 elements and the number of levels is 5. The processor uses a pipeline architecture to achieve the parallelism, with a quoted performance of 800 k Fuzzy Inferences per Second (kFIPS).

H. Ekerol and D.C. Hodgson in [17], describe a dedicated control system which uses Transputers and fuzzy logic. The defuzzification method used is weighted average. The fuzzy logic control algorithm uses 10 rules to control the intensity of an air-jet that is a component in a sorting system involving a conveyor belt. The control actions are pre-calculated and stored in a look-up table residing in the Transputer's memory.

There are examples in the literature of parallel implementations of fuzzy logic systems [19, 20], which address specific applications, but do not address the combination of issues at which this research is aimed.

1.3 Original Contributions

There is a lack of published work that focuses on using parallel processing architectures for the implementation of rule-based inferencing systems. In particular, there is little work, to the best

of the author's knowledge, that uses Transputer technology to implement flexible fuzzy expert systems. This research examines the following areas;

1. Methods of fuzzy logic processing, including connectives, modifiers, fusion operators, and defuzzification. Some new operations are proposed.
2. Application of parallel processing principles to fuzzy processing.
3. Development of a parallel inferencing algorithm.
4. Implementation of the algorithm using the Occam 2 programming language, and running the software on the Inmos T800 Transputer.
5. Investigation into human-computer interfaces.
6. Development of a human-machine-interface (HMI) front end for an expert system.
7. Investigation into the effects of different inferencing options on data processing.
8. Demonstration of inferencing methodologies by way of a number of examples.
9. Design and development of a Transputer Interface Module (TIM), that provides control functions and data collection capability to the TransFuzien system. This involved hardware, firmware, and software, design.

The system which is developed in this study, aims to be as flexible as possible. To provide flexibility in the choice of inferencing options, and so on, impacts directly on the development of the various processing algorithms. The INMOS Transputer has been selected as the processing engine for this study as it has many desirable characteristics which make it suitable for this work. These include:

1. Ease of mapping data flow description onto the architecture through the application of Occam in writing the code.
2. A scaleable performance is achievable by devolving sections of the processing to additional Transputers, or by applying multiple Transputers in a farm arrangement.

To achieve a multiple-input-multiple-output (MIMO) processor, a generic fuzzy inferencing program is run on the T800 Transputer. This program is loaded onto the Transputer system via a command issued by the user, and booted to establish communications with the user interface program which is running on the PC.

The knowledge base interface which runs on the PC, is used to define the parameters of the fuzzy processor. These parameters include, the rule base, the membership function definitions, and the input and output names and mapping. When the fuzzy processor is running

and processing input from what ever source, the user then has the option to alter the above mentioned parameters.

The system which is developed can operate in one of two modes. The first mode, is where input data is processed by a set of rules, output calculated and displayed. This one-off processing behavior is applicable in a system for assisting a person to come to a decision or conclusion, such as in medical diagnosis, or data recognition/classification.

The second mode of operation, processes data continuously. The inputs are sampled at a regular time interval, and the rule-base processes the data for each new interval. This is applicable to control of processes, and requires an appropriate response time from the inference engine, according to the time constants of the controlled plant.

1.4 Thesis Outline

This dissertation is set out in the following manner. Chapter 2 introduces fuzzy set theory, and describes the membership function, various methods of fuzzy inferencing, and the defuzzification process. Some of the advantages of using a fuzzy approach to information processing are given. The union and intersection operations between fuzzy sets is examined and several methods are presented that have been explored in the literature. At this stage, additional operators for these connectives are proposed, with examples to illustrate their affect. Likewise, the process of fusing information that is represented by multiple fuzzy sets, is examined, and an information measure is derived. Further, two additional methods are proposed to fuse fuzzy sets.

With the basic concepts covered, the next chapter examines the process of fuzzy inferencing using *IF-THEN* production rules as the method of knowledge encapsulation. The process is examined to explore how fuzzy inferencing can be performed in a parallel domain, and to identify the driving issues and choices that must be considered. A software structure is developed to perform fuzzy inferencing. The development of a structure for fuzzy inferencing naturally leads to the requirement to configure, test, and apply this structure.

To obtain a physical realisation of the ideas developed in Chapter 3, they must be implemented on the Transputer target system. This process is described in Chapter 4. A means of user interaction is essential in this system, and the type and function of the interface are described in Chapter 5.

This research has the requirement of being able to process data from a variety of physical sources. The Transputer Interface Module (TIM) gives the system the ability to exchange data with external electronic equipment. The TIM is described in Chapter 6.

A number of case studies have been completed using the software and hardware developed in this research. There are four of these studies which each exercise various aspects of the system, and show the results of the fuzzy inferencing process. In Chapter 7, there is an example of each of the following;

1. Multiple input-multiple output (MIMO) system,
2. Fuzzy pattern classification,
3. Fuzzy modeling of a function, and
4. Fuzzy control.

The results for each example are given and discussed. Chapter 8 presents the conclusions of this research, and identifies areas of further work.

Chapter II

INTRODUCTION TO FUZZY SETS AND FUZZY LOGIC

2.1 Overview

In this chapter, fuzzy sets are introduced, together with the basic theory of fuzzy logic operations. This chapter begins by making a comparison between 'classical' set theory, and fuzzy set theory. This leads to a description of fuzzy set operators and membership functions, followed by fuzzy logic, being the method by which fuzzy sets and operators interact. At this stage, two operators are proposed by the author, with illustrative examples given.

In this discussion, the fuzzy rule is introduced as a means of encapsulating domain knowledge[1]. A mathematical description of the inferencing mechanism is presented.

The process by which rules that relate to a common output are combined, is a form of data fusion, and hence this process is called fusion in this work. There are a number of rule fusion or aggregation methods [21] that are in common use, and some of these are presented.

The information content of fused data sets is addressed, and an Information Measure is proposed. Then, two additional methods are proposed by the author, for fuzzy set fusion.

Some of the common defuzzification methods are presented, followed by a discussion on adaptive fuzzy systems. Lastly, temporal aspects of fuzzy logic are considered, with a new concept, in relation to the fuzzy rule, being proposed.

2.2 A Contrast of Styles - Boolean (Crisp) vs Fuzzy

Classical set theory is very clear about what it means to belong to a particular set of objects. In classical set theory, there are no *degrees of belonging* or *degrees of truth*. With Boolean logic there are no shades of gray, only black and white, true or false. However with fuzzy logic, we are dealing with *continuous-valued logic*. There is a continuous transition between one extreme and another, thus enabling us to assign degrees of truth to propositions, such as, *"it is quite cold in this room"*. Fuzzy logic allows such statements, and provides a

[1] Knowledge about a particular process or event

mathematical framework by which it can be evaluated, providing a *degree of truth* or *degree of fulfillment* for the statement.

Fuzzy logic is a powerful tool for dealing with information using approximate reasoning. In fact, humans make subjective decisions all the time, based on minimal and often vague information. *We are, all of us, experts in fuzzy logic*. In the expert system developed herein, the processing of information is accomplished using linguistic descriptions in the form of *IF-THEN* rules. These are the embodiment of the knowledge of a process or event that is available for inferencing operations. These rules are derived from the knowledge of a system's behavior.

The derivation of the fuzzy rules is an interesting field of study, and can be accomplished by interviewing an 'expert', to derive a series of *'if-then'* statements that reasonably cover the range of expected behaviour of the process that is to be modeled. The rules comprise a particular syntax (and are linguistic in form) and a number of fuzzy operators. These are described in the following sections.

An important aspect about conventional rule-based expert systems [2, 22] is that when the system is evaluating its rule-base, it is searching for a match between its knowledge base and the input data. In traditional binary logic, if a rule statement does not match exactly with the input data, then the rule has no further effect on the outcome of the expert system. Hence, if the input data is almost, say 99%, but not quite a match, then this information is lost to the system.

In the case of fuzzy logic, every rule will contribute, to the degree of fulfillment of that rule, to the outcome. Hence the 'quality'[2] of the outcome is higher, as there is no loss of information. This could be important where the quality and the quantity of available data is limited. A system that uses fuzzy logic for inferencing uses all the information at its disposal, and wastes nothing. This is one of the advantages of an expert system based on fuzzy inferencing. By generating a list of behavioural rules for a particular system, and applying the mathematics of fuzzy set operations, the linguistic rules of a knowledge base can be translated into a flexible numerical domain [23].

The common approach to decision making or process control, is to define accurate mathematical models of the plant or process and use the sensory data to monitor its performance, and make appropriate control decisions. This points to one of the main advantages of fuzzy processing. A system need not be described by an exact mathematical model.

[2] Defined by the author as the number of rules activated in a Boolean system, compared to the number of rules activated in a fuzzy system, for the same rulebase.

In the case where the model is known and comprises high order differential equations, solving these equations in a real-time situation places considerable demand on the processing architecture and software. A fuzzy approach allows complicated systems to be described with linguistic rules. Also, where there is insufficient information[3] about the internal operation of a particular system, then it can still be modeled based on the observed behaviour using the fuzzy approach.

Put another way, fuzzy logic enables a system designer to incorporate qualitative and non-linear behaviour into the system model. An example of this situation is where there are a limited number of sensor outputs available to the processor, as in the case of some industrial control situations [24].

In classical control theory, for example, a transfer function of the plant is required, and exact data is used to control a plant of some description [25]. Data from the plant is applied to a mathematical model of the plant to derive some new control inputs to the system. The behaviour or response of the plant is determined by how well the mathematical model represents the plant.

A rule-based fuzzy logic controller applies the input data to the rule base, with all rules being processed in parallel. For each individual output their may be many contributing rules. The outputs from these rules are then combined and de-fuzzified to produce a crisp output. Consider the example where an air-conditioning plant must maintain the air temperature in a room at a pre-defined setting. One rule to accomplish this may look like:

- IF outside temperature IS high AND inside temperature is (high and increasing)
  THEN turn cooler on high

In the following sections the components of fuzzy inferencing are addressed. It is worth noting at this point, that the fuzzy approach is just another tool to solve problems. As with any method of scientific investigation, the tools are there to aid in the understanding of the problem at hand.

[3] There may not be enough sensors within an apparatus to properly describe a required parameter of the system.

2.3 The Membership Function

A membership function (MF) is a mathematical description of a linguistic model or class, such as 'tall person'. It is not the presence of random variables, but the presence of vagueness and imprecision which is the central theme when defining a membership function [3][4]. Membership functions are not probability distribution functions, but are descriptions of linguistic terms or labels, that are relevant in the particular instance.

As an example of a membership function, consider an experiment where the temperature of a room in a house is being measured. A thermometer is placed in the room and allowed to stabilize before a reading is taken. The reading may be 30 degrees Celsius, which is classified by most people as hot. If the temperature was 12 degrees, then this may be classified as cool. The numerical range over which an input variable is defined is called the Universe of Discourse [3]. In this case the universe of discourse may range of temperatures tolerable to humans (10 degrees to 40 degrees).

Now consider if an air conditioner is turned on and a series of temperature measurements are taken at regular intervals. As the temperature drops, the classification of the temperature would change from hot to 'not too hot', to 'warm', to 'slightly cool', and maybe further to 'cold' and finally, to 'very cold'.

Each of the classifications mentioned above, are examples of classes in fuzzy logic which are called membership functions. Consider the first temperature reading in our example of 30 degrees. In fuzzy nomenclature, it has a high degree of membership to the class 'hot', say 95%, but a lesser degree of membership to the class warm, say 60%, and an even lower degree of membership to the class cold, say 5%. Figure 2.1 illustrates this idea. The key point is, that a variable measurement (eg. temperature) can simultaneously belong to other membership sets to varying degrees.

The definition of a membership function, that is, its shape, is an area worthy of some explanation. Dubois and Prade [21] give an interesting discussion entitled, "Where do they come from?". Where indeed! When an MF is defined, it is often representing some empirical data. The MF can be thought of as a transform from crisp space to fuzzy space. The MF is the 'handle' by which humans translate linguistic operators into the realm of mathematics processes for use by computers. An analysis of this subject can be found in Chapter 6 of [4].

Membership functions are often mathematically modeled by a continuous or discrete trigonometric function, polynomial, or linear expression. Some examples of membership function profiles are;

1. Gaussian function

2. Polynomial
3. Linear equation
4. Trigonometric function
5. S and Π functions

Equation 2.1 generates functions with varying characteristics, dependent upon the parameters A (maximum amplitude chosen to match the range of the input data), S (spread factor), and C, which is the fuzzy number or center point of the fuzzy set.

$$\mu(x) = A \cdot e^{[-S([1-x]-C)^2]} \tag{2.1}$$

and

$$\mu(x) = \frac{A}{\left[1+\left[\frac{1}{S}\cdot(x-C)\right]^B\right]} \quad : S>0 \tag{2.2}$$

where $x \in [0,100], \quad S \in [1,100]$

Equation 2.2 is has been used to generate a default family of nine membership functions for the expert system software developed in this work. These are shown in Figure 2.2. The GUI is described in further detail in chapter five

In this work, a discrete representation of membership functions is adopted, as shown in (2.3), with a range of *[-50..+50]*.

$$\mu(\hat{x}) = \left\{\frac{\mu_0}{x_0}, \frac{\mu_1}{x_1}, \ldots\ldots, \frac{\mu_{n-1}}{x_{n-1}}, \frac{\mu_n}{x_n}\right\} \tag{2.3}$$

where $\frac{\mu_i}{x_i}$ is the value of the membership function at x_i.

The number of points that are used has a direct bearing on the time required to process the function. As the number of points increases, so does the time required to perform transforms on the membership functions.

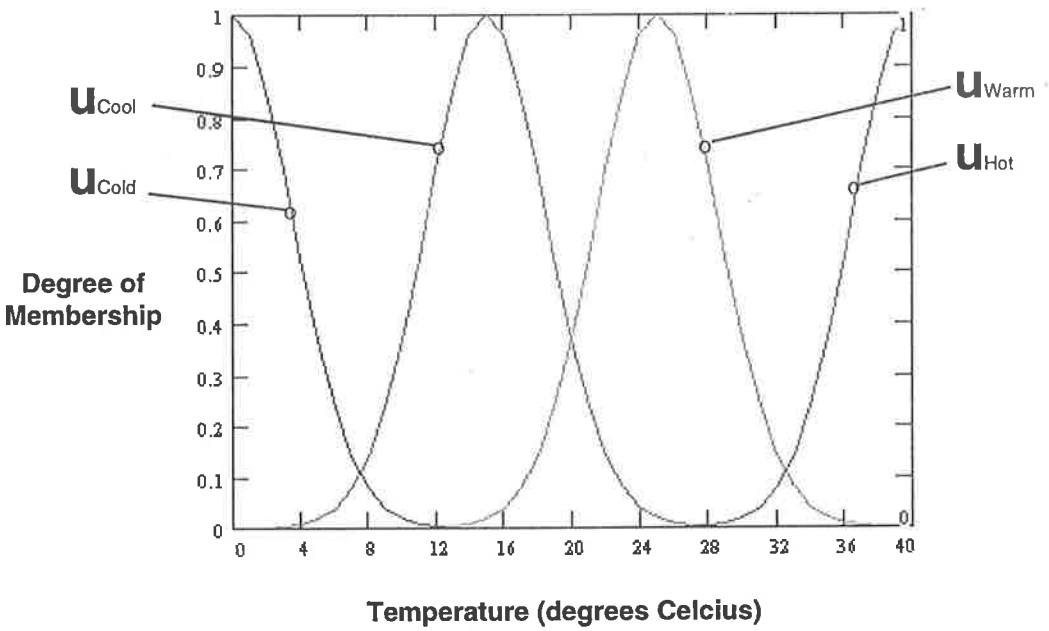


Figure 2.1: Set of Membership Functions that describe a range of temperatures. The position of the peaks and the spread of each function are chosen to suit the application.

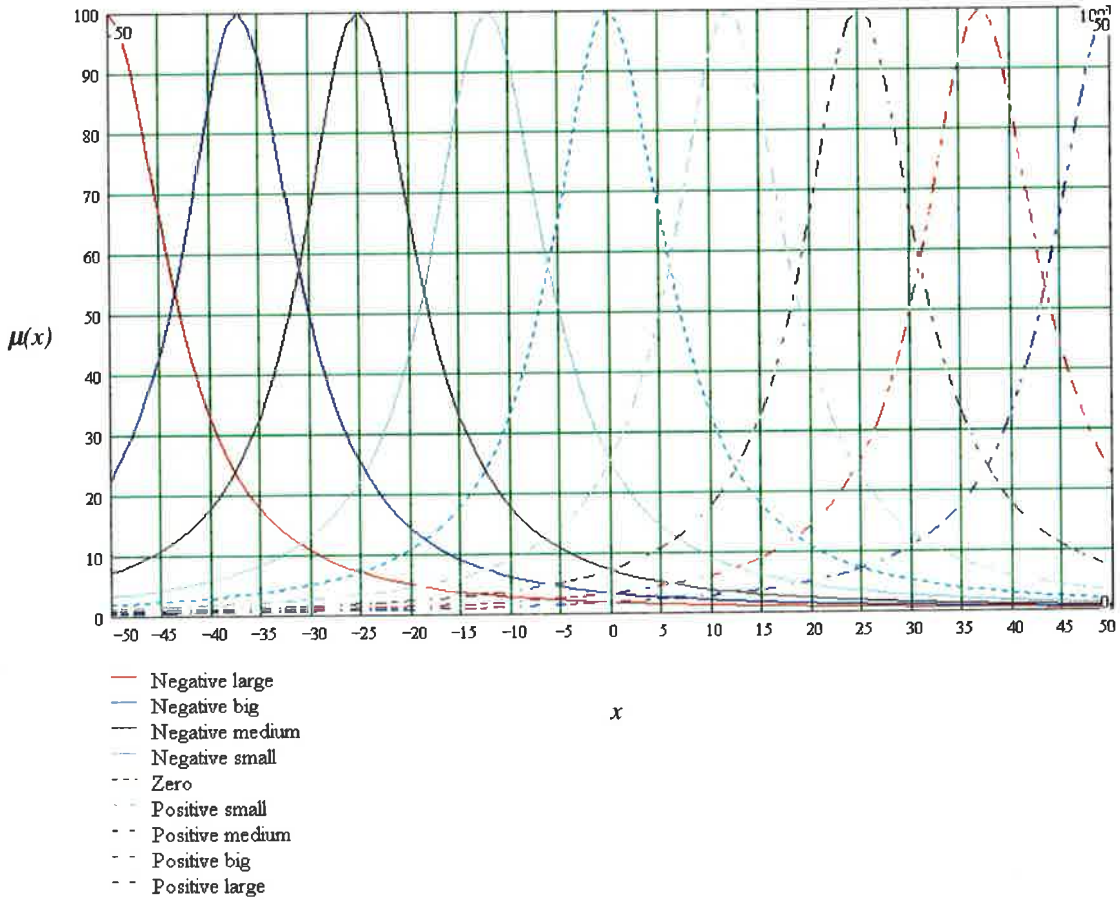


Figure 2.2: Set of Membership Functions generated by equation 2.4, with varying parameters.

2.4 Fuzzy Numbers

Just as the fuzzy sets for *small* or *medium* are represented by a membership function, so numbers can also be described. For instance, a crisp real number N is converted to a fuzzy number by specifying the form of a membership function, centered on N. An example of this could occur as follows:

- IF temperature IS close_to 30 degrees THEN fan_speed IS HIGH

The premise 'temperature IS close_to 30 degrees', allows specific values to be used in the rule base, compared with the more generic terms of small and high.

Fuzzy arithmetic can be performed between fuzzy numbers. An example of fuzzy addition using the discrete fuzzy set representation, as introduced in the previous section, is as follows:

0/0, 0/1, 0/2, 1/3, 2/4, 3/5, 5/6, 3/7, 2/8, 0/9 + 9/ 0, 8/1, 5/2, 2/3, 0/4, 0/5 , 0/6, 0/7, 0/8, 0/9

= 9/0 ,8/1 , 5/2 , 2/3 , 2/4 , 3/5 , 5/6 , 3/7 , 2/8 , 0/9

Fuzzy multiplication and division can be performed, and Dubois and Prade give a detailed treatment of this subject in Chapter 2 of [21].

2.5 Linguistic Hedges

Linguistic modifiers or *hedges* [21, 26, 27] provide a means to emphasize or de-emphasize the effect of a premise. Hedges add to the grammar for writing fuzzy rules, and enhance the ability of the expert to express ideas in a rule form. Table 2.1 lists some of the definitions for hedges. Figure 2.3 illustrates the effect of hedges on membership functions.

Hedges can be used in fuzzy rules to modify the membership functions, as shown in this example:

- IF X1 IS very large AND X2 IS quite small THEN Z1 IS zero

Hedge	Description
normalisation:	$\mu_{norm(A)}(u) = \mu_A(u) / (sup\ \mu_A)$
concentration (eg. very)	$\mu_{con(A)}(u) = [\ \mu_A(u)\ ]^2$
dilution: (eg. some-what)	$\mu_{dil(A)}(u) = [\ \mu_A(u)\ ]^{1/2}$
contrast intensification:	$\mu_{int(A)}(u) = \begin{cases} 2.\mu_A{}^2(u) : \mu_A(u) & \varepsilon\ [\ 0\ ,\ 0.5\ ], \\ 1 - 2.(\ 1 - \mu_A(u)\ )^2 & \text{otherwise} \end{cases}$

Table 2.1: Linguistic hedges commonly used for modifying the shape of membership functions.

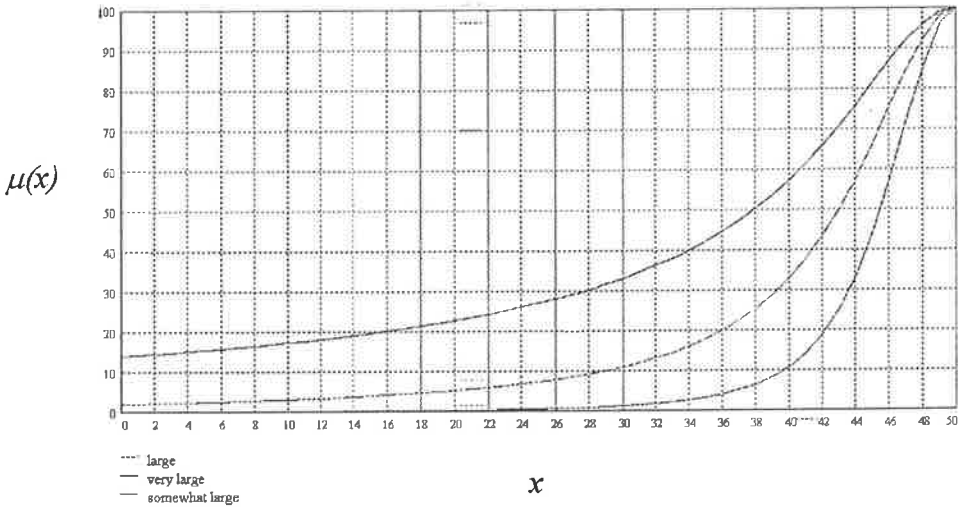


Figure 2.3: The shape of a membership function $\mu(x)$ can be modified by a linguistic hedge. This figure shows the effect of both concentration (square) and dilution (square root) operators on a membership function.

2.6 Fuzzy Set Operations

In this section the concepts of fuzzy set operation are reviewed. The original paper on fuzzy sets by Zadeh [3], described the operations that can be performed between fuzzy sets. Fuzzy sets share the same properties of associativity, distributivity, and commutativity, and so on, as crisp sets. Figure 2.4 illustrates classical Boolean logic, where two sets A and B, intersect. Further, the data points (denoted by +) that are scattered throughout the figure, may belong to either set A, set B, both set A and B, or to neither set. In Figure 2.5 the same data points now belong to both set A and B, with a degree of membership denoted m_A and m_B.

These definitions describe the standard operations for fuzzy sets. They make use of the operators *AND*, *OR*, and *NOT*, all of which have clear definitions for crisp logic. In this work, these operators are referred to as *connectives*. There are various interpretations of how these connectives, *AND, OR*, and *NOT* are implemented. Dubois and Prade [23] discuss the operations on fuzzy sets in detail, and define a number of alternatives. Klir and Folger [27] give a good overview of some of the various operations on fuzzy sets, and include mathematical proofs. A number of these methods have been incorporated into the TransFuzien software, giving the user a more flexible system for fuzzy computation. This is discussed in detail in Chapter 5.

The original paper on fuzzy sets by Zadeh, described the operations that could be performed between fuzzy sets. The operations of union, intersection, and complement are defined in this section.

Consider a function $\mu_A(x)$ and a variable $x : [0, 1]$, and let $\mu_A(x)$ and $\mu_B(x)$ be fuzzy sets on the interval $[0, 1]$. The complement of a fuzzy set is defined as:

- $1 - \mu_A(x)$

The fuzzy '*AND*' function is the intersection of two or more sets, and is evaluated by calculating the minimum of the membership grades within any of the subsets.

- $\mu_A(x) \quad AND \quad \mu_B(x) \quad = MIN(\ \mu_A(x) \ , \ \mu_A(x) \ )$

The 'OR' function is the union of two or more sets, and is evaluated by calculating the maximum of the membership grades within any of the subsets.

- $\mu_A(x) \quad OR \quad \mu_B(x) \quad = MAX(\ \mu_A(x) \ , \ \mu_B(x) \ )$

Fuzzy sets posses the properties of associativity, distributivity, and commutativity [3].

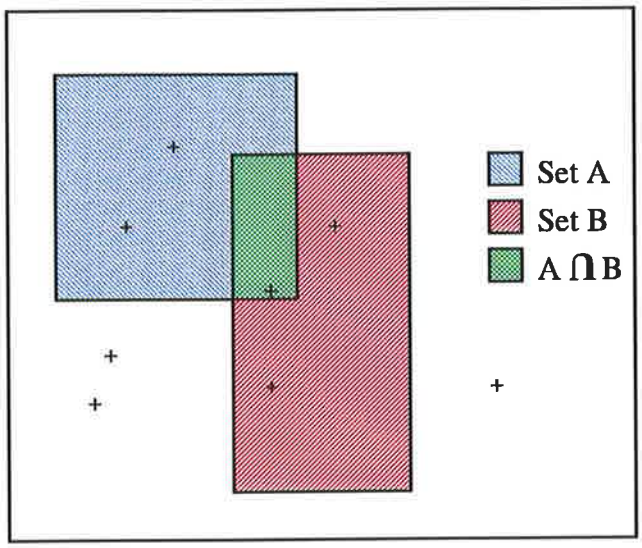


Figure 2.4: Two sets A and B define regions of interest in the parameter space. Data points may belong to either A or B, or the intersection of A and B. The data may also lie outside these sets.

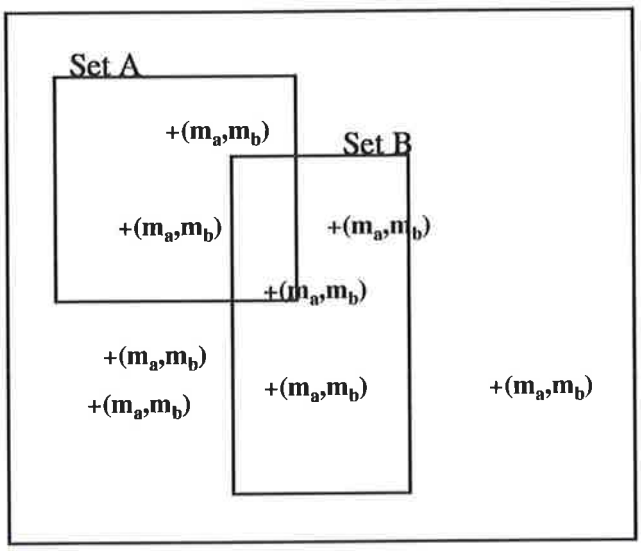


Figure 2.5: With fuzzy classification, each data point has a membership to each set. In this case, the memberships are denoted m_a and m_b.

The two fuzzy sets A and B obey the rules described in table 2.1. The range of the parameter which is being classified is known as the 'Universe of Discourse'.

These definitions are the *standard operations* of fuzzy theory. There are however, other definitions for the fuzzy set operators. Table 2.3 shows some of the fuzzy operators in current use. Figures 2.6, 2.7, 2.8 and 2.9 illustrate examples of these operators. Some of the operators listed in Table 2.3 have been incorporated into the inferencing kernel program that runs on the Transputer system. These operators provide the user with a choice of inferencing strategies for fuzzy computation.

The connectives, AND and OR are also called T-operators, or t-norm and t-conorm respectively. Gupta and Qi in [28] summarized the properties of the t-norm and the t-conorm. The definitions are as follows:

Definition 2.1: T-norm:

Let $t: [0,1]\ x:[0,1] \rightarrow [0,1]$, then t is a T-norm iff the following are true:

1. $t(0,0) = 0$
2. $t(x,1) = x$
3. $t(w,x) <= t(y,z)$ if $w <= y$ and $x <= z$ (monotonicity)
4. $t(x,y) = t(y,x)$ (symmetry)
5. $t(x,t(y,z)) = t(t(x,y),z)$ (associativity)

Definition 2.2: T-conorm

Let $t^*: [0,1]\ x:[0,1] \rightarrow [0,1]$, then t* is a T-conorm iff the following are true:

1. $t^*(0,0) = 0$
2. $t^*(x,0) = x$
3. $t^*(w,x) <= t^*(y,z)$ *if* $w <= y$ *and* $x <= z$ (monotonicity)
4. $t^*(x,y) = t^*(y,x)$ (symmetry)
5. $t^*(x,t^*(y,z)) = t^*(t^*(x,y),z)$ (associativity)

Definition 2.3: Negation

Let $N:[0,1] \rightarrow [0,1]$, N is a negation function iff the following conditions are true:

1. $N(0) = 1, N(1) = 0$
2. $N(x) <= N(y), x >= y$
3. $N(x)$ is continuous
4. $N(x) < N(y)$, *for* $x > y$ *for all* x,y *in* $[0,1]$
5. $N(N(x)) = x$, *for all* x *in* $[0,1]$

Property	Description
Commutativity:	$A \cup B = B \cup A$
Associativity:	$A \cup (B \cup C) = (A \cup B) \cup C$
Idempotency:	$A \cup A = A \quad A \cap A = A$
Distributivity:	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
DeMorgan Laws:	$\overline{(A \cap B)} = \overline{A} \cup \overline{B}$ $\overline{(A \cup B)} = \overline{A} \cap \overline{B}$

Table 2.2 : Properties of fuzzy sets.

Table 2.3 : The fuzzy operators for *AND*, *OR*, and *NOT* have been expressed in varying ways by different authors.

Type	Fuzzy Intersection : t-norm (AND)	Fuzzy Union : t-conorm (OR)	Fuzzy NOT	Range
Schweitzer & Sklar [1961]	$\max(0, x^{-p} + y^{-p} - 1)^{-1/p}$	$1 - \max(0, (1-x)^p + (1-y)^p - 1)^{1/p}$		
Zadeh [1973]	$\min(x, y)$	$\max(x, y)$	$1 - x$	
Giles [1976]	$\max(x + y - 1, 0)$	$\min(x + y, 1)$	$1 - x$	
Lukasiewicz Logics	$\frac{x.y}{(x + y - x.y)}$	$\frac{x + y - 2.x\,.\,y}{(1 - x.y)}$	$1 - x$	
Weber [1983], Bandler et al [1980]	$x\,.\,y$	$x + y - x\,.\,y$	$1 - x$	
Hamacher [1985]	$\frac{x.y}{\gamma + (1 - \gamma)(x + y - x.y)}$	$\frac{x + y - (2 - \gamma)\,x.y}{1 - (1 - \gamma)\,x.y}$	$1 - x$	$\gamma \geq 0$
Frank [1979]	$\log_s\left[1 + \frac{(s^x + 1)(s^y - 1)}{(s - 1)}\right]$	$1 - \log_s\left[1 + \frac{(s^x - 1)(s^{(1-y)} - 1)}{(s - 1)}\right]$		$\gamma \neq 0$ $0 < \lambda < \infty$
Yager [1980]	$1 - \min\left[(1, (1-x)^w + (1-y)^w)^{1/w}\right]$	$\min(1, (x^w + y^w)^{1/w})$		
Dubois & Prade [1980]	$\frac{x.y}{\max(x, y, \lambda)}$	$1 - \frac{(1 - x)(1 - y)}{\max(1 - x, 1 - y, \lambda)}$	$1 - x$	$0 < \lambda < \infty$
Dombi [1980]	$\frac{1}{1 + [(1/x - 1)^{\lambda} + (1/y - 1)^{\lambda}]^{-1/\lambda}}$	$\frac{1}{1 + [(1/x - 1)^{-\lambda} + (1/y - 1)^{-\lambda}]^{-1/\lambda}}$	$1 - x$	
Yu Yandong [1985]	$\max((1 + \lambda)(x + y - 1) - (\lambda.x.y), 0)$	$\min(x + y + \lambda.x.y, 1)$	$1 - x$	$\lambda \geq -1$

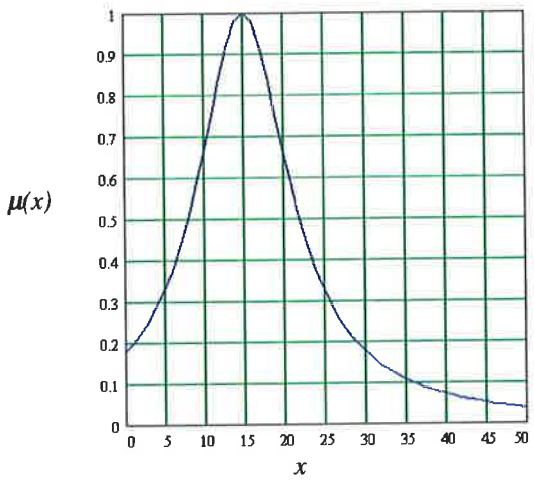


Figure a: Membership function for 'low'.

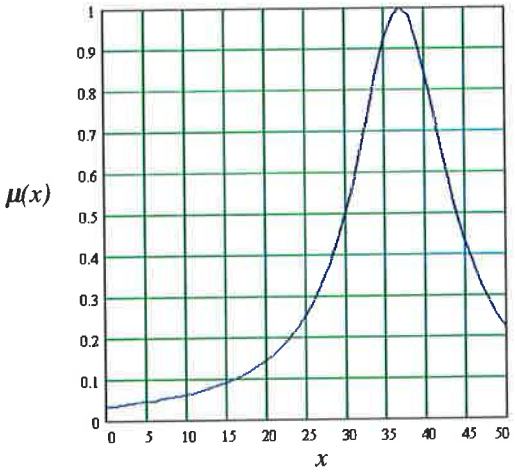
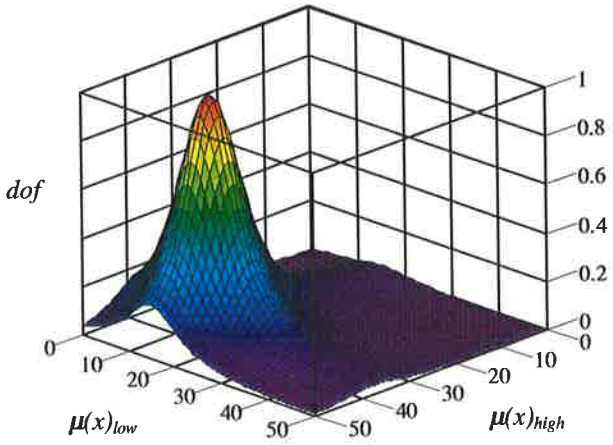
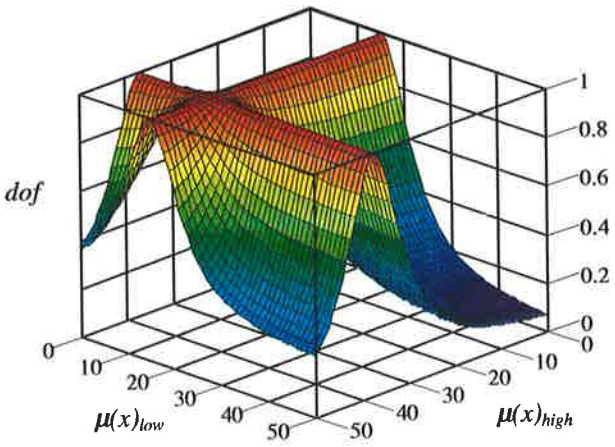


Figure b: Membership function for 'high'.



S1

Figure c: Degree of fulfillment surface S2 for AND, using Weber's interpretation.



S2

Figure d: Degree of fulfillment surface S2 for OR, using Weber's interpretation.

Figure 2.6: An example of the interaction between two membership functions using Weber's definition of the AND operator and the OR operator. The degree of fulfillment surfaces for a two input system is a useful visualization tool for fuzzy inferencing.

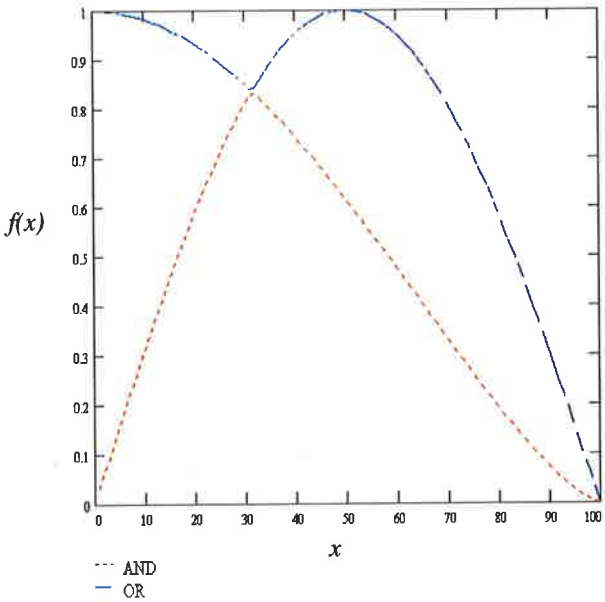


Figure 2.7a: The simplest fuzzy set operators to implement are those originally proposed by Zadeh. Graph showing the outcomes, of the fuzzy set operations AND and OR .

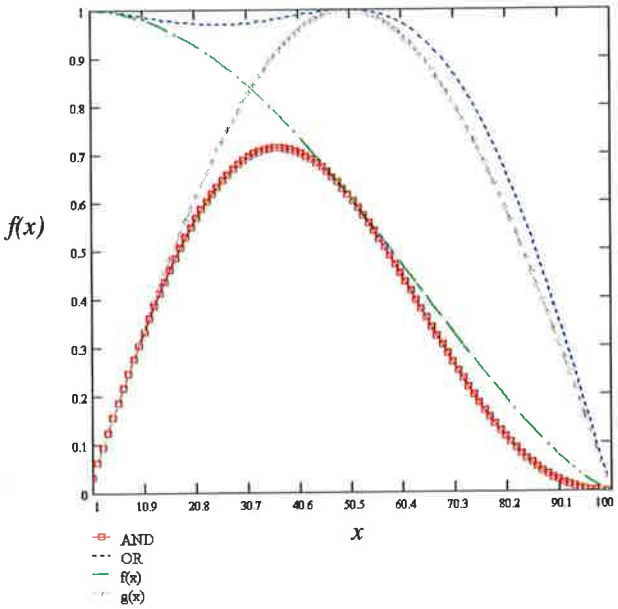


Figure 2.7b: Graph showing Weber AND and OR functions.

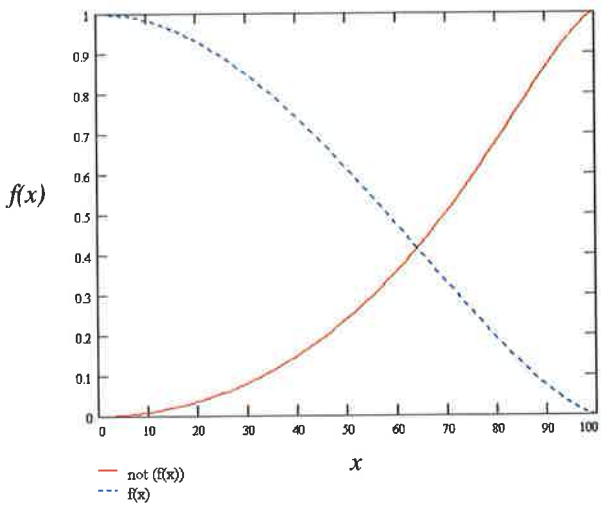


Figure 2.7c: Graph showing Weber NOT function.

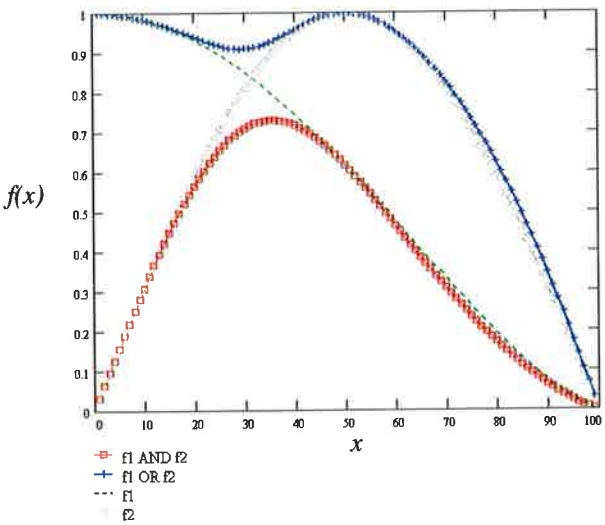


Figure 2.7d: Graph showing Hamacher AND and OR functions.

Figure 2.7: There are several definitions for fuzzy set operators. The effects of each operator vary, as shown in this sequence of graphs which depict the AND, OR and NOT operators. The input is shown as x and the result is depicted as $f(x)$.

From table 2.3, it is clear that the choice of operators will impact on both the outcome of the inferencing process, and the time required to calculate the outputs, due to the calculations steps required for each method. Zadeh's operators are the simplest, whilst others offer parameters that may be *tuned* to suit the individual application.

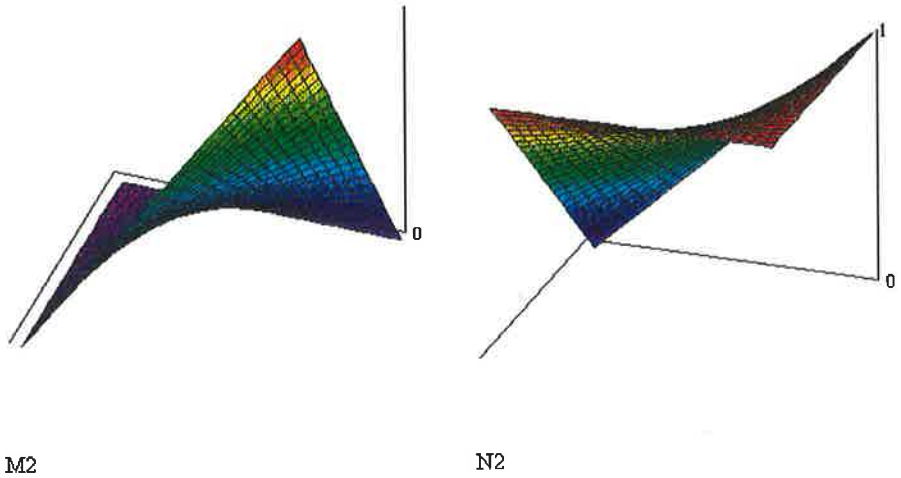


Figure 2.8: Graphs showing Weber t-norm (M2) and t-conorm (N2) with both inputs varying from 0 to 1. This interpretation of T-operators is quite simple to compute and from these surface plots, it can be seen that they conform to an intuitive definition of AND and OR logic.

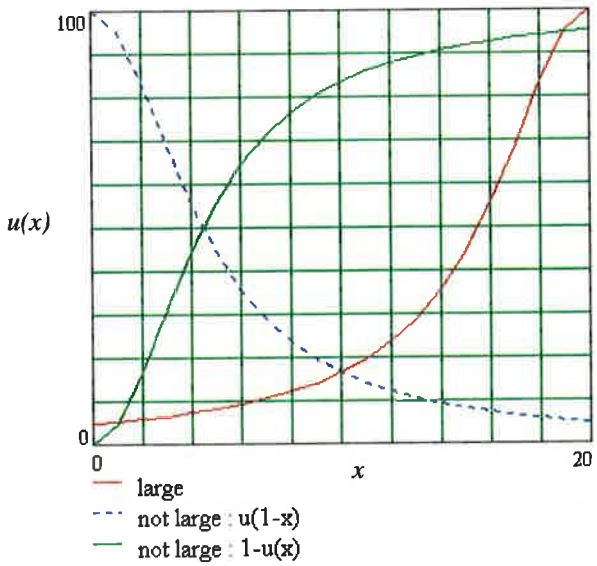


Figure 2.9: The NOT operator can be interpreted in at least two ways, as shown in this figure.

2.7 Proposed Fuzzy Set Operators

The OR operation is a union operation between sets. This t-conorm is based on a product of the minimum of the variables x and y, modified by a factor dependent on the absolute distance between x and y. The equation is modified by the parameter γ. The calculations are simple to perform (especially if $\gamma = 1$), comprising of comparison operators MIN and MAX, and arithmetic operations, so they can be simply implemented in software.

The t-conorm and t-norm proposed in this work are shown in equations 2.4 and 2.7 respectively.

$$t^*(x,y)=MIN\left[MAX(x,y)\cdot\left(1+\frac{|x-y|}{\gamma}\right),1\right] \tag{2.4}$$

where γ = integer, $\gamma \neq 0$ and $x, y \in [0,1]$

A t-norm can be derived [9] from an appropriate t-conorm by applying equation (2.5). If $t^*(x,y)$ is a t-conorm, then:

$$t(x,y)=1-t^*(1-x,1-y) \tag{2.5}$$

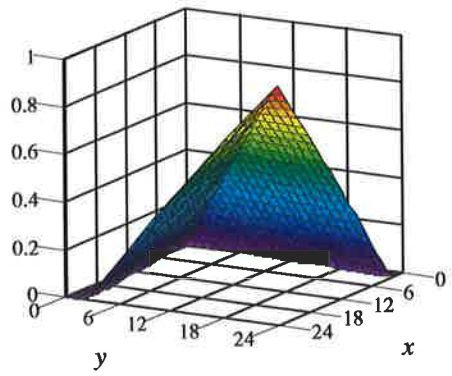
Substituting (2.4) into (2.5) gives (2.6):

$$t(x,y)=1-\left[MIN\left[MAX\left[(1-x),(1-y)\right].\left[1+\frac{|(1-x)-(1-y)|}{\gamma}\right]\right],1\right] \tag{2.6}$$

Expanding terms and simplifying gives (2.7).

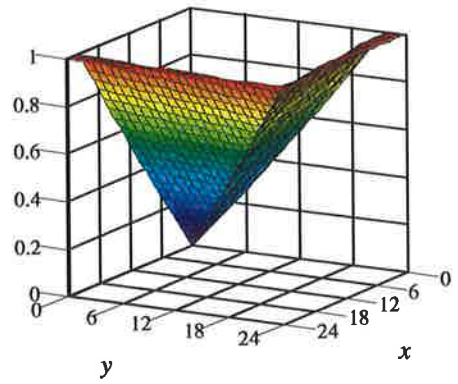
$$t(x,y)=1-\left[MIN\left[MAX\left[(1-x),(1-y)\right].\left[1+\frac{|x-y|}{\gamma}\right]\right],1\right] \tag{2.7}$$

Figures 2.10 and 2.11 illustrate the effect of these operators. It can be shown that these new operators possess the properties of Definitions 2.1 and 2.2 for $\gamma \to \infty$. They offer a measure of tuning (enhance or diminish) by altering the parameter γ, as is clearly illustrated in Figure 2.11.



M

Figure 2.10a: Proposed t-norm as a function of x and y. The values shown on the x and y axes are the indices of the matrix.



N

Figure 2.10b: Proposed t-conorm as a function of x and y. The values shown on the x and y axes are the indices of the matrix.

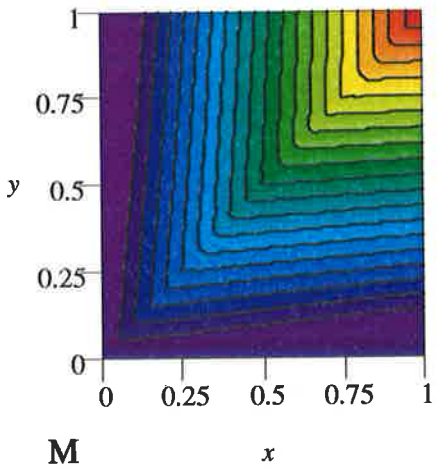


Figure 2.10c: Contour plot of the t-norm. The actual values of x and y are shown on this plot.

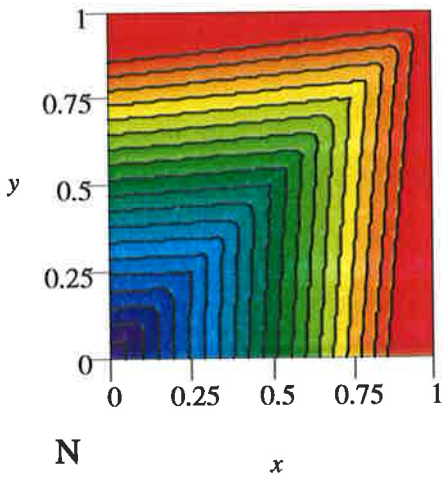
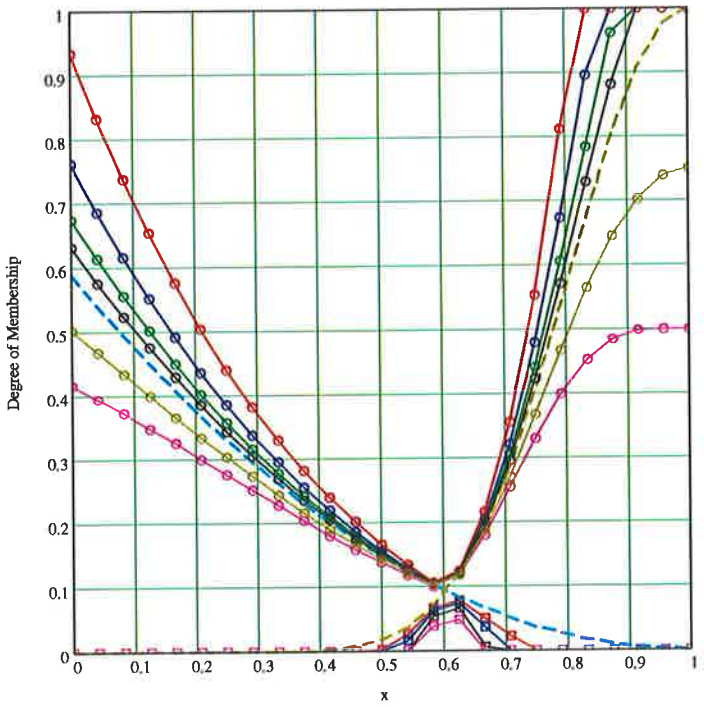
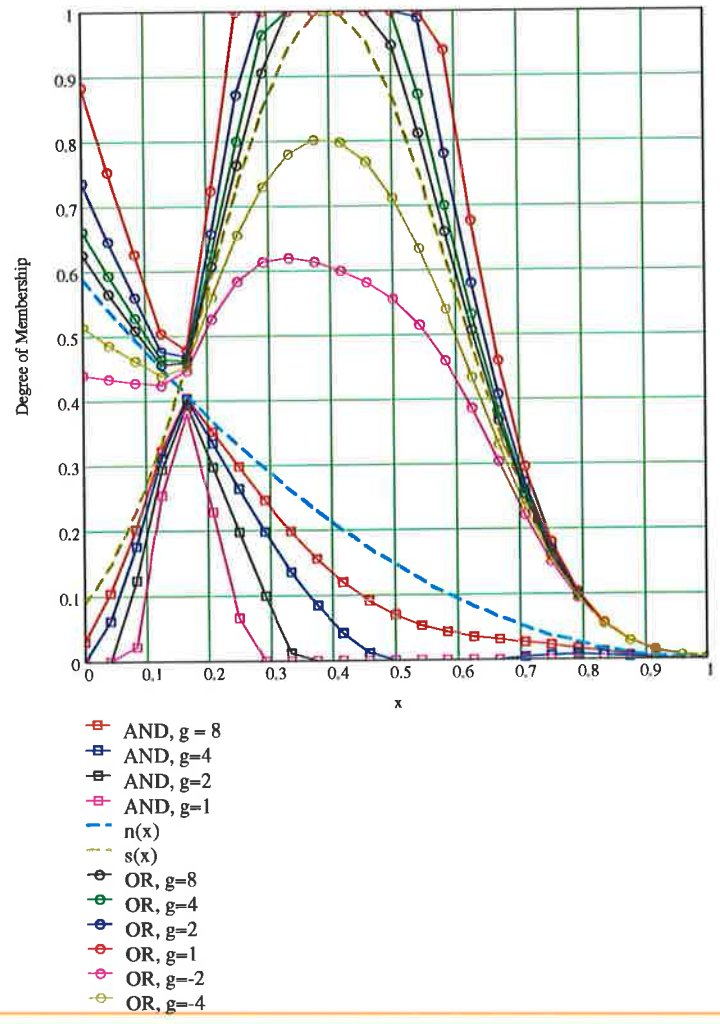


Figure 2.10d: Contour plot of the t-conorm. The actual values of x and y are shown on this plot.

Figure 2.10: Graph (a) showing an example of the proposed t-norm (M) and t-conorm (N). The colour blue represents the minimum of 0, and the colour red represents the maximum of 1.



Example 1



Example 2

Figure 2.11: Two examples where the proposed t-norm and t-conorm operate on functions n(x) and s(x) for various values of γ (shown as g). Function s(x) is changed to illustrate the affect of the operators.

2.8 Knowledge Encapsulation

An expert system relies on the knowledge of the expert or experts who program it, encapsulating the appropriate knowledge. The quantity and quality of knowledge and information required for an expert system will vary according to the task, and according to the standards by which the knowledge is collated, assessed, and represented in the system. In other words, the standards by which information is judged are variable. With this in mind, however, the work presented here assumes that a reasonable effort will be made for each task.

To fully model the behavior of a process by an expert system, the expert needs to be able to transfer an accurate picture of their knowledge of the process. This impacts directly on the specification for a user interface for an expert system. This knowledge must have a framework in which to reside, and must offer the human expert sufficient flexibility and features, to encode their knowledge accurately, and to allow a non-expert to make use of the system for data analysis. This thesis develops such a framework which is described in Chapter 4.

A secondary issue, but one that is still very important, is that of how knowledge is manipulated and used in a system. There is secondary knowledge such as how numbers and facts are represented in a system. This is called knowledge about knowledge, or meta-knowledge [2.1], and is addressed at the implementation stage of developing an expert system.

To encode an expert's knowledge, the IF-THEN production rule model is often used, as this provides a simple method of encoding knowledge in a linguistic form. That is, simple textual rules are written to describe the knowledge, rather than mathematical expressions. These rules are then processed in a mathematical way to arrive at an output. The rules used in fuzzy logic have the form:

- IF (X1 IS A1) AND (X2 IS A2) AND ..... AND (Xn IS An) THEN (Z1 IS B1)

where the left side of the rule is the antecedent and the right side is the consequent. The terms on the left are the rule premises, the Xi are the crisp inputs, and the Ai are the membership functions with labels such as high, low, medium. The terms (Xi IS Ai) are the rule premises. The label can just as easily identify a fuzzy number such as 30. The membership functions are selected according requirements of the system that is being modeled.

Another useful concept is that of Fuzzy Associative Memories (FAMs) [23]. These are usually illustrated as two dimensional maps between the input space and the output space. Figure 2.12 shows an example of a FAM. The input variables are shown along the axes, and the intersection of the inputs gives the appropriate output fuzzy set. FAMs can be used to design models for simple systems, as is shown later in chapter 8.

This thesis uses the fuzzy production rule (described in detail in chapter 4) as the means of encapsulating the expert's knowledge.

The membership function, whether specifically designed or selected from a range of available functions, conveys further knowledge about the process in question. The expert selects suitable functions that convey the desired meaning. The names, or linguistic labels, given to MFs are important when generating a rule base. To say that *'the voltage is high'* is easier to interpret than *'the voltage is function #3'*.

X Y	NL	NS	ZE	PS	PL
NL	ZE		PL		ZE
NS			PS		
ZE	NL	NS	ZE	NS	NL
PS			PS		
PL	ZE		PL		ZE

Figure 2.12: A Fuzzy Associative Memory. The inputs to the FAM are sets X and Y. The membership labels are NL, NS, ZE, PS and PL.

2.9 Fuzzy Inferencing

Fuzzy inferencing is the process of applying expert knowledge, which has been encoded in some way, to the processing of data, in order to obtain an output data set. There are a number of methods [1, 23] of performing fuzzy inferencing. The inferencing process is really two processes in series. The first process calculates a resultant fuzzy set which is the fuzzy representation of the output variable. The next step, is to obtain a non-fuzzy (crisp) value from this output set. This step is optional but is usually always implemented.

As with the inferencing process, there are a number of methods to perform the transformation from fuzzy-space to crisp-space. This process is called *defuzzification*, and is described in section 2.10.

As previously stated, a fuzzy rule has a *DOF* determined by the inputs to the rule. After the *DOF* has been calculated, the RMF is calculated by *modifying* the consequent membership function. Modifiers that are selectable with the GUI are shown in Table 2.4. Some well known methods to *evaluate* fuzzy rules are correlation-minimum encoding - with truncation, and correlation-product - with scaling [23]. An example of each method is shown in Figure 2.13. The power method has the affect of spreading the influence of the consequent membership function over a broader range.

The antecedent contains the conditional statements referring to the input variables, and describes the degree of fulfillment of the rule. The consequent is modified by the antecedent premise (see Figure 2.14).

Consider a single rule node [30] as shown in Figure 2.15. It can be seen that the process of rule evaluation can be decomposed into two high-level processes, the model for which directly follows the format of a fuzzy production rule. That is, there is an antecedent and a consequent process. This data flow diagram also shows library modules for storing the relevant membership functions for these components.

Function decomposition is a powerful method of analyzing a problem's structure, and hence determine data dependencies, and opportunities for parallel implementation.

Modifier	Description
(1) scaling :	$rmf[i] = z[i] \times dof$ $z[i]$ = consequent MF i = integer
(2) truncation :	$rmf[i] = z[i]$ if $z[i] \leq dof$ $= dof$ if $z[i] > dof$ (popular with VLSI implementations [13]of fuzzy processors)
(3) power:	$rmf\,[\,i\,] = z\,[\,i\,]^{(1-dof)}$ where : $dof \in (0,1)$

Table 2.4 :Modifier functions commonly used for transforming resultant membership functions.

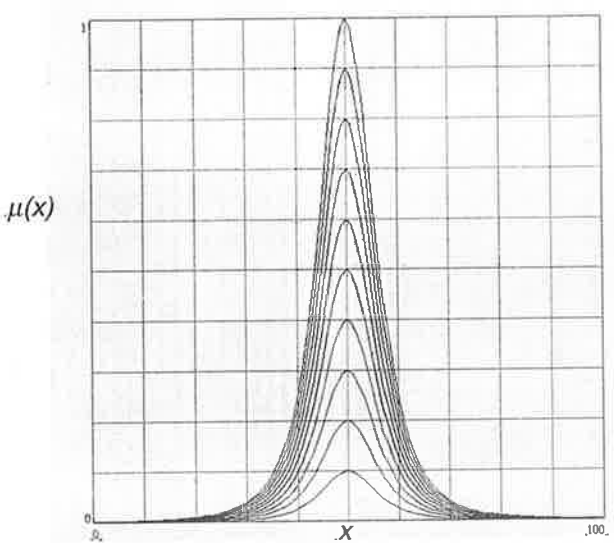


Figure 2.13a : Scaling modifier- the function is scaled by the DOF .

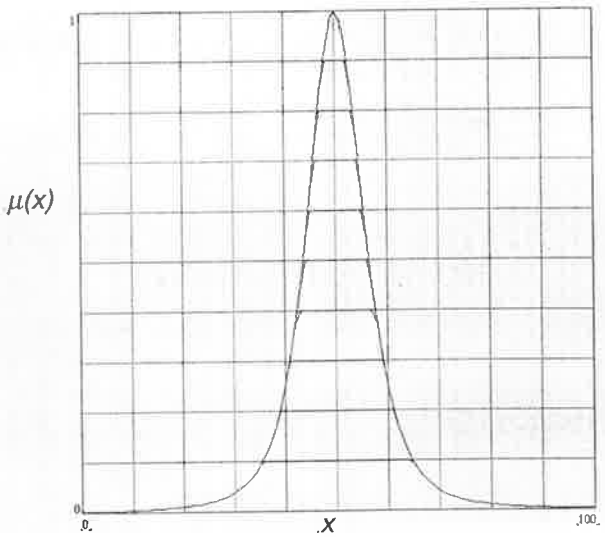


Figure 2.13b : Truncation modifier- the function is truncated at the level of the DOF.

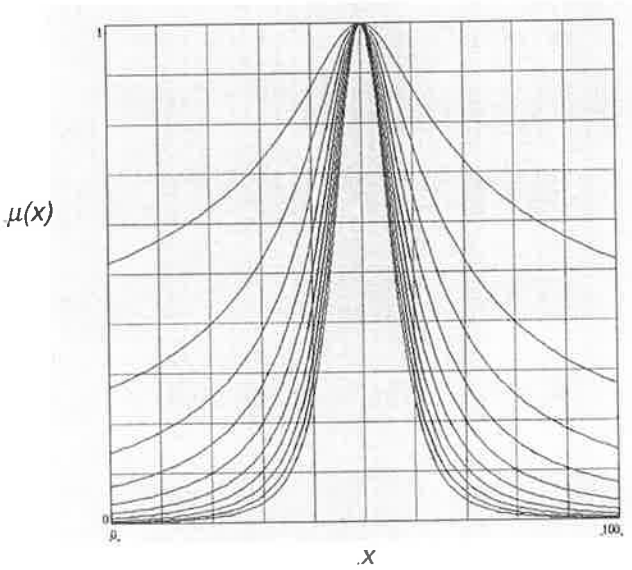


Figure 2.13c : Power modifier- as the power factor increases, the function spreads.

Figure 2.13: As the DOF of a rule varies, the consequent membership function, μ(x), will be modified accordingly. Figure 2.13a shows the affect of scaling as the DOF varies, and Figure 2.13b is the simple case of truncation. Figure 2.13c shows how the consequent membership function is spread as the DOF increases.

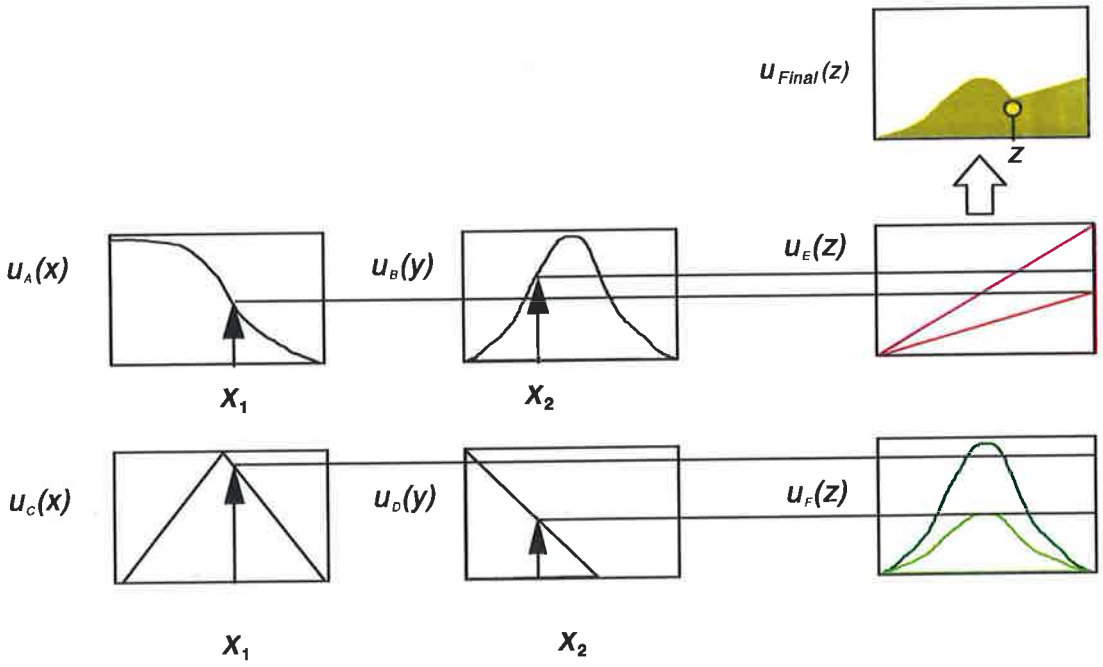


Figure 2.14a : Fuzzy Inference by Product-Sum Method. The inputs are applied to the antecedent membership functions, and the minimum (AND) of the two values is used to scale the consequent membership function. Rule fusion occurs by superimposing each resultant function on the same axes, and taking the maximum profile of each resultant function .

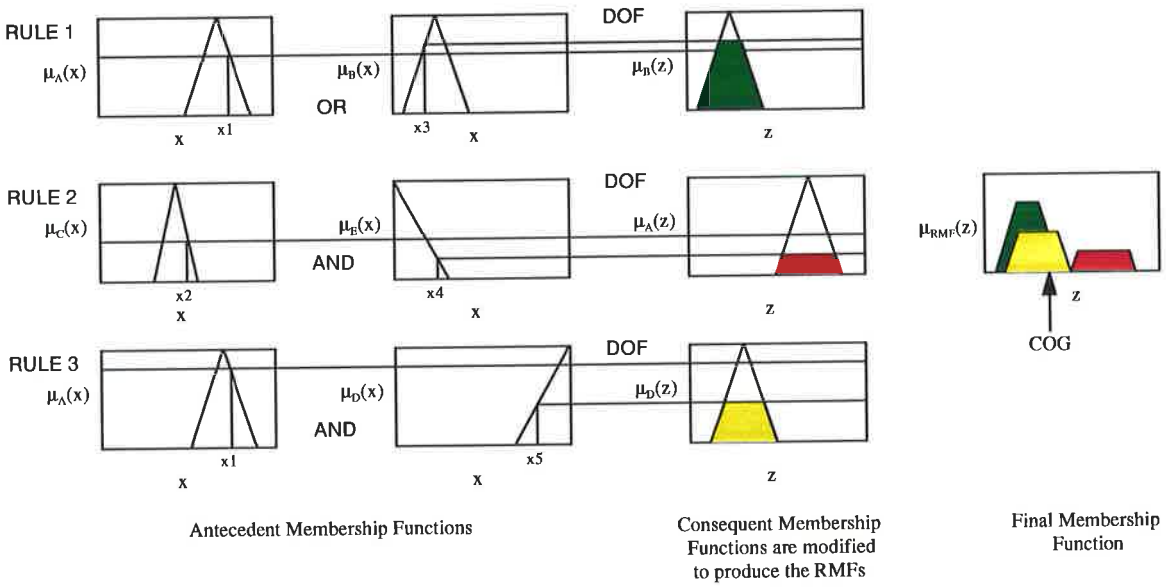


Figure 2.14b : Fuzzy Inference by Truncation-Maximum Profile Method, for a three rule system, with two premises per rule and one output. Center of gravity defuzzification is used to determine the crisp output value.

Figure 2.14: Examples of fuzzy inferencing using (a) scaling and (b) truncation methods.

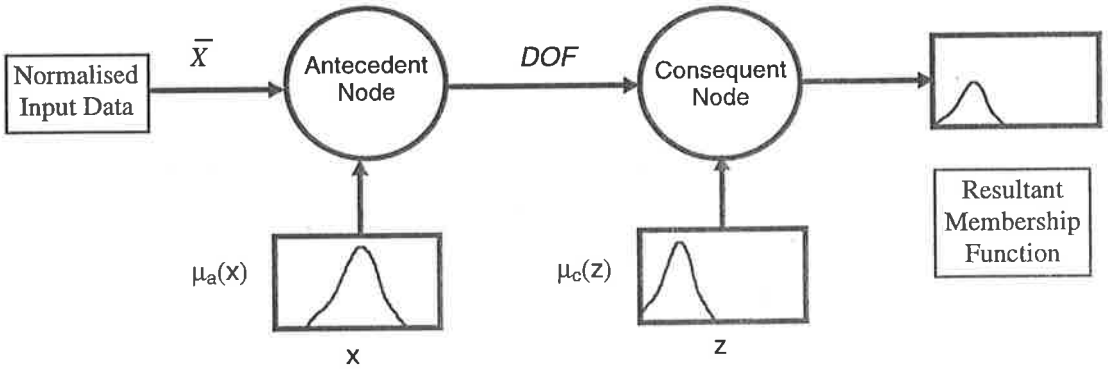


Figure 2.15:A Rule Evaluation Node comprises a process to evaluate the degree of fulfillment of a rule, and a process to evaluate the resultant membership function.

2.10 The Fusion Transform

2.10.1 A Summary and Description of Fusion Methods

The final membership function for a particular output variable, is determined by combining the resultant membership functions of a set of related rules. The process of combination is a data fusion operation.

When there are many RMFs to fuse, the output set may become congested, and the particular biases that each RMF introduces into the final membership function (FMF) may be swamped. This is the case where the maximum profile method is used to establish the bounds for the FMF set. There are several RMF fusion methods to choose from, some of which take this situation into account. The commonly used methods are described here, and an additional fusion operator is introduced.

One simple method builds the FMF by simply placing one set on top of another, over the selected universe of discourse. This method can lead to a $\mu > 1$ (fuzzy sets are usually normalize to some maximum value, often 1). The defuzzification process which follows, will take care of this anomaly. [Kosko p314]. This method is described in equation 2.8, illustrated in Figure 2.4.

$$z(k) = \sum_{k=0}^{N-1} \left[\sum_{r=0}^{P-1} \mu_r(k) \right] \qquad (2.8)$$

where k = 0,1,2,.... (integer)

N = number of intervals in the universe of discourse (integer)

r = 0,1,2,.... (integer)

P = number of rules for this RMF

$\mu_r(k)$ = resultant membership function for rule r.

Another method to calculate the FMF is to form the union of the RMFs using the maximum profile of the RMFs over the universe of discourse. This is expressed in equation 2.9.

$$z(k) = \bigcup_{r=0}^{P-1} \mu_r(k) \qquad (2.9)$$

Table 2.5 summarizes these fusion operations, that are often used for combining the modified consequent functions. The a_i are the values of the resultant membership functions at a particular value of k. Some examples of these methods are illustrated in Figure 2.16. (a_{ik} = result of RMF_i at value k). In the context of this research, the each a_i in these equations is the value of the RMF for a single rule, and n is the number of rules that contribute to the final membership function.

Fusion (Aggregation) Functions
Simple Summation $F_k(a_0, a_1, \ldots, a_{n-1}) = \sum_{i=0}^{n-1} a_{ik}$
Maximum Profile $F_k(a_0, a_1, \ldots, a_{n-1}) = \max(a_i)$
Generalized Means $F_k(a_0, a_1, \ldots, a_{n-1}) = \left(\frac{a_0^k + a_1^k + \ldots + a_{n-1}^k}{n}\right)^{1/k}, \quad k \in R \quad (k \neq 0)$
Geometric Mean $F(a_0, a_1, \ldots, a_{n-1}) = (a_0 \times a_1 \times \ldots \times a_{n-1})^{1/n}, \quad (n \neq 0)$
Arithmetic Mean $F(a_0, a_1, \ldots, a_{n-1}) = \frac{1}{n}(a_0 + a_1 + \ldots + a_{n-1}), \quad (n \neq 0)$
Harmonic Mean $F(a_0, a_1, \ldots, a_{n-1}) = \left(\frac{n}{\frac{1}{a_0} + \frac{1}{a_1} + \ldots + \frac{1}{a_{n-1}}}\right)$
Weighted Generalized Mean $F_\alpha(a_0, a_1, \ldots, a_{n-1}\ ;\ w_0, w_1, \ldots, w_{n-1}) = \left(\sum_{i=0}^{n-1} w_i\ a_i^\alpha\right)^{1/\alpha}, \quad (w_i > 0, \alpha \neq 0)$

Table 2.5: This figure shows the definitions for five fusion functions.

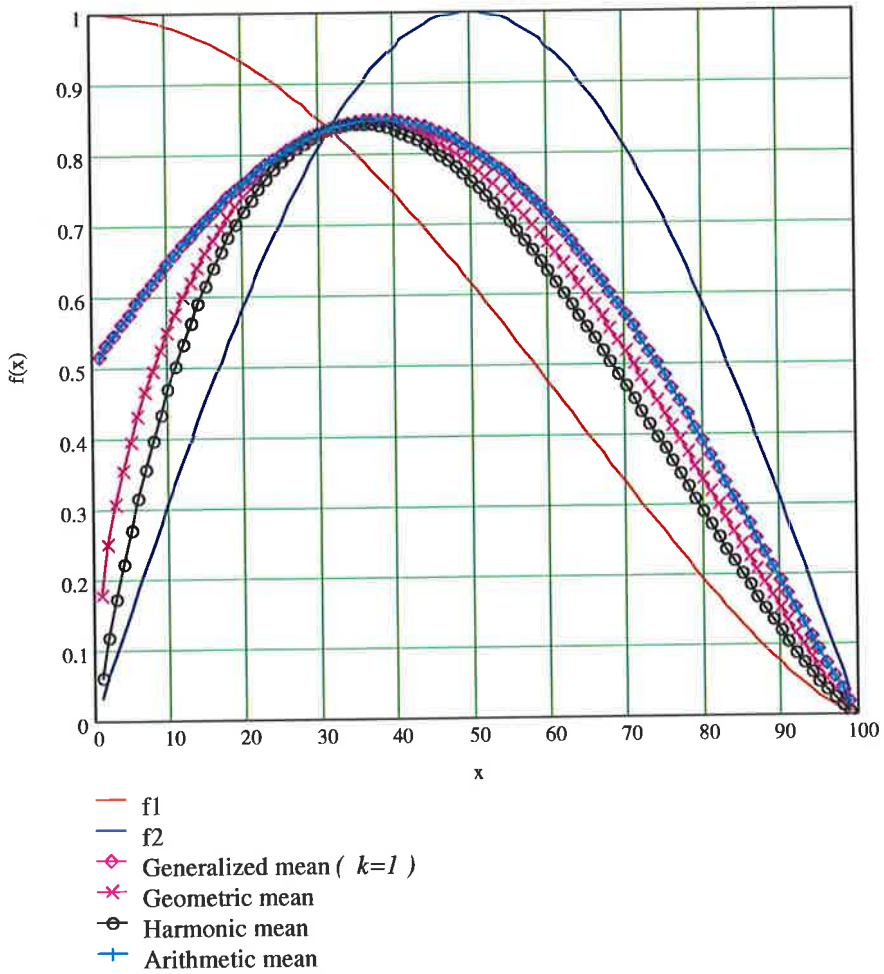


Figure 2.16: The fusion functions combine data with varying effects as shown in this figure.

2.10.2 *A Method to Determine Information Measure*

When information from various sources is combined to formulate a decision, one should be aware of the origin of the information and the degree to which the information segments may either agree or contradict each other. From information theory [31] there are various methods of measuring the information content of a system as new data is added.

With this present research, the fusion process that generates the Final Membership Functions, is an example of a process which is sensitive to the degree of agreement in the data. The greater the agreement between fuzzy sets, then the higher will be the information content of the fuzzy set formed by the fusion of multiple fuzzy sets.

It is here proposed that the RMFs which are combined to generate a single output fuzzy set, the FMF, should be treated with this premise as being central to developing a fusion method. This section deals with the derivation of a measure of fuzzy information, and then applies this to the formulation of an alternative fusion function for fuzzy sets.

Definition 2.1

Define a group of ordered fuzzy sets $M = \{\mu_0(x), \mu_1(x), \mu_2(x), \ldots, \mu_{P-1}(x)\}$

where $\mu_J(x)$ is a fuzzy set, and $x \in X$

and X is the support of M. P = number of sets which divide the domain of X.

then set index $i \in [0, P-1]$

Figures 2.17 a & b show the representation used for this definition. Notice that each set overlaps at most one other set in Figure 2.19a. This will not always be the case, as shown in Figure 2.19b where the triangular functions have varying height and spread parameters. The membership functions may also have different forms such as gaussian or *S* shape, in which case these parameters will affect the outcome.

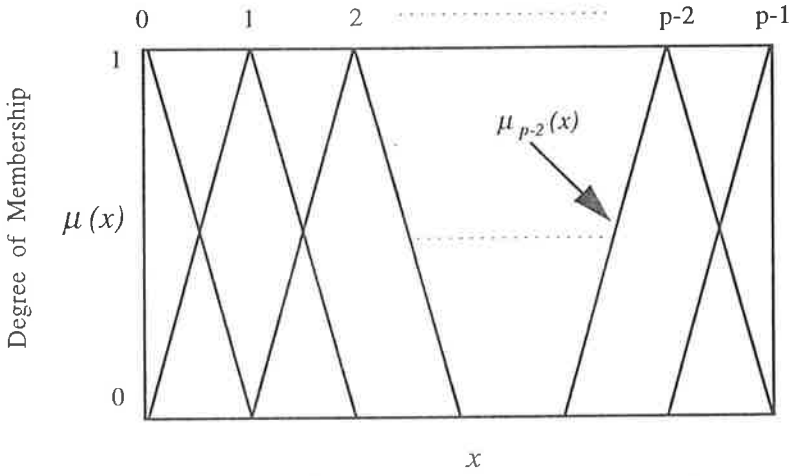


Figure 2.17a: This diagram illustrates the ordered fuzzy sets that constitute the membership functions that may be define for a particular inferencing system.

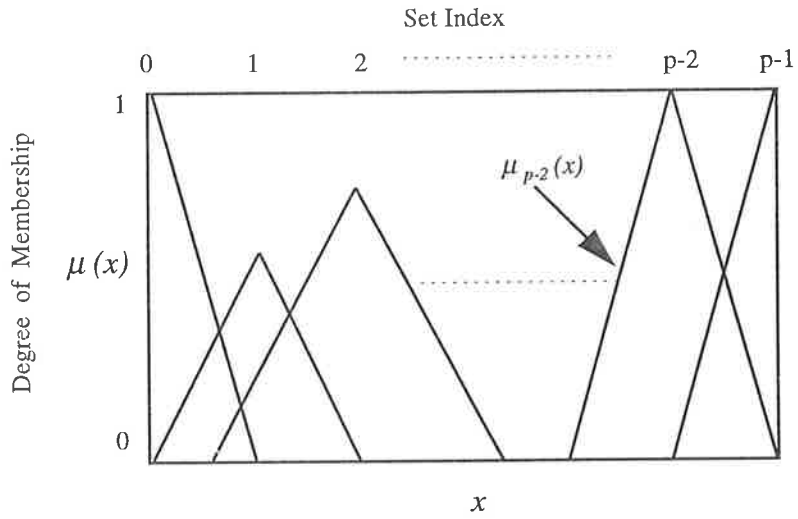


Figure 2.17b: This diagram illustrates the ordered fuzzy sets where the RMFs have varying amplitude and spread. Notice that sets 0, 1 and 2 overlap each other.

This index is now used to compare how well sets *agree* with each other. In combining the RMFs, the index of the modified consequent is now required.

Definition 2.2

Define an Agreement Matrix $A_{i,j}$, the elements of which are the degree to which set ***i*** and set ***j*** agree in their respective propositions[4].

$$A_{i,j} = \frac{\left[P - \left| L_i - L_j \right| \right]}{P} \tag{2.10}$$

where $i, j \in [0, P-1]$ are integers, and L is a vector that maps the RMF number i, to its location in X. A is a P x P matrix.

Definition (2.2) states that the closer that two set are to each other, the greater will be the agreement factor A. Likewise, as the two sets move further apart, their mutual agreement decreases. This leads to the definition of the inverse operator for A, namely the Contradiction Factor C.

Definition 2.3

Define a contradiction factor : $C_{i,j} = 1 - A_{i,j}$ (2.11)

Clearly, when i = j, the two sets agree exactly with $A_{i,j} = 1$ and $C_{i,j} = 0$.

This definition of the agreement matrix ignores the magnitude of the contribution of each set, which is the outcome of the computation of the Degree of Fulfillment ω of the antecedent of the fuzzy rule. This factor can be accounted for by weighting the elements of A accordingly.

Let $$\hat{\omega} = \begin{Bmatrix} \omega_0 \\ \omega_1 \\ \cdot \\ \cdot \\ \cdot \\ \omega_{P-1} \end{Bmatrix}$$

Then a matrix A^C can be defined that compensates for the DOF of each RMF.

[4] The situation here is illustrated as: *Output is Low*, compared with *Output is High*.

Definition 2.4

Define a Compensated Agreement Matrix A^C as follows:

$$A^C{}_{i,j} = \frac{\left[P - \left| L_i \cdot \omega_i - L_j \cdot \omega_j \right| \right]}{P} \tag{2.12}$$

Equation (2.12) says the following:

1. If two sets i and j are to be combined, with $|L_i - L_j| > 0$, that is, non-identical propositions, and $\omega_i \geq \omega_j$, then the agreement between i and j will increase because j is less dominant than i. (ie. the opposition of j to i is weakened)
2. If $\omega_i = \omega_j$, then the agreement is determined by the relative distance between the sets.

$$A^C{}_{i,j} = \frac{\left[P - \omega_i \cdot \left| L_i - L_j \right| \right]}{P} \tag{2.13}$$

1. As both $\omega_i \to 0$ and $\omega_j \to 0$ then even if they are mutually exclusive premises (eg. ON vs OFF) then they will not decrease the agreement value.
2. If only $\omega_i \to 0$ and ω_j remains constant then agreement will increase, as set i becomes less dominant and the contradiction decreases.

2.10.3 Important Properties of Fuzzy Information Measure

Let a and b be two fuzzy sets, then the similarity between a and b is defined as S(a, b).

Reflexivity

$A_{i,j} = 1$ iff $i = j$

Testing equation 2.3 against this property, gives:

$$A^C{}_{i,i} = \frac{\left[P - \left| L_i \cdot \omega_i - L_i \cdot \omega_i \right| \right]}{P} = 1$$

Naturally if two sets are equal, then they are exactly similar.

Symmetry

$A_{i,j} = A_{j,i}$

With these definitions in place, the fuzzy information measure can be derived. A reasonable assumption is that information is additive, hence;

Definition 2.5

Define Fuzzy Information Measure (FIM) as:

$$\mathrm{FIM} = \frac{1}{\mathrm{N}} \sum_{i=0}^{P-1} \sum_{j=0}^{P-1} A_{i,j}^{C} \tag{2.14}$$

where A^C is the compensated agreement matrix with N entries.

An Illustrative Example

Given $$A^C = \begin{pmatrix} 1.0 & 0.3 & 0.1 \\ 0.3 & 1.0 & 0.5 \\ 0.1 & 0.5 & 1.0 \end{pmatrix},$$

then $$FIM = \frac{1}{9} \cdot (1.0 + 0.0 + 0.1 + 0.3 + 1.0 + 0.5 + 0.1 + 0.5 + 1.0) = 0.53$$

This number indicates a moderate level of confidence in the information. A value of 1 would indicate the highest level of confidence and a value of 0, the lowest level.

2.10.4 *Proposed A-Matrix Based Fusion Method*

Another method of performing data fusion is now described. In this method, proposed by the author, the agreement matrix is applied to the RMFs. Discrete data sets are assumed in this method. The key idea is to combine RMF data and accounting for the relationship between the RMFs.

Algorithm 2.1 for RMF fusion

1. Form the compensated agreement matrix A^{C} of Rank P

2. For each column j of the P x M data matrix

   2.1 Form the column vector c

   2.2 Form c^T

   2.3 Calculate $y_j = c^T . A^{C}$

   2.4 Track $y_j max$ (to normalise Z)

3. Calculate $Z = (Y / ymax) . a$

   where a = scale factor

Example 2.2

Given $$A^C = \begin{pmatrix} 1.0 & 0.4 & 0.2 \\ 0.4 & 1.0 & 0.1 \\ 0.2 & 0.1 & 1.0 \end{pmatrix},$$

The defuzzified output is calculated by the Center of Gravity method. The crisp output for the proposed fusion method is 6.816. The crisp output for the arithmetic average method is 6.526.

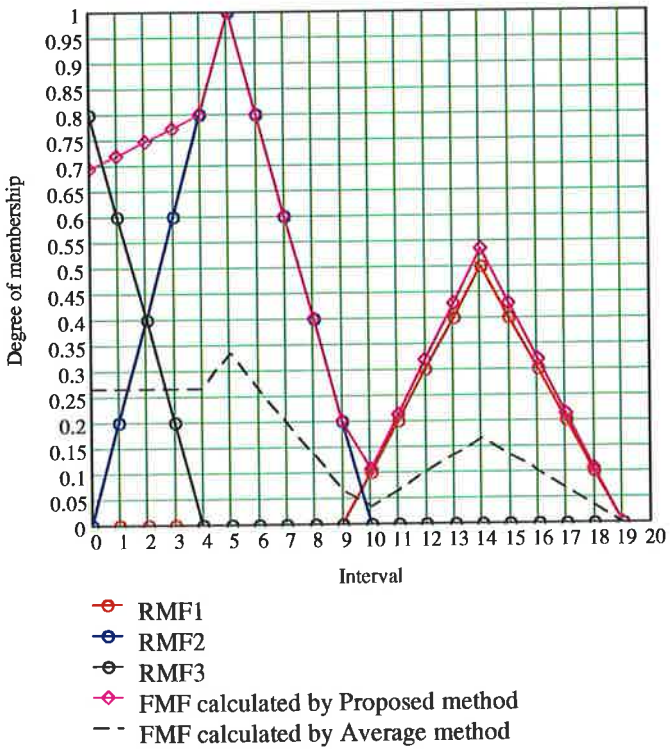


Figure 2.18: This diagram illustrates the proposed fusion method, being applied to 3 Resultant Membership Functions. The arithmetic average of the 3 functions is also shown as a comparison.

2.10.5 *Proposed Sliding Window Averaging Based Fusion Method*

This method is based on the idea of the simple moving average from statistics. This method defines a window of width w ($w < N/2$), which slides across the data set. The data points which lie within the window are averaged to produce a representative data point. The data can further be weighted by a function W(x`). The complexity of this weighting function will have a direct influence on the processing performance, and so it should be selected with this constraint in mind.

Figure 2.19 illustrates the operation of the SWA algorithm. It can be seen that the center of gravity for each FMF varies with the selection of window width and weighting function, and becomes less sensitive to abrupt variations in the data (noise) as the window width increases.

Algorithm 2.2 for SWA fusion

For P RMFs each with N intervals,

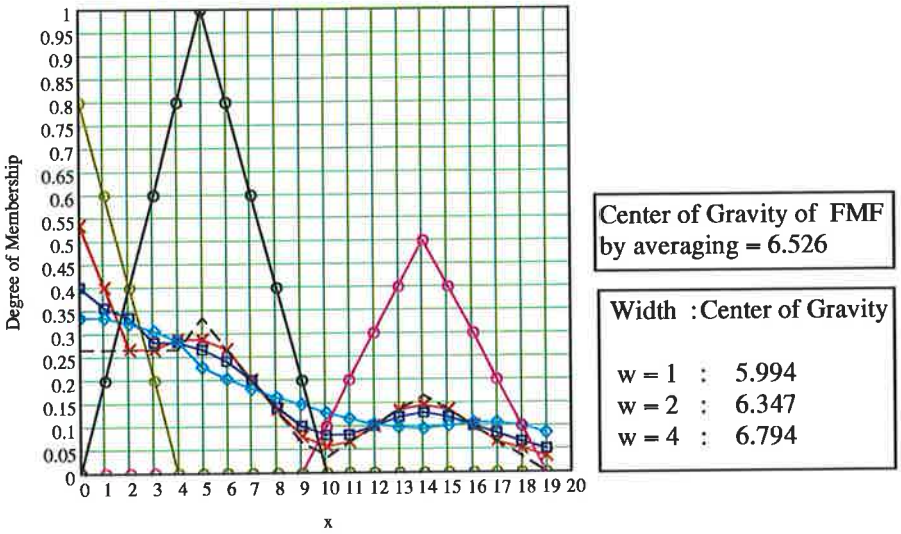
1. Form the data matrix D. Each row of D is the transpose of an RMF vector.
2. Select a window width w: $0 < w \le \frac{N}{2}$
3. Select a window weighting function W(x`)
4. For $i = 0$ *to* $N-1$:

$$if \quad i \le w: \qquad FMF_i = \frac{\sum_{j=0}^{i+w}\sum_{k=0}^{P-1} D_{k,j} \cdot W\left(\frac{|i-j|}{2 \cdot w}\right)}{(i+w) \cdot P}$$

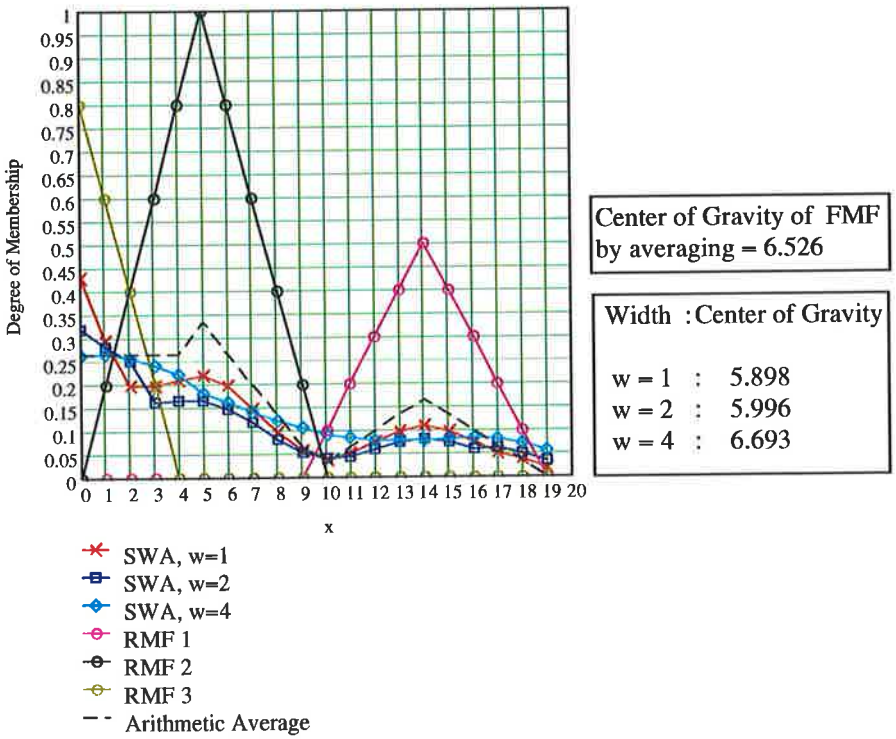
$$else\ if \quad w \le i \le N-w: \qquad FMF_i = \frac{\sum_{j=i-w}^{i+w}\sum_{k=0}^{P-1} D_{k,j} \cdot W\left(\frac{|i-j|}{2 \cdot w}\right)}{(2 \cdot w+1) \cdot P}$$

$$else: \qquad FMF_i = \frac{\sum_{j=i-w}^{N}\sum_{k=0}^{P-1} D_{k,j} \cdot W\left(\frac{|i-j|}{2 \cdot w}\right)}{(w+N-i) \cdot P}$$

End For Loop.



Example 1 : W(x) = 1



Example 2: $W(x) = e^{-x}$

Figure 2.19: Two examples of the SWA fusion method being applied to 3 RMFs to form the corresponding FMF. The window widths w, are as shown, and the weighting function $W(x)$, is set to unity for example 1, and $e^{-x'}$ for example 2. The arithmetic average of the 3 functions is shown as a comparison.

2.10 Defuzzification

The process of converting data from the fuzzy domain to the crisp domain is called defuzzification. The principle idea is to convert a fuzzy set representation of a number, to a non-fuzzy value that best represents that fuzzy set. There are numerous methods that have been proposed to achieve this transformation, the most common of which are listed in Table 2.6.

The center of gravity method is perhaps the most intuitive, and is most commonly used for defuzzification. Other methods may be used based on their simplicity to implement in hardware. The maximum method, where the crisp value is defined by the point at which the fuzzy set reaches its maximum, is the simplest to implement in software.

If there are multiple maxima, then an average of the maxima is taken, and this value then determines the crisp output. This method is known as the mean of maxima method.

Each method involves a different amount of processing, and so the computing time for each will vary accordingly. For a fuzzy expert system that processes data for a *real-time* application, the choice of inferencing method, is an important consideration.

An example illustrates the defuzzification process. In this work, the processing times for the defuzzification of a 1000 point[5] final membership function, running on a single 20MHz T800 Transputer, have been determined and are shown in Table 2.7.
The center of gravity method performs multiplication, summation, and a division operation to determine the crisp value. The maximum method however, simply compares each data point of the fuzzy set with the previous value, keeping track of the maximum. This process involves logical comparisons and storage operations. From Table 2.7, it can be seen that there is significantly more processing required for the COG method.

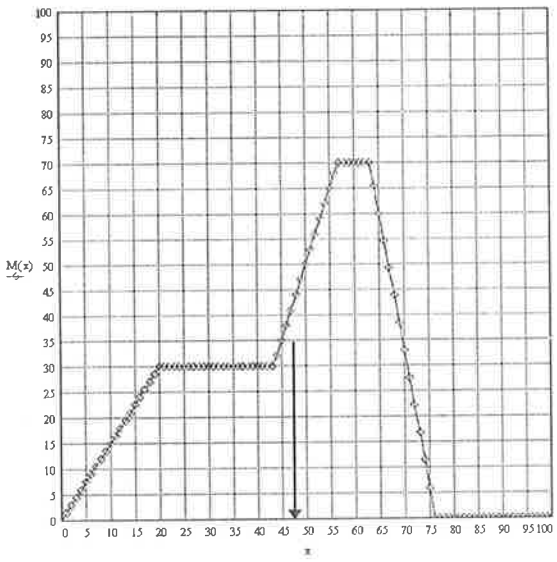
[5] This refers to a discrete, rather than a continuous representation of a membership function.

(1) Center of Gravity
$$z = \frac{\sum_{k=0}^{N-1} F(k) \cdot k}{\sum_{k=0}^{N-1} k}$$ where $F(k)$ = composite membership function, z = center of gravity of F, k = interval value
(2) Maximum
$$z = k \mid Max\,[F_j(k)]$$ where $F_j(k)$ = j'th Final membership function, z = value of k for which F is a maximum
(3) Mean of Maxima
$$z = \frac{\sum k_i^{max}}{r}$$ where k_i^{max} are the values of k for which there are maxima. r = number of maxima

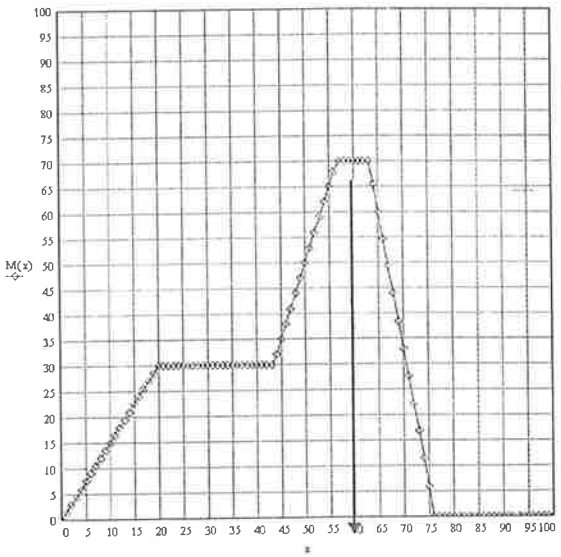
Table 2.6 : The methods of defuzzification will impact on the computation required and hence the time to produce the crisp output. The center of gravity method is widely used.

Defuzzification Method	Time (mS)
center of gravity	4.8
maximum	2.9

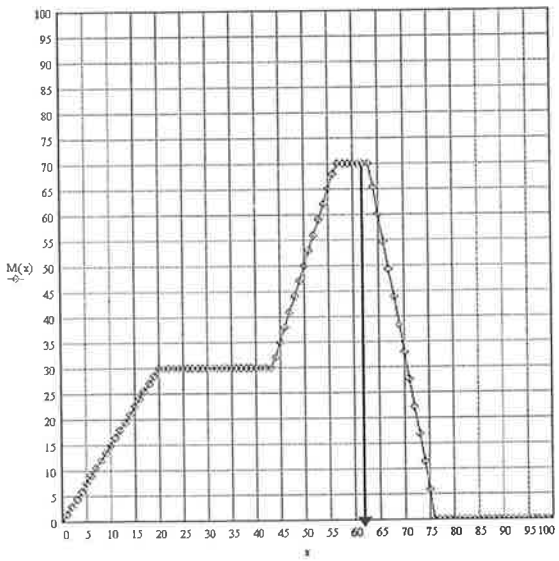
Table 2.7 : Defuzzification times for a 1000 point fuzzy set.



a) Center of gravity method:



b) Mean of maxima method: The algorithm averages the maximum values, hence the result coincides with the mid-point of the peak.



c) Maximum method: The algorithm tracks the largest value of the discrete fuzzy set and returns the value of x at which it occurred..

Figure 2.20: These diagrams show examples of the defuzzification methods for a particular final membership function.

2.11 Adaptive Fuzzy Systems

An adaptive fuzzy system is defined here as one which modifies its behavior, based on a feedback mechanism, according to a performance measure. In fuzzy control situations, the difference between the desired system outputs and the actual system outputs are used to adjust the controller to best match the desired outputs.

There are many methods of introducing adaptive behavior to a fuzzy system, among which are the following;

1. modify the rule weights
2. modify the membership functions
3. adjust the parameters for the connectives
4. add or delete fuzzy rules from the rule base.

Rule weight adjustment is implemented by multiplying the *DOF* of each rule by a factor w. As the processing progresses, the weights are adjusted for best effect (ie. reducing the error).

Adjusting the membership functions is also a legitimate method of system tuning. It could be argued that the membership functions should not be altered as they embody an important component of the expert's knowledge. However, it is just as valid to say that the expert may not know the precise shape that will deliver the optimal performance. Certainly, it would be reasonable to place limits on the degree to which a membership function, and indeed a rule weight factor, may be changed.

As shown in section 2.3, the choice of connective has an effect on the inferencing process.

2.12 Fuzzy Logic and Time

When one considers time in the processing of data, the concepts of past, present, and future trends in the data are naturally introduced. A data processing system that uses feedback from the output data to the input data space, is able to modify its behavior based on the outcome of processing of the previous input data. If the controlling algorithms of such a causal system are realised as a collection of fuzzy rules, which use the fuzzy verb IS, then this places a limitation on the author of the rules, to constrain their knowledge of the processing, to the immediate time instant. A system that simply processes input data and produces an output

data set, is an example of a non-causal system. That is, a system where there is no feedback[6] and has no sense of time. A system that relies on direct feedback from a plant, is an example of a causal system. This type of system is often encountered in fuzzy control, and shows the importance of the temporal aspect of a system's behavior.

Consider now the fuzzy rule as introduced in section 2.7. The fuzzy rule operator *'IS'*, refers to the present and conveys no sense of history or future. The author of a rulebase will have greater flexibility in expressing the desired controlling rules for a system, if the grammar supports additional fuzzy linguistic operators. To this end, two additional operators are introduced and their functions defined.

If a system is to make use of the past events of its behavior, then another operator is required. This operator, here called a temporal fuzzy operator, is '*WAS*'. Likewise, to suggest some predicted behavior, it would be useful to have an operator that conveys a sense of the future. This operator is called '*WILL BE*'. At this point the operators are defined, with definition 2.3 being the current interpretation of the IS operator, and definitions 2.6 and 2.7 being proposed by the author. The inferencing software has provision for implementing this operator in the form of a history buffer that contains a record of the previous output results. The WILLBE operator can be implemented as a separate process, where the calculations for forecasting of future data can be done.

Definition 2.6

For a membership function $\mu_A(x)$, then the fuzzy temporal operator *IS* is defined such that:

if the premise $P_i = x_i$ IS A then:

$\alpha_t = \mu_A(x_i[k])$, where k = integer,

is the *k'th* sampled input, and α_t is the degree to which this premise P_i is satisfied.

Definition 2.7

For a membership function $\mu_A(x)$, then the fuzzy temporal operator *WAS* is defined such that:

if the premise $P_i = X_i$ WAS A then:

$\alpha_t = \mu_A(x_i[k\text{-}1])$, *where* k = *integer*, > 0,

and α_t is the degree to which this premise P_i is satisfied.

Definition 2.8

For a membership function $\mu_A(x)$, then the fuzzy temporal operator *WILLBE* is defined such that:

if the premise $P_i = X_i$ *WILLBE A* then:

[6] It is worth noting that the user of the system may modify the knowledge base of the processing system as the result of a particular outcome, thus giving a form of feedback.

$\alpha_i = \mu_A(x_i[k+1])$, where $x_i[k+1]$ is the predicted value for the input x_i.

ie. $x_i[k+1] = P(\mu_A(x_i[k], k)$

where P is a predicting or forecast function.

$k = integer, > 0$, and α_i is the degree to which this premise P_i is satisfied.

In the case of an expert system that encodes domain knowledge in the form of production rules, these additional temporal operators are introduced to provide more flexibility for the expert, so they may more freely encapsulate their knowledge, and express temporal aspects of their experience of the system in question.

Examples of how these operators can be used are now examined. First consider the rule that expresses the behavior of a temperature control system.

- IF temperature IS high AND temperature WAS low THEN control IS low

This rule tests two premises;

1) temperature is at this moment high, and

temperature was low when the measurement was last made (ie. Apply previous temperature input to the low membership function.)

Another rule that considers two separate input variables is now considered.

- IF x1 is high OR x2 WILL BE low THEN z1 IS high

This rule tests two premises;

1) input variable x1 is at this moment high, and

2) input variable x2 will be low at interval $k+1$, one sample step into the future.

(ie. Apply input *x2* to the forecast process to determine the value of ω for this premise.)

The *WILL BE* operator provides a mechanism to forecast an event based on experience and observation of established trends (as in the case of a control room operator for a power station).

The interpretation of the operator '*WAS*' is defined here as being the previous value. To achieve this, a history buffer of past outcomes is kept. For the operator '*WILLBE*', the interpretation is more complex.

There are many ways of defining a look-ahead operator. A forecast [29] that is based on a model that has been derived from previous data, is known as univariate. Methods that fall into this category are:

- Extrapolation based on fitting a curve to the data, using polynomial, exponential, or power curves.

$$\tilde{x}(N,k) = F(x_N, x_{N-1}, x_{N-2}, \ldots) \qquad (2.15)$$

where N is the time at which the forecast is made, and k is the number of time intervals F is a function that models the data.

- Exponential Smoothing is used where there are no apparent trends in the data. A forecast can be derived by taking a weighted sum of the previous data samples.

$$\tilde{x}(N,1) = c_0 x_N + c_1 x_{N-1} + c_2 x_{N-2} + \ldots \qquad (2.16)$$

where $\{c_i\}$ are weights which are chosen to apply more weight to recent data and less weight to older data. Chatfield [29] suggests a set of geometric weights having the following form:

$$c_i = \alpha(1-\alpha)^i \qquad \text{and} \qquad i = 0,1,\ldots \qquad (2.17)$$

The calculation of a prediction will impact on the processing performance of the expert system. As the model becomes more complicated, as in the case of a high order polynomial curve fit, more processing effort is required.

temperature WAS high => $\mu_{temp}(k\text{-}1)$

x2 WILL BE medium => $\mu[(x2\{predicted\})]$

where $x2\ \{predicted\} := Q\ .\ (x2(k\text{-}1)\ x2(k))$

In a parallel processing architecture, the prediction of the next input value would be done by a processor that handles all data transactions with the data source. In the architecture

that is describe in this study, such a prediction process would map onto the Database Manager (described in Chapter 5). This module would ideally be assigned to its own Transputer.

2.13 Chapter Summary

This chapter has described the basic concepts of fuzzy logic. Fuzzy set operations, the membership function, and methods for performing fuzzy inferencing have been presented. A number of alternatives exist for fuzzy operators, and for combining fuzzy rules, and some examples of the effects of these operators on fuzzy sets have been examined.

Their are many choices when it comes to implementing fuzzy logic to model a system, and this chapter has provided an overview of the subject, so as to give an appreciation of the effects of these various choices. A t-norm and a t-conorm operator have been proposed, and examples have been used to illustrate their properties.

An information measure has been developed for fusing fuzzy sets. This measure has been applied to develop another fusion operator, for which examples have been evaluated.

A second fusion operator has been illustrated which is based on a sliding window average of the data. This method of fusing fuzzy sets has also been illustrated by way of an example.

New operators have been introduced to attempt to handle temporal aspects in fuzzy rule development. These operators have been proposed to add to the flexibility of knowledge encapsulation.

Chapter III

DEVELOPMENT OF A FUZZY INFERENCING ALGORITHMIC STRUCTURE

3.1 Introduction

This chapter builds on the concepts introduced in Chapter 2 about fuzzy logic, and presents the development of a structure and algorithms that perform fuzzy inferencing in a parallel domain. The structure refers to the data processing and data flow, not to a physical device.

3.2 Evolution of an Inferencing Algorithmic Structure

3.2.1 Operational Constraints and Requirements

Chapter 1 introduced an overview of the expert system that is developed in this thesis. There are many parts to such a system, and it is necessary to develop a consistent and efficient framework or structure in which these components can function.

The expert system will operate within particular constraints. Likewise, there are particular requirements or expectations that should be realised. In formulating such parameters, it is perhaps useful to consider how one person relays information to another in order that some task may be performed. With this in mind, the following list presents issues that are important in this context. The issues include:

1. the expert system requires a mechanism for interacting with the expert (visual prompts), and can store the required expert knowledge in a flexibly manner (memory)
2. knowledge can be changed or updated
3. the expert system has the task of processing data. (Once information is exchanged and complete, it is acted upon.)
4. data processing should occur within a *reasonable* length of time[1]
5. a parallel processing paradigm is adopted

[1] Relative to the time constants that are relevant to the system in question.

6. various inferencing strategies may be employed and evaluated (experimentation)
7. results may be viewed in some convenient form.

The expert system requires configuration, if any choice is to be offered to the user in the way that data is to be processed. There must be a means of user interaction with the system, where rules can be generated and evaluated, and processing options selected. These points are used to guide the development of the expert system. Knowledge transfer and information display is dealt with in greater detail in Chapter 5.

Once the knowledge is entered into the system, it needs to be managed. This knowledge will comprise the initial expert information plus the results of subsequent processing of data. This is achieved by incorporating a process that manages the knowledge flow in the system. Let this be called the Knowledge Base Module (KBM).

Next, the data from sensors, that is to be processed, needs to be managed. A process is assigned to this task, and is called the Data Base Module (DBM). Once the data is available, it is processed by the inference engine. Call this the Fuzzy Inference Engine (FIE). Table 3.1 summarises the major components of the this expert system. This table defines the function of each of the components. This functional division seems logical, though other interpretations would be possible.

The time taken to process the input data will depend on the method chosen. The basic procedure here, is to apply the rule base to the input data, and determine an output. The two methods of processing considered here are:

1. rule by rule evaluation
2. look-up table based evaluation

The first method evaluates each rule in turn, and requires the FIE to understand the text of each rule. This leads to the concept of compiling the rule base, to produce an executable list of instructions that the FIE can understand.

The second method, pre-calculates all possible outcomes, and stores the results in a multi-dimensional look-up table. Clearly, the size of the table is affected by séveral factors which include:

1. the number of inputs that may be referenced in the antecedent of each rule
2. the memory capacity of the host processor
3. the number of increments in the fuzzy sets - for a discrete fuzzy set as described in Chapter 2, section 3.

Component (Module)	**Function**
User Interface	• rulebase generation • display data • display inferencing options • display results of processing
Supervisor	• oversee and supervise data processing • coordinate message transfers throughout the structure
Data Manager	• store, collect, and disseminate data for processing • store results of processing
Knowledge Manager	• store, collect and disseminate knowledge of the process to be modeled
Inference Engine	• process data according to the available knowledge
Rule Node	• calculate the RMF for the rule passed to it by the FIE

Table 3.1 : The components of the expert system each have specific functions to perform.

The number of rules in the rule base does not influence the size of such a table, but does impact on the time to calculate the outputs for the table. For a look-up table approach, as illustrated in figure 3.1, the memory capacity m required may be calculated as follows:

Assume a multiple input- single output system:

Let p = number of inputs to the system

q = number of outputs to the system

s = support (range) of the fuzzy sets with $x_i(k) : k \in [0, s]$

b = number of bytes to represent the inputs and outputs (ie. 8 bit data gives rise to 256 levels to each input).

then, for each output:

$$m = s^{p} \cdot b \tag{3.1}$$

and for a system with q outputs, (3.1) becomes

$$m = q \cdot s^{p} \cdot b \tag{3.2}$$

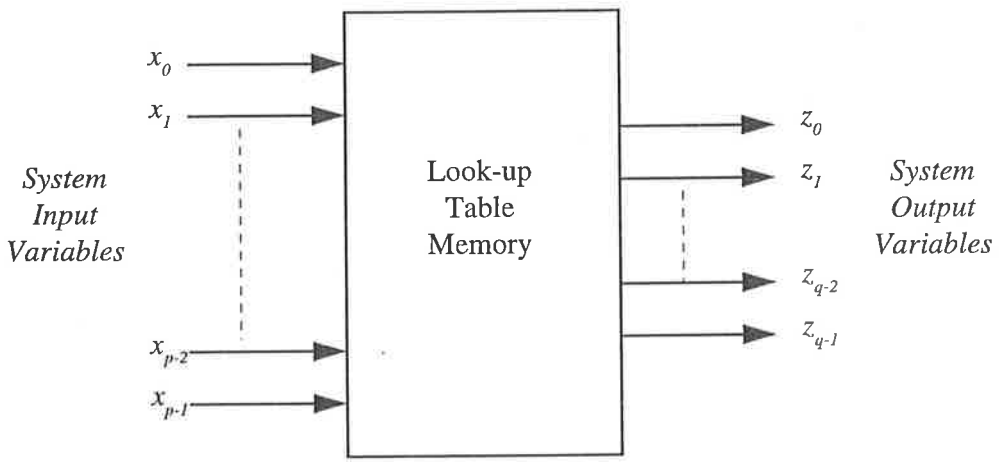


Figure 3.1: A look-up table for a multiple input- multiple output fuzzy system.

Example 1: $s = 100$

$p = 2$

$q = 1$

$b = 1$

from (3.2), $m = 1 \cdot 100^{2} \cdot 1 = 10k$ bytes

Example 2: $s = 100$

$p = 4$

$q = 1$

$b = 1$

from (3.2), $m = 100^4 = 100M$ bytes

Assume a multiple input- multiple output system:

Example 3: $s = 100$

$p = 4$

$q = 4$

$b = 1$

from (3.2), $m = 4 . (100^4) = 400M$ bytes

Clearly, as the complexity of the system increases, by increasing s, p, q, or b, the memory requirements for pre-calculated output values, increases exponentially.

Restricting the number of input variables that may be used in combination with each other, limits the diversity and flexibility of the rule representation. The user of the system would have to reformulate their knowledge in terms of the restrictions. For example consider a case where three or more inputs are required to properly describe the experience of the expert.

Example:

X1 IS A1 AND X2 IS A2 AND X3 IS A3 AND X4 IS A4 THEN Z IS B1

The system may not exhibit behavior that the expert can simply decompose into a number of rules with only two inputs per rule. Thus the structure needs to retain a high degree of knowledge representation flexibility. Hence rule by rule evaluation is suitable.

3.3 An Algorithm for Rule Evaluation

Given that the expert system accepts rules in the form shown above, a processing structure is required that supports this form. A simple single rule processing node [30] is

shown in Figure 3.2, where input data is processed by the inference engine, according to the knowledge contained in the knowledge base.

Figure 3.3 shows how the knowledge base can be split into two separate modules. The first contains the rules to process the data, and the second contains information about how the inference engine will actually perform that processing. The inferencing structure must be able to support a method for altering these options. This also impacts upon the design of the GUI, described later in Chapter 5, as it must provide a mechanism for offering the user a selection of inferencing procedures[2].

The rule node is now examined to define the data transformations that occur. This top-down decomposition of process function is used as a method to develop the inferencing structure. By using the fuzzy production rule as the model, the rule node is split into two processes, one to handle the antecedent part of the rule, and the other to handle the consequent part of the rule.

[2] As described in chapter 2.

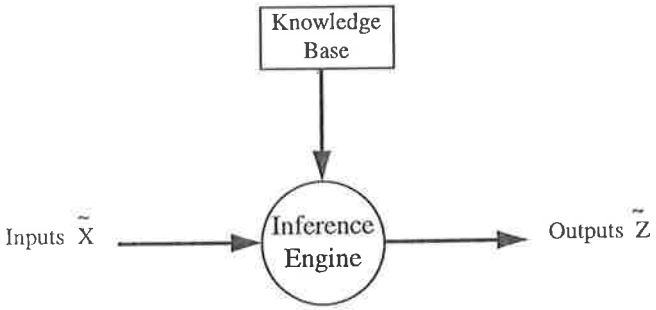


Figure 3.2: A single inferencing node uses rules and other knowledge, contained in the knowledge base, to process the input data. The Data Flow Diagram (DFD) does not relate information about timing, just data transformation.

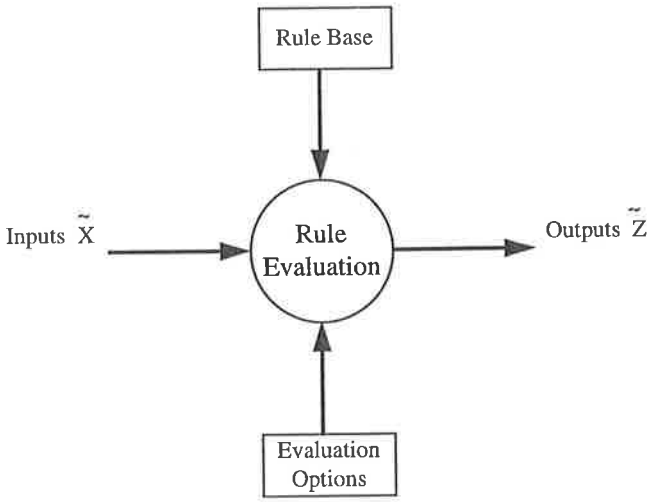


Figure 3.3: The inference node comprises a single rule evaluation node. The knowledge base comprises the rules base and other system information. An additional source of information is required that sets the inferencing options in the Rule node.

With reference to Figure 3.4, there are sets of data that are required by the antecedent and consequent processes. This data is derived from the expert who initially defines the rule base, and the user, who selects various inferencing options to process their data according to their requirements. This data is stored in separate locations, depending on where it is required. The output from the rule evaluation process is the resultant membership function for each rule as it is read from the rule base.

It would be desirable to evaluate all the rules in a rule base concurrently. If there are N rules, this would require N processing elements capable of evaluating a fuzzy rule each. Figure 3.5 shows a data flow diagram where all rules belonging to the same output set[3], are evaluated, and then combined to produce an output. The input data is distributed to each rule node, where the resultant membership function is calculated. These functions are then combined by the fusion node, to form the final membership function. Lastly, defuzzification gives the crisp output value for the rule set.

The FMF is a fuzzy set that represents the outcome of processing the input data by the rule base. The next step is to transform this fuzzy set into a crisp number. This is achieved by the defuzzification process. There are many methods of performing the fusion and defuzzification transformations. Each of these processes requires a store to contain configuration information, and these are shown in the DFD.

With the Occam programming language, the data flow diagrams are readily implemented, whereby each process bubble of the DFD becomes an Occam process, and each data channel becomes an Occam channel

Each rule will be represented by its own process. This approach will require that the number and form of each rule is known before compiling the source code, as Occam is a static programming language. This would be less of a concern in the case of a static rule base, where the rules remain constant. An example of such a system, is in an embedded controller application, where the processor performs the same task repeatedly, and the environment is *well understood.*

For the case where the rulebase is dynamic, as is often the case with systems which adapt to their environments, then an alternative approach, is a more appropriate choice. This alternative approach involves a single fuzzy rule node which can access a rule base. It accepts input data as before, but now it applies all the rules in its rule base to that data, before accepting the next input data.

[3] A set of fuzzy rules that applies to a particular output.

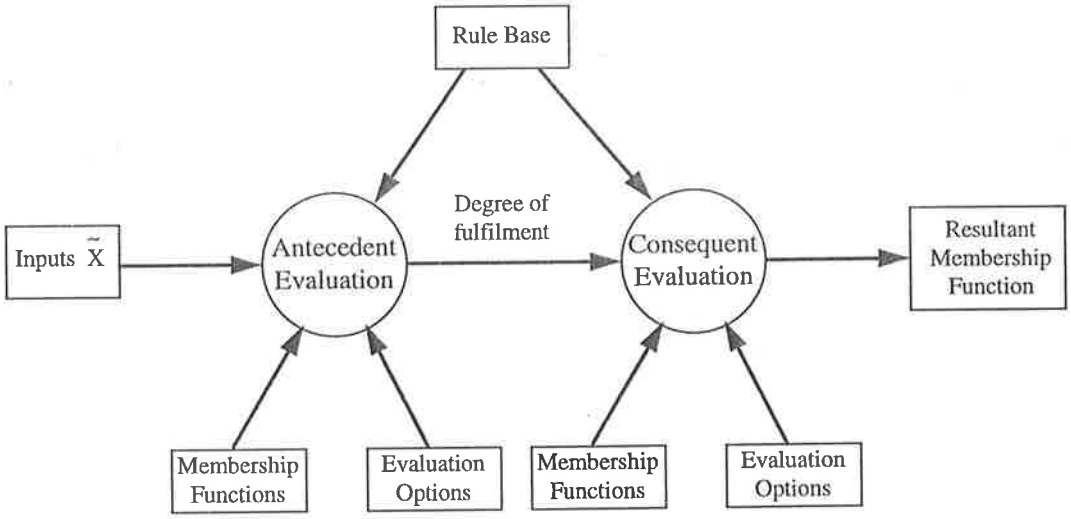


Figure 3.4: The rule node comprises lower level processes. The first is the antecedent process, which calculates the degree of fulfillment (DOF) for the rule. The DOF determines the extent to which the consequent mf is modified. The result is the resultant membership function for the rule.

The structures developed in this chapter comprise processes, data path ways, and data storage elements. The data storage elements perform two functions. The first is to store the appropriate configuration information required for each processing element. The second is to store the results of processing.

The support structure which is developed and described in Chapter 5, must allow the user to manipulate all resources of this structure.

Algorithm 3.1 defines the processing stages, the output of which are the RMFs for the rule base. These RMFs are then combined to form the final membership function. Figure 3.4 shows a DFD where there are multiple rule processes. Indeed, there could be as many processes as rules. The fusion process collects all the RMFs and generates the FMF.

This processing strategy relies on presenting the fuzzy rules to the inferencing engine in a particular format. The next section describes this format and the tool by which linguistic rules are transformed.

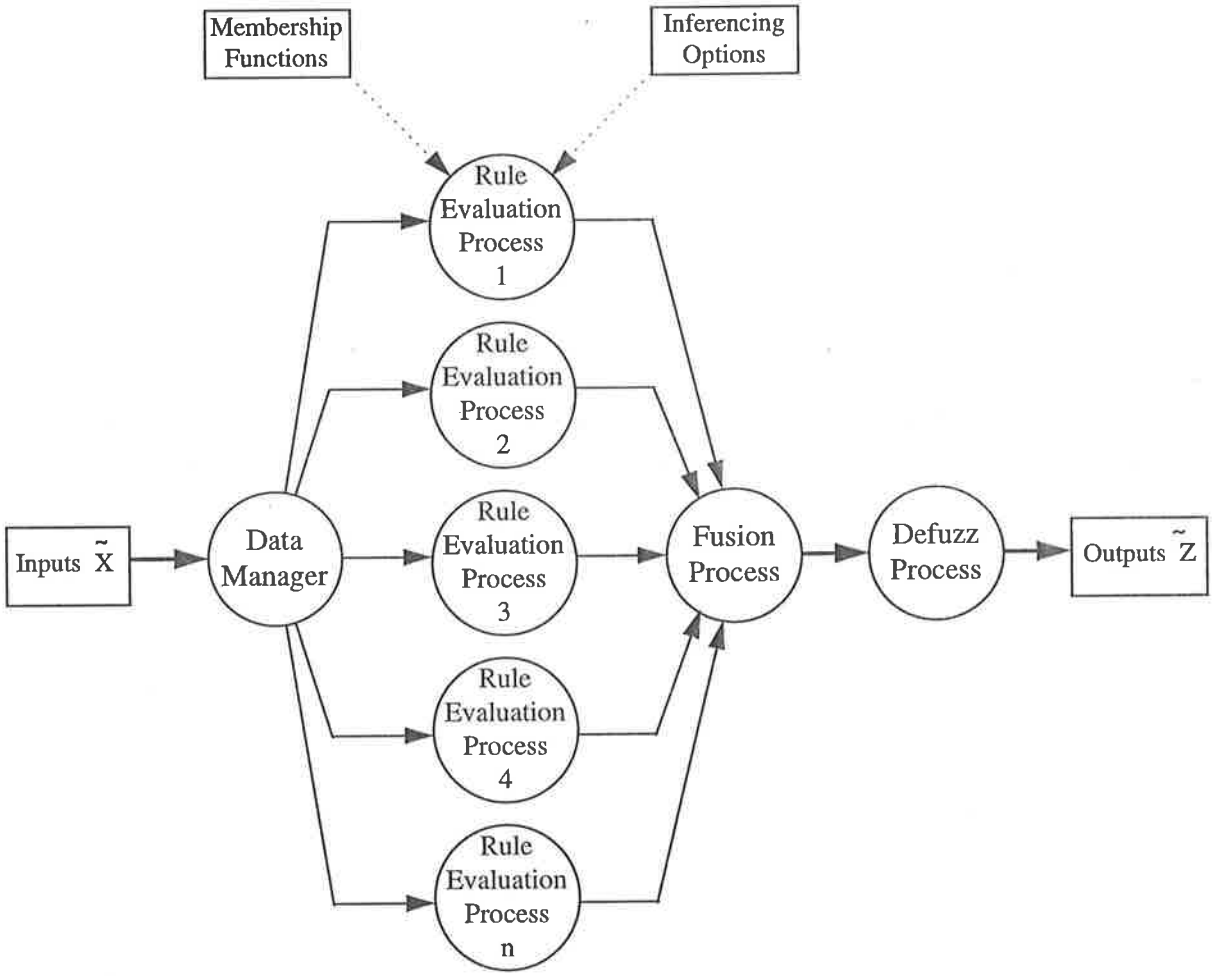


Figure 3.5: A parallel processing system may be achieved by calculating each fuzzy rule simultaneously. This is achieved by assigning a single rule node process to each rule in the rule base. For *n* rules this will require *n* processes running concurrently.

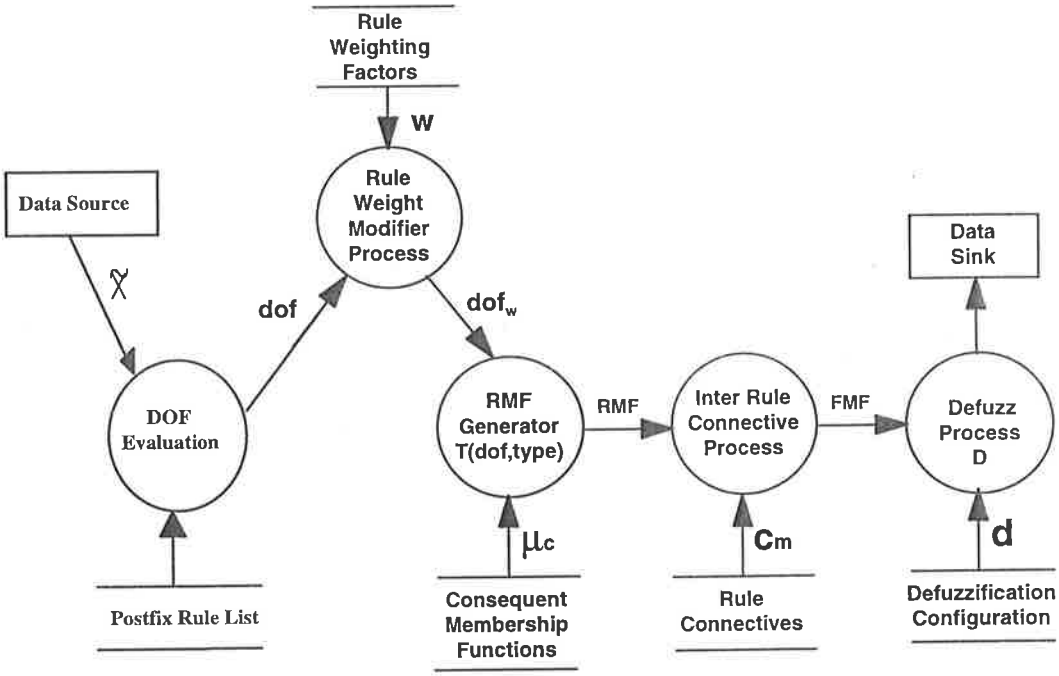


Figure 3.6 : The DFD for fuzzy inferencing that is adopted in this study, evolves from the logical sequence of events for processing a fuzzy rule, and defines the structure. Each of the circles represents a stage of processing, whilst the directed lines represent the flow of data between the processes. This DFD includes additional processes for; (i) collecting the RMFs and fusing them, and (ii) transforming the fuzzy sets to crisp output values (defuzzification).

Algorithm 3.1: Rule base evaluation

BEGIN

while the processor is in run mode

SEQ

Get input data vector $\hat{x} = x_0, x_1, \ldots, x_{n-1}$

SEQ i = 0 for Number of rules in the rulebase

... calculate dof_i for this rule

... calculate the weighted *dof* for this rule

... calculate resultant membership function (*RMF*)

... calculate weighted resultant membership function (*WRMF*)

... store RMF in data base

... store WRMF in data base

SEQ j = 0 for Number of outputs

... calculate final membership (FMF_j)

... store FMF_j in data base

... defuzzify FMF_j

... send crisp outputs to DBM

... check system state, held in KBM

END.

3.4 The Fuzzy Rule and the Rulebase Compiler

3.4.1 Description

To facilitate the process of generating fuzzy rules, a rule editor has been developed that allows the user to compose textual rules for the rule base. The syntax of the rule has been developed to be simple, yet capable of expressing and encapsulating the user's logic. This syntax is shown in Figure 3.7. The linguistic rule is translated (compiled) to a form that has been developed for processing by the inferencing machine.

The syntax of the rule in Backaus-Naur [32] form is as shown in Figure 3.8. This leads to the format of the rule data (which is a representation of the textual rule) that is processed by the fuzzy inference module.

The strategy for processing the rulebase is to evaluate the degree of fulfillment, and then calculate the resultant membership function (RMF), for each rule. The RMF's which belong to the same output function are combined by another process, to create the final final membership function. This function is defuzzified to form the crisp output value.

This strategy led to the idea of using a postfix representation of each rule. The major advantages of this method are;

1. the inference machine can accommodate any length of rule.
2. the syntax of the textual rule is very flexible
3. this strategy enables farming of rule evaluation to multiple processors.

A simple rulebase comprising two rules is shown in Figure 3.4. The output from the rule compiler shows how the text of each rule has been transformed. This example incorporates the use of brackets and linguistic hedges, and shows how these rules are analysed.

This new representation of the rule is now in a form that can be processed in a serial manner by the FIE. The FIE executes each sub-statement as it is encountered, and stores the results in an internal stack. The output of the FIE is a Resultant Membership Function.

3.4.2 Operation of the Rule Compiler

As described in the previous section, the rule compiler translates the textual rules into a form that is suitable for processing by the inference engine. The rules must obey the syntax as described in Figure 3.7.

```
<rule>          := <rule no.> IF <antecedent> THEN  <consequent>

<rule no.>      := <integer>

<antecedent>    := <premise> | <premise> <connective> <antecedent>

<premise>       := <input variable> <temporal operator> <hedge> <membership>

<temporal operator> := {WAS | IS | WILL BE}

<consequent>    := <output variable> IS <hedge> <membership>

<hedge>         := { Absolutely | Very | Almost | Quite | NOT   }

<connective>    := { AND | OR }
```

Figure 3.7: The proposed syntax for the fuzzy rule as used in this thesis comprises a number of components. This form fully describes the legal rule structure which the rule compiler can translate into an executable sequence of parameters for the inference engine.

```
<Rule List>       := <rule number> <output number> <operator list>

<rule number>     := <integer>

<output number>   := <integer>

<operator list>   := <operator type> <operator value> <operator list>

<operator type>   := <integer>

<operator value>  := <integer>
```

Figure 3.8: The rule list syntax describes the output from the rule compiler. For each rule in the rulebase, there will be a corresponding rule list which is executed by the inference engine.

TransFuzien can process compound fuzzy rules, such as:

- IF (X1 IS A1) AND ((X2 IS A2) OR (X3 IS A3)) THEN Z IS B1

which then becomes:

- X1 A1 iS X2 A2 IS X3 A3 IS OR AND Z B1 IS .

Another example is:

- If (angle IS small) and ((position is zero) or (position is small)) then force is medium

after parsing (the first stage of rule compilation) this becomes:

- If angle small IS position zero IS position small IS or and force medium IS then

Hence, a compound fuzzy rule can be represented in a postfix notation.

To illustrate how this process works, the Rule Editor has been used to produce four fuzzy rules with varying degrees of complexity. These rules have then been processed by the Rule Compiler to produce the corresponding rule lists. The results are shown in Figures 3.8(a) to (d). The first rule is a simple example of a single input-single output statement, incorporating a linguistic hedge. The second rule (b), comprises multiple inputs and a linguistic hedge.

The next two rules incorporates brackets, and are examples of compound rule statements. The corresponding rule list shows how the brackets have been accounted for in the rearrangement of the logic representation.

Each rule in the rule base is compiled and converted to a Postfix [32] representation. This new form eliminates the need for brackets that would otherwise be required to define the *scope* of logical operations. The Postfix fuzzy rules then become a part of the knowledge base, where they are available to the Inference Engine for later processing. This compiling process preserves the fuzzy logical properties of the rules and their operators.

The input and output variables, and the membership functions are each assigned unique identifiers by the software as each rule is compiled. An ordered postfix rule list formed, that is then passed to the inference engine for evaluation.

The OPCODEs are derived from the syntax for the fuzzy rule. The first number represents the opcode for the inference engine, and the second part represents additional

information for the opcode. This is similar to micro-code for conventional microprocessors [33]. Each opcode has a data field that is used to identify particular membership functions, or I/O variable mapping. An entry of *99* indicates a fuzzy operator such AND, OR, and NOT.

OPCODE	DATA	Description
0	99	IF
1	n	input, source
2	99	IS
3	n	input membership function
4	n	hedge, type 3
5	99	AND
6	99	OR
7	99	NOT
8	99	THEN
9	n	output, sink
10	n	output membership function

Table 3.2 : There are 11 opcodes which the inference engine interprets.

Rule List	Textual Interpretation
0 => 99	IF
1 => 0	input 0
3 => 1	membership 1
4 => 2	hedge 2
2 => 99	IS
9 => 0	output 0
10 => 3	output membership 0
4 => 3	hedge 3
2 => 99	IS
8 => 99	THEN

(a) IF x0 IS Very Big+ THEN y IS Quite Small+

Rule List	Textual Interpretation
0 => 99	IF
1 => 0	input 0
4 => 1	membership 1
3 => 2	hedge 2
2 => 99	IS
1 => 2	input 2
4 => 4	membership 4
2 => 99	IS
5 => 99	AND
9 => 0	output 0
10 => 0	output membership 0
2 => 99	IS
8 => 99	THEN

(b) IF x1 IS Very Big+ AND x2 IS Zero THEN z1 IS Large+

Rule List	Textual Interpretation
0 => 99	IF
1 => 0	input 0
4 => 1	membership 1
2 => 99	IS
1 => 2	input 2
4 => 4	membership 4
2 => 99	IS
1 => 4	input 4
4 => 2	membership 2
2 => 99	IS
6 => 99	OR
5 => 99	AND
9 => 1	output 1
10 => 3	output membership 3
2 => 99	IS
8 => 99	THEN

(c) IF x1 IS Big+ AND (x2 IS Zero OR x3 IS Med+) THEN z2 IS Small+

Rule List	Textual Interpretation
0 => 99	IF
1 => 0	input 0
3 => 2	membership 2
4 => 4	hedge 4
2 => 99	IS
1 => 1	input 1
3 => 4	membership 4
4 => 5	hedge 5
2 => 99	IS
1 => 0	input 0
3 => 1	membership 1
2 => 99	IS
6 => 99	OR
5 => 99	AND
9 => 0	output 0
10 => 0	output membership 0
2 => 99	IS
8 => 99	THEN

(d) IF x1 IS Slightly Medium+ AND (x1 IS Almost Zero OR x3 IS Big+) THEN y IS Large+

Figure 3.9: The rule lists produced by the rule compiler include information about the type of operation to be performed, and parameters that specifies the input source and the output destination. These parameters are used by the data management system of the inference engine.

3.5 Chapter Summary

In this chapter, a method has been proposed for representing the fuzzy production rule in the form specified by the syntax of Figure 3.7. The advantages of this form have been discussed, and sample outputs that were generated by the compiler are included. The issue of how best to process a rule base has been addressed, and a solution proposed. This is described by algorithm 3.1.

Processing performance can be enhanced by suitable selection of a repetitious component of the data processing, and farming this component to as many available Transputers as possible.

Chapter IV

THE GRAPHICAL USER INTERFACE

4.1 Chapter Purpose

This chapter describes the Graphical User Interface (GUI) that has been developed to provide configuration and control of the software suite, and of the hardware resources introduced in Chapter 1. The preprocessing of data is addressed and each of the dialog screens are described.

4.2 Function of the GUI

The functional development of the GUI was guided by the requirements of the inferencing structure, and ease of use. The GUI provides the means to:

1. compose fuzzy rules for the rule base,
2. compile the rule base,
3. create membership functions,
4. select inferencing options for the inference engine,
5. view results and,
6. control resources attached to the transputer interface module.

The GUI is written in C++ [34], and runs on an IBM compatible PC under the Microsoft Windows operating system. This program is event driven from the user's perspective. That is, an event is generated by the program each time the user interacts with the GUI via the mouse or keyboard. The subsequent command packet is transmitted to the Transputer network via the link interface on the B008 motherboard. The GUI is served by the program running on the Transputer network.

The GUI comprises several software modules, each of which performs a specific task. The names of the files and their description are listed in Table 4.1.

There are several dialog boxes that the user can interact with, to enter the fuzzy rules, and to configure the system. Some of the most important features will be described here. The GUI has been designed to keep the process of interacting with the inferencing system as simple as possible.

The main control panel of the GUI, shown in Figure 4.1, displays the following information;

1. The DOFs of each rule are displayed.
2. The values of the inputs are displayed.
3. The rule weights are displayed.
4. Graphs of the resultant membership functions can be selected for display.
5. Graphs of the final membership functions can be selected for display.
6. The output variable names and their crisp values are displayed.
7. The inferencing process can be stopped, and then continued.
8. Hardware resource controls (via the Transputer Interface Module).

Module Name	Description
fuzzyapp.cpp	the main module
details.cpp	generates a window for entering textual information about the current project
b008cdlg.cpp	generates the control window and handles the communications with the Transputer system
iodlg.cpp	generates the input and output variables definition window
ruldlg.cpp	generates the rule editor window and contains the rule structure definitions
rcomp.cpp	generates the compiler window, and compiles the rulebase
equtndlg.cpp	the membership function editor
infrncmd.cpp	generates the inference methods selection window

Table 4.1: Software module definitions.

Listings[1] 4.1 and 4.2 illustrate how the program responds to events generated by the user. The Inferencing methods can be transmitted at any time, even whilst the inference engine is processing the rulebase. The current cycle is completed first, then the new methods are employed. There are some lines of code, commented out, which were used during the commisioning of the software. These lines generated message boxes on the screen of the PC, that indicated the progress of the processing.

[1] A full listing of the GUI program software was not included, for clarity, due to the large size of the files.

```
void B008CommDlg::BNLoadRulesClicked ()
{
int i, j;
int comms_status = 0;
   //Tell the system how many rules there are.
   comms_status = OutByte(t_number_of_rules); // Send message header followed by
   OutWord(NumberOfRules);     // the number of rules
   // MessageBox("NUMBER OF RULES SENT", "Information", MB_OK);
   //Tell the system how many outputs there are.
   comms_status = OutByte(t_number_of_outputs); // Send message header followed by
   OutWord(NumberOfOutputs);     // the number of outputs
  // Send the rulebase record
   for(i=0; i<NumberOfRules; i++)
     {
      comms_status = OutByte(t_add_rule);    // add a rule
     // MessageBox("ADD RULE TOKEN SENT", "Information", MB_OK);
      comms_status = OutWord(i);            // send rule identity number
      comms_status = OutWord(80);           // send size = 2*40
     //These 40 pairs of integers comprise a single rule list
      comms_status = OutWord(i);/*send rule number*/
      comms_status = OutWord(RuleBase[i].RuleOutput);/*send output number*/
     for(j=0; j<39; j++)
       {
        comms_status = OutWord(parsed_rule_buffer[i].symbol[j]);/*send data*/
        comms_status = OutWord(parsed_rule_buffer[i].value[j]); /*send data*/
       }
     }
    if( comms_status < -290)
        {
        MessageBox("COMMUNICATIONS FAILURE !!!!", "Information", MB_OK);
        }
    else
        {
        MessageBox("RULEBASE SENT", "Information", MB_OK);
         }
}
```

Listing 4.1: The "load rulebase" routine.

```
void B008CommDlg::BNConfigureEngineClicked ()
{
int comms_status = 0;
   comms_status = OutByte(t_inference_methods); /* send config. selections */
   comms_status = OutWord(ProjectInfo.ConnectiveMethod);      // send connective
   comms_status = OutWord(ProjectInfo.RuleModifierMethod);    // send modifier
   comms_status = OutWord(ProjectInfo.RuleFusionMethod);      // send fusion
   comms_status = OutWord(ProjectInfo.DefuzzificationMethod); // send defuzzification
         if( comms_status < -290)
           {
           MessageBox("COMMUNICATIONS FAILURE !!!!", "Information", MB_OK);
           }
         else
           {
         MessageBox("INFERENCE ENGINE CONFIGURED","Information", MB_OK);
    }
}
```

Listing 4.2: The "configure inference engine" routine.

The procedure in using TransFuzien is outlined in Figure 4.2. Firstly, establish the requirements of the task, and then use the graphical user interface to create a knowledge base. The knowledge base is then converted to a form suitable for interpretation by the inferencing software, that operates by processing a single rule at a time. The user can select an appropriate inferencing method for the task in hand, by using the inference selection dialog box.

In a typical session with the package, the user can produce a new Project File that has facilities for entering the name and a brief description of the project (Figure 4.3). There are a number of pre-defined membership functions and hedges, but input and output variables must be entered before rules can be created by the Fuzzy Rule Editor (Figure 4.4). This editor allows rules to be built by a simple point-and-click method. A Membership Function Generator (Figure 4.5) is available to the user, to create their own discrete membership functions.

Once again, this is achieved by entering an equation by a point-and-click method. The equation is displayed for verification purposes. The function is normalised so the maximum value is one.

As pointed out in Chapter 2, there are a number of methods for interpreting fuzzy operators. TransFuzien provides a number of these methods for the user to select. Figure 4.4 shows a dialog box that lists methods for;

1. interpreting fuzzy *AND, OR*, and *NOT*,
2. modifying the consequent membership function,
3. combining or fusing rules that belong to the same rule set, and defuzzification.

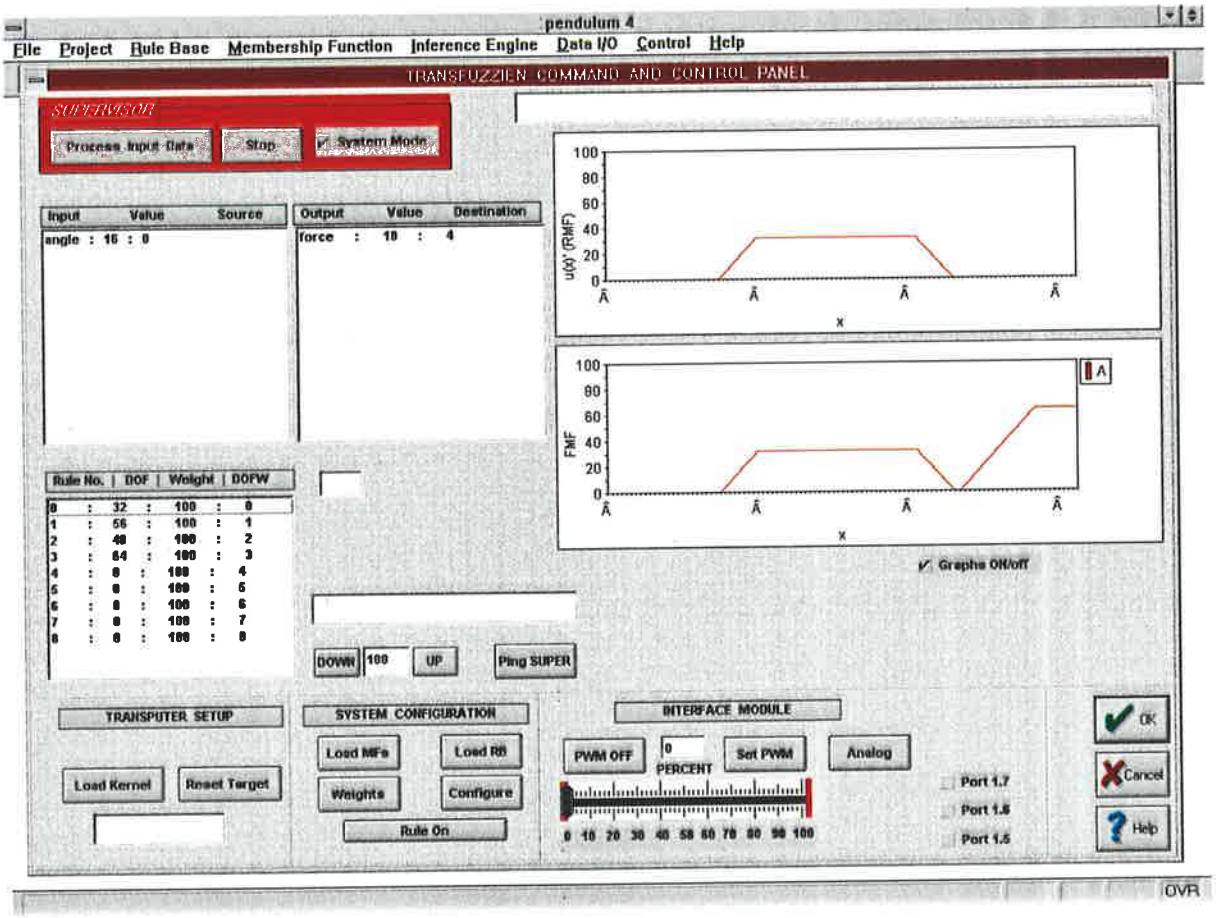


Figure 4.1: The control panel displays the input data and the results of processing, and provides user control over external hardware resources connected to the Transputer Interface Module.

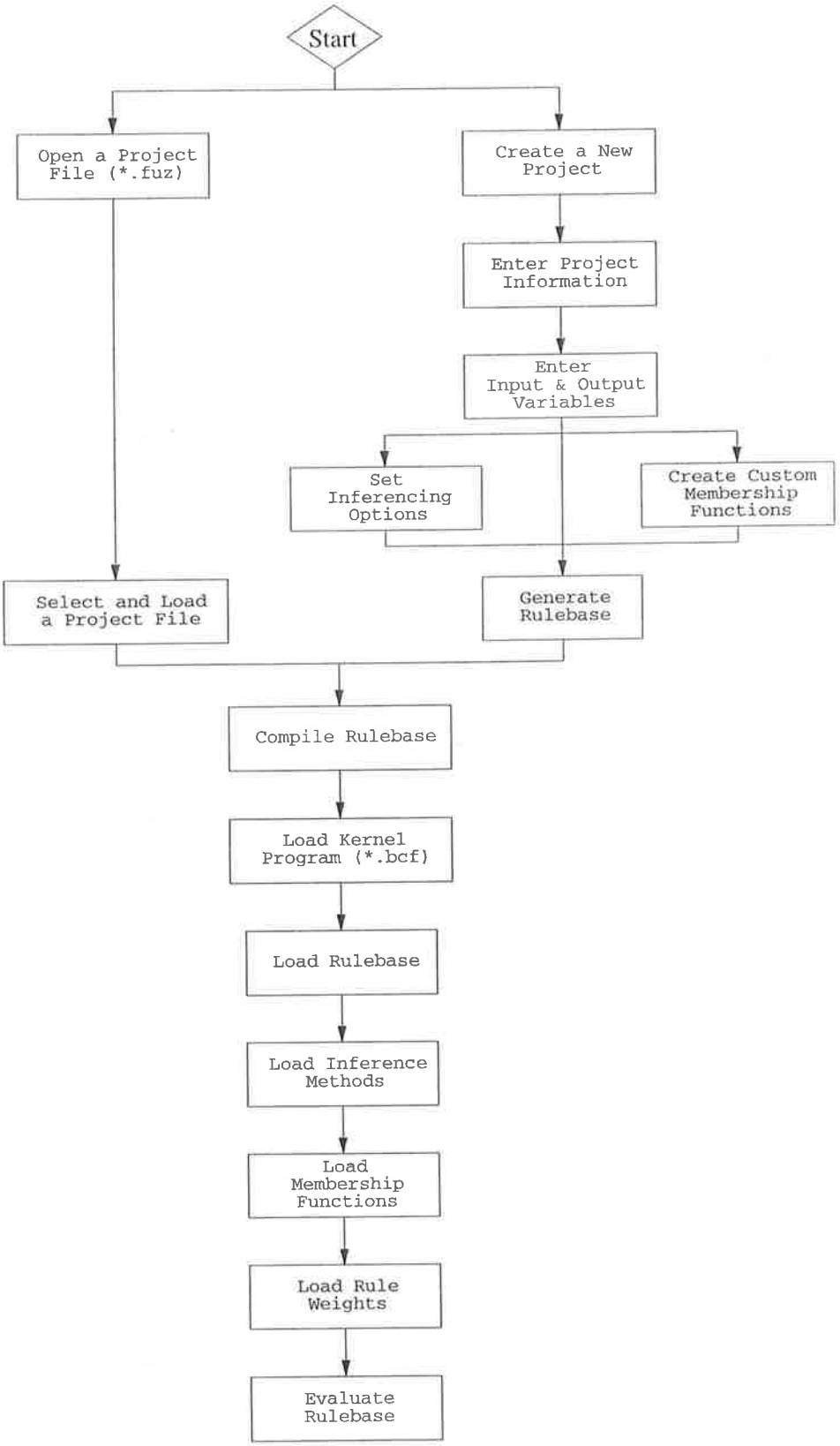


Figure 4.2 : This flow chart shows the sequence of events necessary to run the TransFuzien software package.

File Contents	Description
Linear Function	Project name
Richard Bowyer	Author
1.0	Version
1995	Date
Linear function modeling using fuzzy rules.	Description of the project
0	Connective type
0	Modifier type
0	Fusion type
0	Defuzzification type
1	Input variable source identifier
x	Input variable name
1	Output Variable sink identifier
y	Output variable name
9	Number of rules in the rule base
0	Input variable
R>0:IF x IS Large- THEN y IS Large- I,>0,S,~8,T,<0,S,&8,	Rule text
0	Parsed rule text
R>1:IF x IS Big- THEN y IS Big- I,>0,S,~7,T,<0,S,&7,	Output number
0	
R>2:IF x IS Medium- THEN y IS Medium- I,>0,S,~6,T,<0,S,&6,	
0	
R>3:IF x IS Small- THEN y IS Small- I,>0,S,~5,T,<0,S,&5,	
0	
R>4:IF x IS Zero THEN y IS Zero I,>0,S,~4,T,<0,S,&4,	
0	
R>5:IF x IS Small THEN y IS Small I,>0,S,~3,T,<0,S,&3,	
0	
R>6:IF x IS Medium+ THEN y IS Medium+ I,>0,S,~2,T,<0,S,&2,	
0	
R>7:IF x IS Big+ THEN y IS Big+ I,>0,S,~1,T,<0,S,&1,	
0	
R>8:IF x IS Large+ THEN y IS Large+ I,>0,S,~0,T,<0,S,&0,	
0	

Figure 4.3 Listing of a typical Project file. The membership tables are aslo written to this file but are excluded for clarity.

4.3 Description of the Operator Interface Dialog Boxes

This section explains the function of each screen, and provides bitmap images of each dialog box. At present there are several redundant sections of code present, which were used during development and debugging of the software suite. These were useful for diagnostic work, but could be removed for the next version of the software. One of these features is a dialog box which displays a confirmation message when certain commands are initiated by the operator. For example, when the rule base, or the membership functions, or the inferencing configuration is loaded to the transputer network, a confirmation dialog box is displayed.

The rule editor, shown in Figure 4.4, has been designed such that when entering a fuzzy rule, using the mouse (a single click of the left mouse button for buttons, and a double click of the left mouse button for listbox items), the operator is moving from left to right. Also, items have been grouped in a logical manner, such as the placement of the hedge list box, being adjacent to the membership listbox.

The input and output names, must be entered prior to generating the rule base. They are then displayed in their own listboxes. As rules are composed and added to the rule base, they are displayed in the Rule Base dialog box.

The inferencing options for connectives, consequent function modification, rule fusion, and defuzzification, are selectable from the Inference Method dialog box.

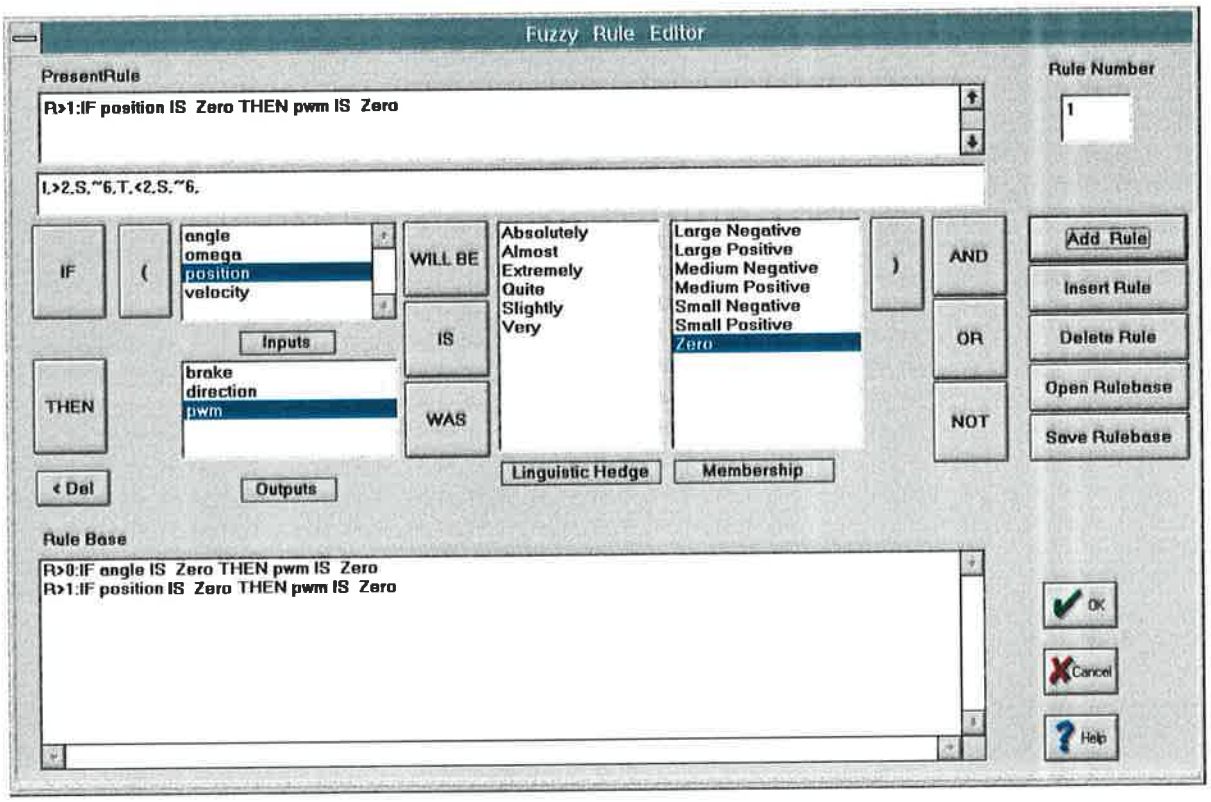


Figure 4.4 : The Rule Editor dialog screen provides the functionality to compose fuzzy rules. The present rule appears at the top of the screen, whilst completed rules are listed below.

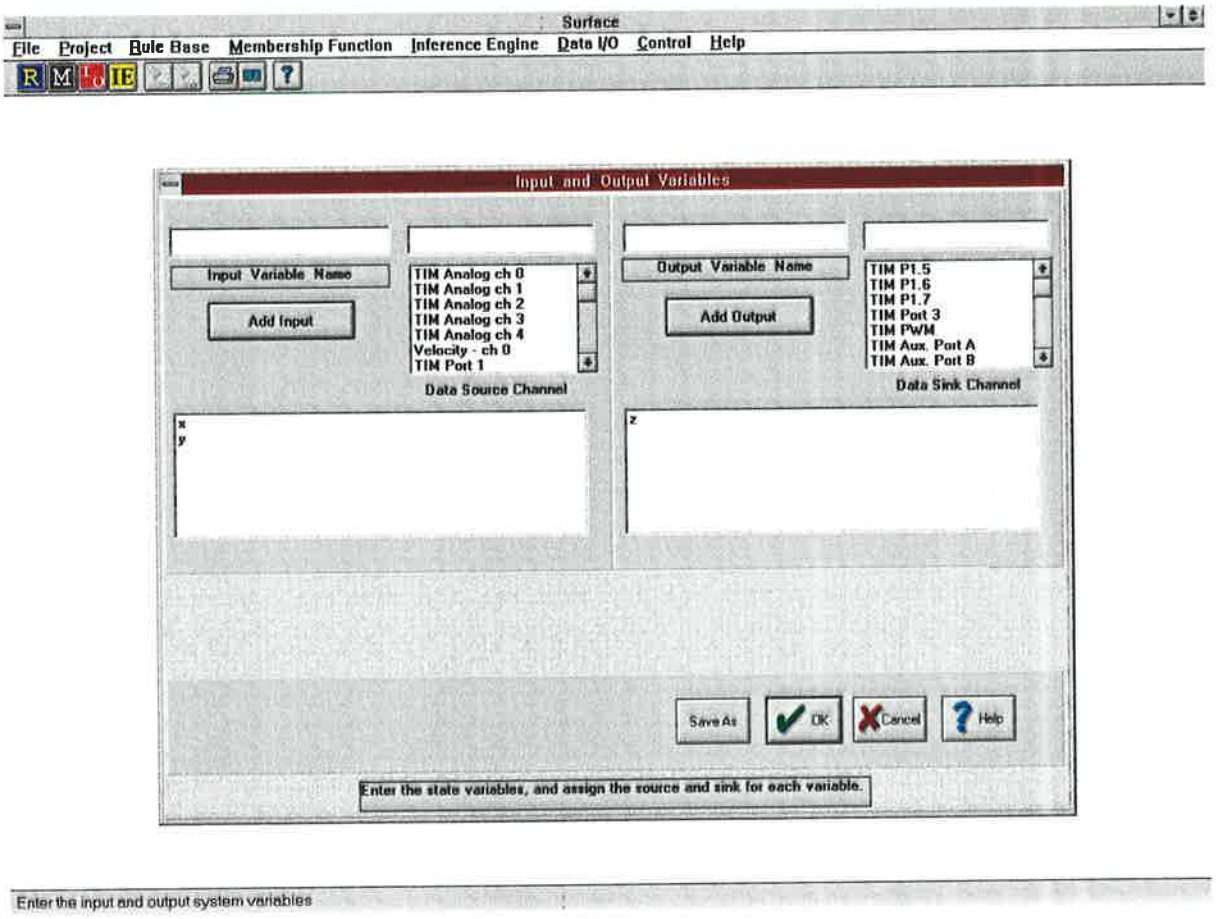


Figure 4.5 : The input and output variables for the system to be modeled or controlled, are entered with the variables dialog box. The name and physical source or sink are defined here.

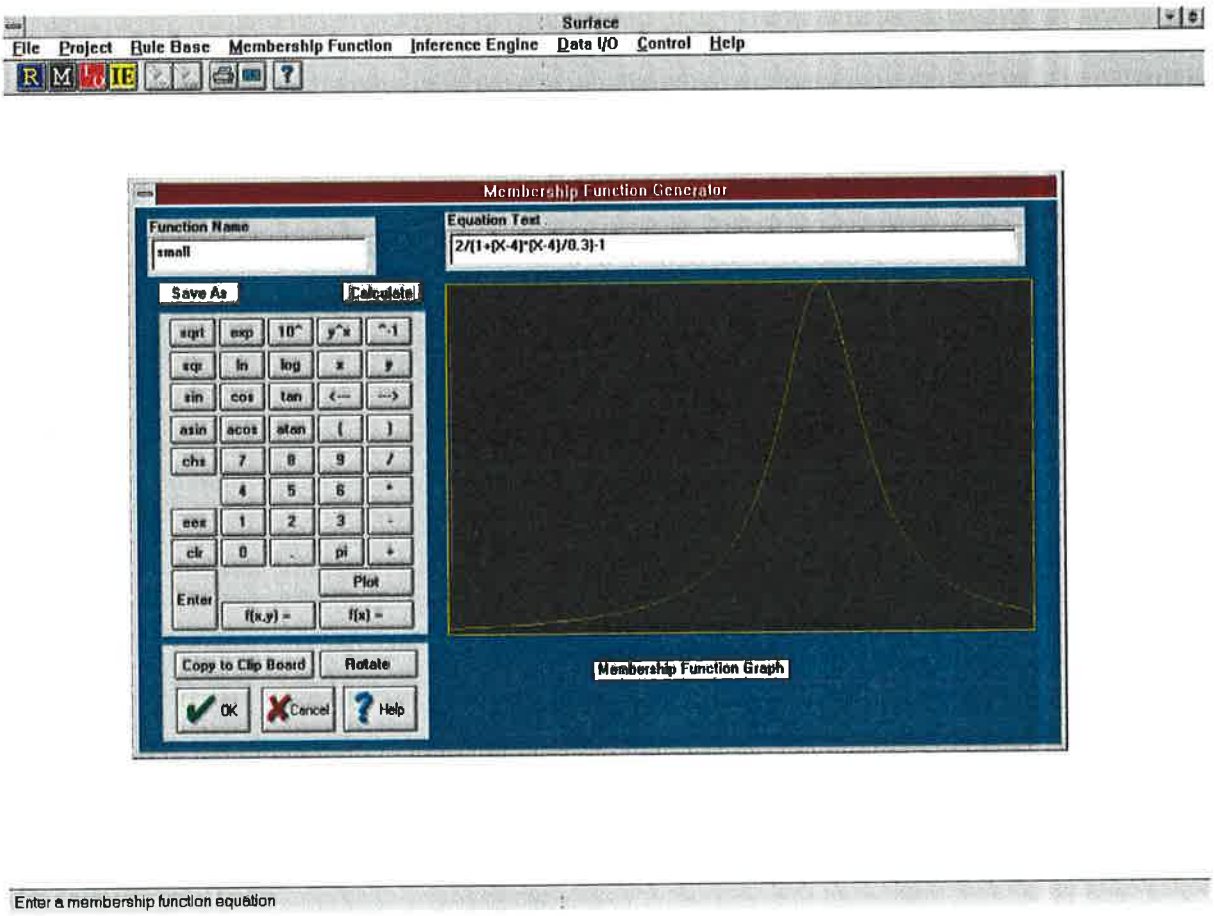


Figure 4.6 : The membership function editor provides an equation building and display facility. The equation is entered by either typing with the keyboard, or by using the mouse to hit the keypad.

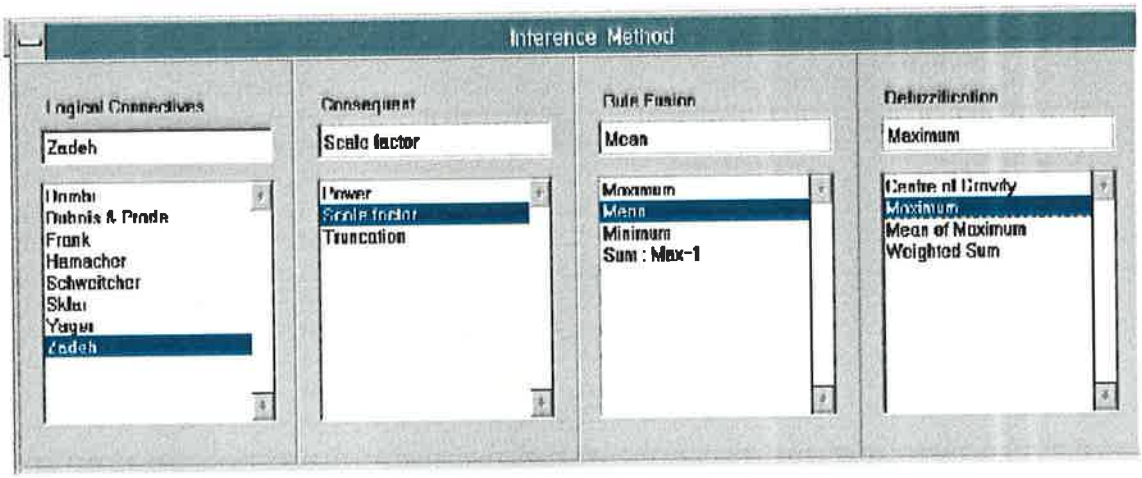


Figure 4.7 : Dialog box that enables the selection various inferencing methods.

4.4 Chapter Summary

This chapter has described the graphical user interface of the TransFuzien software suite. This software has been written in C and runs under the Microsoft Windows operating system. There are several features that were used for diagnosis and debugging that can be removed in the next version of the software.

The various dialog boxes for system configuration have been described. The development of this GUI, and the functionality, have been governed by the need to first develop a knowledge base, to configure the inference engine, and then to display results of processing in a graphical manner.

Chapter V

THE EXPERT SYSTEM FRAMEWORK AND ALGORITHM IMPLEMENTATION

5.1 Introduction

5.1.1 Chapter purpose

This chapter presents the implementation of the inferencing algorithm that was described previously in Chapter 4, and covers the mapping of the software to the hardware resources. The various processing algorithms do not exist in isolation. There must be a support framework upon which the expert system is built and run. The software and hardware to achieve this are described. The various processes and the data flow through the system are described.

The software routines are written in Occam 2 [35, 36], and were compiled using the Inmos Transputer Development System (TDS) [37].

5.1.2 Chapter overview

The software that resides on the Transputer target system, has a variety of functions to perform. This chapter describes those functions and the interactions that occur throughout the system. In section 5.2, the process functionality is described, together with the communications protocols that are a key feature of Transputer based processing.

The software that implements the inferencing algorithm[1] is first developed to run on a single Transputer. Next, additional processes are added to the code to enable the farming out of the fuzzy rule evaluations. This code is configured to run on a single Transputer.

The final stage is to configure the code to run on a network of Transputers which reside on the B008 motherboard. This is described in section 5.3.

[1] This is the algorithm that interprets the fuzzy rules and calculates the crisp outputs.

5.2 Process Function and Communications

To best understand the implementation of the algorithm, it is helpful to consider the data transformations and data flows within the system. Data flow diagrams (DFDs) provide a powerful method for visualizing process interactions and data transformations, and have been used to describe each of the component processes in this algorithm.

The expert system framework consists of software that allows the user to configure all aspects of the expert system, and to examine data during the processing phase. The graphical user interface is a key component of the support structure, and is described in detail in the following chapter. Likewise, there are software routines that manage data interactions between the system and the external environment. These process supply the information required by the inferencing process.

The software that implements the inferencing algorithm is organized as a number of Occam processes, as shown on Figure 5.1, where each of the processes run concurrently on the Transputer target system. The data packets have been omitted from the figure for the sake of clarity. However, these packets are described in tables later in the chapter which list define the PROTOCOLS of the data flowing along each channel.

A bootable code file (BCF) is extracted by the file handling utility of the TDS, and saved as a *.bcf* file. This file is accessed by the GUI file Project management procedure, and loaded to the Transputer network via the C012 link adapter.

The B008 motherboard links were re-arranged to allow the network to be booted from the links. (The details of the B008 motherboard are described in Appendix A.) Following compilation of the Occam source code, the TDS is used to extract a description of the boot path for the network as described in the configuration statement. This is shown below.

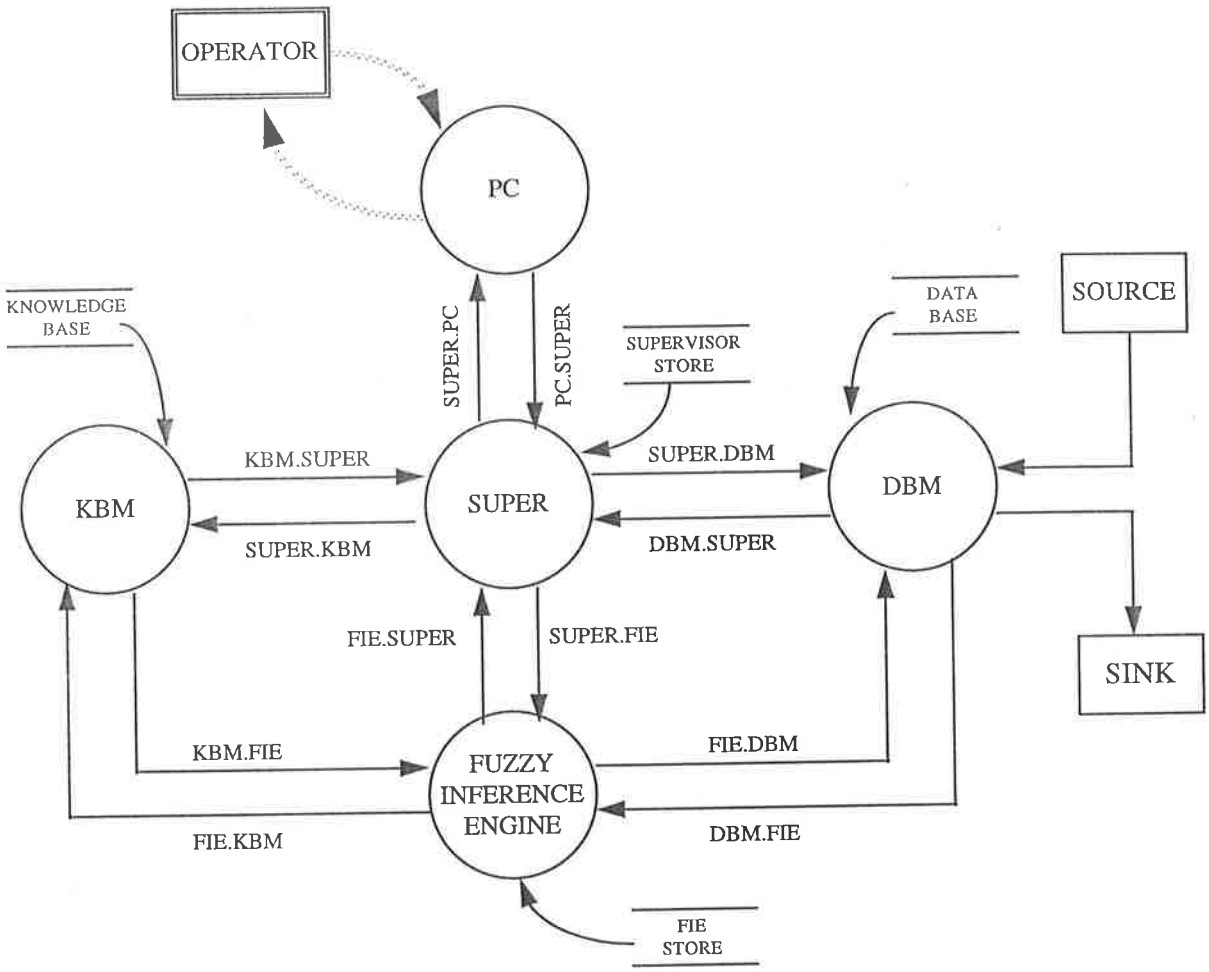


Figure 5.1 : Top level data flow diagram for TransFuzien software suite. The ellipses represent processes, the directed lines represent channels for communications, and parallel lines above and below a label, represent data storage libraries. The data packets have been excluded for clarity.

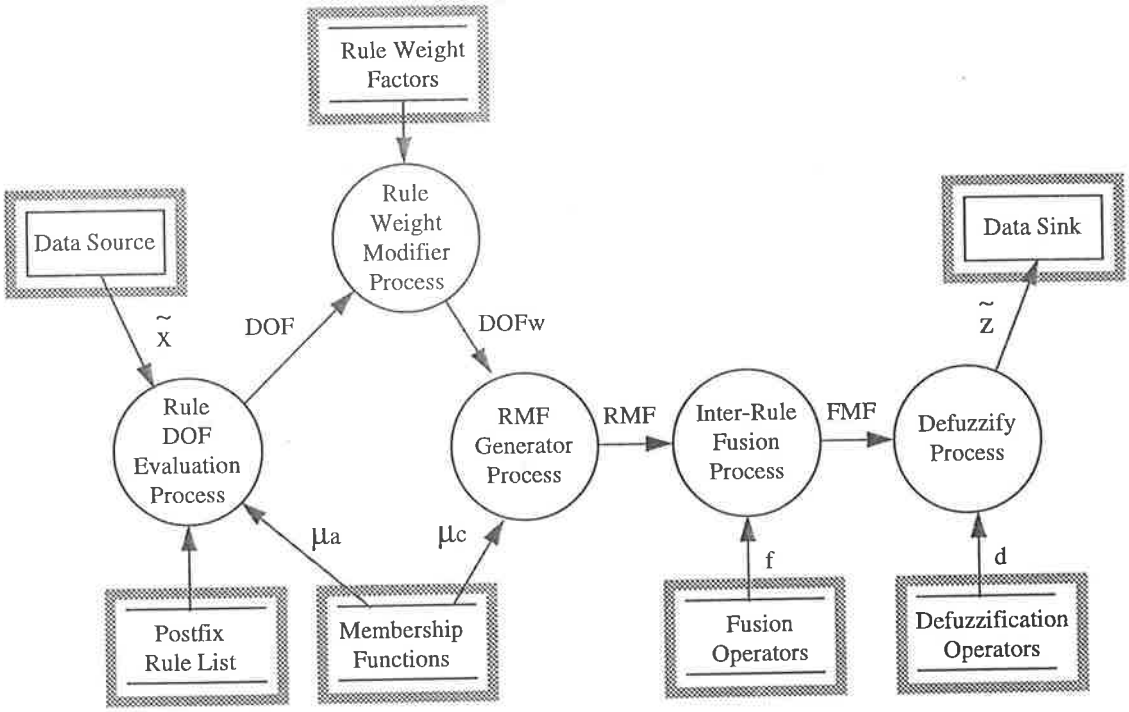


Figure 5.2 : The DFD for the FIE indicates the various data that are required to be present before processing can begin. The structures marked with the patterned rectangles represent elements that need to be configured by the user.

The process that performs the inference calculations resides on the FIE process. The folding editor of the TDS is very useful for developing code in a top-down fashion. Folds are created which can be labeled according to their function. These folds are then entered, and either further folds can be create or Occam source code can be written (or a combination of the two). This segment of code[2] describes the operation of the FIE process. Notice the Occam key words WHILE and SEQ, and the folds which are identified by the three dots.

5.2.1 The PC process function

The PC process resides on the host computer (IBM PC) is written in C, and has two main functions. Firstly, it provides a graphical interface by which the operator interacts with the system. Secondly, it performs the necessary pre-processing of the information that the operator enters into the system.

Operator interaction is facilitated by a number of dialog boxes that may be displayed by selecting the appropriate function from the main menu located at the top of the display screen. These dialog boxes have been described in Chapter 5.

Each dialog box has associated with it software routines that handle the user interaction for that box. The rule editor, for example, has code to generate, save, and view the fuzzy rules. There are routines to establish communications between the PC, and the B008 motherboard.

The system parameters for the current version of the software are listed in Table 5.1. These parameters determine the limitations of the software suite, and highlight where future improvements may be made. (eg. increase the number of rules)

[2] A full listing of the software is contained in appendix A

System Parameter	Value
1. Maximum number of rules that can be processed	100
2. Number of membership functions	9
3. Number of intervals per membership function	100
4. Number of Inferencing Connectives supported	6
5. Number of Fusion Methods supported	4
6. Number of Defuzzification Methods supported	3
7. On-line inference method variation[3]	yes

Table 5.1: TransFuzien System Parameters

[3] Ability to change the inferencing strategies via the GUI whilst in the continuous processing loop mode.

The system parameters were determined by a consideration of several factors. These included:

1. Number of rules: 100 was thought to be sufficient for modeling typical problems encountered by the author.
2. Number of membership functions: Again, typical domains are divided into 5 to 9 regions.
3. Number of intervals per membership function: The number of intervals impacts directly on the processing time to evaluate a fuzzy rule (see later). The final choice allows for good byte size data representation, which simplifies implementation, whilst providing a reasonable dynamic range for the input data.
4. Inferencing methods: Provide typical methods.
5. Size of memory required on both the PC and the Transputer modules. As these numbers increase, the memory requirements will increase.

The final numbers being a trade-off between flexibility and system (and code) complexity.

5.2.2 The Supervisor (SUPER) process function

The Supervisor process is written in Occam, and resides on the Transputer target. It responds to commands from the operator via the GUI, and co-ordinates their execution on the Transputer system. Commands from the PC must belong to the protocol for the channel connecting the PC to the Super process (these protocols are described later.), and are routed to the appropriate destination.

There are three other input channels to the SUPER process, each of which is monitored for messages. Figure 5.3 shows how this process responds to messages. The SUPER process acts as a gate-way for messages to and from the rest of the system.

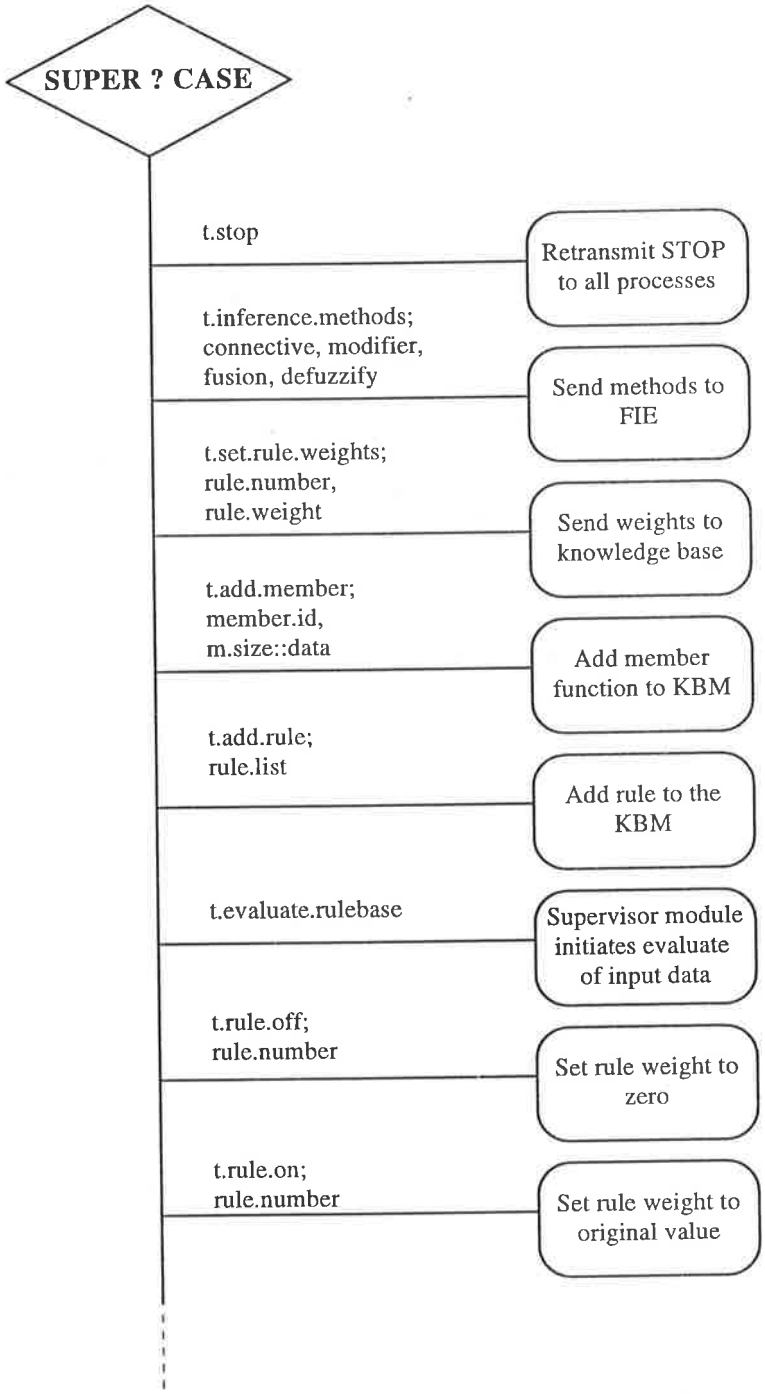


Figure 5.3: The Supervisor process interprets messages from the PC process, and serves each message as it arrives. The Occam CASE statement acts as a selector, to distinguish which function is to be initiated.

5.2.3 *The Knowledge Base Module (KBM) function*

The knowledge base module stores information for the inference engine, and distributes this information to the other processes as required. The information that is managed includes:

1. the rule base
2. inferencing configuration information
3. the present state
4. number of rules
5. number of inputs and outputs

The contents of the KBM may be updated as the software is running, which adds to the flexibility of the system.

5.2.4 *The Data base (DB) process function*

The Database Module (DBM) controls communications with the data source and sink, and stores results from current and previous (history) calculations. Data can originate from the TIM or from the PC, or it can receive the results of calculations performed by the FIE.

The calculation of the resultant membership functions and the final (composite) membership functions are computationally intensive processes within the software suite. The results from each process are transmitted to the database manager and stored. When the GUI issues a request for data, then the DBM will receive the request via the SUPER process, and send the data stored in the appropriate data structure.

Type	Variable name
BOOL	manager.running:
INT	output.number, output.value, vector.length, status:
INT	input.source:
[20] BYTE	PC.input.vector:
[20] INT	PC.history.buffer: -- store previous vector
[20] BYTE	Plant.input.vector:
[20] INT	Plant.history.buffer: -- store previous vector
[20] BYTE	output.name:
INT	size, stop.char:
INT	input.number, input.value:
BYTE	pwm.value:

Table 5.2. List of variable declarations for the DBM Process

5.2.5 *The Fuzzy Inference Engine (FIE) process function*

The Fuzzy Inference Engine (FIE) calculates the resultant membership function for each rule in the rule base. The results from each process are transmitted to the database manager and stored.

Figure 5.4 describes the processing of the rulebase, and shows the sequence of events involved in evaluating the RMF and WRMF sets[4]. The flowchart shows that after these sets have been evaluated, the system calculates the FMF and finally the crisp output. The evaluation of a single rule is described in Algorithm 5.1.

Algorithm 5.1 for Rule List Evaluation

While more rules to process
Do
Read rule list Opcode and Data
CASE Opcode

IF	:continue
X	:read input vector x
μ_{input}	:lookup m(x)
H	:apply hedge - calculate h(μ(x))
IS	:Place results on stack
	stack pointer = stack pointer + 1
AND, OR	:pop last 2 entries from stack
	apply connective
THEN	:set more flag = False
	(DOF is now on stack)
μ_{output}	:Calculate Resultant membership function

Z :calculate weighted DOF and store results
End.

[4] These sets comprise 101 integer values.

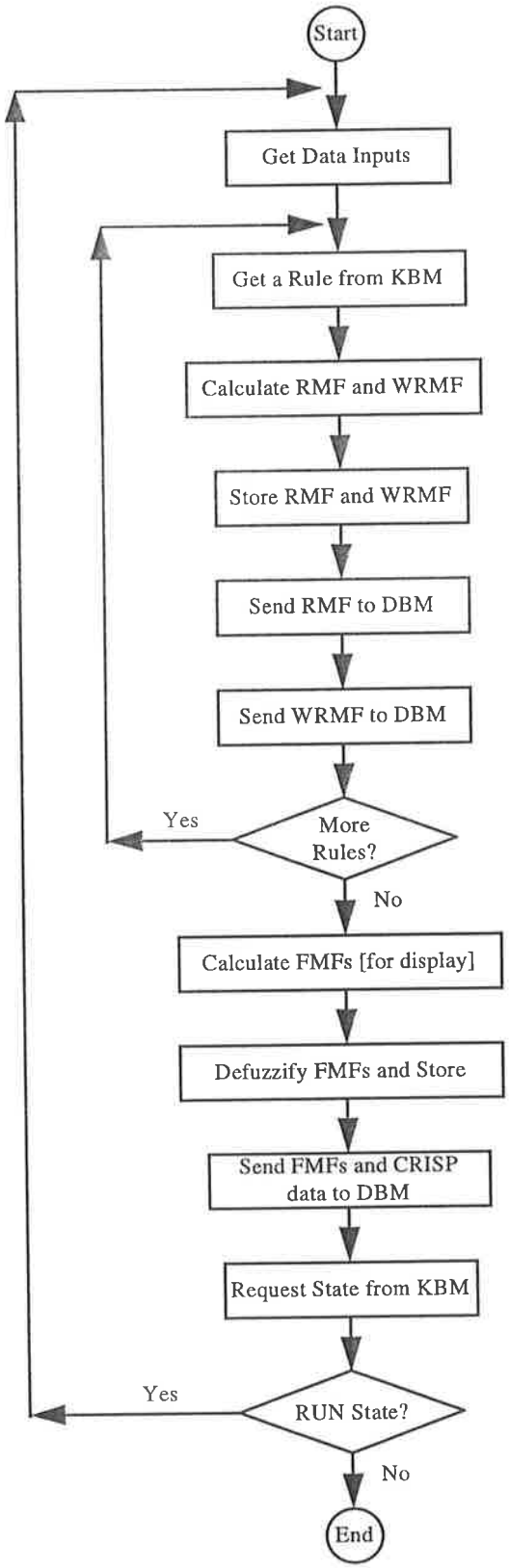


Figure 5.4 :Flowchart for the rulebase evaluation phase of processing for a single processor architecture. The evaluation of the RMF and WRMF occur in the Fuzzy Inference Engine process labeled FIE. Processing continues if the system is in RUN state, but will execute one pass through the rulebase otherwise.

5.2.6 *Process communications*

The communications between processes is an important area in this work. The method adopted here is to define a number of commands, which may include some data, that are recognized by the processes. The commands and data are passed between processes in an ordered manner, that must ensure that each communication can complete, and that deadlock does not occur. Channel protocols are defined for each Occam channel, which fully defines the message type and the data types that are permitted.

A Client-Server approach is taken to the communications between processors. As one process (the Client) issues a message (eg. a request for data) then the process that receives that request (the Server) will attend to the request.

All command packets that are issued from the GUI[5] are routed through the Supervisor process (SUPER), which directs the command to the appropriate destination. The destination of each packet is determined by the tag that begins the packet. The tags are defined in the PROTOCOL statement. Table 5.3 shows the protocols that are used.

The commands include initialization processes with information about inferencing methods, membership function data, and so on. Each command causes a particular sequence of events within the inferencing processes, and, depending on the command, a particular reply is expected by the GUI.

In order to keep track of the many commands and their effects, a graphical representation of process interactions has been used. These graphs are called Process Event Graphs (PEGs), and show the temporal dependencies that exist between processes. PEGs can be quite complex, and are particularly useful in highlighting the interactions between processes that are operating in parallel. PEGs also help to develop code that is not susceptible to deadlock. That is, the PEG provides a visual tool for planning and analysing communications events. Figure 5.6 shows an example of a PEG which illustrates the initiation of a process rulebase command from the GUI.

The Process Event Graph comprises a column containing boxes for each process, with horizontal lines drawn from each box. The line represents a time axis. Directed lines are then drawn from one horizontal line (the source), to the destination line. The nature of the process interaction is shown by labels on the directed line.

[5] The GUI is running on the PC under Microsoft Windows.

Process	Tag Identifier		Data Description
All	0	t.stop	
	1	t.execute	
	2	t.report.type	INT
SUPER	3	t.system.mode	INT -- mode
	4	t.Super.status	
	5	t.supervisor.response	BYTE
	6	t.dof.information	INT::[]BYTE --length, dofs
	7	t.evaluate.rulebase	
DBM	8	t.send.crisp.data	INT --output number
	9	t.send.fmf.data	INT --output number
	10	t.DBM.reply	
	11	t.input.name	INT::[]BYTE
	12	t.output.data	INT::[]INT; INT::[]INT
	13	t.file.input.vector	INT::[]INT --file
	14	t.DBM.ack.stop	
	15	t.get.input.vector	
	16	t.input.vector	INT::[]INT --size, data
	17	t.request.input.vector	INT --input number
	18	t.the.inputvalue	INT::[]INT --size, Plant input vector
	19	t.PC.input.vector	INT::[]INT --user supplied input
	20	t.send.dof.data	--send dofs to GUI
	21	t.send.rmf.data	INT --send rmfs to GUI
KBM	22	t.KBM.status	
	23	t.knowledgebase.ping	INT
	24	t.number.of.rules	INT
	25	t.add.rule	INT;INT;INT::[]INT --rule,output,size,data
	26	t.delete.rule	INT --rule.no
	27	t.add.member	INT; INT::[]INT --member no, size, array
	28	t.set.rule.weight	INT; INT --rule no, weight
	29	t.rule.on	INT -- rule no, evaluate
	30	t.rule.off	INT -- rule no, don't evaluate
	31	t.rule.list	INT; INT; INT::[]INT --rule, o/p, size, data
	32	t.rule.info	INT::[]BYTE
	33	t.send.rules.in.rulebase	
	34	t.send.rule	INT --send rule list for this rule
	35	t.request.membership.value	INT; INT -- member.number, x.value
	36	t.send.mfs.to.FIE	
	37	t.number.of.outputs	INT
FIE	38	t.inference.ping	INT
	39	t.FIE.status	INT
	40	t.request.status	INT
	41	t.inference.methods	INT; INT; INT; INT
	42	t.forced.rule	INT; INT::[]INT --rule.no, size, rule.list
	43	t.rule.data	INT; INT::[]INT --rule.no, size, rule.list
	44	t.evaluate.rule	--tell FIE to evaluate a rule
	45	t.membership.value	INT; INT --member id, member value
	46	t.membership.functions	INT::[]INT -- Array of MFs.
TIM	47	t.reset.plant	
	48	t.shutdown.plant	
	49	t.set.pwm	BYTE
	50	t.pulse.width.mod	BYTE -- 0 = 100% on, 255 = off
	51	t.setoutput	INT; INT -- output number, value
	52.	t.input.data	INT::[]INT
	53.	t.request.state	

Table 5.3 : Tag identifiers for system processes.

Tag Identifier	Data Description
t.rmf.data	INT; INT::[]INT --rule number, size, array
t.fmf.data	INT; INT::[]INT --output number, length, values
t.weighted.dof.data	INT::[]INT --dofweighted
t.dof.data	INT::[]INT
t.crisp.data	INT --crisp output
t.error	BYTE -- error tag followed by error type
t.send.input.data	INT -- source
t.input.data	INT::[]INT
t.crisp.data	INT::[]INT; INT::[]INT
t.crisp.value	INT

Table 5.4 : Protocol of the RESULTS channels for the processes.

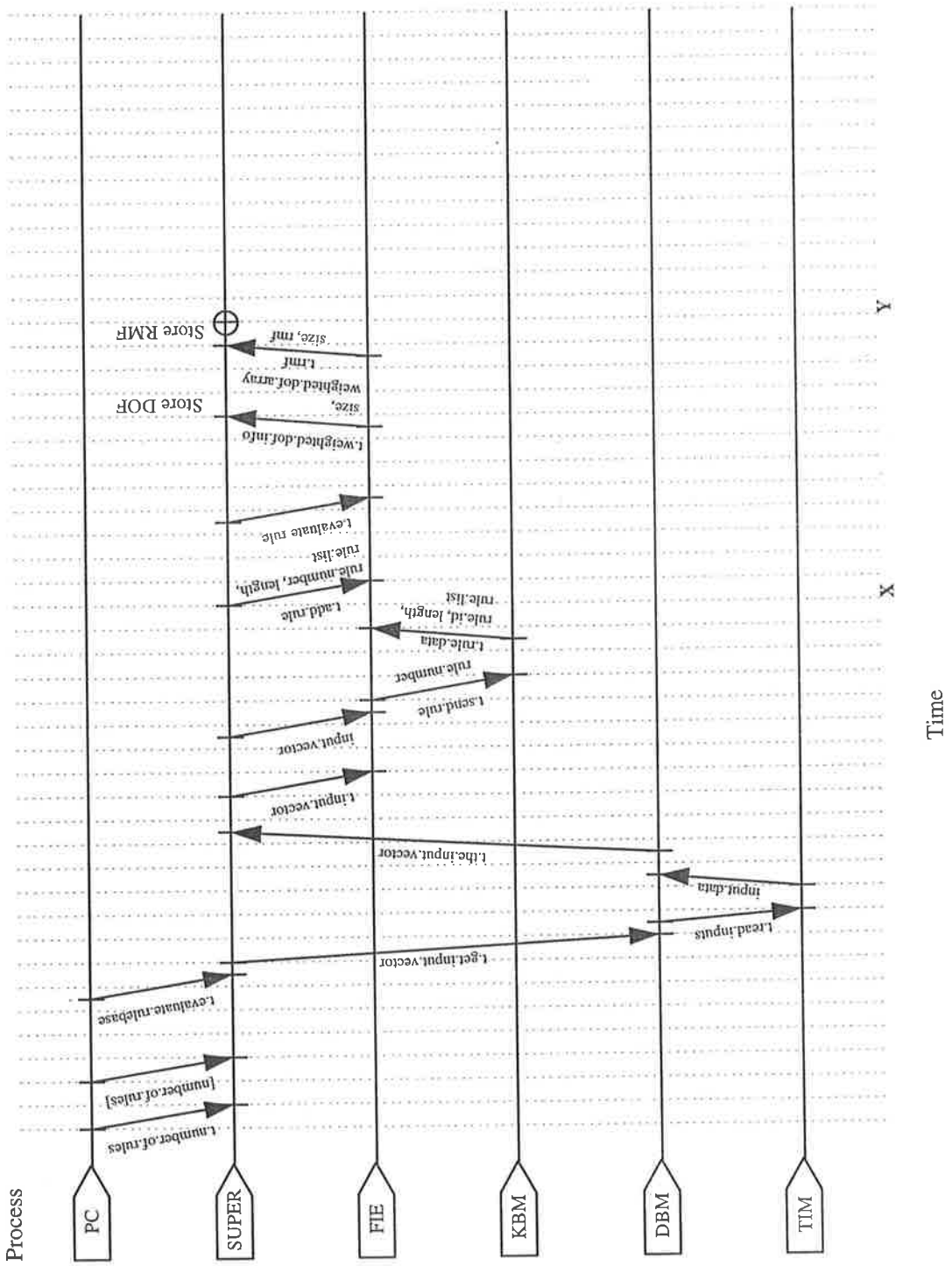


Figure 5.5 :A Process Event Graph gives a graphical representation of process interactions. The events between the markers X and Y are repeated for each rule in the rulebase. In this study, circles are used to define event cycles. The time axis is not to scale.

5.3 Processing on Multiple Transputers

5.3.1 Requirements

In a given rule base, there may be only a few rules, or there may be hundreds of rules to evaluate. If each rule is assigned to its own processing element, then this would require hundreds of processors which is clearly not practical in most cases. Another method must be found to evaluate the rule base, whilst using the available number of processing elements in an efficient manner.

The solution adopted in this work is that of process farming [38, 39, 40, 41, 42]. With this system, each processor is given a task to perform, being the evaluation of a single rule. When it is complete, it hands the result to a supervisor process, and accepts another task. This system allows for a performance increase as the number of processors is increased.

Farming leads to a near linear speed-up, as more worker processors are added to the architecture. However, a point is reached where the communications bottle neck becomes significant compared with the processing time. The T800 Transputer only has four communications links and so the direct connectivity is limited. The B008 motherboard has a programmable cross bar switch that can be used to connect Transputers into various physical architectures (see Appendix A).

An important issue in parallel processing is that of data dependency. Consider two processes P1 and P1, that both connect to a third process P3. P1 and P2 are assigned tasks by P3, and required data from P3 to complete their individual tasks. Further, the results of the tasks on P1 and P2 are required by P3 to perform a calculation. Clearly, this is a situation where the availability of data controls the flow of events.

To minimise the data dependency in the architecture being developed in this work, as much information as possible, that is required by the worker processes, is distributed to them prior to the run-time phase. For example, the membership function tables are loaded to each worker before execution.

5.3.2 Process Timing

At this stage it is worth considering which factors influence the processing time for a given task. With reference to figure 5.2, and limiting the number of Transputers to one, the following factors are relevant:

1. The communications over-head per rule, T_{comms}, to send commands from the GUI to update the information display.
2. The number of rules in the rule base, N_r.
3. Time to perform the Fusion of the RMFs, T_{fusion}.
4. Time to perform the Defuzzification of the FMF, T_{defuz}.

The communication time T_{comms} , will increase as the number of tasks increases. Clearly, the strategy that is adopted for updating the information displayed by the GUI, will impact on the communications over-head. The more often results are updated on the GUI, the more time will be spent passing the result data from the DBM to the SUPER process, onto the GUI. This is one reason why a facility has been developed (the Transputer Interface Module, described in Chapter 6) to interact directly with peripheral electronic hardware. Obviously, the nature of the processing task, dictates how sensitive or crucial timing will be, and this will be a deciding factor in the way the GUI is operated.

The total processing time T_{proc}, per output is the sum of all the component times. The Occam code may be written with PAR constructs, but as the Transputer is time-slicing these parallel segments of code, the outcome is much the same as if purely sequential code was used. Hence, the total time is described by equation (5.1).

$$T_{proc} = T_{commsr} + \sum_{i=1}^{N_r} T_{p_i} + T_{fusion} + T_{defuz} \tag{5.1}$$

Equation 5.1 shows that as N_r increases, the processing time for evaluating the RMFs of the rules, becomes a larger proportion of the total time. T_p is the time required to evaluate a single rule and pass the results to the DBM, and will vary with the form of each rule. This will be the case if each rule in the rulebase has the same number of premises in the antecedent, and further, each premise comprises the same number of fuzzy operators. To illustrate this point consider the three cases below:

Case 1:

IF *x1* IS large AND *x2* is small THEN *z1* IS zero

◊ *There are 3 operations to perform.*

Case 2:

IF *x1* IS VERY large AND *x2* is small THEN *z1* IS zero

◊ *There are 4 operations to perform.*

Case 3:

IF (*x1* IS large AND *x2* is small) AND (*x3* IS small OR *x4* IS medium) THEN *z1* IS small

◊ *There are 7 operations to perform.*

Clearly, the total processing time will be determined by the longest processing interval. In order to increase the processing through-put of the system, a process farm is proposed, that will allow the rule evaluation process to be replicated on additional Transputers. One of the advantages of the rule representation which has been developed in this work, is that it enables rules to be evaluated by other processes, then combine the Resultant Membership Functions to form the Final Membership Functions.

The algorithm that is developed in chapter 4, is suitable for mapping onto multiple Transputers. By decomposing the fuzzy inferencing process into the component parts, it has been shown that the rule evaluation is one area where multiple processors can be used to enhance processing through-put.

To run the code developed for a single Transputer (see previous section) on a network of Transputers, several changes need to be made. The first change involves process mapping. There are two T800 Transputer modules available to the system, the first of which has been used up to this point for all processing. The second rule evaluation process is now mapped to the second T800 Transputer. This is accomplished by altering the configuration statements as shown in the code segment below. An additional PROCESSOR statement has been added, together with the name of the process that is to run on this processor, as shown in Listing 5.1.

```
PLACED PAR
  PROCESSOR 0 T8
    PLACE from.C012 AT link0in:
    PLACE to.C012   AT link0out:
    PLACE Plant.DBM AT link1in:
    PLACE DBM.Plant  AT link1out:
    PLACE FIE.Node AT link2out:
    PLACE Node.FIE AT link2in:
    FuzzienProg(from.C012, to.C012, FIE.Node0, Node0.FIE,
                FIE.Node1, Node1.FIE, DBM.Plant, Plant.DBM)


  PROCESSOR 1 T8
    PLACE FIE.Node AT link0in:
    PLACE Node.FIE AT link0out:
    RuleNode1(FIE.Node1, Node1.FIE)
```

Listing 5.1: Process mapping to processors.

The next change involves the communications channels between processes. Two additional Occam channels have been created to handle communications between the new rule node and the rest of the processes. The data flow diagram for this configuration is shown in Figure 5.6.

A further change is required to the FIE process to incorporate a task scheduler, or farmer, which allocates rule lists to each Rule Evaluation Node (REN). The scheduler also distributes the configuration information to the RENs which includes the membership function tables, the inferencing options, the input vector, and the system mode.

5.3.3 Task Scheduling

There are many possible methods of assigning tasks to processing resources. The method adopted in this work is to allocate tasks to each available processor in the network. In this case, the network consists of two processors connected in a tree configuration.

In this work a flag called the RuleNodeState is defined for each REN. This flag defines the present activity state of the process. A '0' represents idle and a '1' represents busy. The farmer initially allocates a task to each REN, and the corresponding flag is set to indicate Busy status. As the results (RMF) returns from each REN the farmer collects the data (harvest) and allocates the next task to that REN. This process repeats until all rules in the rulebase have been

processed. Algorithm 5.2 describes the scheduling process, which can be extended to N Transputers.

Consider the hardware architecture of a multiple Transputer system as shown in Figure 5.7. The processes, as described previously in Figure 5.6, are mapped onto this architecture. All of the processes, excluding the second rule evaluation node, are mapped onto Transputer T0. The second rule evaluation node is mapped onto Transputer T1.

There are two processes that evaluate the degree of fulfillment and the resultant membership function for each rule that is input to that process. This division has been used for the following reasons;

1. There are often many rules in a rule base that must be evaluated. This method is one way to utilize available Transputer resources.
2. The fusion and defuzzification processes occur less frequently. That is, there are usually multiple rules that contribute to a single output.

Algorithm 5.2 for Task Scheduling for Two Processing Nodes

```
While more.work
    Do
     If
       (Node 1 is free) AND (rules to evaluate > 0)
         Get the next rule from the KBM
             Send the rule to Node 1
             Set the busy flag for Node 1
             Decrement the number of rules to evaluate
          Increment the rule counter
       else if
         (Node 2 is free) AND (rules to evaluate > 0)
         Get the next rule from the KBM
             Send the rule to Node 2
             Set the busy flag for Node 2
             Decrement the number of  rules to evaluate
             Increment the rule counter
       If both Nodes are busy OR work all tasks allocated
         Read a data packet from the first Node to send its results
         If all work has been collected
            more.work = FALSE.
End.
```

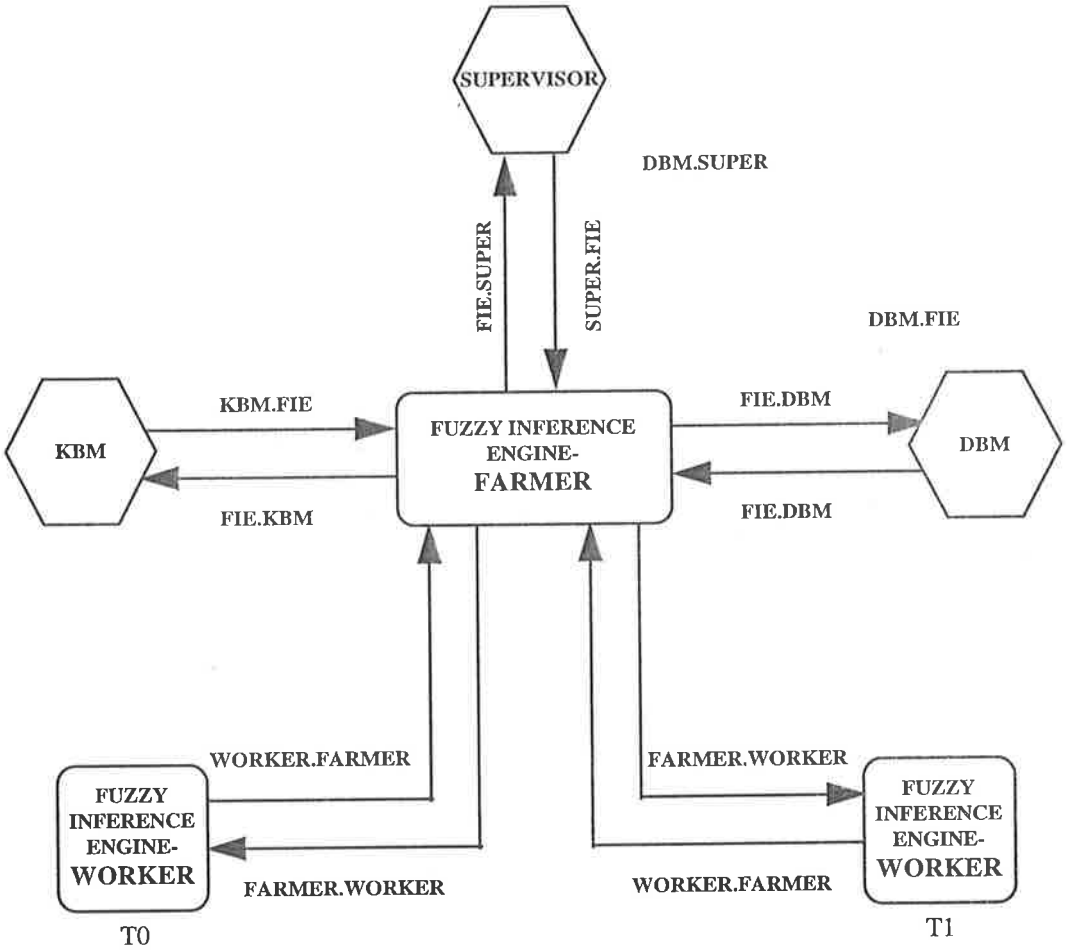


Figure 5.6 : Data Flow Diagram showing the two rule evaluation processes, each of which runs on its own Transputer. The FIE now handles task scheduling between available worker nodes. The FARMER process runs on Transputer T0 and controls the allocation of rule evaluation tasks to the WORKER processes. The additional worker process runs on Transputer T1.

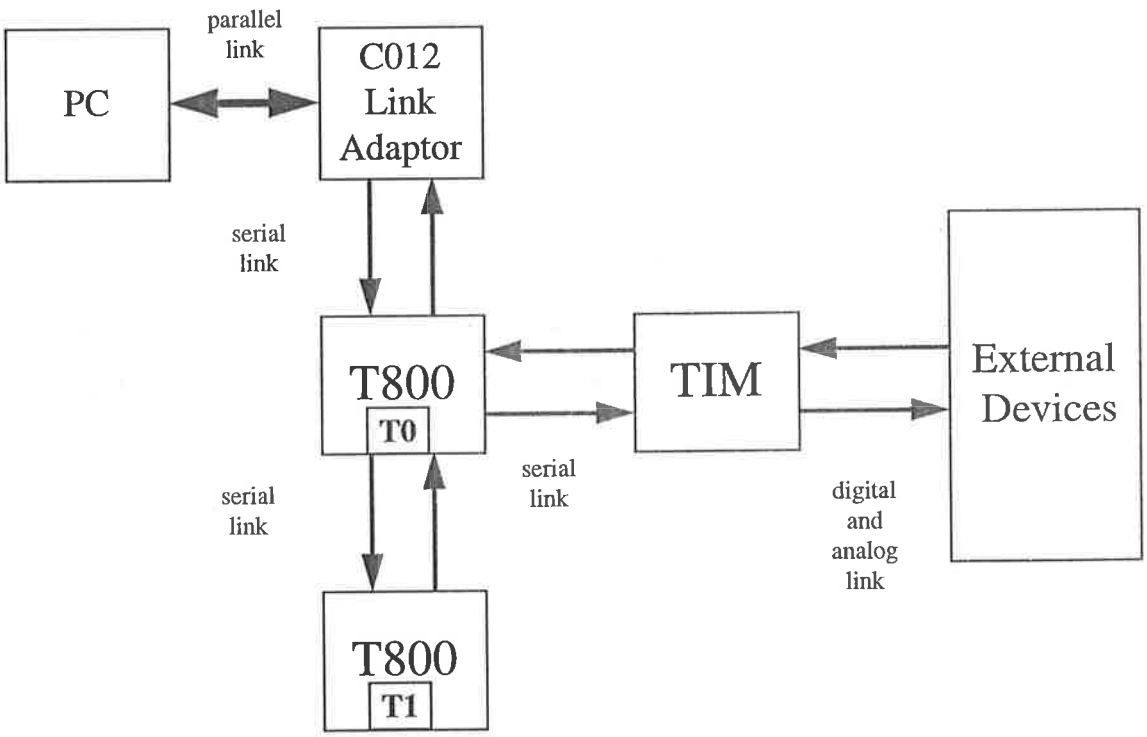


Figure 5.7: The processing hardware for this thesis comprises the personal computer, and two T800 Transputers which are mounted on the B008 motherboard. Transputer T0 accesses external data via the Transputer interface module. Transputers T0 and T1 are connected by a serial link.

5.3.4 *Process Timing Re-visited*

The processing time for a multiple Transputer network will depend on a number of factors, and can be characterized by the times required to perform specific tasks within the processing cycle. These include the following factors:

1. The number of Transputers available, N_p.
2. The communications over-head to send requests between processes via the channels, and to receive responses, T_{comms}.
3. The number of rules in the rule base, N_r. As N_r increases, the processing time for the rules becomes a larger proportion of the total time. T_p is the time required to evaluate a single rule. With each rule evaluation being a sequential operation, the speedup factor is limited in theory (Amdahl's law [38]) to $1/(T_p/T_{proc})$.
4. Time to perform the Fusion of the RMFs, T_{fusion} , increases with the number of RMFs, and the method of defuzzification that has been selected by the operator.
5. Time to perform the Defuzzification of the FMF, T_{defuz}. This is a fixed time that depends only on the method of defuzzification that has been selected by the operator.

Equation 5.1 defined the rulebase processing time for the case of a single Transputer. All times are added in sequence to arrive at a final total, even if the code is written using parallel constructs. With $N_p > 1$, the benefits of parallel processing can be realised. Equation 5.1 is now modified to account for the multiple processor network.

$$T_{proc} = T_{comms} + \frac{\lceil N_r \times T_p \rceil}{N_p} + T_{fusion} + T_{defuz} \qquad (5.2)$$

The adoption of the rule syntax described in Chapter 3, has provide greater flexibility in representing an expert's knowledge, however, this also means that the times T_{pi} , are not necessarily equal (Equation 5.2 assumes they are equal), yet the communication time for each link is the same due to the channel protocol (RESULTS) being identical for each Transputer. This is not always the case as illustrated previously in Section 5.3.2. T_p and T_p are not generally equal. It is worth noting that for the case where there are only 2 rules to evaluate, and two processors available, then the rule that has the most number of operations will determine the processing time. Also, as the results are being collected by a single farmer process, there will be competition between worker processes to deliver their results. The

farmer process de-multiplexes the incoming channels, and hence there is an additional factor introduced, being (N_r x $T_{results}$), where $T_{results}$ is the time required to receive the results from a remote[6] worker process, and includes a component for link communications. Therefore, the processing time for equal length tasks becomes:

$$T_{proc} = T_{comms} + (N_r \times T_{results}) + \frac{\sum_{i=1}^{N_p} T_{p_i}}{N_p} + T_{fusion} + T_{defuz} \qquad (5.3)$$

Taking this analysis one step further, where the task times are not all equal, requires a statistical approach. Stone [42] addresses this situation and derives a stochastic model for the execution times for multiple tasks assigned to multiple processors. He states that the execution time depends on the Expected value of the maximum of the sum of task processing times.

To investigate the timing of the system, six rulebases were written comprising 1, 2, 4, 8, 16 and 32 rules. Figure 5.8a shows the times required to evaluate the rulebase only, for a single Transputer, and using a farm of the two Transputers. Figure 5.8b shows the total processing times (the time to produce the crisp output) for three cases. These are;

1. All of the processing is mapped onto the root Transputer T0 (see Figure 5.7),
2. All rule evaluation is performed on the additional Transputer T1, with subsequent processing handled by T0,
3. The rule evaluation tasks are farmed out to both Transputers T0 and T1, according to scheduling algorithm 5.2.

Figure 5.8a shows that the farming strategy delivers improved performance, with a reduction in the time of about 30% over the single processor implementation. Figure 5.8b shows the following;

- In Case 1 shows that the processing time increases linearly with an increase in the size of the rulebase. This timing follows the single processor model described by equation 5.2, with $N_p = 1$. The link communication is minimised, which saves time.
- In Case 2, it can be seen that the over-head of the additional channel communications between T0 and T1 has caused a degradation of performance relative to case 1. This is in accordance with equation 5.3, with N_p = 1, where the results arrive from the remote processor. In this figure, the times start at a nominal value of about 8mS to process a single rule, which includes communications overhead, fusion, and defuzzification times.

[6] A remote process is defined as one that is mapped to its own processor.

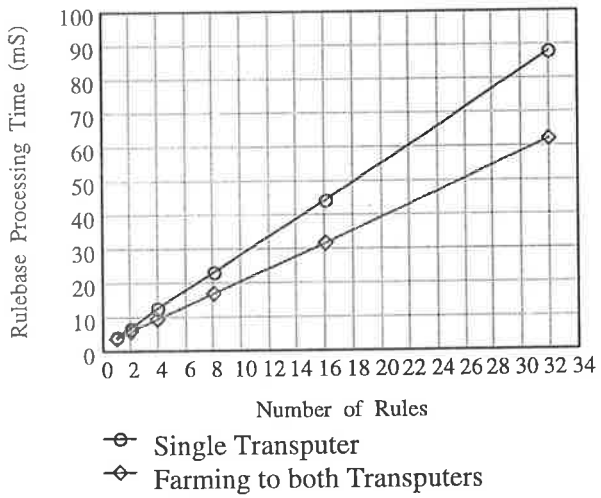
For the single Transputer case, as the number of rules increases, the processing time increases linearly, as expected from equation 5.1.

- The third case shows some improvement in performance by farming the rule evaluation to both Transputers. After the RMFs have been collected from the workers, the data is further processed to make it ready for the fusion process. This post processing of the RMFs, together with the calculation of the FMF data, are computationally intensive, and could be improved in future versions of this software. They contribute significantly to the processing time, and prevent significant performance improvements by farming the rule evaluation tasks alone. Again, this is illustrated in Figure 5.8, where the processing time for the farming case is less than for the case where the farmer would not be performing any substantial, additional processing. In this case, the processing time would be closer to half that for a single processor.

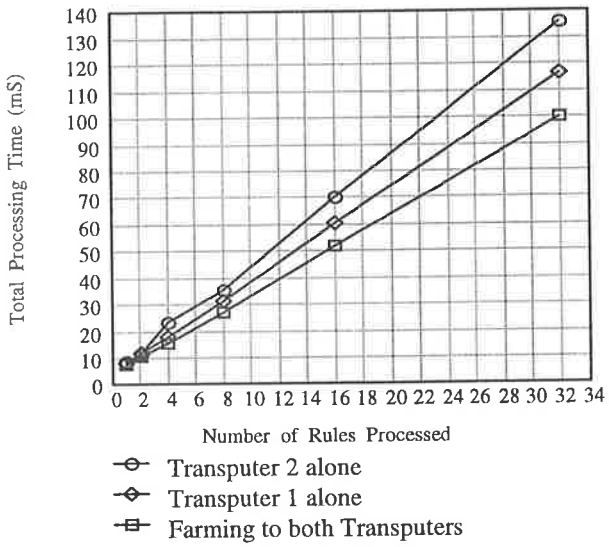
This investigation has identified several key points in relation to developing an architecture based on parallel processing concepts. These include;

1. Communications Bandwidth - The Transputer has four serial links that can be used to connect to other Transputers, creating a variety of architectures. As the number of Transputers (or processors in general) increases, the communications between processors becomes a larger over-head, compared with the actual processing time.
2. Process Identification - The overall process usually has some granularity [42], where it can be decomposed into smaller sub-processes. The nature of the data will often determine how process identification and division is done. This will also assist in determining what type of parallelism can be employed (eg. pipeline, farming, tree), so that processes can be executed concurrently.
3. Process Mapping - When the individual processes have been identified, the associations that exist between them will suggest some logical mapping onto the available processor architecture, or indeed, motivate the development of a specific, purpose-built architecture. Issues that arise at this stage include the communications bandwidth. Moving data around between processors is expensive in time, and should be minimised if possible. Processes that use the same data should be located close to each other, geographically, in the network.
4. Synchronisation - This issue relates to the scheduling algorithm that is employed to distribute work packets to remote processors. It must be able to handle multiple channels, and guarentee to service each worker as results arrive. Workers need not be tightly coupled, and it is better that they are not for reasons of flexibility, but the communications between the worker and the farmer must be free of the possibility of deadlock.

5. Adaptibility - The problem that is to be solved may be just one of many classes of computational problem. The architecture may need to adapt to different data flows generated by different problems. Purpose-built architectures have the advantage of being optimised for their particular task. Adding flexibility to a system may compromise the computational efficiency of that system.



(a) Rulebase processing times for a single Transputer and with two Transputers.



(b) Total processing times for a single Transputer and with two Transputers.

Figure 5.8: Graphs showing (a) the rulebase processing performance, and (b) the total processing performance. Processing time depends on the number of rules that are in the rulebase and the number of processors available in the system. (Transputer 1 corresponds to T0 and Transputer 2 corresponds to T1 in Figure 5.7.)

5.4 Process Interactions

The knowledge base manager (KBM) is loaded with all the rules for the present project. The FIE is configured by the SUPER process, to perform the appropriate inferencing strategies as selected by the user. When all system parameters have been loaded, the user sends a command via the GUI, to the Supervisor, to commence execution. The SUPER process will then perform all functions necessary to complete the processing of the rulebase.

To assist in the planning and development of the process communications and functionality, a chart has been developed that describes the events that occur between processes. This chart is simply called a Process Event Graph, an example of which is shown in Figure 5.3.

The PEG shows the concurrent processes along the left hand side, with lines extending from each label. These lines represent a time axis. Directed lines from one time line to another, indicate the flow of a message. This diagram is useful for visualizing the data interactions between processes.

When execution has been initiated, the FIE requests the latest input data from the DBM. This data is transmitted to the worker nodes. The FIE then enters a loop where it requests the next rule from the KBM and assigns the task to an available worker node.

The worker nodes calculate the RMF for the rule and send it to the FIE, where it is stored in an array of RMFs. The RMFs are also sent to the Supervisor process where they can be accessed and displayed by the GUI.

When all rules have been processed by the FIE, the Supervisor calculates the Final Membership Function (FMF) for each output. These FMFs are then defuzzified according to the selected defuzzification method. The FIE requests the state of the inference methods from the KBM (in the event that the operator has changed them) and updates the current parameters.

If the system is in the closed loop mode, the process is repeated. The FIE will check the mode parameter at the end of each pass.

The KBM maintains the rule base and other configuration information for the system. As information is accessed by the FIE and the Supervisor, this separate process manages this interaction. Likewise for the DBM. The DBM has the additional duty of managing data to and from the TIM. It also keeps a copy of all results so that history data can be used in a feedback situation.

5.5 Running the software suite

5.5.1 The configuration process

The TransFuzien software suite runs on the PC. The program is started by double clicking on the program's icon. The user is then presented with a screen with a number of menu options displayed along the top of the screen. By selecting these menus various aspects of the software suite can be configured.

To begin a new project, select the project menu and click on New. This will display a dialog box for entering information about the project. When this is completed, press OK or Cancel to close, and proceed to define the input and output configuration of the project. This is done by selecting the Data I/O menu, or clicking on the speed button situated just below the menu items. The data definition dialog box is now displayed which enables the user to define the sources and sinks and their names. Again, press OK or Cancel to finish.

Next, select the inferencing options menu where the methods for modifying resultant membership functions, connection logic, rule fusion, and defuzzification are selected.

Finally selecting the Rulebase menu, and clicking on New, will display the rule base editor. The composition of rules is simply a matter of using the mouse to point and click on the items displayed in the list boxes, and the dedicated buttons, to create the rule of choice. As each rule is generated, click Add Rule, which causes the present rule to be appended to the current rule base.

When the rule base is complete, it may be saved to a separate file for later examination or printing, but this is not essential. Pressing OK will complete the composition phase. At this point the project can be saved to a file by again selecting the Project menu item and selecting Save Project. A prompt appears for the name and directory for the file.

The final step is to compile the rulebase. This is accomplished by selecting the Rulebase menu and selecting Compile. The compiler dialog box appears. Pressing the Compile button will process the rule base, displaying the results in the box. Press OK to complete the process.

To run the software, select the Control menu item, and then select 'B008 Communications'. The TransFuzien Command and Control Panel is now displayed. This dialog screen contains the controls to load the kernel file to the Transputer system, to start processing, and to display results of the processing. The kernel program is down-loaded by pressing the 'Load Kernel' button, situated at the bottom left of the screen. This will display

a file selection box where the appropriate bootable code file (.bcf) may be selected. Pressing OK will initiate the down-load process.

After the kernel file is loaded, the Transputer network will execute the code, and will be awaiting commands from the user. The inference engine, data base manager, and knowledge manager processes need to be configured at this point. To load the rulebase, press the 'Load Rule Base' button. To load the inferencing options, press the Configure button, and to load the membership functions, press the 'Load Membership Functions' button. As each button is pressed, an acknowledgment is displayed on the screen. The system is now ready to process data.

5.5.2 Run-time operation

This software suite operates in one of two modes. The first is a single step mode where the input data is processed once, and then the processing stops. Pressing the Process Input Data button (located at the top left of the screen) will perform the single step processing. This is the default state.

The other mode is a continuous mode of processing. To enter this mode, the mode check box is checked, and then the Process Input Data button is pressed. In this mode, the software will continually process the selected inputs until requested to stop. This mode is useful in control applications.

The degree of fulfillment of each rule may be displayed by double clicking on any rule in the rule list box. The resultant membership function for the particular rule selected is displayed in graphical form. Similarly, the final membership functions may be displayed by double-clicking in the output list box.

There are other controls in this dialog screen for sending messages to the Transputer interface module (TIM), for reading analog data, and setting the pulse width modulation output of the TIM. These controls were incorporated at the early stages of the system's development to assist in hardware and software testing. They may be removed or enhanced in a later version of the software.

The operation of the system is more fully explored and described in Chapter 7 of this thesis.

5.6 Chapter Summary

This chapter has described the implementation of the algorithm for fuzzy rule evaluation. The inter-process communication channels and their protocols have been described. Data flow diagrams have been used to design and visualize the process interactions.

The process event graph shows the communication interactions between processes as a function of time. This graphical method is a useful tool that helps one to avoid the problem of communication deadlock. The usefulness of the PEG is evident from the examples given in this chapter.

The timing of process events has been examined for the single processor case, and for a multiple processor case. The algorithm has been mapped onto single and multiple Transputer architectures, and tests performed to determine advantages and problems associated with the two cases. The results show that the farming approach provides an improvement in processing time, however, the time to perform the RMF fusion and the subsequent defuzzification, over-shadow this to a large degree. Improvements in the fusion process in particular, would lead to improvements in performance.

Important issues have been identified that impact on the development of a processing architecture. A scheduling algorithm has been presented and implemented.

The individual processes that comprise the TransFuzien software package have been described.

Chapter VI

THE TRANSPUTER INTERFACE MODULE

6.1 Introduction

This chapter describes the design of the Transputer Interface Module (TIM), which provides a means of connecting the expert system to external digital and analog equipment. The block diagram of the TIM is shown in Figure 6.1. It has been designed to function as an embedded control processor, providing a mechanism for data transfer between external experimental apparatus and the expert system.

6.2 Description

The TIM is a self contained processing module that has its own on-board micro-controller, which communicates with the host Transputer via an Inmos C012 Link Adapter [43]. The C012 is configured to connect directly to port 0 of the Philips 87C752 [44], which is a low cost, 8 bit processor, that includes digital I/O, an on-chip 5 channel A/D converter, and a digital output that can be programmed to produce continuous digital pulses of variable pulse width. The program for the micro-controller is contained in an on-chip 1k EPROM. A number of routines have been programmed into this memory, which allow the 87C752 to perform a variety of tasks that include the following;

1. read the analog to digital converter channels
2. output a particular pulse width modulation to the PWM pin
3. data I/O to the designated port

The schematic design of the TIM is shown in Figure 6.3. The C012 link adapter translates between the serial protocol of the Transputer links, and a parallel 8 bit word that is used for communicating with peripheral devices. Restricting the system to 8 bits limits the dynamic range achieveable, but the TIM can be modified at a later date to provide 10 or 16 bit resolution.

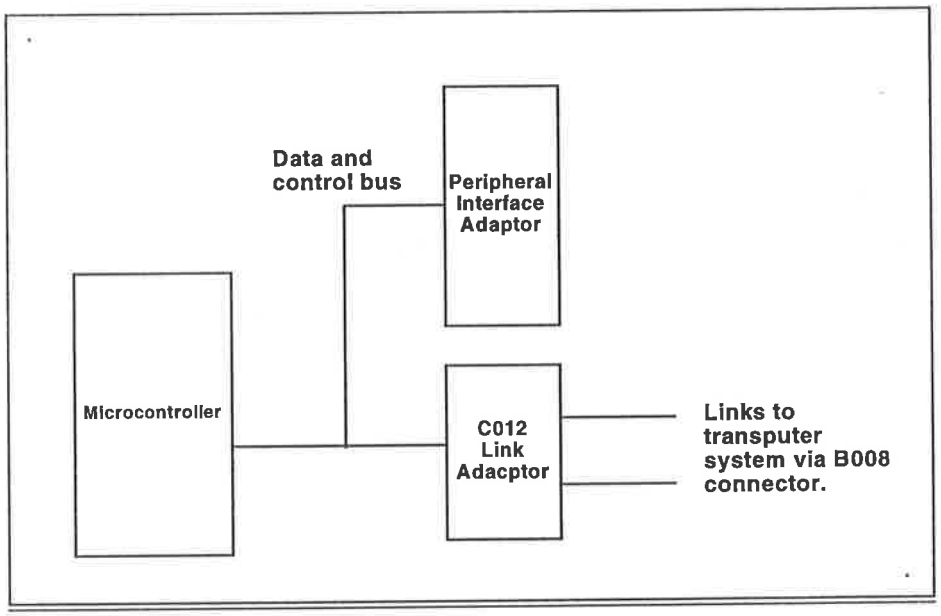
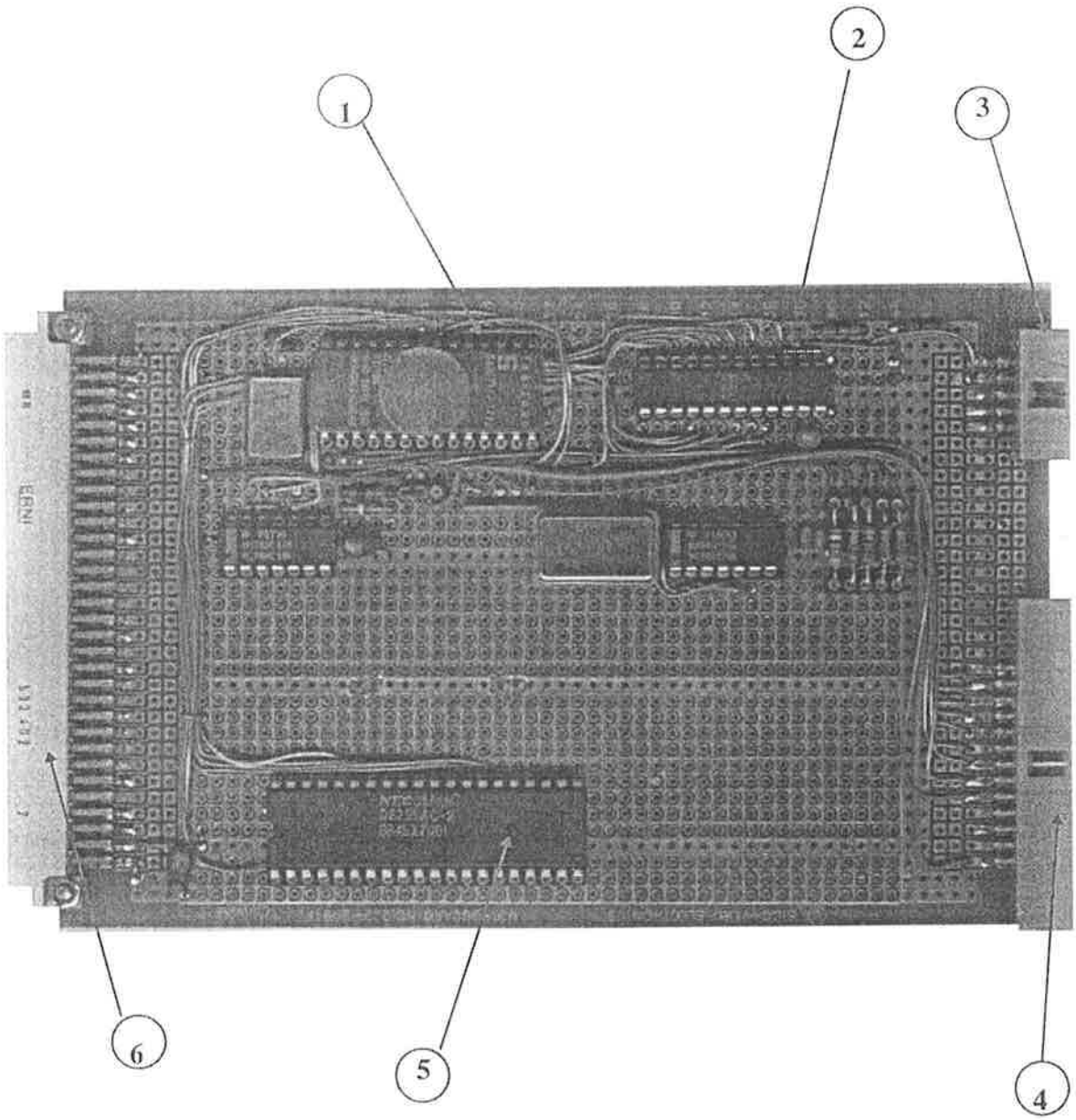


Figure 6.1 : Block diagram of the Transputer Interface Module, consisting of the micro-controller, the link adapter, the peripheral adapter, and signal conditioning hardware.



1. 87C752 Micro-controller
2. C012 Transputer Link Adapter
3. B008 Motherboard to TIM connector
4. Digital and analog inputs, and digital outputs connector
5. Peripheral Interface Adapter (PIA)
6. Euro connector for +5 volt supply

Figure 6.2 : Photograph of the Transputer Interface Module showing the micro-controller, the transputer link adapter and the PIA device.

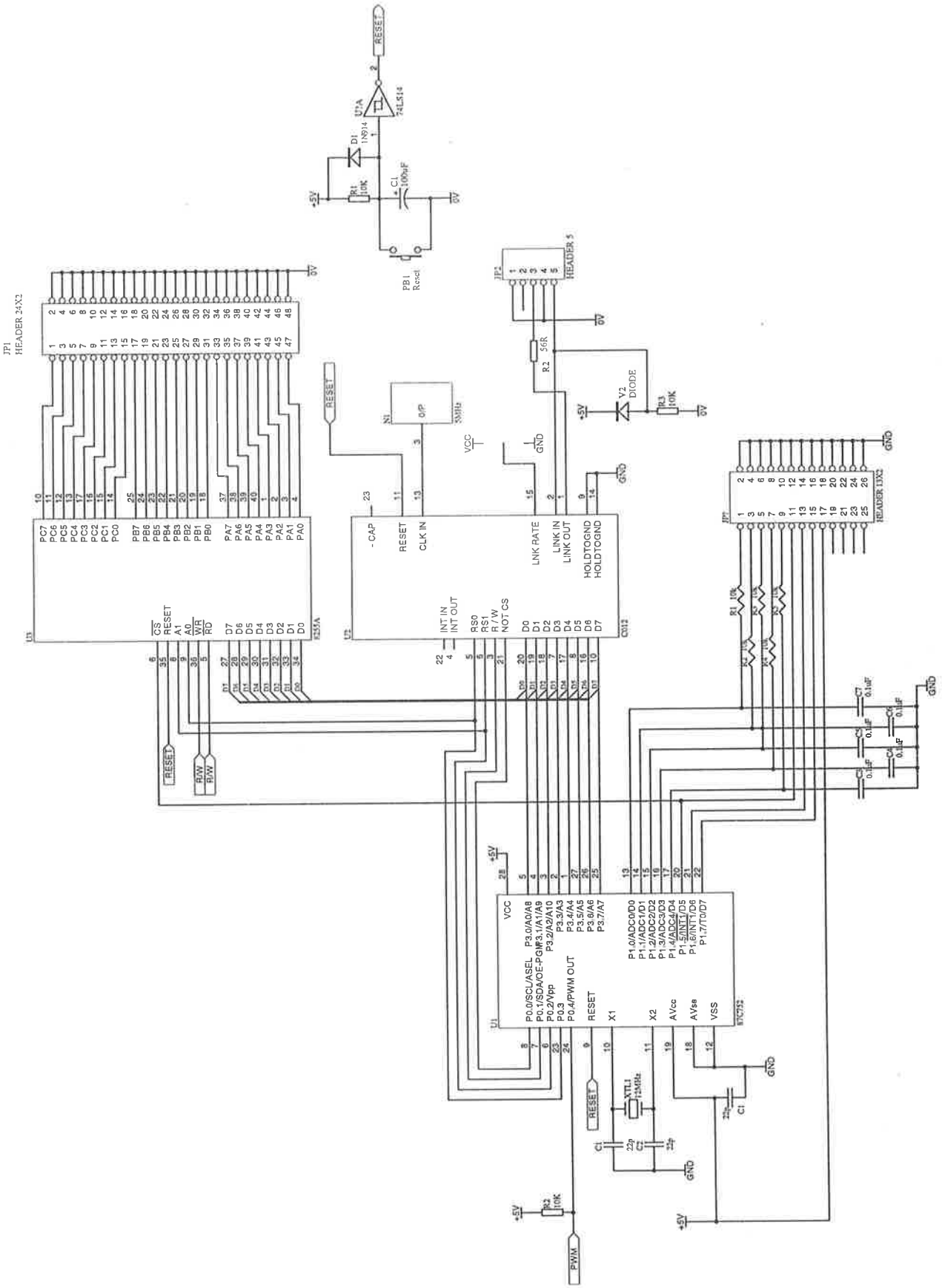


Figure 6.3: Circuit diagram of the Transputer Interface Module

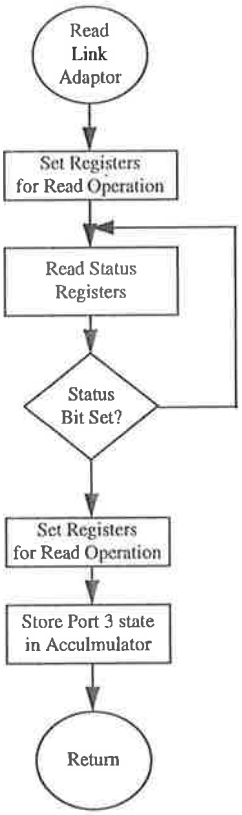


Figure 6.3 Flow chart for reading data from the link adapter

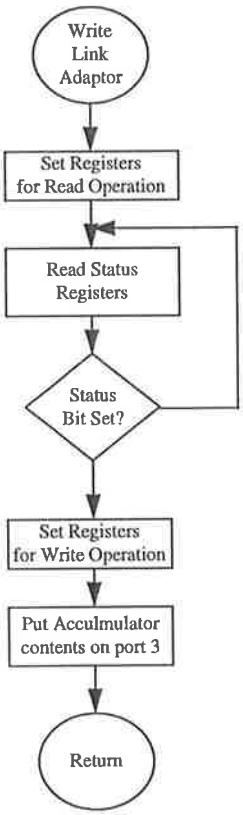


Figure 6.4 Flow chart for writing data to the link adapter

6.3 Operation of the TIM

When the micro-controller is reset, it executes a set of initialisation tasks, after which it waits for a command sequence to arrive from the transputer system via the C012 link adapter. The command is sent by the data base manager process, which in turn, received a command request from either the GUI, the FIE, or the Supervisor module.

The command comprises two bytes, sent consecutively to the TIM. The first byte identifies the type of command to be executed, and the second byte, holds relevant data such as the PWM duty cycle or the port settings for TTL outputs.

On receipt of the command, the TIM executes one of the following sub-routines;

1. Reset
2. Read the analog inputs to the micro-controller.
3. Set the pulse width modulation output
4. Set the TTL outputs P1.5, P1.6, and P1.7, to the desired state (see table 6.1)
5. Read the state of port 1

These routines provide a basic set of operations to interact with hardware that is attached to the TIM. Port 1 provides the 5 analog inputs and 3 digital outputs, and the PIA provides additional digital I/O for the system.

Data Byte	TTL Port Bits		
R2	P1.7	P1.6	P1.5
0	X	X	0
1	X	X	1
2	X	0	X
3	X	1	X
4	0	X	X
5	1	X	X

Table 6.1 Truth table for TTL port bit control

6.3 Chapter Summary

This chapter has described the Transputer Interface Module (TIM), which provides an electronic data connection between the expert system hardware and external peripheral electronic devices. The TIM provides the facilities to;

1. read the analog to digital converter channels
2. output a particular pulse width modulation to the PWM pin
3. read data from, and write data to the designated port of the micro-controller.

Chapter VII

APPLICATIONS OF FUZZY PROCESSING - CASE STUDIES

7.1 Introduction

In this chapter, the TransFuzien system is used to demonstrate fuzzy processing of data for a number of different cases. These include pattern classification, system modeling, and fuzzy control. In the first section, TransFuzien is used in the open loop mode, where various transfer functions are realised. In the second section, TransFuzien is operating in the closed loop mode, performing fuzzy control of an inverted pendulum apparatus.

7.2 Fuzzy Data Processing for a Multiple Input - Multiple Output System

In this section, the TransFuzien system has been programmed to evaluate output values for a multiple input - multiple output (MIMO) system [42]. In this case, there are four input variables X1, X2, X3, and X4, and three output variables Z1, Z2, and Z3. The rule base is shown in Table 7.1 and the inference methods employed are listed in Table 7.2. The results for three sets of input variables (*A, B,* and *C*) are shown in Figure 7.1.

The objective is to present an example of a MIMO configuration with an arbitrarily defined rule base to demonstrate how the outputs vary according to varying inputs.

Rule Number	Rule Text
0	IF x1 IS SmallPos THEN z1 IS SmallPos
1	IF x3 IS MediumPos THEN z1 IS MediumPos
2	IF x3 IS BigPos THEN z3 IS BigPos
3	IF x1 IS SmallPos AND x3 IS BigPos THEN z1 IS MediumPos
4	IF x2 IS LargePos AND x2 IS BigPos THEN z2 IS Zero
5	IF x3 IS Zero OR x2 IS MediumPos THEN z3 IS SmallPos

Table 7.1 There are 6 rules for this example, with 3 inputs and 3 outputs.

Inference Method	Type
Connectives	Zadeh (MIN - MAX)
Consequent Modifier	Scaling
RMF Fusion	Arithmetic Average
Defuzzification	Center of Gravity

Table 7.2 The inferencing methods for the MIMO example.

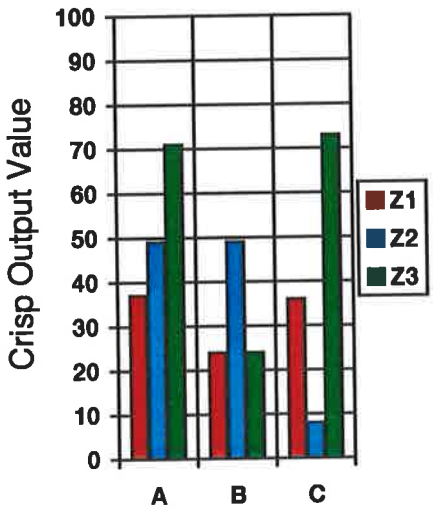


Figure 7.1: Non-fuzzy outputs *Z1*, *Z2*, and *Z3* are produced by a rulebase comprising six fuzzy rules. Three sets of input data give rise to corresponding output data variations.

7.3 Fuzzy Data Classification

In this section, data items are classified into classes. Each item is described by a vector of two features, X1 and X2. There can be more dimensions to the feature, but the results are not easily visualized beyond 3 dimensions. The feature space comprises a number of regions that each represent a prototype pattern. A set of fuzzy rules is applied to each feature, to map that feature to a particular partition of the feature space.

When writing a rulebase for pattern classification, the regions that form the class are described by rules. However, just as important are the rules that describe what regions do not belong to the classes of interest. These rules help to excise unwanted data, that may be considered as noise. This requirement is demonstrated particularly well in the rulebase for the torus example.

The sharpness of the classification process can be improved by using linguistic hedges to alter the shape of the membership values. The data for this example has been read from the transputer interface module's analog voltage inputs.

7.3.1 The Torus

This example has two classes *A* and *B*, that are concentric, with class *A* surrounding class *B* (see Figure 7.2). The rule base for this example is shown in Table 7.8. Rules 0 to 4 define membership of classes, and rules 5 to 21 specify exclusion from those classes.

7.3.2 Discussion

The rulebase has classified the two classes quite well. As the number of rules that govern the description of each class, the discrimination between classes improved. This is demonstrated in Table 7.9, where the output values for each class are shown before, and after the addition of more rules to exclude regions.

The results of the processing are displayed in Figure 7.3, where the two outputs (Class A and Class B) are plotted. Figure 7.3a shows the output for the Class A classifier, whilst Figure 7.3b shows the output for the Class B classifier. The *inverse* relationship is evident from these figures, showing that data that more fully belongs to one class, will belong to the other class with a smaller membership grade.

This case study demonstrates the ability of the system to classify data.

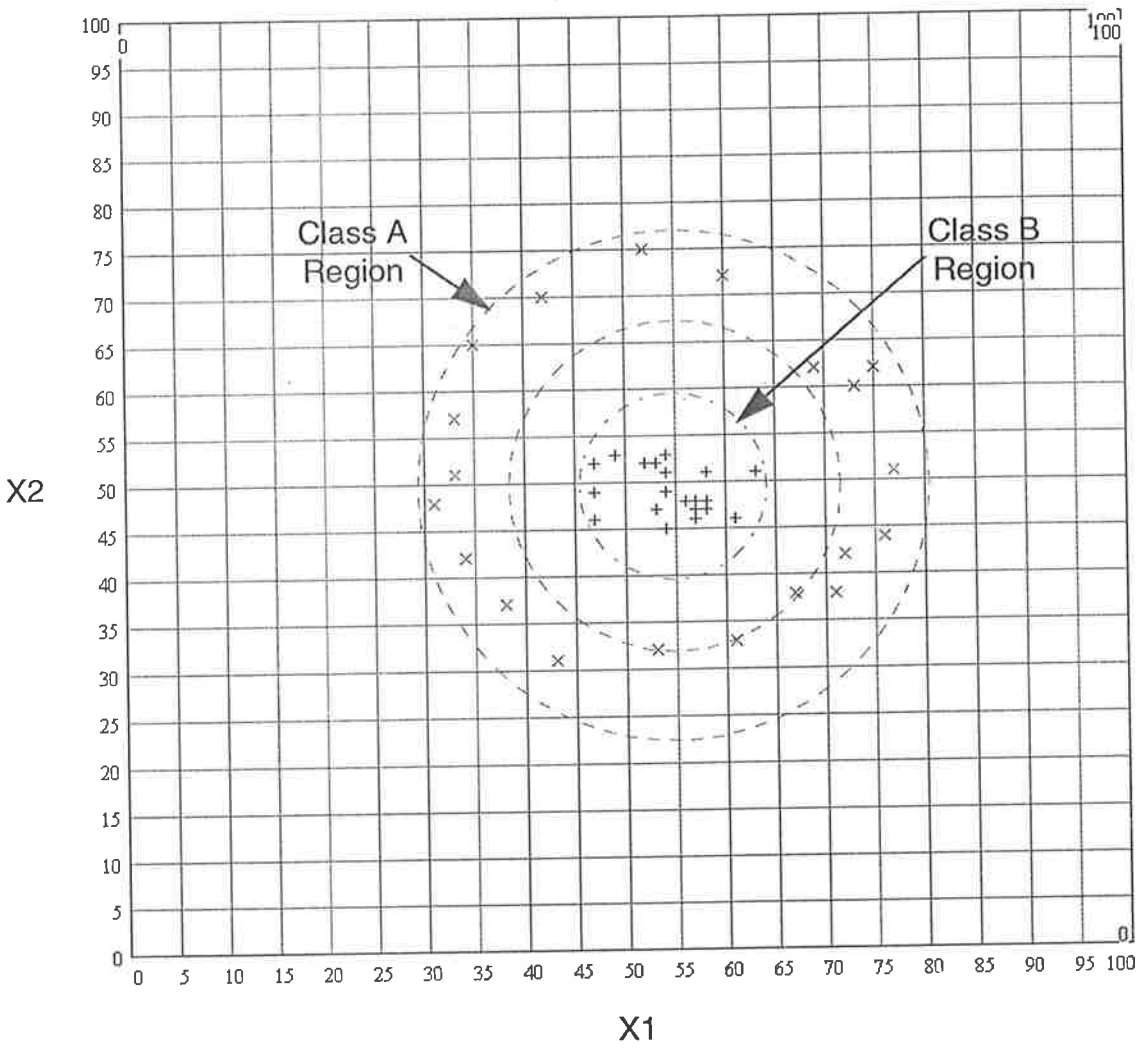


Figure 7.2 : Two regions are defined by the concentric circles shown in this figure. Class B is the central region of the figure, whilst Class A is the annulus that surrounds Class B. Data points which are defined by two coordinates *(x1, x2),* are classified by the rule base, and will possess membership to both classes to some extent.

Rule Number	Rule Text
0	IF x1 IS Small+ AND x2 IS Medium+ THEN classA IS Large+
1	IF x1 IS Big+ AND x2 IS Medium+ THEN classA IS Large+
2	IF x1 IS Medium+ AND x2 IS Big+ THEN classA IS Large+
3	IF x1 IS Medium+ AND x2 IS Small+ THEN classA IS Large+
4	IF x1 IS Medium+ AND x2 IS Medium+ THEN classB IS Large+
5	IF x1 IS Small+ AND x2 IS Small+ THEN classA IS Zero
6	IF x1 IS Large+ AND x2 IS Large+ THEN classA IS Zero
7	IF x1 IS Small+ AND x2 IS Large+ THEN classA IS Zero
8	IF x1 IS Large+ AND x2 IS Small+ THEN classA IS Zero
9	IF x1 IS Small+ AND x2 IS Small+ THEN classB IS Zero
10	IF x1 IS Small+ AND x2 IS Large+ THEN classB IS Zero
11	IF x1 IS Large+ AND x2 IS Small+ THEN classB IS Zero
12	IF x1 IS Large+ AND x2 IS Large+ THEN classB IS Zero
13	IF x1 IS Medium+ AND x2 IS Medium+ THEN classA IS Zero
14	IF x1 IS Medium+ AND x2 IS Large+ THEN classB IS Zero
15	IF x1 IS Medium+ AND x2 IS Zero THEN classB IS Zero
16	IF x1 IS Zero AND x2 IS Medium+ THEN classB IS Zero
17	IF x1 IS Large+ AND x2 IS Medium+ THEN classB IS Zero
18	IF x1 IS Medium+ AND x2 IS Big+ THEN classB IS Zero
19	IF x1 IS Medium+ AND x2 IS Small+ THEN classB IS Zero
20	IF x1 IS Small+ AND x2 IS Medium+ THEN classB IS Zero
21	IF x1 IS Big+ AND x2 IS Medium+ THEN classB IS Zero

Table 7.3: The rule base that classifies the data comprises 22 rules that define the two regions A and B.

	X1	X2	Actual Class	Class A Indicator	Class B Indicator
Before exclusion rules	60	72	Class A	79	86
After exclusion rules	60	72	Class A	79	19

Table 7.4 : This table shows the improvement in discrimination between Class A and Class B, from (79, 86) before, to (79, 19), after the addition of rules that define the class regions more fully. A high value represents a good match, 100 being the maximum value.

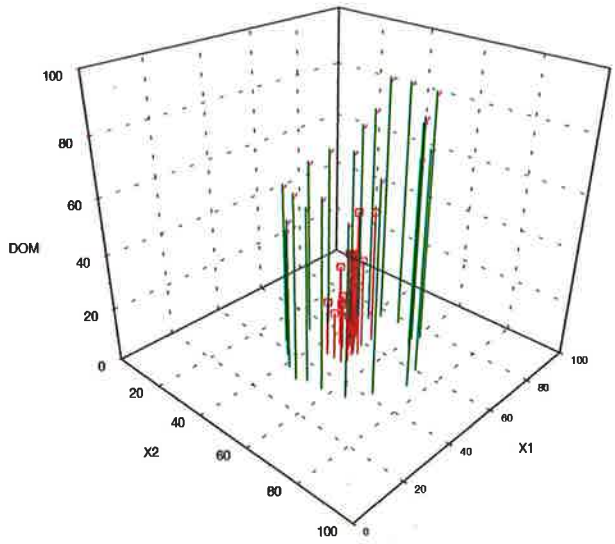


Figure 7.3a : Plot of the Class A output.

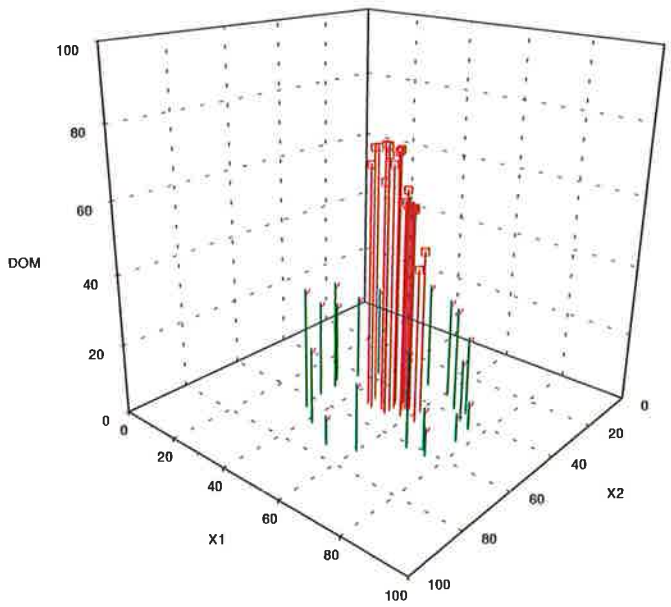


Figure 7.3b : Plot of the Class B output.

Class A data
Class B data

Figure 7.3 : Graph showing the degree to which data belongs to class *A* and Class B.

7.4 Modeling of a function using a fuzzy rule base.

Modeling of processes can involve complex non-linear equations, that must be solved, often in real-time. A fuzzy rule based system can be used to model these complex processes, provided sufficient care is taken in constructing the rulebase.

7.4.1 Linear Approximator

In this example, a rulebase, shown in Table 7.4, describes the behaviour of a simple linear function, $y = x$. Figure 7.4 shows the results of the data processing for several points between 0 and 100. The output values show good agreement with the ideal case. This is an example of a single input - single output (SISO) system where x is the input, and y is the output.

Rule Number	Rule Text
0	IF x IS Large+ THEN y IS Large+
1	IF x IS Big+ THEN y IS Big-
2	IF x IS Medium+ THEN y IS Medium+
3	IF x IS Small+ THEN y IS Small+
4	IF x IS Zero THEN y IS Zero
5	IF x IS Small- THEN y IS Small-
6	IF x IS Medium- THEN y IS Medium-
7	IF x IS Big- THEN y IS Big-
8	IF x IS Large- THEN y IS Large-

Table 7.5: Rule base that models the function $y(x) = x$.

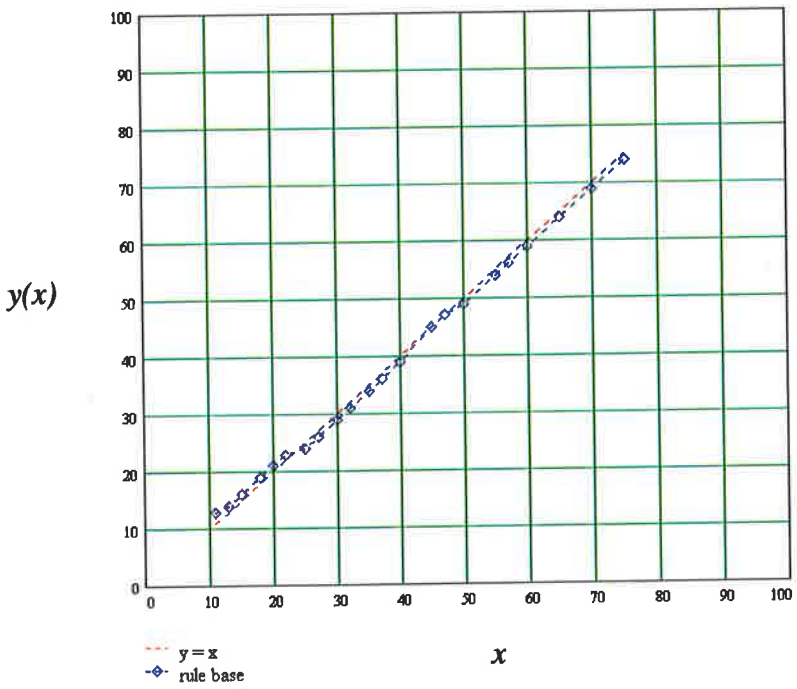


Figure 7.4 : The rule based model of the function $y(x) = x$ closely matches the ideal case. Triangular membership functions were applied in this example.

7.4.2 Complex Function Approximator

More complex functions can be modeled using fuzzy logic, in this case, the function is $z(x,y) = x^2 - y^2$. A useful method for developing models with fuzzy sets, that have a two input-one output relationship, is to use input-output maps, or FAM. The corresponding FAM is shown in Figure 7.5, from which a rulebase is derived. The rules are then read directly from this map and are shown in Table 7.10.

From the resulting plots (see Figures 7.6a-d), it can be seen that there is a close correlation between the theoretical and modeled plots. A finer resolution may be obtained by using additional rules to model the function, but this involves a trade-off between complexity, and speed of processing (if speed is an issue).

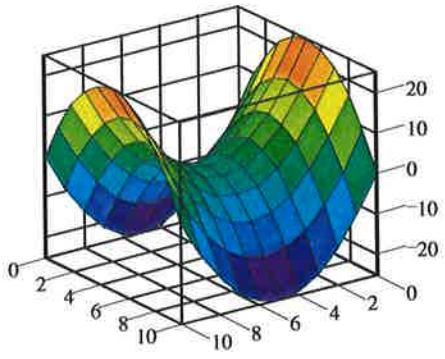
The major features match those of the theoretical plot for z. The scale can be adjusted by the system that uses the output values.

X / Y	NL	NS	ZE	PS	PL
NL	ZE		PL		ZE
NS			PS		
ZE	NL	NS	ZE	NS	NL
PS			PS		
PL	ZE		PL		ZE

Figure 7.5 : Key features of the model are identified using a simple matrix approach that maps the input space to the output space (I/O Map). The inputs to the matrix are x and y. The membership labels are NL, NS, ZE, PS and PL.

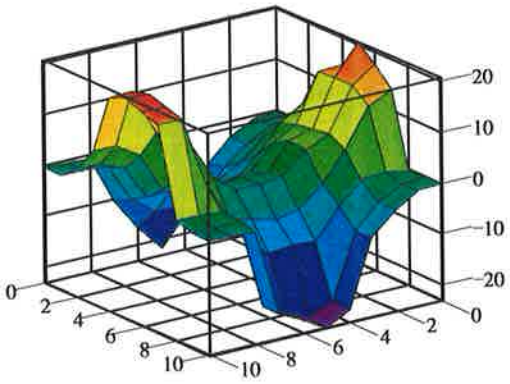
Rule Number	Rule Text
0	IF x IS Zero AND y IS Zero THEN z IS Zero
1	IF x IS Large- AND y IS Large- THEN z IS Zero
2	IF x IS Large+ AND y IS Large- THEN z IS Zero
3	IF x IS Large- AND y IS Large+ THEN z IS Zero
4	IF x IS Large+ AND y IS Large+ THEN z IS Zero
5	IF x IS Large- AND y IS Zero THEN z IS Large-
6	IF x IS Large+ AND y IS Zero THEN z IS Large-
7	IF x IS Zero AND y IS Large- THEN z IS Large+
8	IF x IS Zero AND y IS Large+ THEN z IS Large+
9	IF x IS Zero AND y IS Medium- THEN z IS Small+
10	IF x IS Zero AND y IS Medium+ THEN z IS Small+
11	IF x IS Medium- AND y IS Zero THEN z IS Small-
12	IF x IS Medium+ AND y IS Zero THEN z IS Small-

Table 7.6 : Rule base that describes z, according to the I/O map. The rule base is derived from this mapping as shown in Figure 7.5.



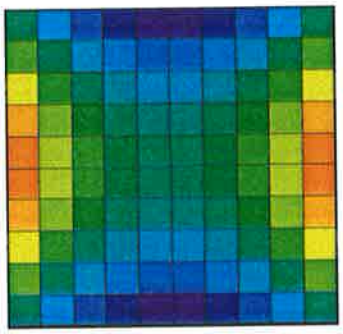
Z

Figure 7.6a : Calculated surface plot of the function $z(x,y) = x^2 - y^2$, $x = [-5..+5]$ and $y = [-5..+5]$.



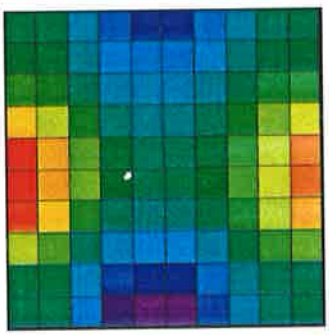
N

Figure 7.6b : Surface plot generated by the expert system using 13 rules.



Z

Figure 7.6c : Calculated surface plot of the function $z(x,y) = x^2 - y^2$. Blue represents negative numbers, green represents zero, and orange represents positive numbers.



N

Figure 7.6d : The surface plot generated by TransFuzien software shows a high degree of correlation with the theoretical plot of Figure 7.7c.

Figure 7.6 : Modeling relies on the identification of key features of a system, and then encoding these with suitable rules. The rule based model of the function $Z(x,y) = x^2 - y^2$ (Figure 7.6b) closely matches the ideal case. The scale is not relevant here, as the expert system output can be adjusted to suit the application.

7.5 Signal Processing

This example illustrates how a fuzzy rule base may be used to perform filtering of information. For the purposes of this case study, the filters perform a bounds classification on the input data. Two types of filter are tested, being a low pass filter and a band pass filter. A filter has a number of important characteristics such as the cut-off point, the roll-off, the band bass and band gap ripple, the band stop rejection, and the insertion loss.

The input and output variables for this case have the following ranges;

Input variable :x1 Range = [0..100]
Output variable :z1 Range = [0..100]

Linguistic hedges are one method by which the sharpness of a filter can be altered. Another is to adjust the shape of the membership functions. The membership sets are fuzzy numbers and can be defined to suit the filter characteristics that are desired. (ie. cut-off value, half power point) In this case, generic terms are used only such as low, medium and high.

7.5.1 A Low Pass Filter

The rule base for the low pass filter is shown in Table 7.11. These rules are derived by considering the desired low pass filter characteristics. Figure 7.7 shows the output from the inferencing process as a function of the input variable x. This figure highlights the influence of the choice of inferencing strategy on the output.

Rule Number	Rule Text
0	IF x1 IS low THEN z1 IS large
1	IF x1 IS small THEN z1 IS large
2	IF x1 IS medium THEN z1 IS small
3	IF x1 IS large THEN z1 IS zero

Table 7.7: Rule base for the low pass filter.

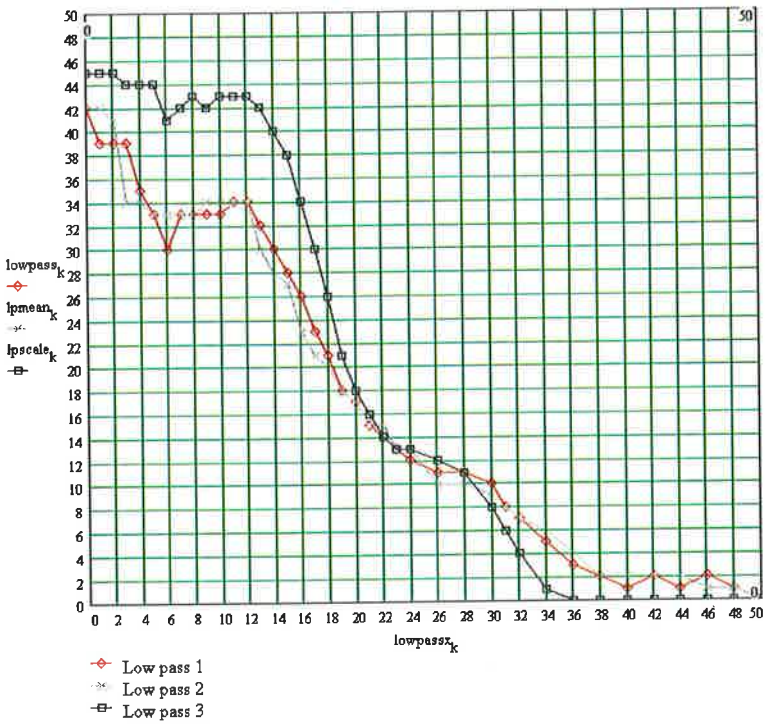


Figure 7.7 : The output for the low pass filter is affected by the choice of inferencing methods.

FILTER	Connective	Modifier	Fusion	Defuzzification
Low pass 1	min-max (Zadeh)	truncation	peak	Center of gravity
Low pass 2	min-max (Zadeh)	truncation	mean	Center of gravity
Low pass 3	min-max (Zadeh)	scale	peak	Center of gravity

Table 7.8: Inference methods for the 3 low pass filters.

7.5.2 A Band Pass Filter

The rule base for the band pass filter is shown in Table 7.12. These rules are derived by considering the desired low pass filter characteristics. Figures 7.8 shows the output from the inferencing process as a function of the input variable x, and highlights the influence of the choice of inferencing strategy on the output.

Rule Number	Rule Text
0	IF x1 IS low THEN z1 IS zero
1	IF x1 IS small THEN z1 IS large
2	IF x1 IS medium THEN z1 IS large
3	IF x1 IS large THEN z1 IS zero

Table 7.9 Rule base for the band pass filter.

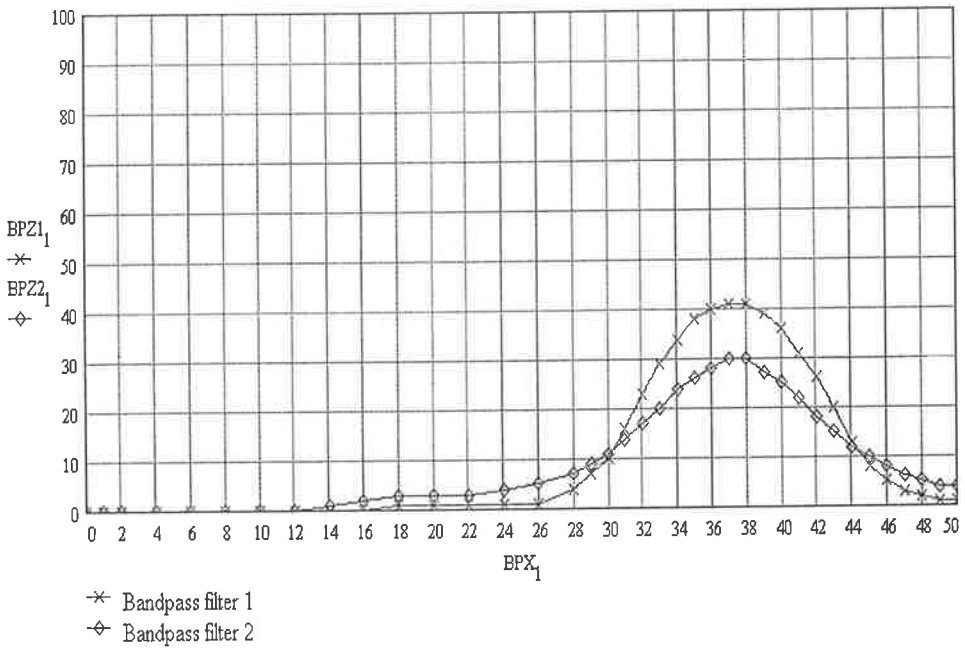


Figure 7.8 : The output responses for two bandpass filters. The variation is due to the fusion method employed for each filter.

The two data sets are the result of applying the same rulebase to identical input data. Bandpass filter #1 used the maximum profile, whilst bandpass filter #2 used averaging for the fusion method. These results show that the fusion method affects the response of the filters.

7.6 Real Time Control - The Inverted Pendulum

To achieve real-time control of a dynamic system, the relevant time constants must be considered. Control decisions based on the sensor input, need to be made quickly enough that the system under control is receiving command inputs that result in the desired behaviour. That is, the bandwidth of the decision making system is large enough to accommodate the time constants of the plant[1].

There are many instances in the literature that describe the inverted pendulum [23, 45, 46] and show the complexity of the equations of motion for this system. In this study, the variable that is measured is the angle of the pendulum. The angle sensor is read by channel 0 of the analog to digital converter on the Transputer Interface Module. More elaborate sensing and rules can be implemented as an extension to this study. The system diagram is shown in Figure 7.9.

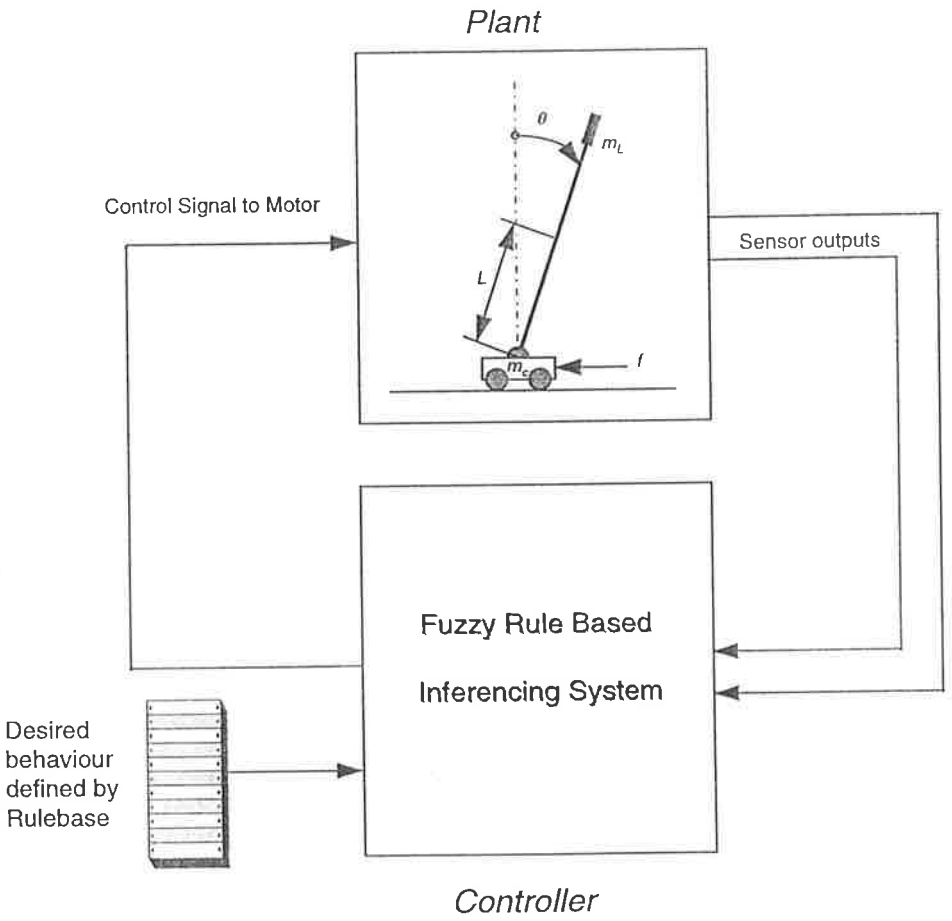


Figure 7.9 : The inverted pendulum apparatus is the plant in this control loop.

[1] The plant is the controlled system.

7.6.1 *Description of the Apparatus*

The inverted pendulum apparatus has been totally designed and built by the author as a test-bed for the real-time control section of this study. Figure 7.10 shows the relevant features of the apparatus. It comprises a small trolley with 8 wheels, that engage a level linear track on each side of the trolley. The pendulum is mounted on the trolley, and is free to rotate about the pivot point.

The position of the trolley on the track, and the angle of the pendulum, are sensed by linear potentiometers. The outputs are connected to the analog inputs on the TIM. The motor is a 24 volt DC servo motor. The motor voltage is controlled by the motor control card (see Figure 7.11), the circuit diagram of which is shown in Figure 7.12.

The pole is of length 2L, the mass of the cart is m_c, and the additional load mass is m_L. The force f applied to the cart is regulated by the PWM output from the TIM, and an additional TTL signal, also from the TIM, which switches the direction of the motor. The rulebase used in this experiment is very simple as shown in Table 7.13.

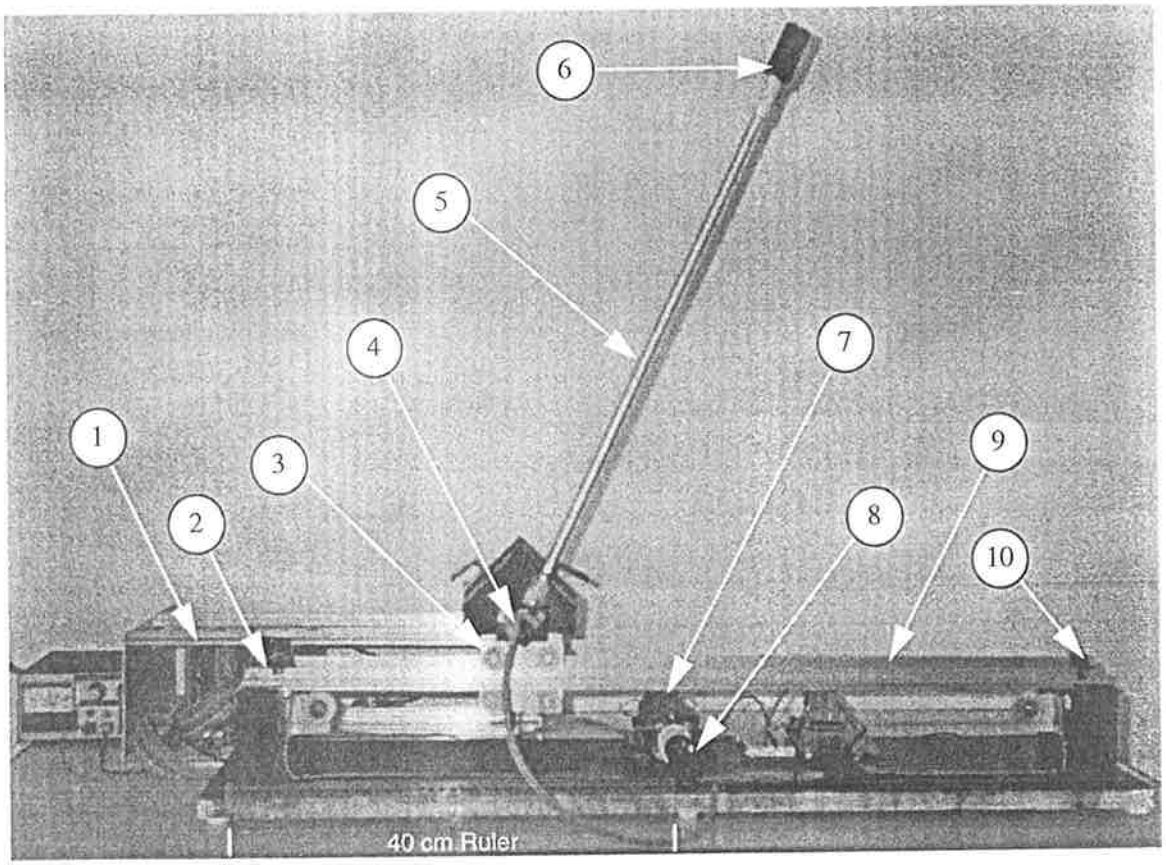
7.6.2 *Pendulum Motor Drive and Sensor Card*

The pendulum motor drive and sensor card, receives commands from the fuzzy processing system and converts them into the appropriate electrical signals for the motor. Data from the angle sensor, the linear position sensor, and the end sensors, are conditioned, then sent to the generic module for processing.

This hardware comprises an H-bridge circuit, and motor current monitoring circuitry. Four IRLZ14 logic level N-channel MOSFETs are used for the H-bridge. Each MOSFET is rated at Id = 10A continuous (40A pulsed). The logic signals required for the H-bridge are generated by an ALTERA 7032 EPLD.

A separate power supply is used for the motor. This supply is isolated from the digital supply by opto-couplers. The PLD output are buffered by an open collector buffer chip (7407), which in turn drive the opto-coupler LEDs.

The angle sensor comprises a 10k linear potentiometer. The end sensors are photo-electric transmitter-receiver devices, which detect an interruption to the beam, however these were not used in this case study.



Description of components.

1. Control electronics
2. Infrared sensor
3. Trolley assembly
4. Angle sensor
5. Aluminium rod
6. Cylindrical weight
7. Electric motor
8. Position sensor
9. Aluminium track for the trolley
10. Infrared sensor

Figure 7.10 : Photograph of the inverted pendulum apparatus.

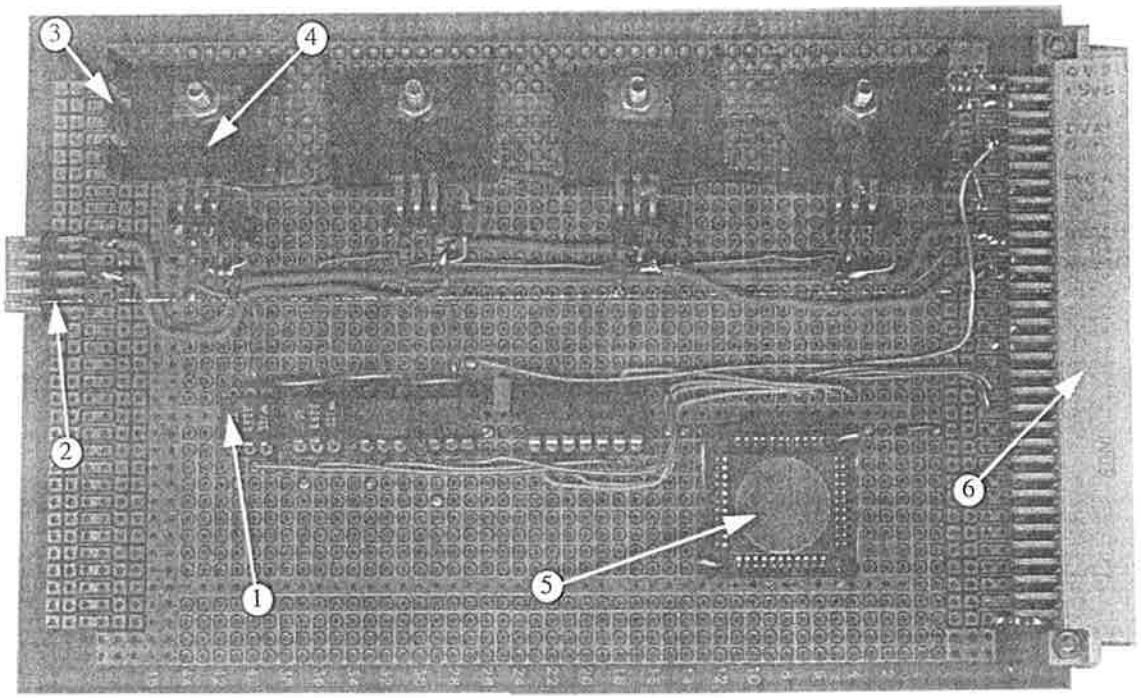


Figure 7.11 : Photograph of the Motor Control Module for the inverted pendulum motor.

Description of components:

1. Opto-coupler
2. Motor Power connector
3. Heatsink
4. Mosfet
5. EPLD
6. Signal and power connector

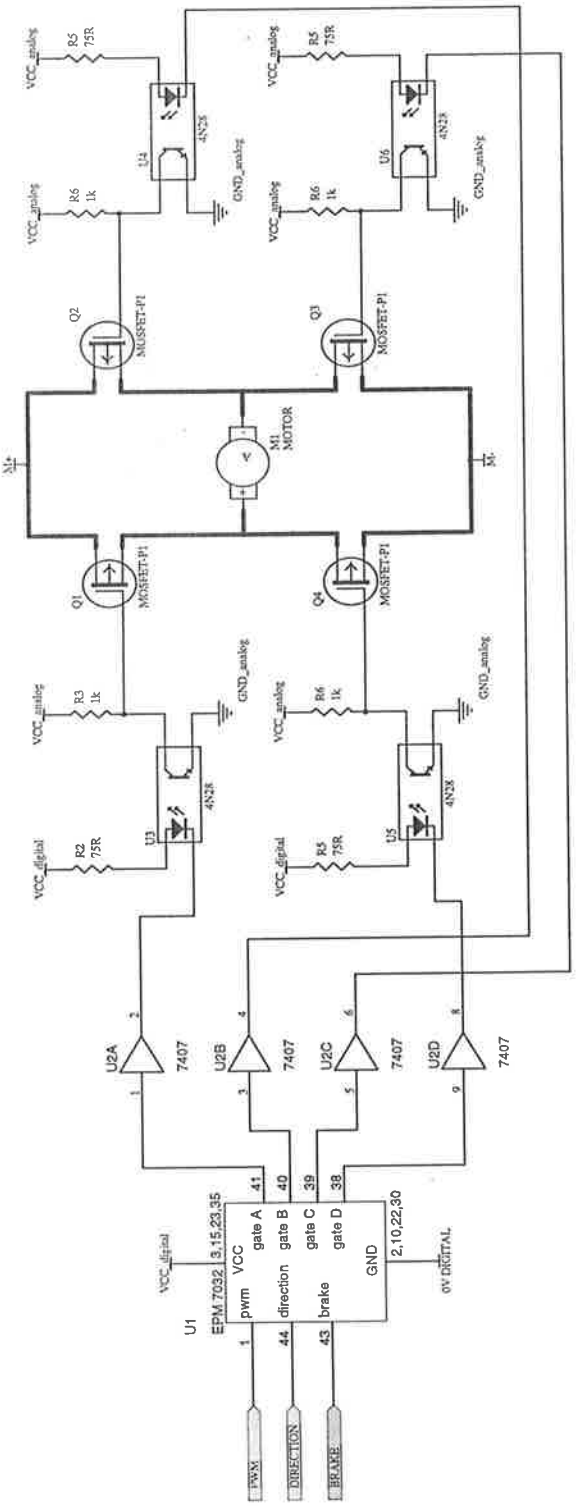


Figure 7.12 : Circuit diagram of the Motor Control Module for the inverted pendulum motor.

7.6.3 Experimental procedure and results

The Rulebase

Rule Number	Rule Text
0	IF angle IS Zero THEN force IS Zero
1	IF angle IS Large THEN force IS Large+
2	IF angle IS Medium+ THEN force IS Large+
3	IF angle IS Small- THEN force IS Large-
4	IF angle IS Medium- THEN force IS Large-

Table 7.13 : The Rulebase for the inverted pendulum.

7.9

Static Performance

In the first phase of the experiment, the system was loaded with the pendulum project and run in loop mode. The time taken to evaluate the rulebase and generate the required output signals for the apparatus was 98 mS. The motor current was switched off, and the angle of the pendulum was adjusted. For each increment from the vertical, the output was recorded. The measurements were repeated for two fusion methods, being the arithmetic mean and the peak follower. The measurements are shown in Figure 7.13.

The graph shows that as the angle from the vertical increases from zero, to either side, the force applied to the cart is increased so as to correct the imbalance. Both methods of rule fusion produce the predicted result, with little difference between them. The processing times for this case are shown in Table 7.2.

Dynamic Performance

The second phase of the experiment tested the dynamic performance of the inverted pendulum controller. The same project file was loaded and the system was then run in the closed loop mode for two situations.

Restrained Test:

The Pendulum was supported in the vertical position (angle = 0) and the motor supply switched on. The pendulum was held in an upright position, then moved to either side, and the motion of the cart observed.

Unrestrained Test:

The next step was to support the pendulum in the upright position, with the cart at the center point of the track, and then release the pendulum and allow the system to control the apparatus.

7.6.4 Observations

Restrained Test:

1. As the angle from the vertical increased, the duty cycle of the drive voltage to the motor increased. The duty cycle output from the TIM was monitored with an oscilloscope.
2. The cart was subsequently driven to correct the error in the angle as detected by the system.
3. This behaviour of the cart was consistent with the behaviour encoded in the rule base file.

Unrestrained Test:

1. When the pendulum was released, it tended to stay in an upright position due to the friction of the pivot and the lack of external disturbance. In this state the motor current was zero, as expected from the rulebase. Again the motor drive modulation was monitored on an oscilloscope, together with the direction output[2].
2. The pendulum was then perturbed to one side to initiate the control action. In this state the cart was driven in accordance with the angular displacement of the pendulum.
3. The best performance, defined as a balanced upright pendulum, was achieved using Zadeh connectives, scaling as the modifier, arithmetic mean as the fusion process, and center of gravity defuzzification. This state was only achievable for a maximum of 8 seconds before the system became unstable (ie. the pole could no longer be kept upright).

[2] The direction output is a digital signal that connects to the EPLD on the motor control board. It controls the switching of the FET H-bridge.

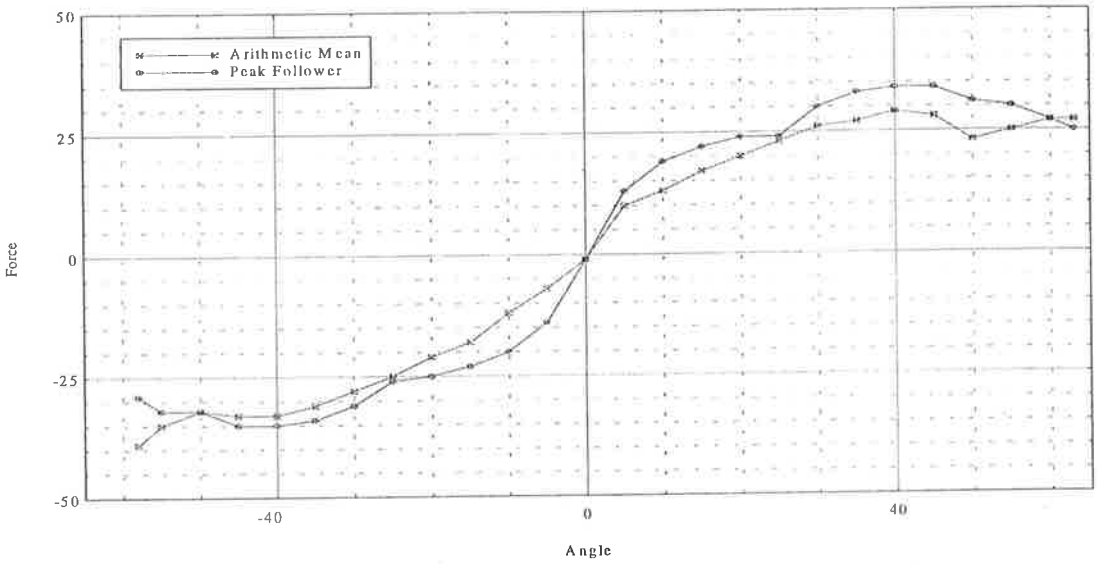


Figure 7.13 : The system output for the pendulum control is determined by the angular displacement of the pole from the vertical. This figure shows the effect of selecting different fusion processes.

7.6.5 Comments

A performance improvement could be obtained by using an additional input that detects the angular velocity of the inverted pendulum. Some mechanical improvements to the pivot bearing would also reduce undesirable movement of the pendulum in the direction perpendicular to the direction of the cart motion.

During development of the test-bed, and subsequent testing, the sensors were placed under considerable mechanical stress, and their performance eventually become somewhat degraded. This is an area that will require attention in future experiments with this test-bed.

There were noticeable changes in operation when different inferencing methods were applied to the closed loop system. This ability of the GUI software to apply changes whilst processing data, proved useful, and was a positive result from this case study.

Further experimentation[3] with inference settings and with the shapes of membership functions, would be useful in exploring the dynamic behaviour of the controller.

[3] However time did not permit further development at this stage.

7.7 Chapter Summary

This chapter has presented a number of case studies that show how fuzzy inferencing may be employed in a variety of data processing tasks. The results for each case testify to the usefulness of the fuzzy approach to information processing, and demonstrate that the TransFuzien system provides a useful research tool for investigating fuzzy inferencing.

Some problems with the mechanical structure of the inverted pendulum apparatus need to be resolved to investigate this real time experiment further. The pulse width modulation of the electric motor could be replaced with a linear power amplifier, which would reduce the harmonics generated in the motor coil by the modulating square wave. Also, velocity sensors can be added to the apparatus to provide an additional input to the expert system. The rule base would be change accordingly to reflect the new input data.

Chapter VIII

THESIS SUMMARY

8.1 Discussion

This research has set out to explore the concepts of fuzzy logic, and to investigate how an inferencing system can be developed using parallel processing concepts. The principles of fuzzy logic have been introduced, and a number of new concepts have been proposed. These include;

1. t-norm and t-conorm operators for fuzzy set interaction
2. operators for handling temporal aspects of knowledge representation
3. fusion operators

As the outcomes in the form of resultant membership functions, are calculated, the inherent information content which is represented by the RMF, will vary according to the degree to which each rule is satisfied. A method for calculating an information measure for fused fuzzy sets has been developed and presented.

The fusing of fuzzy sets has been addressed, with common methods presented, and two alternatives proposed. The first method is based on the proposed information measure, and the second method is based on a sliding window average. Examples of each method have been presented.

In considering a parallel implementation of fuzzy inferencing, it has been shown that a suitable representation of knowledge is important, and has been achieved using fuzzy rules which have the *IF-THEN* format. A fuzzy rule compiler has been presented that translates textual rules into a format suitable for processing by the Inference Engine.

The INMOS Transputer has been used to implement the inferencing strategy proposed in this research. A number of special factors relating to parallel processing systems have been explained. These include data flow processing, and methods for avoiding deadlock in a multi-processor architecture, which include the use of formal protocols for communications. The inferencing modules have been successfully mapped onto a two-Transputer system.

The various software components that have been developed, comprise a complete fuzzy inferencing expert system. This package has been given the name TransFuzien, which stands for *Transputer-based Fuzzy Logic Inference Engine*. The system comprises a number of components that includes, a Graphical User Interface that is designed to facilitate easy composition of fuzzy rules, membership functions, and inferencing strategies. A number of the dialog boxes of the GUI are shown. The GUI provides the human-computer interface to TransFuzien, and it has been shown how the GUI is used to encapsulate and pre-process domain knowledge, configure the inferencing module, and monitor run-time results. The GUI adds to the ability to experiment with different inferencing strategies, and directly observe the results.

Algorithms have been developed and presented which form the basis for fuzzy inferencing, together with a parallel processing architecture, based on the Inmos Transputer. The algorithms that run on the Transputer system are realised in the Occam 2 programming language.

An electronic interface has been designed and constructed by the author to facilitate an electronic connection between the expert system and external equipment. This interface provides a means for reading analog and digital data, and for writing digital data to peripheral devices.

Evaluation of the system has shown the operation of the components of the pre-processing algorithms that form an important part of the graphical user interface. The overall performance of this system has been demonstrated with specific examples. These include fuzzy modeling, fuzzy pattern classification, and fuzzy control.

The results from this research obtained are positive, firstly from the point of view that a working inferencing system using parallel processing concepts, has been developed, and secondly, by demonstrating the outcomes of the research by the above mentioned case studies. A performance enhancement has been demonstrated by applying parallel processing concepts. The limitations of the system, including a limit on the number of rules to 100, and the current number of Transputers to 2, need to be addressed in further developments. There is scope for experimenting with various process farming strategies, which will be particularly relevant for mapping the software to larger transputer networks. Also, the algorithms developed herein could be applied to other types of processor target systems, apart from the Transputer.

This work has shown the importance of taking a systems approach to the investigation, development and implementation of solutions to complex engineering problems.

8.2 Further Work

As the system has evolved, it has become apparent to the author how this current version of the software can be improved in many areas. This is the case with most research and engineering projects, where subsequent improvements are made, based on the things that are discovered during the first phase of development.

Further developments of the system can focus on a number of areas. The first is the processing performance, which may be improved with the use of the T9000 Transputer [47]. The existing Occam code would need to be re-compiled for the T9000, and some other changes made to the hardware interface, including the addition of an IMSC100 Parallel DS-Link Adapter [48]. This device converts the T9000 serial protocol to the T800 serial protocol.

The second area that can be addressed, is the Graphical User Interface. It could be modified to reflect operator preferences, such as the colours used for the various screen displays. Also, the range of inferencing options could be enhanced. The rule compiler can be extended to accommodate multi-consequent rules.

Next, the measure of fuzzy information is a useful quantity to calculate. It could be displayed by the graphical user interface, to give an indication of the quality of the outcomes of the expert system. This would be a subjective measure, but would be relevant and useful to further research into fuzzy information processing.

In Chapter 2, new linguistic operators have been introduced called *WAS* and *WILLBE*. Further analysis using these operators is required, and particularly in the area of the forecasting methods that may be employed.

The TransFuzien system has provided a means of exploring the affects of the choice of inferencing strategies for particular applications. The TransFuzien system could be used to derive a method for recommending the inferencing strategy (perhaps a heuristic method) that is best suited for the task at hand.

Further research would prove fruitful in the area of real-time data processing. The inverted pendulum experiment is a good test-bed for exploring various inferencing methodologies, however, particular attention needs to be given to the mechanical components of such a system.

Increasing the number of Transputers in the system would be useful. This would require some further effort to address the problems raised by an increase in communication complexity, and task assignment between processors.

References

[1] Charniak E. and McDermott D., "Introduction to Artificial Intelligence", Addison-Wesley, 1985.

[2] Rich E., "Artificial Inteligence", McGraw Hill, 5th printing, 1986.

[3] Zadeh L. A., "Fuzzy Sets", Information and Control, Vol. 8, No. 3, pp. 338-353, 1965.

[4] Zimmermann J.,"Fuzzy Sets, Decision Making, and Expert Systems", Kluwer Academic Publisers, 1987.

[5] Hopgood A. A., "Knowledge-Based Sytems for Engineers and Scientists", 1993.

[6] Yasunobu, S. and Miyamoto, S., "Automatic Train Operation System by Predictive Fuzzy Control", in "Industrial Applications of Fuzzy Control", Sugeno M. editor, Elsevier Science, 1985.

[7] Harbison-Briggs K. and McGraw K., "Knowledge Acquisition - Principles and Guidelines", Prentice Hall, 1989.

[8] INMOS, "The Transputer Development and iq Systems Databook", 1st edition, 1989.

[9] Bardossy A. and Duckstein L., "Fuzzy Rule-based Modeling with Applications to Geophysical, Biplogical and Engineering Systems", CRC Press, 1995.

[10] Motorola Semiconductor Products, "Motorola Fuzzy Logic Data Sheet", 1996.

[11] Infra-Logix Data Sheet.

[12] Seimens Data Sheet "Fuzzy Logic Coprocessor, SAE 81C99".

[13] VLSI Technology Inc., Data Sheet "VY86C500 12-bit Fuzzy Computational Accelerator", 1993.

[14] Liu L, "Optical Implementation of Parallel Fuzzy Logic", Optics Communications,

[15] Hasnain S. B.and Linkens D. A., "The Use of AI Methodology in Control Applications of Transputers", IEE Colloquium on Recent Advances in Parallel Processing for Control", Volume 7, pp 1-10, 1988.

[16] Stachowicz M. S., "Parallel Architecture for a Multi-Input Fuzzy Logic Controller", SPIE Vol. 1707, Applications of Artificial Intelligence X: Knowledge-Based Systems, pp. 137-145, 1992.

[17] Ekerol H and Hodgson D. C., "Angular position control of objects using a transputer-based vision system and fuzzy logic techniques", Proceedings of Institution of Mechanical Engineers, Vol. 207, 1993.

[18] Stachowicz M. S., "A hardware accelerator for linguistic data processing", SPIE Vol. 1607, Intelligent Robots and Computer Vision X: Algorithms and Techniques, pp. 439-445, 1991.

[19] Zapata E. L., Rivera F. F, Plata O. G. and Ismail M.A., "Parallel Fuzzy Clustering on Fixed Size Hypercube SIMD Computers", Parallel Computing II, pp. 289-303, 1989.

[20] Liu L., "Optical Pattern Fuzzy Logic", Japanese Journal of Applied Physics, Vol. 29, No. 7, pp L1281-L1283, July 1990.

[21] Dubois D. and Prade H., "Fuzzy Sets and Systems: Theory and Applications", Academic Press, 1980.

[22] Estathiou J., "Expert Systems in Process Control", Longman, 1989.

[23] Kosko B, "Neural Networks and Fuzzy Systems", Prentice Hall, 1992.

[24] Yagishita, O. Itoh, O. and Sugeno M., "Application of Fuzzy Reasoning to Water Purification Process", in "Industrial Applications of Fuzzy Control", pp. 19-39, Sugeno M. editor, Elsevier Science, 1985.

[25] D`Azzo J. J. and Houpis C. H., "Feedback Control System Analysis & Synthesis", 2nd edition, McGraw Hill, 1982.

[26] Klir G. J. and Folger T. A., "Fuzzy Sets, Uncertainty, and Information", Prentice Hall, 1988.

[27] Zadeh, L., "Fuzzy languages and their Relation to Human and Machine Intelligence", 1970.

[28] Gupta M. M. and Qi J.,"Connectives (AND, OR, NOT) and T_Operators in Fuzzy Reasoning", pp. 211-233, in "Conditional Logic in Expert Systems", Elsevier Science B.V. (North Holland), 1991.

[29] Chatfield, C., "The Analysis of Time Series", 4th edition, Chapman Hal, 1989.

[30] Bowyer R.S., "Implementation of a Parallel Fuzzy Logic Controller", ATOUG-4 The Transputer in Australasia, IOS Press, Amsterdam, pp. 13-18, Sept. 1991.

[31] Haykin S., "Communications Systems", 2nd edition, Wiley, 1983.

[32] Aho A., Sethi R. and Ullman J., "Compilers : Principles, Techniques, and Tools", Addison Wesley, 1977.

[33] Mick J. and Brick J., "Bit-Slice Microprocessor Design", McGraw-Hill, 1980.

[34] Borland, "C++ Programmer's Guide", Version 4.0, 1993.

[35] Pountain D. and May D., "A Tutorial Introduction to OCCAM Programming", BSP Professional Books, 1987.

[36] INMOS, "OCCAM 2 Reference Manual", Hoare C. A. ed., 1988.

[37] INMOS, "Transputer Development System", Prentice Hall, 1988.

[38] Fox G., Johnson M., et al., "Solving Problems on Concurrent Processors, Volume 1, General Techniques and Regular Problems", Prentice Hall, 1988.

[39] Sturrock S. S. and Salmon I., "Application of Occam to Biological Sequence Comparisons", in "Occam and the Transputer", Edwards J. ed., IOS Press 1991.

[40] Jones G. and Goldsmith M., "Programming in Occam 2", Prentice Hall, 1988.

[41] INMOS, "The Transputer Applications Notebook, Architecture and Software", 1st Edition, 1989.

[42] Stone H. S., "High-Performance Computer Architectures", Addison-Wesley, 1987.

[43] INMOS, "The Transputer Databook", 1992.

[44] Philips, "Microcontroller Data Book", 1992.

[45] Chen Y., "Stability Analysis of Fuzzy Control - A Lyapunov Approach", Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics, pp. 1027-1031, 1987.

[46] Yamakawa T. ,"Stabilisation of an Inverted Pendulum by a High-Speed Fuzzy Logic Controller Hardware System", In "Fuzzy Sets and Systems", 32, pp. 161-180, North Holland, 1989.

[47] INMOS, "The T9000 Transputer Hardware Reference Manual", 1st edition, 1993.

[48] INMOS, "The T9000 Transputer Product Overview Manual", 1st edition, 1991.

[49] INMOS, "IMSB008 User Guide and Reference Manual", 1988.

[50] Dowsing R. D.,"Introduction to Concurrency Using Occam", p15, 1988.

Appendix A

INTRODUCTION TO THE TRANSPUTER AND OCCAM

A.1 Purpose

The purpose of this appendix is to introduce the Inmos Transputer1 [43], and the Occam 2 programming language. In the first section, the architecture and the performance specifications of the Transputer are presented. The next section explains key aspects of the Occam 2 which is the native language of the INMOS Transputer.

A parallel processing paradigm is central to this study, and this chapter explains why Occam is useful in developing parallel algorithms.

A.2 The Transputer

The T800 Transputer is a 32 bit microcomputer combining features such as 4K bytes of on-chip memory, four serial links, a 32 bit floating point unit, and event handling hardware. The block diagram of the T800 Transputer is shown in figure A.1. The serial communications links on each processor allow for a high degree of inter-connectability, giving the system designer the ability to hardwire a variety of topologies. Figure A.2 shows some of the possible configurations.

Additional architectural flexibility can be achieved by using a 32 way programmable cross-bar switch, that allows Transputers to be connected to one another in a more dynamic structure. This facility is not being used in this research. The links of the Transputer connect directly to the C004 device, and the switch is programmed via it's own serial channel.

One of the development products manufactured by Inmos, is the B008 Motherboard [49], which provides ten positions for Transputer modules (TRAMs). Each TRAM contains a Transputer, local memory, and some interface logic. The B008 is a 3U size card that plugs into one of the slots of an IBM compatible personal computer.

The Transputer links enable processing architectures to be developed that can closely map the data flow requirements of the problem.

[1] In this study, T800 Transputers are used. The T9000 Transputer has a different architecture to the T800, and will not be described herein.

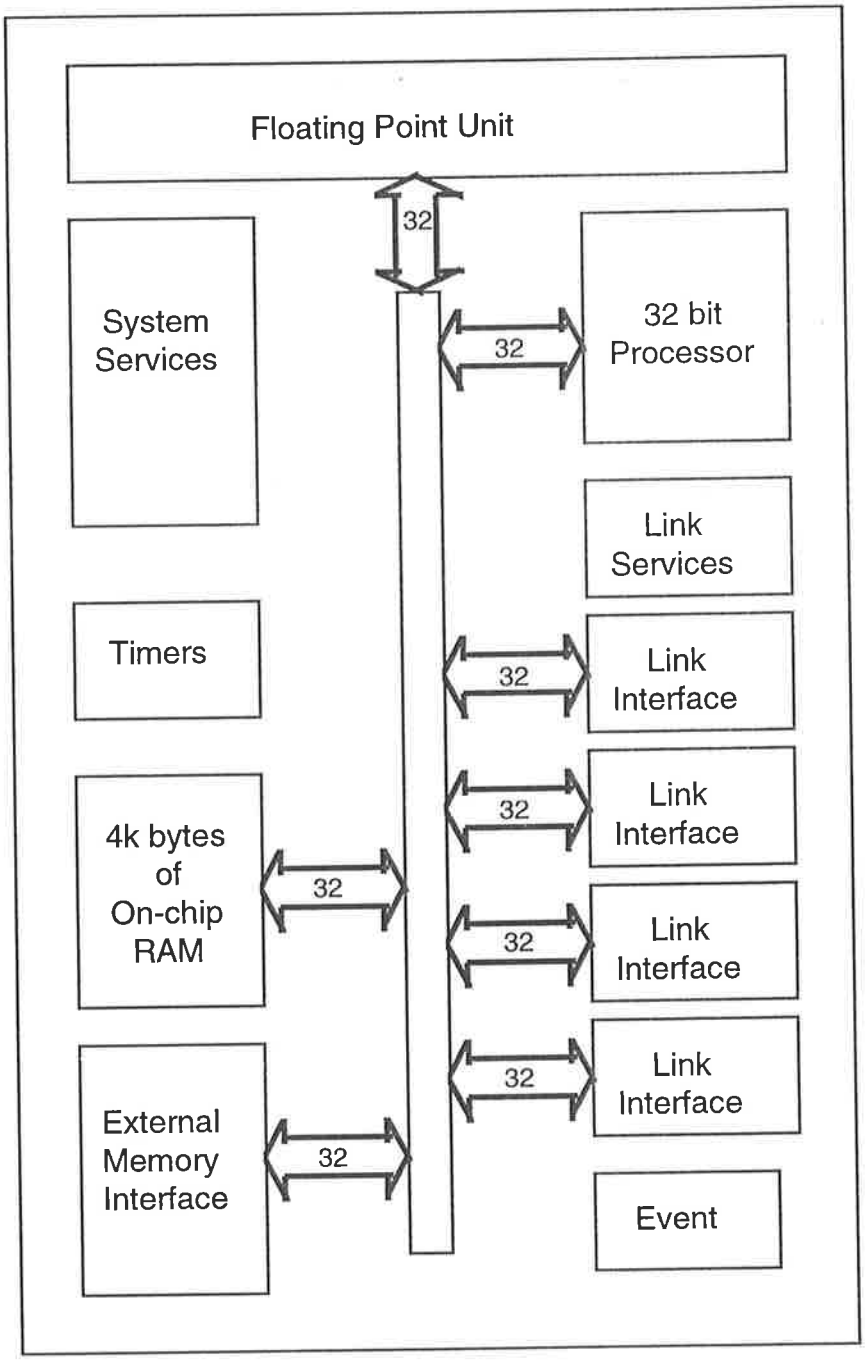


Figure A.1 : Block diagram of the T800 Transputer Architecture

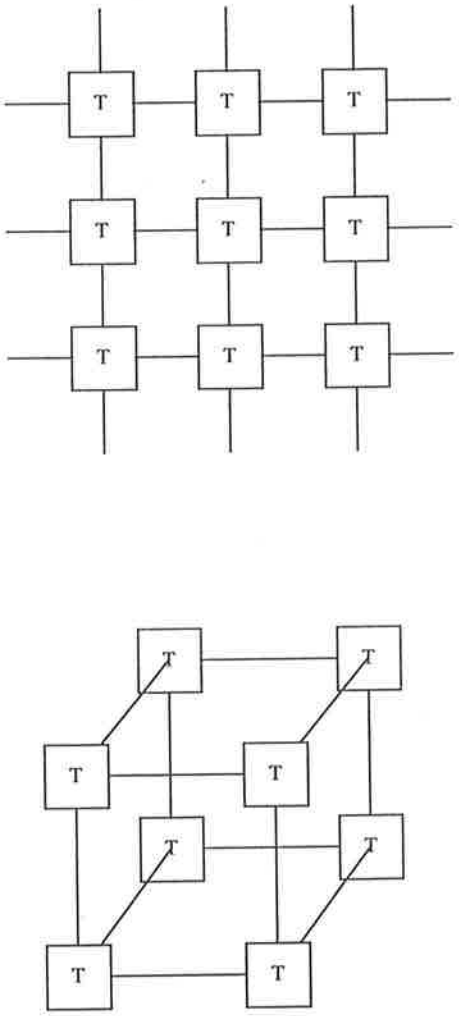


Figure A.2 : The four serial links allow various architectures to be created using the Transputer. This ability to connect Transputers to each other directly[2], without 'glue' logic, makes them particularly useful in building hardware architectures that best suit a particular data processing algorithm .

[2] Within the limitation of the four serial links.

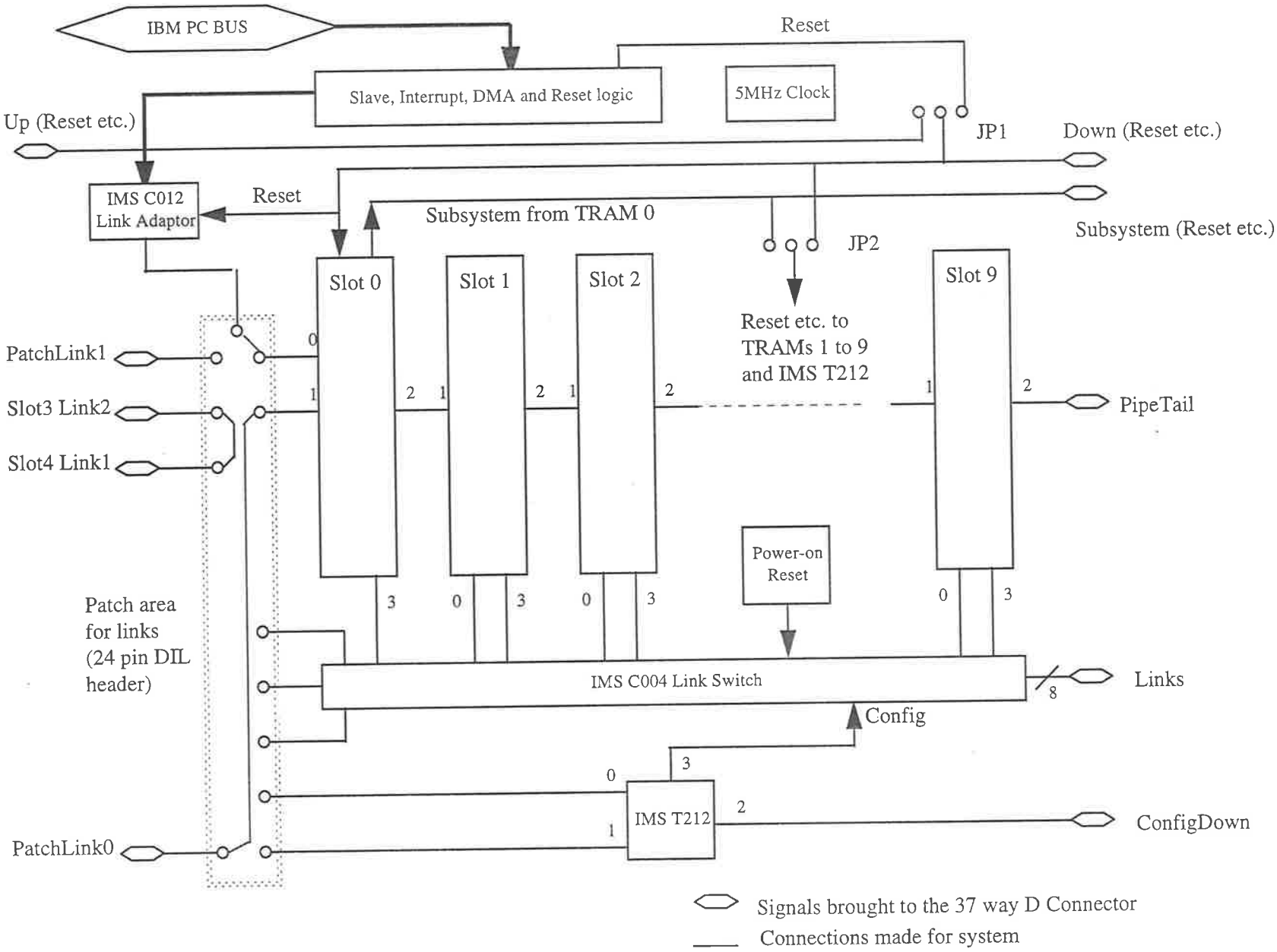


Figure A.2: IMS B008 Motherboard Functional Block Diagram.

A.3 The Occam programming Language

The Occam programming language [35] was developed specifically for the INMOS Transputer. Occam is a concurrent language, as distinct from C, FORTRAN, PASCAL, and other common languages which are designed for Von Neumann processing architectures.

An important point to be made here, is that Occam is a static language [50], in that all of the storage structures must be defined before compiling the code. This fact impacts on this work, as will be pointed out in a later chapter.

In Occam, the fundamental unit is the process. A process is a module which performs some specific task of data transformation (eg. calculate the sum of a series of numbers and output the result). A process P must be able to communicate with it's neighbourhood, and this is done via channels. A process can have any number of logical channels declared, but let us consider for now a simple case of one input channel and one output channel. In our example, P receives the numerical data on it's input channel, performs the sum, and when all input data has been received, outputs the result on it's output channel.

There are some fundamental constructs in Occam, being PAR and SEQ. The SEQ statement means perform the following processes in sequence. The PAR statement means perform the following processes in parallel.

For example:

```
SEQ                    PAR
  process A              process A
  process B              process B
```

Communication between processes is synchronous and is accomplished by channels which must be declared. Communication protocols can be established for each channel. The CHAN statement declares a communication channel for passing data between two processes. For example, CHAN input INT, declares a channel which can pass only integers.

A simple example illustrates how channels interact with processes. Here, an integer is read from the input channel, and is then sent out along the output channel.

```
SEQ
  input.channel.name ? the.number
  output.channel.name ! the.number
```

Particular attention must be payed to the scope of a statement. The SEQ or PAR constructs define the beginning of a sequence of process statements. The scope is defined by using indenting. For example,

```
PAR
 SEQ                              -- Block A
  input.channel ? the.number
  output.channel ! the.number
 SEQ                              -- Block B
  variable := 42
```

This code segment defines two process blocks, A and B, which are performed in parallel. Each block is indented by two spaces from the PAR statement.

With Occam it is possible to define the type associated with a channel. This is called the channel ***protocol***. The protocol statement appears in the channel declaration, and defines the type of data which is permitted to flow in this channel. More than one data type can be associated with a channel, as long as the types are listed in the protocol statement. For example, consider a process with two input channels, channel1 and channel2.

```
CHAN OF INT channel1;
CHAN OF BYTE channel2;
PAR
  INT x,y:
  BYTE z;
  SEQ
    --- do some processing
    channel1 ! x;  -- output the results
    channel2 ! y;
    channel2 ! z;
:
```

This section of code can be simplified, and made more readable, by using the PROTOCOL statement to combine all communications into one channel.

```
PROTOCOL special IS INT; INT; BYTE;
CHAN OF special channel;
PAR
  INT x,y:
  BYTE z;
  SEQ
    --- do some processing
    channel ! x,y, z;  -- output the results
:
```

Another special communication feature of Occam is the ***Tagged Protocol***. This protocol allows a channel to carry any one of a list of data types which have been declared by the Protocol statement. Protocols are used extensively in this study. An example of a protocol statement is the following;

```
PROTOCOL mix
      CASE
        data; INT; INT; BYTE; REAL32
        control; [2]BYTE
        stop.signal
:
```

Channel can be declared to *carry* mixed protocols.

```
CHAN OF mix forward, backward;
[4]CHAN OF distributor;
```

Each message is tagged by a meaningful name. There is no value associated with a tag. Tagged protocols are used in this system as a means of identifying the various messages which flow from one process to another.

Repetition and loops are supported in Occam using the SEQ and WHILE constructs. These have the following form:

```
SEQ i = 0 FOR n
   some.process
```

```
SEQ
   i := 0
   sum := 0
   WHILE i < n
      SEQ
         sum := sum + i
```

The ALT construct provides a method to select an input from a number of inputs. The input that is the first to have data ready, is the one which is read by the process. A multiplexer is an example of the use of the ALT construct.

```
ALT i = (0 FOR number.of.inputs)
  multiplexer.input.channels[i] ? x
  multiplexer.output.channel ! x
```

Shared variables, or global variables, can only be read in Occam. This avoids possible indeterminate results if two or more parallel processes tried to write to the same variable at the same time.

A major advantage of Occam is it's ability to be used to implement algorithms that have been designed using structure software techniques. In particular, processing systems that are specified using data flow diagrams (DFDs) can readily be coded in Occam. The data flow is represented by channels connecting processes

A.4 Summary

This appendix has introduced the Transputer and its programming language, Occam. The architecture of the Transputer provides many features that make it attractive for multi-processor networks, and for implementing parallel algorithms using Occam.

Occam as a programming language does not provide the flexibility of other high level languages such as C, but as it has been designed specifically to run on the Transputer architecture, it is efficient. Occam also provides a means of implementing processing algorithms, which have been specified using data flow techniques, in a straight forward manner. This is an advantage when writing software.

Appendix B

Listing of the Main OCCAM Software Routines for the Inference Engine.

Protocols for Channel Communications

```
--NOTE This list of protocols is divided into DESTINATION headings.
--The protocols are listed according to where the message is going.
PROTOCOL MESSAGE
  CASE
    --*** messages for all modules ***
    t.stop
    t.execute
    t.report.type;              INT
    --*** messages for supervisor ***
    t.system.mode;              INT --open loop or control loop mode
    t.Super.status
    t.supervisor.response;      BYTE
    t.dof.information;          INT::[]BYTE  --length, list of dofs originating  from FIE module.
    t.evaluate.rulebase  --tell the SUPER to evaluate rulebase
    --*** messages for data base  ***
    t.send.crisp.data;          INT   --request crisp data for this output
    t.send.fmf.data                   INT   --request fmf data for this output
    t.DBM.reply
    t.input.name;               INT::[]BYTE
    --t.output.data;            INT::[]BYTE
    t.output.data;              INT::[]INT; INT::[]INT
    t.file.input.vector;        INT::[]INT  --data vector from a file
    t.DBM.ack.stop
    t.get.input.vector              --get input vector from plant
    --t.input.vector;           INT; INT::[]INT --time stamp, size, vector data
    t.input.vector;             INT::[]INT  --size, vector data
    t.request.input.vector;     INT         --send request to read an input
    t.the.inputvalue;           INT::[]INT  --size, Plant input vector
    t.PC.input.vector;          INT::[]INT  --user supplied input vector
    t.send.dof.data;            INT         --rule number
    t.send.rmf.data;            INT         --send rmfs to GUI
    --*** messages for knowledge base ***
    t.KBM.status
    t.knowledgebase.ping;       INT   --can return the number sent by PC
    t.number.of.rules;          INT  --Tell KB how many rules there are.
    t.add.rule;                 INT;INT;INT::[]INT --rule.no, output.id, size, array
    t.delete.rule;              INT  --rule.number
    t.add.member;               INT; INT::[]INT --member no, size, array
    --add a membership function to the membership store
    t.set.rule.weight;          INT; INT  --rule number, weight
    --rule number, followed by percentage, 0 to 100%
    t.rule.on;                  INT     -- rule number, evaluate
    t.rule.off;                 INT     -- rule number, don't evaluate
    t.rule.list;                INT; INT; INT::[]INT   --rule no., output no., size, array
    t.rule.info;                INT::[]BYTE
    t.send.rules.in.rulebase     --a command to tell the KBM to tell the FIE how many rules there are to process.
    t.send.rule;                INT  --send rule list for this rule number
                                --a request to send a rule, received from the inference engine
    t.request.membership.value; INT; INT -- member.number, x.value
                                -- lookup a membership value with this x value
    t.send.mfs.to.FIE           --t.request.membership.functions
    t.number.of.outputs;        INT
    --*** messages for inference engine ***
    t.inference.ping;           INT
    t.FIE.status;               INT
    t.request.status;           INT
    t.inference.methods;        INT; INT; INT; INT      --connective, modifier, fusion, defuz
```

```
  t.forced.rule;            INT; INT::[]INT --rule.number, size, rule.list
  t.rule.data;              INT; INT::[]INT --rule.number, size, rule.list
                            --Data sent from the knowledge base in response to a request
                            --sent by the inference engine.
                            --This data is a single rule from the rule store.
  t.evaluate.rule           --tell the inference engine to evaluate a rule
  t.membership.value;       INT; INT --membership value from the knowledge store
                            --member id, member value
  t.membership.functions;   INT::[]INT -- Array of MFs.
                            -- First 100 values are MF zero
                            -- Second 100 "    "  MF one etc.
  t.reset.plant
  t.shutdown.plant
  t.set.pwm;                BYTE
  t.pulse.width.mod;        BYTE     -- 0 = 100% on, 255 = off
  t.setoutput;              INT; INT    -- output number, value
  t.input.data;             INT::[]INT
  t.request.state
  t.actual.output.id;       INT::[]INT
  t.rules.per.output;       INT::[]INT
  t.rule.output.map;        INT::[]INT
  t.send.inference.methods
:
--These messages are then acted upon by the destination process.

PROTOCOL RESULTS  --Results sent by rule node back to communicator
 CASE
  -- tag followed by information
  t.get.input.data
  t.rmf.data;               INT; INT::[]INT  --rule number, size, array
  t.fmf.data;               INT; INT; INT::[]INT  --output #, crisp value, length, array
  t.dofs.data;              INT::[]INT    --dofs
  t.weighted.dof.data;      INT::[]INT    --dofweighted
  t.dof.data;               INT
  t.error;                  BYTE  -- error tag followed by error type
  t.send.input.data;        INT  -- source
  t.input.data;             INT::[]INT
  t.crisp.data;             INT::[]INT
  t.crisp.value;            INT
  t.output.destinations;    INT::[]INT  --sink ids.
:
```

```
VAL link0out IS 0  :
VAL link1out IS 1  :
VAL link2out IS 2  :
VAL link3out IS 3  :
VAL link0in  IS 4  :
VAL link1in  IS 5  :
VAL link2in  IS 6  :
VAL link3in  IS 7  :
CHAN OF RESULTS  DBM.Super, FIE.DBM, DBM.FIE, Node1.FIE, Node2.FIE:
CHAN OF MESSAGE  Super.KBM, KBM.Super, Super.FIE, FIE.Super, Super.DBM,
          FIE.KBM,   KBM.FIE, FIE.Node1, FIE.Node2:

VAL number.of.sensors IS 5:
VAL PC.module IS  100:
VAL DBM.module IS 101:
VAL KBM.module IS 102:
VAL FIE.module IS 103:
VAL acknowledge.stop IS 200:
VAL DBM.ack.stop IS 205:
VAL default.rule.length IS 80:
VAL default.rmf.size IS 101:
VAL default.fmf.size IS 101:

VAL ID.very IS 0:
VAL ID.slightly IS 1:
CHAN OF MESSAGE from.C012:   -- internal 'logical' channels
CHAN OF ANY to.C012:   -- internal 'logical' channels
CHAN OF BYTE DBM.Plant, Plant.DBM:

#USE userio
#USE snglmath
--NOTE This list of protocols is divided into DESTINATION headings.
--The protocols are listed according to where the message is going.

PROC FuzienProc(CHAN OF MESSAGE from.C012, CHAN OF ANY to.C012,
          CHAN OF BYTE   DBM.Plant, Plant.DBM,
          CHAN OF MESSAGE FIE.Node2,
          CHAN OF RESULTS Node2.FIE)
  VAL link0out IS 0  :
  VAL link1out IS 1  :
  VAL link2out IS 2  :
  VAL link3out IS 3  :
  VAL link0in  IS 4  :
  VAL link1in  IS 5  :
  VAL link2in  IS 6  :
  VAL link3in  IS 7  :
  CHAN OF RESULTS  DBM.Super, FIE.DBM, DBM.FIE, Node1.FIE, Node2.FIE:
  CHAN OF MESSAGE  Super.KBM, KBM.Super,
            Super.FIE, FIE.Super,
            Super.DBM,
            FIE.KBM,   KBM.FIE,
            FIE.Node1, FIE.Node2:

  VAL number.of.sensors IS 5: --ditto
  VAL PC.module IS  100:
  VAL DBM.module IS 101:
  VAL KBM.module IS 102:
  VAL FIE.module IS 103:
  VAL acknowledge.stop IS 200:
  VAL DBM.ack.stop IS 205:
```

```
VAL default.rule.length IS 80:
VAL default.rmf.size IS 101:
VAL default.fmf.size IS 101:

VAL ID.very IS 0:
VAL ID.slightly IS 1:
CHAN OF MESSAGE from.C012:    -- internal 'logical' channels
CHAN OF ANY to.C012:    -- internal 'logical' channels
CHAN OF BYTE DBM.Plant, Plant.DBM:

PLACE from.C012 AT link0in :
PLACE to.C012 AT link0out  :
PLACE DBM.Plant AT link1out:
PLACE Plant.DBM AT link1in:

PLACE Node2.FIE AT link2in:
PLACE FIE.Node2 AT link2out:
PROC delay (VAL INT delay)
  TIMER clock:
  INT current.time:
  SEQ
    clock ? current.time
    clock ? AFTER current.time PLUS delay
:
```

```
PROC Supervisor(CHAN OF MESSAGE from.C012, CHAN OF ANY to.C012,
          CHAN OF MESSAGE Super.DBM, Super.FIE, FIE.Super,
                    Super.KBM, KBM.Super,
          CHAN OF RESULTS DBM.Super )
  VAL INT TIM.P1.7  IS 0:
  VAL INT TIM.P1.6  IS 1:
  VAL INT TIM.P1.5  IS 2:
  VAL INT TIM.PORT3 IS 3:
  VAL INT TIM.PWM   IS 4:
  VAL INT PC.GUI    IS 5:
  VAL INT PC.FILE   IS 6:
  VAL INT size.of.Plant.vector IS 20:
  VAL INT length.of.rule      IS 80:
  VAL INT size.of.pipeline    IS 1:
  VAL INT max.message.length  IS 4000:
  VAL INT data.size           IS 101:
  BOOL running:
  INT    system.mode:
  INT    dof, pointer, op, crisp.index, index1, index2, base.index:
  INT    v,i,j,k, stop.char, report.type, status, ping.number, N:
  INT    connective, modifier, fusion, defuz:
  INT    m.size, member.number, number.of.outputs:
  [101]INT membership.data:
  INT    length, status, name.size, temp1, temp2:
  INT    rule.number, rule.weight, list.length, dumm, output.id:
  [80]INT rule.list:
  [20]BYTE input.name:
  [20]INT output.data:    --holds crisp results of inferencing
  [100][101]INT rmf.store:  -- [rule.number][index]
  [100][101]INT cmf:  -- [rule.number][index]
  [101]INT fmf:
  [20]INT crisp.output:
  [10]INT output.destination:
  BYTE   dummy, pwm.value, temp:
  INT    input.source, output.number, output.value, output.id:
  [20]INT PC.input.data:
  [20]INT file.input.data:
  [20]INT Source.input.vector:  -- vector of 20 input sensor values
  INT      number.of.rules, rule.id:
  [100]INT weighted.dof.array:  -- 0% to 100%
  [100]INT dof.store:
  INT      size, size1, size2, crisp.value:
  [101]INT rmf.from.DBM:
  [100]BYTE dof.information:
  BYTE   input.label:
  [5]BYTE analog.data:
  SEQ
    -- There are 4 bi-directional I/O channels
    -- PC, KnowledgeBase, DataBase, FuzzyEngine
   running := TRUE
   number.of.rules := 0
   list.length := 80
   SEQ i = 0 FOR 20
    Source.input.vector[i] := 0
   SEQ i = 0 FOR 101
    rmf.from.DBM[i] := 0
   SEQ i = 0 FOR 100
    SEQ j = 0 FOR 101
     rmf.store[i][j] := 0
   SEQ i = 0 FOR 100
```

```
    weighted.dof.array[i] := 0
  SEQ i = 0 FOR 101
    fmf[i] := 0
  SEQ i = 0 FOR 20
    crisp.output[i] := 0
WHILE running
  SEQ
    from.C012 ? CASE --get data from end of pipeline
      t.stop
        SEQ
          --Super.Com  ! acknowledge.stop --send acknowledge back to PC
          Super.FIE ! t.stop
          Super.DBM ! t.stop
          Super.KBM ! t.stop
      t.execute
        SKIP
      t.system.mode; system.mode
        Super.KBM ! t.system.mode; system.mode
      t.Super.status
        SKIP
      t.supervisor.response; dummy
        SKIP
      t.number.of.rules; number.of.rules
        Super.FIE ! t.number.of.rules; number.of.rules
      t.evaluate.rulebase  -- START processing
        Super.FIE ! t.evaluate.rulebase
      t.send.dof.data; rule.number  -- send the dofs to the GUI
        SEQ
          Super.DBM ! t.send.dof.data; rule.number
          DBM.Super ? CASE
            --receive dof
            t.dofs.data; size::dof.store
              SKIP
          SEQ i = 0 FOR number.of.rules
            to.C012 ! dof.store[i]
      t.send.rmf.data; rule.number  -- send rmf for this rule
        SEQ
          Super.DBM ! t.send.rmf.data; rule.number
          DBM.Super ? CASE
            t.rmf.data; rule.number; length::rmf.from.DBM
              SKIP
          --NOW SEND CORRESPONDING RMF
          SEQ j = 0 FOR 101
            to.C012 ! rmf.from.DBM[j]
      t.send.fmf.data; output.number
        SEQ
          Super.DBM ! t.send.fmf.data; output.number
          DBM.Super ? CASE
            t.fmf.data; output.number; crisp.value; length::fmf
              SKIP
          --Send FMF to the GUI
          to.C012 ! crisp.value
          SEQ i = 0 FOR 101
            to.C012 ! fmf[i]
      t.send.crisp.data; output.number
        SEQ
          Super.DBM ! t.send.crisp.data; output.number
          DBM.Super ? CASE
            t.crisp.data; size::crisp.output
              SKIP
```

```
    SEQ i = 0 FOR number.of.outputs
      SEQ
        to.C012 ! i
        to.C012 ! crisp.output[i]
  t.input.name; name.size::input.name
    SKIP
  t.file.input.vector; length::file.input.data
    SKIP
  t.setoutput; output.number; output.value
    Super.DBM ! t.setoutput; output.number; output.value
  t.get.input.vector  -- Read input vector independantly
    SEQ
      Super.DBM ! t.get.input.vector          --REQUEST INPUT VECTOR
      DBM.Super ? CASE                        --WAIT HERE FOR REPLY
        t.input.data; length::Source.input.vector
          SKIP
      SEQ i = 0 FOR size.of.Plant.vector    --SEND PLANT VECTOR TO GUI
        to.C012 ! Source.input.vector[i]
  t.PC.input.vector; length::PC.input.data
    Super.DBM ! t.PC.input.vector; length::PC.input.data
  t.KBM.status
    Super.KBM ! t.KBM.status
  t.knowledgebase.ping;  ping.number
    SKIP
  t.inference.methods; connective; modifier; fusion; defuz
    Super.KBM ! t.inference.methods; connective; modifier; fusion; defuz
  t.add.rule; rule.number; output.id; list.length::rule.list
    Super.KBM ! t.add.rule; rule.number; output.id; list.length::rule.list
  t.delete.rule; rule.number
    Super.KBM ! t.delete.rule; rule.number
  t.rule.on; rule.number
    SKIP
  t.rule.off; rule.number
    SKIP
  t.send.mfs.to.FIE
    Super.KBM ! t.send.mfs.to.FIE
  t.number.of.outputs; number.of.outputs
    SEQ
      Super.KBM ! t.number.of.outputs; number.of.outputs
      Super.FIE ! t.number.of.outputs; number.of.outputs
  t.FIE.status; status
    Super.FIE ! t.request.status; status --request FIE status
  t.set.rule.weight; rule.number; rule.weight
    SKIP
  t.add.member; member.number; m.size::membership.data --id=0,1,2
    Super.FIE ! t.add.member; member.number; m.size::membership.data
  t.pulse.width.mod; pwm.value
    Super.DBM ! t.pulse.width.mod; pwm.value --set the pwm
```

```
PROC DataManager(CHAN OF MESSAGE  Super.DBM,
           CHAN OF RESULTS  DBM.Super, FIE.DBM, DBM.FIE,
           CHAN OF BYTE     Plant.DBM, DBM.Plant)
 --These constants identify the source of data vectors that flow
 --into the Data manager.
 --Data may come from a file, the plant, the printer port lpt1,
 --or the serial port com1.
 --Other sources can be added later.
 VAL INT data.size    IS 101:

 VAL INT length.of.dof.store IS 100:
 VAL INT maximum.input.store.depth IS 5:
 VAL INT second       IS 5000:
 VAL INT file.source  IS 0:
 VAL INT plant.source IS 1:
 VAL INT lpt1.source  IS 2:
 VAL INT com1.source  IS 3:
 VAL INT PC.source    IS 4:

 VAL INT TIM.PORT.1.7 IS 0:
 VAL INT TIM.PORT.1.6 IS 1:
 VAL INT TIM.PORT.1.5 IS 2:
 VAL INT TIM.PORT3    IS 3:
 VAL INT TIM.PWM      IS 4:
 VAL INT PC.GUI       IS 5:
 VAL INT PC.FILE      IS 6:
 VAL INT HISTORY      IS 7:
 BOOL  manager.running:
 INT  system.mode:
 TIMER dbm.time:
 INT   i, j, k, dof, number.of.outputs, number.of.rules:
 INT  sink, sink.word, output.number, output.value, vector.length, status:
 --   store input vectors of length 20, from up to 4 sources
 --   eg. file, plant, lpt1, com1
 INT   input.source, data.word:
 [20]  INT PC.input.vector:
 [20]  INT history.buffer: -- just store previous vector
 -- 20 input channels
 [20][maximum.input.store.depth]  INT input.vector.store:
 [20]  INT Source.input.vector:
 [20]  INT Source.history.buffer: -- just store previous vector
 [20]  INT output.data:          -- value
 [10]  INT output.destination:    -- destination
 INT   size, data.size, destination.size, stop.char, store.pointer:
 INT   input.number, input.value, rule.number:
 BYTE  pwm.value:
 BYTE   data.byte:
 [100]INT weighted.dof.array:
 [100]INT dof.store:
 [101]INT rmf:            -- [output number][index]
 [100][101]INT rmf.store:   -- [rule.number][index]
 [101]INT fmf:            -- [output number][index]
 [10][101]INT fmf.store:    -- [output number][index]
 [101]INT transmit.buffer:
 [5]BYTE analog.data:
 [20]INT crisp.data:
 SEQ
   manager.running := TRUE
   number.of.rules := 0
   output.value := 0
```

```
data.byte := 0(BYTE)
SEQ i = 0 FOR 20
  Source.input.vector[i] := 0
-- Initialsie dofs for rules.
SEQ i = 0 FOR 100
  dof.store[i] := 0
SEQ i = 0 FOR 100
  SEQ j = 0 FOR 101
    rmf.store[i][j] := 0
SEQ i = 0 FOR 100
  weighted.dof.array[i] := 0
SEQ i = 0 FOR 20
  output.data[i] := 0
SEQ i = 0 FOR 10
  output.destination[i] := 0
SEQ i = 0 FOR 101
  fmf[i] := 0
SEQ i = 0 FOR 10
  SEQ j = 0 FOR 101
    fmf.store[i][j] := 0
SEQ i = 0 FOR 20
  crisp.data[i] := 0
SEQ i = 0 FOR 101
  transmit.buffer[i] := 0
WHILE manager.running
  PRI ALT  -- monitor the incoming channels
    --The storage structures can be updated by any of
    --the following channels as this is a sequential operation.
    Super.DBM ? CASE
      t.system.mode; system.mode
        SKIP
      t.number.of.outputs; number.of.outputs
        SKIP
      t.get.input.vector
        -- Command '2' is the Read Analog command for the 87C752.
        SEQ
          DBM.Plant ! 2(BYTE)              -- Send analog value
          DBM.Plant ! 0(BYTE)              -- Analog channel number {0,1,2,3,4}
          Plant.DBM ? data.byte
          DBM.Plant ! 2(BYTE)              -- Send analog value
          DBM.Plant ! 0(BYTE)              -- Analog channel number {0,1,2,3,4}
          Plant.DBM ? data.byte
          --                data.byte = [0..255]
          --Read the value, then scale it to [-50..+50]
          Source.input.vector[0]:=(( (INT data.byte) *100)/255)-50
          --Plant.input.vector[0] := 20(BYTE)-- Read from C012 link
          DBM.Plant ! 2(BYTE)        -- Send analog value
          DBM.Plant ! 1(BYTE)        -- Analog channel number {0,1,2,3,4}
          Plant.DBM ? data.byte
          DBM.Plant ! 2(BYTE)        -- Send analog value
          DBM.Plant ! 1(BYTE)        -- Analog channel number {0,1,2,3,4}
          Plant.DBM ? data.byte
          --Read the value, then scale it to [-50..+50]
          Source.input.vector[1]:=(((INT data.byte )*100)/255)-50
          DBM.Plant ! 2(BYTE)        -- Send analog value
          DBM.Plant ! 2(BYTE)        -- Analog channel number {0,1,2,3,4}
          Plant.DBM ? data.byte
          DBM.Plant ! 2(BYTE)        -- Send analog value
          DBM.Plant ! 2(BYTE)        -- Analog channel number {0,1,2,3,4}
          Plant.DBM ? data.byte
```

```
    --Read the value, then scale it to [-50..+50]
    Source.input.vector[2] := (((INT data.byte )*100)/255)-50
    DBM.Plant ! 2(BYTE)        -- Send analog value
    DBM.Plant ! 3(BYTE)        -- Analog channel number {0,1,2,3,4}
    Plant.DBM ? data.byte
    DBM.Plant ! 2(BYTE)        -- Send analog value
    DBM.Plant ! 3(BYTE)        -- Analog channel number {0,1,2,3,4}
    Plant.DBM ? data.byte
    --Read the value, then scale it to [-50..+50]
    Source.input.vector[3] := (( (INT data.byte)*100)/255)-50
    DBM.Plant ! 2(BYTE)        -- Send analog value
    DBM.Plant ! 4(BYTE)        -- Analog channel number {0,1,2,3,4}
    Plant.DBM ? data.byte
    DBM.Plant ! 2(BYTE)        -- Send analog value
    DBM.Plant ! 4(BYTE)        -- Analog channel number {0,1,2,3,4}
    Plant.DBM ? data.byte
    --Read the value, then scale it to [-50..+50]
    Source.input.vector[4] := (((INT data.byte)*100)/255)-50
    DBM.Super ! t.input.data; 20(INT)::Source.input.vector
  t.setoutput; output.number; output.value --come her to test TIM
   --The output.value defines the bit address and it's state.
   SEQ
    IF
     (output.number = TIM.PORT.1.5) AND ( output.value = 0)
      SEQ
       DBM.Plant ! 8(BYTE)    -- Send command type 2^3=8
       DBM.Plant ! 0(BYTE)    -- Send command value
     (output.number = TIM.PORT.1.5) AND ( output.value = 1)
      SEQ
       DBM.Plant ! 8(BYTE)    -- Send command type 2^3=8
       DBM.Plant ! 1(BYTE)    -- Send command value
     (output.number = TIM.PORT.1.6) AND ( output.value = 2)
      SEQ
       DBM.Plant ! 8(BYTE)    -- Send command type 2^3=8
       DBM.Plant ! 2(BYTE)    -- Send command value
     (output.number = TIM.PORT.1.6) AND ( output.value = 3)
      SEQ
       DBM.Plant ! 8(BYTE)    -- Send command type 2^3=8
       DBM.Plant ! 3(BYTE)    -- Send command value
     (output.number = TIM.PORT.1.7) AND ( output.value = 4)
      SEQ
       DBM.Plant ! 8(BYTE)    -- Send command type 2^3=8
       DBM.Plant ! 4(BYTE)    -- Send command value
     (output.number = TIM.PORT.1.7) AND ( output.value = 5)
      SEQ
       DBM.Plant ! 8(BYTE)    -- Send command type 2^3=8
       DBM.Plant ! 5(BYTE)    -- Send command value
     output.number = TIM.PWM
      -- Command '4' is the PWM command for the 87C752.
      SEQ
       DBM.Plant ! 4(BYTE) -- 0 = 100% on, 255 = off
       DBM.Plant ! (BYTE output.value) -- 0 = 100% on, 255 = off
     TRUE
      SKIP
  t.stop
   SKIP
  t.PC.input.vector; vector.length::PC.input.vector
   SEQ
    --Increment store pointer
    store.pointer := store.pointer + 1
```

```
    IF
      store.pointer > maximum.input.store.depth
        store.pointer := 0
      TRUE
        SKIP
    SEQ i =0 FOR vector.length
      input.vector.store[i][store.pointer] := PC.input.vector[i]
t.reset.plant
  SKIP
t.shutdown.plant
  SKIP
t.pulse.width.mod; pwm.value        -- 0 = 100% on, 255 = off
  -- Command '4' is the PWM command for the 87C752.
  SEQ
    DBM.Plant ! 4(BYTE) -- 0 = 100% on, 255 = off
    DBM.Plant ! pwm.value -- 0 = 100% on, 255 = off
t.send.dof.data; rule.number
  SEQ
    DBM.Super ! t.dofs.data; 100(INT)::dof.store
t.send.rmf.data; rule.number
  SEQ
    SEQ i = 0 FOR 101
      transmit.buffer[i] := rmf.store[rule.number][i]
    DBM.Super ! t.rmf.data; rule.number; 101(INT)::transmit.buffer
t.send.fmf.data; output.number
  SEQ
    SEQ i = 0 FOR 101
      transmit.buffer[i] := fmf.store[output.number][i]
    output.value  := output.data[output.number]
    DBM.Super ! t.fmf.data; output.number;output.value;101(INT)::transmit.buffer
t.send.crisp.data; output.number
  DBM.Super ! t.crisp.data; 20(INT)::output.data
FIE.DBM ? CASE
 t.get.input.data
  SEQ
    -- Command '2' is the Read Analog command for the 87C752.
    DBM.Plant ! 2(BYTE)             -- Send analog value
    DBM.Plant ! 0(BYTE)             -- Analog channel number {0,1,2,3,4}
    Plant.DBM ? data.byte
    DBM.Plant ! 2(BYTE)             -- Send analog value
    DBM.Plant ! 0(BYTE)             -- Analog channel number {0,1,2,3,4}
    Plant.DBM ? data.byte           -- data.byte = [0..255]
    --Read the value, then scale it to [-50..+50]
    Source.input.vector[0]:=(( (INT data.byte) *100)/255)-50
    DBM.Plant ! 2(BYTE)       -- Send analog value
    DBM.Plant ! 1(BYTE)       -- Analog channel number {0,1,2,3,4}
    Plant.DBM ? data.byte
    DBM.Plant ! 2(BYTE)       -- Send analog value
    DBM.Plant ! 1(BYTE)       -- Analog channel number {0,1,2,3,4}
    Plant.DBM ? data.byte
    --Read the value, then scale it to [-50..+50]
    Source.input.vector[1]:=(((INT data.byte )*100)/255)-50
    DBM.Plant ! 2(BYTE)       -- Send analog value
    DBM.Plant ! 2(BYTE)       -- Analog channel number {0,1,2,3,4}
    Plant.DBM ? data.byte
    DBM.Plant ! 2(BYTE)       -- Send analog value
    DBM.Plant ! 2(BYTE)       -- Analog channel number {0,1,2,3,4}
    Plant.DBM ? data.byte
    --Read the value, then scale it to [-50..+50]
    Source.input.vector[2] := (((INT data.byte )*100)/255)-50
```

```
    DBM.Plant ! 2(BYTE)        -- Send analog value
    DBM.Plant ! 3(BYTE)        -- Analog channel number {0,1,2,3,4}
    Plant.DBM ? data.byte
    DBM.Plant ! 2(BYTE)        -- Send analog value
    DBM.Plant ! 3(BYTE)        -- Analog channel number {0,1,2,3,4}
    Plant.DBM ? data.byte
    --Read the value, then scale it to [-50..+50]
    Source.input.vector[3] := (( (INT data.byte)*100)/255)-50
    DBM.Plant ! 2(BYTE)        -- Send analog value
    DBM.Plant ! 4(BYTE)        -- Analog channel number {0,1,2,3,4}
    Plant.DBM ? data.byte
    DBM.Plant ! 2(BYTE)        -- Send analog value
    DBM.Plant ! 4(BYTE)        -- Analog channel number {0,1,2,3,4}
    Plant.DBM ? data.byte
    --Read the value, then scale it to [-50..+50]
    Source.input.vector[4] := (((INT data.byte)*100)/255)-50
    DBM.FIE ! t.input.data; 20(INT)::Source.input.vector
t.send.input.data; input.source
  --send value in the data buffer
  IF
    input.source = plant.source
      DBM.FIE ! t.input.data; vector.length::Source.input.vector
    input.source = PC.source
      DBM.FIE ! t.input.data; vector.length::PC.input.vector
    TRUE
      SKIP
  IF
    input.source = history
    --send the data stored in the history buffer
    DBM.FIE ! t.history.vector; vector.length::history.buffer
t.dofs.data; size::dof.store
  SKIP
t.weighted.dof.data; size::weighted.dof.array
  SKIP
t.rmf.data; rule.number; size::rmf
  SEQ i= 0 FOR size
    rmf.store[rule.number][i]:=rmf[i]
t.fmf.data; sink.word; data.word; size::fmf
  SEQ
    output.data[sink.word] := data.word
    SEQ i= 0 FOR size
      fmf.store[sink.word][i]:=fmf[i]
    history.buffer[sink.word]:= output.data[sink.word] --store previous vector
    --The output.value defines the bit address and it's state.
      --  ================================================
      --  |   VALUE   |  PORT 1.7  |  PORT 1.6  |  PORT 1.5 |
      --  ================================================
      --  |     0     |     X      |     X      |     0     |
      --  |-----------|------------|------------|-----------|
      --  |     1     |     X      |     X      |     1     |
      --  |-----------|------------|------------|-----------|
      --  |     2     |     X      |     0      |     X     |
      --  |-----------|------------|------------|-----------|
      --  |     3     |     X      |     1      |     X     |
      --  |-----------|------------|------------|-----------|
      --  |     4     |     0      |     X      |     X     |
      --  |-----------|------------|------------|-----------|
      --  |     5     |     1      |     X      |     X     |
      --  ================================================
    SEQ
      sink := output.destination[sink.word]
      output.value  := data.word
      IF
        (sink = TIM.PORT.1.5) AND ( output.value = 0)
          SEQ
```

```
          DBM.Plant ! 8(BYTE)    -- Send command type 2^3=8
          DBM.Plant ! 0(BYTE)    -- Send command value
          SKIP
        (sink = TIM.PORT.1.5) AND ( output.value = 1)
         SEQ
          DBM.Plant ! 8(BYTE)    -- Send command type 2^3=8
          DBM.Plant ! 1(BYTE)    -- Send command value
          SKIP
        (sink = TIM.PORT.1.6) AND ( output.value = 2)
         SEQ
          DBM.Plant ! 8(BYTE)    -- Send command type 2^3=8
          DBM.Plant ! 2(BYTE)    -- Send command value
        (sink = TIM.PORT.1.6) AND ( output.value = 3)
         SEQ
          DBM.Plant ! 8(BYTE)    -- Send command type 2^3=8
          DBM.Plant ! 3(BYTE)    -- Send command value
        (sink = TIM.PORT.1.7) AND ( output.value = 4)
         SEQ
          DBM.Plant ! 8(BYTE)    -- Send command type 2^3=8
          DBM.Plant ! 4(BYTE)    -- Send command value
        (sink = TIM.PORT.1.7) AND ( output.value = 5)
         SEQ
          DBM.Plant ! 8(BYTE)    -- Send command type 2^3=8
          DBM.Plant ! 5(BYTE)    -- Send command value
        sink = TIM.PWM
         -- Command '4' is the PWM command for the 87C752.
         SEQ
          DBM.Plant ! 4(BYTE) -- 0 = 100% on, 255 = off
          DBM.Plant ! BYTE (output.value) -- 0 = 100% on, 255 = off
        TRUE
         -- Command '4' is the PWM command for the 87C752.
         SEQ
          DBM.Plant ! 4(BYTE) -- 0 = 100% on, 255 = off
          DBM.Plant ! 45(BYTE) -- 0 = 100% on, 255 = off
      SEQ
       DBM.Plant ! 8(BYTE)    -- Send command type 2^3=8
       DBM.Plant ! 1(BYTE)    -- Send command value
      SEQ
       DBM.Plant ! 8(BYTE)    -- Send command type 2^3=8
       DBM.Plant ! 0(BYTE)    -- Send command value
t.output.destinations; size::output.destination
 SKIP
t.crisp.data; size::output.data
 SEQ
  SEQ i = 0 FOR size
   history.buffer[i]:= output.data[i] -- just store previous vector
  --The output.value defines the bit address and it's state.
```

VALUE	PORT 1.7	PORT 1.6	PORT 1.5
0	X	X	0
1	X	X	1
2	X	0	X
3	X	1	X
4	0	X	X
5	1	X	X

```
  SEQ i = 0 FOR number.of.outputs
   SEQ
    output.number := output.destination[i]
```

```
output.value := output.data[output.number]
IF
 (output.number = TIM.PORT.1.5) AND ( output.value = 0)
  SEQ
   DBM.Plant ! 8(BYTE)   -- Send command type 2^3=8
   DBM.Plant ! 0(BYTE)   -- Send command value
 (output.number = TIM.PORT.1.5) AND ( output.value = 1)
  SEQ
   DBM.Plant ! 8(BYTE)   -- Send command type 2^3=8
   DBM.Plant ! 1(BYTE)   -- Send command value
 (output.number = TIM.PORT.1.6) AND ( output.value = 2)
  SEQ
   DBM.Plant ! 8(BYTE)   -- Send command type 2^3=8
   DBM.Plant ! 2(BYTE)   -- Send command value
 (output.number = TIM.PORT.1.6) AND ( output.value = 3)
  SEQ
   DBM.Plant ! 8(BYTE)   -- Send command type 2^3=8
   DBM.Plant ! 3(BYTE)   -- Send command value
 (output.number = TIM.PORT.1.7) AND ( output.value = 4)
  SEQ
   DBM.Plant ! 8(BYTE)   -- Send command type 2^3=8
   DBM.Plant ! 4(BYTE)   -- Send command value
 (output.number = TIM.PORT.1.7) AND ( output.value = 5)
  SEQ
   DBM.Plant ! 8(BYTE)   -- Send command type 2^3=8
   DBM.Plant ! 5(BYTE)   -- Send command value
 output.number = TIM.PWM
  -- Command '4' is the PWM command for the 87C752.
  SEQ
   DBM.Plant ! 4(BYTE) -- 0 = 100% on, 255 = off
   DBM.Plant ! BYTE output.value -- 0 = 100% on, 255 = off
 TRUE
  -- Command '4' is the PWM command for the 87C752.
  SEQ
   DBM.Plant ! 4(BYTE) -- 0 = 100% on, 255 = off
   DBM.Plant ! 45(BYTE) -- 0 = 100% on, 255 = off
```

```
PROC KnowledgeManager(CHAN OF MESSAGE Super.KBM, KBM.Super,
                      FIE.KBM,   KBM.FIE)
  --The knowledge manager stores all the information required to
  --describe the system that is being controlled, or the data that
  --is being processed.

  --The FIE is loaded with a complete set of membership functions
  --during the configuration phase.

  --In this way, the movement of data is kept to a minimum, and
  --the communication protocols are simplified.

  --The FIE is sent one rule list at a time.
  VAL INT idle.state        IS 0:
  VAL INT run.state         IS 1:
  VAL INT ready.state       IS 4:

  VAL INT data.size         IS 101:
  VAL INT default.rule.length IS 80:
  VAL INT TIM.P1.7          IS  0:
  VAL INT TIM.P1.6          IS  1:
  VAL INT TIM.P1.5          IS  2:
  VAL INT TIM.PORT3  IS  3:
  VAL INT TIM.PWM           IS  4:
  VAL INT PC.GUI            IS  5:
  VAL INT PC.FILE           IS  6:
  INT system.mode, member.number, x.value:
  INT connective, modifier, fusion, defuz:
  BOOL running:
  INT i,j, k, temp, r.size, m.size, m.value:
  INT rule.number, number.of.rules, rule.weight:
  INT rule.to.delete:
  INT output.id:
  [80]INT rule.list:
  INT member.id:                    --identifies the function
  [100]INT rule.output.map:         --Associates rule a number to an output number
  [10]INT  rules.per.output:
  [10]INT  output.flag:
  [10]INT  actual.output.id:
  INT number.of.outputs:
  INT rules.in.rulebase:   --The number of rules actually stored in the transputer's memory.
  [4]BYTE inference.method:       --array to store the selected inference methods
  [100][80]INT rulebase.store:    --array to store 100 rules
  [100]INT rule.output.map:       --association between rule and output
  [80]INT transmit.buffer:        --array used for transmitting a rule
  [100][80]INT rulebase.image:    --array to store image of rulebase
  [25][101]INT membership.store: --array to store 25 functions
  [10]INT io.map:                  --array to store io mapping
  BYTE pwm.value:
  SEQ
    running := TRUE
    system.mode:= idle.state
    connective:=0
    modifier:=0
    fusion:=0
    defuz:=0
    --initialisation
    number.of.rules := 0
    number.of.outputs := 0
    SEQ i = 0 FOR 100
```

```
  rule.output.map[i] := 0
SEQ i = 0 FOR 10
  SEQ
    rules.per.output[i] := 0
    output.flag[i] := 0
    actual.output.id[i] := 0
WHILE running
  PRI ALT  -- monitor incoming channels
    Super.KBM ? CASE
      t.system.mode; system.mode
        SKIP
      t.inference.methods; connective; modifier; fusion; defuz
        SKIP   --FIE will interogate KBM to get methods
      t.number.of.rules; number.of.rules
        SKIP
      t.number.of.outputs; number.of.outputs
        SKIP
      t.add.rule; rule.number; temp; r.size::rule.list
        SEQ
          --number.of.rules := number.of.rules + 1
          rule.output.map[rule.number] := rule.list[1] --output id is second item
          SEQ i = 0 FOR r.size
            rulebase.store[rule.number][i] := rule.list[i]
          output.id := temp
          rule.output.map[rule.number] := output.id --output id is second item
          --This code determines what the output sinks are.
          IF
            output.id = TIM.P1.7
              SEQ
                rules.per.output[0] := rules.per.output[0] + 1
                output.flag[0] := 1  --if this output is used, set flag
            output.id = TIM.P1.6
              SEQ
                rules.per.output[1] := rules.per.output[1] + 1
                output.flag[1] := 1  --if this output is used, set flag
            output.id = TIM.P1.5
              SEQ
                rules.per.output[2] := rules.per.output[2] + 1
                output.flag[2] := 1  --if this output is used, set flag
            output.id = TIM.PORT3
              SEQ
                rules.per.output[3] := rules.per.output[3] + 1
                output.flag[3] := 1  --if this output is used, set flag
            output.id = TIM.PWM
              SEQ
                rules.per.output[4] := rules.per.output[4] + 1
                output.flag[4] := 1  --if this output is used, set flag
            output.id = PC.GUI
              SEQ
                rules.per.output[5] := rules.per.output[5] + 1
                output.flag[5] := 1  --if this output is used, set flag
            output.id = PC.FILE
              SEQ
                rules.per.output[6] := rules.per.output[6] + 1
                output.flag[6] := 1  --if this output is used, set flag
            output.id = 7
              SEQ
                rules.per.output[7] := rules.per.output[7] + 1
                output.flag[7] := 1  --if this output is used, set flag
            output.id = 8
```

```
        SEQ
          rules.per.output[8] := rules.per.output[8] + 1
          output.flag[8] := 1  --if this output is used, set flag
        output.id = 9
          SEQ
            rules.per.output[9] := rules.per.output[9] + 1
            output.flag[9] := 1  --if this output is used, set flag
        TRUE
          SKIP
    --NOW CHECK WHICH OUTPUTS ARE USED BY TESTING WHICH FLAGS ARE SET
      j := 0
      SEQ i = 0 FOR 10
        IF
          output.flag[i] = 1
            SEQ
              actual.output.id[j] := i
              j := j + 1   --j will equal the number of outputs
          TRUE
            SKIP
      KBM.FIE ! t.rules.per.output; 10(INT)::rules.per.output
      KBM.FIE ! t.actual.output.id; 10(INT)::actual.output.id
      KBM.FIE ! t.rule.output.map;  100(INT)::rule.output.map
    t.delete.rule; rule.to.delete
      SEQ
      --update number of rules
      number.of.rules := number.of.rules - 1
      INT16 k:
      SEQ i = 0 FOR number.of.rules
        IF
          i <> rule.to.delete
            SEQ k = 0 FOR 50   --length of a rule list
              rulebase.image[i][k] := rulebase.store[i][k]
          TRUE  -- don't copy this rule
            SKIP
      SEQ i = 0 FOR number.of.rules
        SEQ k = 0 FOR 50   --length of a rule list
          rulebase.store[i][k] := rulebase.image[i][k]
    t.rule.on; rule.number
      SKIP  --turn this rule on if it is off (set weight to 100%)
    t.rule.off; rule.number
      SKIP  --turn this rule off if it is on (set weight to 0%)
    t.send.mfs.to.FIE
      SEQ  --send a copy of the 25 membership functions to FIE
        k :=0
        -- Fill read transmit buffer
        SEQ i = 0 FOR 25
          SEQ j = 0 FOR 101
            SEQ
              transmit.buffer[k]:=membership.store[i][j]
              k := k + 1
        --buffer is filled, so now transmit to FIE
        KBM.FIE ! t.membership.functions; k::transmit.buffer
    FIE.KBM ? CASE
      t.send.rules.in.rulebase
        KBM.FIE ! t.number.of.rules; number.of.rules
        --tell FIE how many rules there are
      t.send.inference.methods
        KBM.FIE ! t.inference.methods; connective; modifier; fusion; defuz
      t.send.rule; rule.number
        --forces KBM to send a rule to FIE
```

```
    SEQ
      SEQ i = 0 FOR 80(INT)
        transmit.buffer[i] := rulebase.store[rule.number][i]
      KBM.FIE ! t.rule.data; rule.number; default.rule.length::transmit.buffer
  t.request.membership.value; member.number; x.value
    SEQ
      --lookup the value requested
      m.value := membership.store[member.number][x.value]
      --now send the value
      KBM.FIE ! t.membership.value; member.number; m.value
  t.request.state
    KBM.FIE ! t.system.mode; system.mode
:
```

```
PROC InferenceEngine(CHAN OF MESSAGE Super.FIE, FIE.Super,
                     KBM.FIE, FIE.KBM, FIE.Node1, FIE.Node2,
                     CHAN OF RESULTS Node1.FIE, Node2.FIE,
                     CHAN OF RESULTS FIE.DBM, DBM.FIE)
  VAL Process.Stopped     IS 0:
  VAL Process.Idle        IS 1:
  VAL Process.Running     IS 2:
  VAL size.of.rmf         IS 101:
  VAL AND.logic           IS 0:
  VAL OR.logic            IS 1:
  VAL NOT.logic           IS 2:
  VAL Zadeh.name          IS 0:
  VAL Giles.name          IS 1:
  VAL Weber.name          IS 2:
  VAL Hamacher.name       IS 3:
  VAL Yager.name          IS 4:
  VAL Dubois.name         IS 5:
  VAL INT default.rmf.size        IS 101:
  VAL INT default.fmf.size        IS 101:
  VAL NumberOfTransputers         IS 2:
  VAL INT data.size       IS 101:
  VAL INT idle.state      IS 0:
  VAL INT run.state       IS 1:
  VAL INT ready.state     IS 4:
  VAL INT t.IF            IS 0:
  VAL INT t.INPUT         IS 1:
  VAL INT t.IS            IS 2:
  VAL INT t.MEMBER        IS 3:
  VAL INT t.HEDGE         IS 4:
  VAL INT t.AND           IS 5:
  VAL INT t.OR            IS 6:
  VAL INT t.NOT           IS 7:
  VAL INT t.THEN          IS 8:
  VAL INT t.OUTPUT        IS 9:
  VAL INT t.OUTPUT.MEMBER IS 10:
  VAL INT t.INPUT.MEMBER  IS 11:

  [100]INT dof.store:
  INT    system.mode, i, index, temp2, length, data.value: --Idle, Running, Stop
  INT    connective, modifier, fusion, defuz:
  BOOL   running, DONE.FLAG:
  INT    connective, modifier.type, fusion, defuzzify:
  INT    vector.time.stamp, vector.length:
  [20]INT input.vector.data:
  INT    number.of.rules, rule.id, rule.count, FIE.rule.count, output.id:
  [80]INT Rule.Array:
  [20]BYTE Input.Vector:          -- used by procedure "CalcRuleList"
  [100]INT rule.weights:          -- stores the rule weights set by the GUI
  [100]INT weighted.dofs:         -- stores weighted degrees of fulfilment
  INT    rule.weight:             -- stores weight for this rule
  INT    rule.number, r.size:
  INT    member.value, member.id:
  INT    m.size:                  -- m.size = number.of.MFs x 100
  [2525]INT receive.buffer:       -- accepts membership functions
  [25][101]INT membership.store:  -- local copy of the membership functions
  INT    size, number.of.outputs:
  [101]INT transmit:       --transmit buffer
  [20]INT  crisp.output:
  [101]INT defuz.array:
  [101]INT fuze.array:
```

```
  INT     temp1, index1, index2, pointer, N, base.index, k, j:
  [10]INT  rules.per.output:
  [10]INT  actual.output.id:
  [100]INT rule.output.map:    --Associates rule a number to an output number
  INT     op:
  [100]INT rule.weight.store:  -- array to store rule weights
  [101]INT rmf:                -- storage array for rmf for this process
  [100][101]INT rmf.store:     -- [rule.number][index]
  [100][101]INT cmf:           -- [rule.number][index]
  [10][101]INT  fmf:           -- [output.number][index]
  [101]INT r.member:           -- receive buffer for membership function
                               -- It then gets stored in the membership store array.
  [NumberOfTransputers]INT RuleNodeState:
  INT     count, m, temp:
  INT     xvalue,dof, index, u, output.number:
  INT     operator.type, operator.value, membervalue:
  INT     stack.pointer, u.of.x, u.x.take.1, u.x.take.2:
  INT     x.take.1, x.take.2:
  BOOL    more:
  [20]INT result.stack:  --Place to keep interim results
  INT     output.member.id:
  INT     rule.sent:
  [2]INT RuleNodeState:

SEQ
    running := TRUE
    system.mode := run.state
    number.of.rules := 0
    RuleNodeState[0] := 0   --T800 number 0
    RuleNodeState[1] := 0   --T800 number 1
    SEQ i = 0 FOR 100
      rule.output.map[i] := 0
    SEQ i = 0 FOR 10
      SEQ
        rules.per.output[i] := 0
        actual.output.id[i] := 0
    SEQ i = 0 FOR 100
      rule.weights[i] := 1
    SEQ i = 0 FOR 100
      dof.store[i] := 0
    SEQ i = 0 FOR 100
      weighted.dofs[i] := 1
    SEQ j = 0 FOR 25(INT)
      SEQ i = 0 FOR 101
        membership.store[j][i]:=0  -- local copy of the membership functions
    connective:=0
    modifier.type:=0
    fusion:=0
    defuzzify:=0
    SEQ i = 0 FOR 20
      Input.Vector[i] := 0(BYTE)
    SEQ i = 0 FOR 101
      rmf[i] := 1
    SEQ i = 0 FOR 10
      SEQ j = 0 FOR 101
        fmf[i][j] := 1
    SEQ i = 0 FOR 20
      crisp.output[i] := 0
    SEQ i = 0 FOR 101
      transmit[i] := 0
```

```
SEQ i = 0 FOR 101
 SEQ
  fuze.array[i] := 0
  defuz.array[i] := 0
FIE.KBM ! t.send.inference.methods
KBM.FIE ? CASE
 t.inference.methods; connective; modifier.type; fusion; defuz
  SEQ
   FIE.Node1 ! t.inference.methods; connective; modifier.type; fusion; defuz
   FIE.Node2 ! t.inference.methods; connective; modifier.type; fusion; defuz
WHILE running
 ALT
  Super.FIE ? CASE -- get a message from the SUPERVISOR
   t.stop
    SEQ
     FIE.Node1 ! t.stop
     FIE.Node2 ! t.stop
     SEQ
      running := FALSE
   t.inference.methods; connective; modifier.type; fusion; defuz
    SEQ
     FIE.Node1 ! t.inference.methods; connective; modifier.type; fusion; defuz
     FIE.Node2 ! t.inference.methods; connective; modifier.type; fusion; defuz
   t.set.rule.weight;  rule.number; rule.weight
    rule.weight.store[rule.number] := rule.weight  --store weight
   t.add.member; member.id; m.size::r.member
    SEQ
     FIE.Node1 ! t.add.member; member.id; m.size::r.member
     FIE.Node2 ! t.add.member; member.id; m.size::r.member
     SEQ i = 0 FOR m.size
      membership.store[member.id][i] := r.member[i]
   t.number.of.outputs; number.of.outputs
    SEQ
     FIE.Node1 ! t.number.of.outputs; number.of.outputs
     FIE.Node2 ! t.number.of.outputs; number.of.outputs
   t.rule.on; rule.number --master instruction to FIE
    rule.weight.store[rule.number] := 100 (INT)  --store weight
   t.rule.off; rule.number  --master instruction to FIE
    rule.weight.store[rule.number] := 0 (INT)  --store weight
   t.input.vector; size::input.vector.data      --receive the input vector
    SEQ
     FIE.Node1 ! t.input.vector; size::input.vector.data
     FIE.Node2 ! t.input.vector; size::input.vector.data
   t.number.of.rules; number.of.rules
    SEQ
     FIE.Node1 ! t.number.of.rules; number.of.rules
     FIE.Node2 ! t.number.of.rules; number.of.rules
   t.system.mode; system.mode
    SEQ
     FIE.Node1 ! t.system.mode; system.mode
     FIE.Node2 ! t.system.mode; system.mode
```

```
t.evaluate.rulebase
        SEQ
         system.mode := run.state

         WHILE system.mode = run.state
          SEQ
           --Get latest input vector from the DBM
           FIE.DBM ! t.get.input.data
           DBM.FIE ? CASE
            --the Plant input vector contains data from source
            t.input.data; length::input.vector.data
              SKIP
           FIE.Node1 ! t.input.vector; length::input.vector.data
           FIE.Node2 ! t.input.vector; length::input.vector.data
           rule.count := 0
           ResultsToCome := number.of.rules
           TasksToDo := number.of.rules
           RuleNodeState0 := 0
           RuleNodeState1 := 0
           more.work := TRUE

           WHILE more.work
            SEQ
                IF
                 (RuleNodeState0 = 0) AND (TasksToDo > 0)
                  SEQ
                   --Get a rule from the KBM
                   FIE.KBM ! t.send.rule; rule.count
                   KBM.FIE ? CASE
                    t.rule.data; rule.id; r.size::Rule.Array
                      SKIP
                   FIE.Node1 ! t.rule.data; rule.id; r.size::Rule.Array
                   RuleNodeState0 :=1
                   TasksToDo := TasksToDo - 1
                   rule.count := rule.count + 1
                 (RuleNodeState0 = 0) AND (TasksToDo > 0)
                  SEQ
                   ... Get next rule from KBM
                   FIE.Node1 ! t.rule.data; rule.id; r.size::Rule.Array
                   RuleNodeState0 :=1
                   TasksToDo := TasksToDo - 1
                   rule.count := rule.count + 1
                 TRUE
                  SKIP

              ALT
                Node1.FIE ? CASE
                  SEQ
                      ... Collect Results
                      ... Send results to DBM
                      RuleNodeState0 := 0
                      ResultsToCome := ResultsToCome-1
                      IF
                        ResultsToCome = 0
                          more.work := FALSE
                      TRUE
                        SKIP
               Node2.FIE ? CASE
                  SEQ
                      ... Collect Results
```

```
                ... Send results to DBM
                RuleNodeState1 := 0
                ResultsToCome := ResultsToCome-1
                IF
                  ResultsToCome = 0
                    more.work := FALSE
                TRUE
                  SKIP

--Calculate the Final Membership Function
          --Fuse all rules that belong to the same output variable
          --RMFs are in rmf.store[][] array
          --SEQ i = 0 FOR number.of.outputs
          --Each rule belongs to a particular output variable.
          --We have a rule map array that identifies which output a particular
          --rule belongs to. This is supplied by the pre-processing software.
          --
          --initialise cmfs
          --SELECT RMFs FOR EACH OUTPUT, AND PUT INTO A MATRIX THAT IS THEN
                    FUZED.
          --Should be able to delete this step by using intelligent pointers
          --SORT ACCORDING TO OUTPUT
          index1 := 0
          --MUST COMPARE AGAINST THE ACTUAL OUTPUT SINK, NOT JUST THE
          --LOOP COUNTER !!!!!!
          SEQ i = 0 FOR number.of.outputs
            SEQ
              --NOW GET THE NEXT OUTPUT NUMBER IN THE LIST OF OUTPUTS
              -- EG 0,1,2,4,6  NOTE...NOT CONSECUTIVE ORDER !!!!
              temp2 := actual.output.id[i]
              SEQ j = 0 FOR number.of.rules
                SEQ
                  --GET THE OUTPUT ID. FOR THIS RULE
                  temp1 := rule.output.map[j]
                  IF        -- this rule belongs to this output, then...
                    temp1 = temp2
                      --cmf is an array of rmfs, placed in output order
                      SEQ
                        SEQ k = 0 FOR 101  --PLACE THIS RMF INTO THE ARRAY
                          cmf[index1][k] := rmf.store[j][k]
                        index1 := index1 + 1
                    temp1 <> temp2   -- if it does not belong, then do nothing
                      SKIP
                    TRUE    -- placed here to cover all possibilities
                      SKIP
          op := 0
          index1 := 0
          index2 := 0
          N := 0
          base.index := 0
          k := 0
          pointer := 0
          SEQ i = 0 FOR number.of.outputs  -- FOR EACH OUTPUT, CALCULATE FMF[]
            SEQ
              pointer := actual.output.id[i] --added pointer will equal the output sink number
              --Pointer must be going larger than 9 for this to fail.
              --There can be up to 100 rules for a single output, and there can be
              --upto 10 outputs. Therefore, dimension of rules.per.output is [0..9]
              --N can have a value between 0 and 99.
              N := rules.per.output[pointer] --was i GET NUMBER OF RULES FOR THIS OUTPUT
```

```
            --GET THE DESTINATION FOR THIS OUTPUT
            SEQ j = 0 FOR 101
              SEQ
                --N = number of rules to this output
                SEQ k = 0 FOR N  -- LOAD THE FUZE ARRAY
                  SEQ
                    fuze.array[index1] := cmf[index2][j]
                    index1 := index1 + 1
                    index2 := index2 + 1
                index1 := 0  -- reset fuze array pointer
                index2 := base.index
                fmf[i][j] := Fuze(fuze.array, N, fusion)  --save fmf results
              --LOAD ARRAY WITH THE FINAL MEMBERSHIP FUNCTION FOR THIS OUTPUT
              SEQ n = 0 FOR 101
                defuz.array[n] := fmf[i][n]
              --CALCULATE THE CRISP OUTPUT and STORE in array
              --LOAD CRISP VALUE INTO THE CORRECT POSITION IN THE CRISP ARRAY
              data.value := (Defuzzify (defuz.array, defuz)) - 50
              crisp.output[pointer] := data.value
              base.index := base.index + N  -- calculate the new base index
            FIE.DBM ! t.output.destinations; 10(INT)::actual.output.id
            SEQ i = 0 FOR number.of.outputs
              SEQ
                --Load transmit buffer
                SEQ j = 0 FOR 101
                  transmit[j] := fmf[i][j]
                --Send to DBM
                FIE.DBM ! t.fmf.data; i; data.value; 101(INT)::transmit
            --NOW SEND DATA TO DBM TO UPDATE ACTUAL OUTPUTS
            --The crisp.output array contains the defuzzified data. The order within
            --this array relates to the order of the output listbox in the GUI.
            --Therefore, sink identifiers are not required at this point.
            FIE.KBM ! t.send.inference.methods
            KBM.FIE ? CASE
              t.inference.methods; connective; modifier.type; fusion; defuz
                SKIP
            FIE.KBM ! t.request.state
            KBM.FIE ? CASE
              t.system.mode; system.mode
                SKIP
          KBM.FIE ? CASE
            t.number.of.rules; number.of.rules
              SKIP
            t.actual.output.id; size::actual.output.id
              SKIP
            t.rules.per.output; size::rules.per.output
              SKIP
            t.rule.output.map; size::rule.output.map
              SKIP
:
```

Worker Node (Node 2 has the same code.)

```
PROC Node1(CHAN OF MESSAGE FIE.Node1, CHAN OF RESULTS Node1.FIE)
  VAL Process.Stopped IS 0:
  VAL Process.Idle    IS 1:
  VAL Process.Running IS 2:
  VAL size.of.rmf   IS 101:
  VAL AND.logic     IS 0:
  VAL OR.logic      IS 1:
  VAL NOT.logic     IS 2:
  VAL Zadeh.name    IS 0:
  VAL Giles.name    IS 1:
  VAL Weber.name    IS 2:
  VAL Hamacher.name IS 3:
  VAL Yager.name    IS 4:
  VAL Dubois.name   IS 5:
  VAL INT default.rmf.size IS 101:
  VAL INT default.fmf.size IS 101:
  VAL NumberOfTransputers IS 2:
  VAL INT data.size  IS 101:
  VAL INT idle.state IS 0:
  VAL INT run.state  IS 1:
  VAL INT ready.state  IS 4:
  VAL INT t.IF     IS 0:
  VAL INT t.INPUT  IS 1:
  VAL INT t.IS     IS 2:
  VAL INT t.MEMBER IS 3:
  VAL INT t.INPUT.HEDGE  IS 4:
  VAL INT t.AND    IS 5:
  VAL INT t.OR     IS 6:
  VAL INT t.NOT    IS 7:
  VAL INT t.THEN   IS 8:
  VAL INT t.OUTPUT IS 9:
  VAL INT t.OUTPUT.MEMBER IS 10:
  VAL INT t.INPUT.MEMBER IS 11:
  VAL INT t.OUTPUT.HEDGE IS 12:

  [100]INT dof.store:
  INT     system.mode, i, index, temp2, length, data.value: --Idle, Running, Stop
  INT     connective, modifier, fusion, defuz:
  BOOL    running, DONE.FLAG:
  INT     connective, modifier.type, fusion, defuzzify:
  INT     vector.time.stamp, vector.length:
  [20]INT input.vector.data:
  INT     number.of.rules, rule.id, rule.count, FIE.rule.count, output.id:
  [80]INT Rule.Array:
  [20]BYTE Input.Vector:      -- used by procedure "CalcRuleList"
  [100]INT rule.weights:      -- stores the rule weights set by the GUI
  [100]INT weighted.dofs:     -- stores weighted degrees of fulfilment
  INT     rule.weight:        -- stores weight for this rule
  INT     rule.number, r.size:
  INT     member.value, member.id:
  INT     m.size:             -- m.size = number.of.MFs x 100
  [25][101]INT membership.store:  -- local copy of the membership functions
  INT     size, number.of.outputs:
  INT     temp1, index1, index2, pointer, N, base.index, k, j:
  INT     op:
  [100]INT rule.weight.store:  -- array to store rule weights
  [101]INT rmf:                -- storage array for rmf for this process
```

```
[100][101]INT rmf.store:    -- [rule.number][index]
[101]INT r.member:          -- receive buffer for membership function
                            -- It then gets stored in the membership store array.
[NumberOfTransputers]INT RuleNodeState:
INT    count, m, temp:
INT    xvalue,dof, index, u, output.number:
INT    operator.type, operator.value, membervalue:
INT    stack.pointer, u.of.x, u.x.take.1, u.x.take.2:
INT    x.take.1, x.take.2:
BOOL   more:
[20]INT result.stack:  --Place to keep interim results
INT    output.member.id:
INT FUNCTION modifier (VAL INT a, b, type)
 INT modvalue:
 VALOF
  IF
   type = 0
    modvalue := (a*b)/100 -- could be 100*100/100 !
   type = 1
    SEQ
     IF
      a > b
       modvalue := b  --truncate
      a < b
       modvalue := a  --below cut-off value
      a = b
       modvalue := a
   (type <> 0) OR (type <> 1)
    modvalue := a      -- catch anything else
  RESULT modvalue
:
INT FUNCTION MIN (VAL INT a, b)
 INT min:
 VALOF
  IF
   a < b
    min := a
   b < a
    min := b
   a = b
    min := a
  RESULT min
:
INT FUNCTION MAX (VAL INT a, b)
 INT max:
 VALOF
  IF
   a > b
    max := a
   b > a
    max := b
   a = b
    max := a
  RESULT max
:
```

```
INT FUNCTION Connective (VAL INT x, y, gamma, logic.type, name)
  INT min, max, temp, answer:
  VALOF
    IF
      logic.type = AND.logic
        CASE name
          Zadeh.name
            SEQ
              IF
                x <= y
                  min := x
                y <= x
                  min := y
                TRUE
                  SKIP
              answer := min
          Giles.name
            SEQ
              answer := MAX ((x+y)-100, 0) --x,y could be 0..100
          Weber.name
            answer := (x*y)/100 --Normalise
          Hamacher.name
            answer:=((x+y)-((2-gamma)*(x*y)))/(1-((1-gamma)*(x*y)) )
          Yager.name
            answer := 100-MIN(1, ((1-x) ))
          Dubois.name
            answer := (x*y)/( MAX(x, MAX(y, gamma) ) )
          ELSE
            SEQ
              IF
                x <= y
                  min := x
                y <= x
                  min := y
                TRUE
                  SKIP
              answer := min
      logic.type = OR.logic
        CASE name
          Zadeh.name
            SEQ
              IF
                x >= y
                  max := x
                y >= x
                  max := y
                TRUE
                  SKIP
              answer := max
          Giles.name
            answer := MIN(x + y, 100)
          Weber.name
            answer := (x+y)-((x*y)/100)
          Hamacher.name
            SEQ
              temp := x*y
              answer := temp / ( gamma+((1-gamma)*((x+y)-temp)) )
          Yager.name
            answer := (x+y) - ((x*y)/100)
          Dubois.name
```

```
        SEQ
          temp := 100- (((100-x)*(100-y))/100)
          answer :=temp / ( MAX(100-x, MAX(100-y, gamma) ) )
        ELSE
        SEQ
          IF
            x >= y
              max := x
            y >= x
              max := y
            TRUE
              SKIP
          answer := max
      logic.type = NOT.logic
       CASE name
        Zadeh.name
          answer := (1 - x)
        Giles.name
          answer := (1 - x)
        Weber.name
          answer := (1 - x)/(1+(gamma*x))
        Hamacher.name
          answer := (1 - x)
        Yager.name
          answer := (1 - x)/(1 + (gamma*x) )
        Dubois.name
          answer := 1 - x
        ELSE
          answer := (1 - x)
      TRUE
       SKIP
    RESULT answer
:
SEQ
  -- Initialisation
  running := TRUE
  system.mode := run.state
  number.of.rules := 0
  SEQ i = 0 FOR 100
    rule.weights[i] := 1
  SEQ i = 0 FOR 100
    dof.store[i] := 0
  SEQ i = 0 FOR 100
    weighted.dofs[i] := 1
  SEQ j = 0 FOR 25(INT)
    SEQ i = 0 FOR 101
      membership.store[j][i]:=0  -- local copy of the membership functions
  connective:=0
  modifier.type:=0
  fusion:=0
  defuzzify:=0
  SEQ i = 0 FOR 20
    input.vector.data[i] := 0
  SEQ i = 0 FOR 101
    rmf[i] := 1
  WHILE running
    FIE.Node1 ? CASE
      t.stop
        running := FALSE
      t.inference.methods; connective; modifier.type; fusion; defuz
```

```
    SKIP
  t.set.rule.weight; rule.number; rule.weight
    rule.weight.store[rule.number] := rule.weight  --store weight
  t.add.member; member.id; m.size::r.member
    SEQ i = 0 FOR m.size
      membership.store[member.id][i] := r.member[i]
  t.number.of.outputs; number.of.outputs
    SKIP
  t.rule.on; rule.number
    rule.weight.store[rule.number] := 100 (INT) --store weight
  t.rule.off; rule.number
    rule.weight.store[rule.number] := 0 (INT)   --store weight
  t.input.vector; size::input.vector.data    --receive the input vector
    SKIP
  t.number.of.rules; number.of.rules
    SKIP
  t.system.mode; system.mode
    SKIP
  t.rule.data; rule.id; length::Rule.Array
    SEQ
      --The procedure is passed the rule list that is to be evaluated, together with the input vector data.
      --The procedure calculates the dof, and the weighted dof of the rule, and the rmf of the rule.
      --These results are stored in FREE variables that are declared outside of the PROC, and therefore have
        scope that covers the procedure.
      -- Rule.Array is passed to the knowledge base for storage, and is
      -- then passed to the inference engine during the evaluation phase.
      -- Rule.Array has the following format:

      -- <Rule.Array>     = <rule number><output number><operator list>
      -- <rule number>    = <integer>
      -- <output number>  = <integer>
      -- <operator list>  = <operator type><operator value><operator list>
      -- <operator type>  = <integer>
      -- <operator value> = <integer>

      -- Example:
      --  Rule Text : if v1 is zero and v0 is small then z2 is zero
      --  After parsing this becomes:
      --  # out v1  zero  is  v0  small  is  AND  z2  zero  is  then
      --  [2,2][1,1][4,6][2,99][1,0][4,4][2,99][5,99][9,2][4,6][2,99][8,99]
      -- When the rule evaluator reads the output identifier, it knows to
      -- stop processing the rule and pass the dof to the next phase of processing.
      -- This will multiply the dof by the weight for this rule, then calculate
      -- the resultant membership function for the rule.
      SEQ
        stack.pointer := 0
        more := TRUE
        --same as rule.id
        rule.number   := Rule.Array[0]  --first element of array
        output.number := Rule.Array[1]
        count := 2      --start at the second data pair of array
        WHILE more
          SEQ
            --process the list, return the dof for the antecedent part
            operator.type  := Rule.Array[count]
            operator.value := Rule.Array[count+1]
            CASE operator.type
              t.IF
                SKIP
              t.INPUT
```

```
  SEQ
    xvalue := INT(input.vector.data[operator.value])
 t.MEMBER
  --access the local membership store
  SEQ
    index := xvalue + 50
    u.of.x := membership.store[operator.value][index]
 t.INPUT.HEDGE
  --pop x
  SEQ
    u := (u.of.x * u.of.x)/100    -- VERY ( could be 100x100 )
    u.of.x := u  --re-assign u.of.x ready to be pushed onto the stack
 t.IS
  SEQ
    result.stack[stack.pointer] := u.of.x   -- put value onto stack
    stack.pointer := stack.pointer + 1      -- increment pointer
 t.AND, t.OR   -- t.not
  SEQ
    stack.pointer := stack.pointer - 1
    u.x.take.1 := result.stack[stack.pointer]
    stack.pointer := stack.pointer - 1
    u.x.take.2 := result.stack[stack.pointer]
    temp := Connective (u.x.take.1, u.x.take.2, 1, operator.type, connective)
    result.stack[stack.pointer] := temp
    stack.pointer := stack.pointer + 1
 t.THEN
  --This is the end of the list
  --The rmf array has been calculated.
  more := FALSE
 t.OUTPUT.MEMBER
  SEQ
    output.member.id := operator.value
    -- CALCULATE THE RESULTANT MEMBERSHIP FUNCTION
    SEQ m = 0 FOR 101
      SEQ
        --member[][] is the local store for all membership functions
        membervalue := membership.store[output.member.id][m]
        --The value becomes modified by this function
        rmf[m] := modifier(membervalue, dof,modifier.type)
    more := FALSE
  t.OUTPUT
   --get the dof and apply weighting factor w
   -- The last item pushed onto the stack was the result of the Fuzzy
   -- Logical Operators (AND, OR, ...). This last one will be the
   -- Degree of Fulfilment (dof) for this rule.
   SEQ
     stack.pointer := stack.pointer - 1   -- adjust the stack pointer
     IF
       stack.pointer < 0
         stack.pointer := 0
       stack.pointer >= 0
         dof := result.stack[stack.pointer]   -- get the dof
       TRUE
         SKIP
     --Now calculate the weighted dof for this rule, and store it.
     weighted.dofs[rule.number] := dof * rule.weights[rule.number]
     dof.store[rule.id] := dof
     weighted.dofs[rule.id] := dof * rule.weights[rule.id]
   t.OUTPUT.HEDGE
     --This will follow IMMEDIATELY after rmf[] calculation
```

```
            SEQ
              u := (rmf[m] * rmf[m])/100  --VERY ( could be 100x100 )
              rmf[m] := u  --re-assign
            ELSE
              more := FALSE
          count :=count + 2  -- while loop
        Node1.FIE ! t.rmf.data; rule.id; 101(INT)::rmf
        Node1.FIE ! t.dofs.data; 100(INT)::dof.store
        Node1.FIE ! t.weighted.dof.data; 100(INT)::weighted.dofs
:

--RMF.matric is the column vector from the cmf array
--The column contains data values for contributing rmfs.

INT FUNCTION Fuze(VAL []INT RMF.matrix, VAL INT Num, type.of.fusion)
  INT answer, m:
  REAL32 temporary, temp, Div, sum:
  REAL32 temp1,temp2:
  VAL Arithmetic.Mean.Type IS 0:
  VAL Harmonic.Mean.Type  IS 1:
  VAL Geometric.Mean.Type  IS 2:
  VAL Peak.Follower.Type  IS 3:
  VALOF
   SEQ
    sum := 0.0(REAL32)
    temporary := 0.0(REAL32)
    Div := (REAL32 ROUND Num)
    IF
     Num <= 0
      Div := 1.0(REAL32)
     TRUE
      SKIP
    IF
     type.of.fusion = Arithmetic.Mean.Type
      SEQ
       SEQ m = 0 FOR Num  -- N = number of rules for this output
        sum := sum + (REAL32 ROUND RMF.matrix[m]) --sum items in column
       answer := (INT ROUND (sum / Div)) --divide by number of items to get average
     type.of.fusion = Harmonic.Mean.Type
      SEQ
       SEQ m = 0 FOR Num       -- N = number of rules to this output
        SEQ
         temp := (REAL32 ROUND RMF.matrix[m])  --get the data
         IF
          temp = 0.0(REAL32) --test for divide by zero
           SKIP
          temp >= 1.0(REAL32)
           temporary := temporary + (1.0(REAL32)/temp)
          TRUE
           SKIP
       temporary := temporary / (REAL32 ROUND Num)
       answer := (INT ROUND temporary)
     type.of.fusion = Geometric.Mean.Type
      SEQ
       temporary := (REAL32 ROUND RMF.matrix[0])--initialise
       temp1 := REAL32 ROUND Num
       temp2 := 1.0(REAL32) / temp1
       SEQ m = 1 FOR Num
        temporary := temporary * (REAL32 ROUND RMF.matrix[m])
       temporary := POWER(temporary, temp2)
```

```
        answer := (INT ROUND temporary)  --convert to integer
      type.of.fusion = Peak.Follower.Type
       SEQ
        answer := RMF.matrix[0]  -- initialise
        SEQ m = 1 FOR Num-1  -- N = number of rules for this output
         IF
          RMF.matrix[m] > answer
           answer := RMF.matrix[m]
          TRUE
           SKIP
      TRUE
       SEQ
        SEQ m = 0 FOR Num  -- N = number of rules for this output
         sum := sum + (REAL32 ROUND RMF.matrix[m]) --sum items in column
        answer := (INT ROUND (sum / Div)) --divide by number of items to get average
    RESULT answer
:

INT FUNCTION Defuzzify (VAL []INT final.mf, VAL INT type.of.defuz)
  INT Centre.of.Gravity, numerator, denominator:
  INT Maximum, answer:
  VALOF
   SEQ
    answer := 0
    numerator := 0
    denominator := 0
    Maximum := 0
    IF  --can add other methods here.
     type.of.defuz = 0
      SEQ
       SEQ n = 0 FOR default.fmf.size
        SEQ
         numerator := numerator + (n * final.mf[n])
         denominator := denominator + final.mf[n]
         IF
          denominator = 0    --case where the CMF is zero set
           denominator := 1
          TRUE
           SKIP
       Centre.of.Gravity := numerator / denominator
       answer := Centre.of.Gravity
     type.of.defuz = 1
      SEQ
       SEQ i = 0 FOR default.fmf.size
        IF
         final.mf[i] >= Maximum
          Maximum := i -- value at which function is a maximum
         TRUE
          SKIP
       answer := Maximum
     type.of.defuz = 2
      SEQ
       SEQ i = 0 FOR default.fmf.size
        IF
         final.mf[i] >= Maximum
          Maximum := i
         TRUE
          SKIP
       answer := Maximum
     TRUE
```

```
      SEQ
        SEQ n = 0 FOR default.fmf.size
          SEQ
            numerator := numerator + (n * final.mf[n])
            denominator := denominator + final.mf[n]
            IF
              denominator = 0    --case where the CMF is zero set
                denominator := 1
              TRUE
                SKIP
        Centre.of.Gravity := numerator / denominator
        answer := Centre.of.Gravity
    RESULT answer
:
```

Channel Declarations

```
CHAN OF ANY     to.C012:
CHAN OF RESULTS  DBM.Super, FIE.DBM, DBM.FIE, Node1.FIE,
           Node2.FIE:
CHAN OF MESSAGE  Super.KBM, KBM.Super,
           Super.FIE, FIE.Super,
           Super.DBM, from.C012,
           FIE.KBM,   KBM.FIE,
           FIE.Node1, FIE.Node2:
CHAN OF BYTE     DBM.Plant, Plant.DBM:
```

FuzienProc Process inter-connection

```
PAR -- These processes run concurrently on the B008 board
  Supervisor(from.C012, to.C012, Super.DBM, Super.FIE,
        FIE.Super, Super.KBM, KBM.Super, DBM.Super)
  KnowledgeManager(Super.KBM, KBM.Super, FIE.KBM, KBM.FIE)
  DataManager(Super.DBM, DBM.Super, FIE.DBM, DBM.FIE, Plant.DBM, DBM.Plant)
  InferenceEngine(Super.FIE, FIE.Super, KBM.FIE, FIE.KBM,
           FIE.Node1, FIE.Node2, Node1.FIE, Node2.FIE, FIE.DBM, DBM.FIE)
  Node1(FIE.Node1, Node1.FIE)
:
```

Process Mapping

```
PLACED PAR
 PROCESSOR 0 T8
  PLACE from.C012 AT link0in:
  PLACE to.C012   AT link0out:
  PLACE Plant.DBM AT link1in:
  PLACE DBM.Plant   AT link1out:  --interface to card
  PLACE FIE.Node2 AT link2out:
  PLACE Node2.FIE AT link2in:

  FuzienProc(from.C012, to.C012,
        DBM.Plant, Plant.DBM, FIE.Node2, Node2.FIE)

 PROCESSOR 1 T8
  PLACE FIE.Node2 AT link1in:
  PLACE Node2.FIE AT link1out:

  Node2(FIE.Node2, Node2.FIE)
```

Appendix C

Listing of the Control Software for the Transputer Interface Module Micro-controller.

```
;*************************************************************************
; This program controls the interface between the plant, in this case,
; the inverted pendulum, and the transputer system.
;  This program first reads all five A/D channels and stores the values in
;  on-chip memory.  This process is repeated continuously.
;*************************************************************************
$Title(C012 Interface control)
$MOD752
;*************************************************************************
PWMVal   DATA     13h             ;Holds next value for updating PWM
Flags    DATA     20h
ADFlag   BIT      Flags.1         ;A/D conversion complete flag.
;*************************************************************************
; ANALOG CHANNEL STORAGE
AnCh0   DATA    13h    ;Store for analog channel 0 value
AnCh1   DATA    14h    ;Store for analog channel 1 value
AnCh2   DATA    15h    ;Store for analog channel 2 value
AnCh3   DATA    16h    ;Store for analog channel 3 value
AnCh4   DATA    17h    ;Store for analog channel 4 value
;*************************************************************************
outdata DATA    18h    ;store for output data value
;*************************************************************************
;      PORT ASSIGNMENT
;P0.0   TTL OUT CS/ C012
;P0.1   TTL OUT R/W C012
;P0.2   TTL OUT A1 C012
;P0.3   TTL OUT A2 C012
;P0.4   TTL OUT PWM

;P1.0   A/D Input
;P1.1   A/D Input
;P1.2   A/D Input
;P1.3   A/D Input
;P1.4   A/D Input
;P1.5   TTL OUT
;P1.6   TTL OUT
;P1.7   TTL OUT

;P3.0   TTL IN/OUT/D0
;P3.1   TTL IN/OUT/D1
;P3.2   TTL IN/OUT/D2
;P3.3   TTL IN/OUT/D3
;P3.4   TTL IN/OUT/D4
;P3.5   TTL IN/OUT/D5
;P3.6   TTL IN/OUT/D6
;P3.7   TTL IN/OUT/D7
```

```
;*****************************************************************************
;
; Interrupt Vectors
        ORG     0               ;Reset vector.
        AJMP    Reset
;*****************************************************************************
;
Reset:  MOV     SP,#30h
        MOV     Flags,#0        ;Clear flags.
        MOV     TCON,#00h       ;Set up timer controls.
        MOV     IE,#82h         ;Enable timer 0 interrupt.
        SETB    P0.0            ;notCS = 1
START:
;*****************************************************************************
;       NOW READ C012 PORT (PORT 3) TO GET COMMAND.
;*****************************************************************************
        ACALL   C012Read        ;Command returned in A
        MOV     R1,A            ;Put command into R1 for now
        ACALL   C012Read        ;Get data to go with this command
        MOV     R2,A            ;Put data into R2 for now.
;*****************************************************************************
; NOW INTERPRET THE COMMAND THAT IS IN R1  =  1 OF 8 COMMANDS  =
;*****************************************************************************
        MOV     A,R1            ;Get the command into acc.
        JB      ACC.0,COMD_0    ;If bit 0 set, jump to command 0 :Reset
        JB      ACC.1,COMD_1    ;If bit 1 set, jump to command 1 :read A/D channel #data
        JB      ACC.2,COMD_2    ;If bit 2 set, jump to command 2 :set PWM to #data
        JB      ACC.3,COMD_3    ;If bit 3 set, jump to command 3 :set P.15,P1.6,P1.7 to [0,1]
        JB      ACC.4,COMD_4    ;If bit 4 set, jump to command 4 :read port P1
; IF NO COMMANDS MATCH THEN JUMP BACK TO THE START
        JMP     START
;*****************************************************************************
;
; Command 0 = Reset
COMD_0: JMP     Reset
;*****************************************************************************
;
; COMMAND 1 = SEND ANALOG VALUE TO C012
;       Sample the A/D input identified by data in R2.
;       ADConv returns after completion of the conversion.
;*****************************************************************************
;
COMD_1:
        MOV     A,R2            ;Set A/D channel to #data.
        ACALL   ADConv          ;Start A/D conversion.
        MOV     AnCh0,A         ;Store A/D value
        MOV     A,AnCh0         ;Put A/D channel value into A
        ACALL   C012Write       ;Send data to C012
        JMP     START           ;Begin loop again
;*****************************************************************************
;
```

```
; COMMAND 2 = SET PWM
;*****************************************************************************
COMD_2:
        MOV  PWMP,#80h      ;Set up PWM prescaler.
        MOV  PWCM,R2        ;Set PWM value, data for which is in R2.
        MOV  PWENA,#01h     ;Start PWM.
        JMP  START
;*****************************************************************************
; COMMAND 3 = DIGITAL OUT , set a TTL output P1.5 P1.6 P1.7 to 0 or 1.
;*****************************************************************************
                ; data byte tells which bit and its' state.
                ;     R2  P1.7  P1.6  P1.5
                ;     0   X     X     0
                ;     1   X     X     1
                ;     2   X     0     X
                ;     3   X     1     X
                ;     4   0     X     X
                ;     5   1     X     X
;*****************************************************************************
COMD_3:
        MOV    A,R2
        CJNE   A,#00h,NOT_CP15
        CLR    P1.5
NOT_CP15:
        CJNE   A,#01h,NOT_SP15
        SETB   P1.5
NOT_SP15:
        CJNE   A,#02h,NOT_CP16
        CLR    P1.6
NOT_CP16:
        CJNE   A,#03h,NOT_SP16
        SETB   P1.6
NOT_SP16:
        CJNE   A,#04h,NOT_CP17
        CLR    P1.7
NOT_CP17:
        CJNE   A,#05h,NOT_SP17
        SETB   P1.7
NOT_SP17:
        JMP    START
;*****************************************************************************
; Command 4 = TTL INPUT ; get state of port 1
COMD_4:
        MOV    P1,#0FFh       ;Put port into read mode
        MOV    A,P1           ;Put port 1 data into A
        ACALL  C012Write      ;Send data to C012
```

```
        JMP    START
;*****************************************************************************
; Command 5 =
;*****************************************************************************
;COMD_5:     JMP    START
;*****************************************************************************
; Command 6 =
;*****************************************************************************
;COMD_6:     JMP    START
;*****************************************************************************
; COMMAND 7 = TEST ROUTINE
;*****************************************************************************
;COMD_7:     JMP  START
;*****************************************************************************
; C012 Link Adaptor Initialise Routine
C012Init:
        RET
;*****************************************************************************
; C012 Link Adaptor Read Routine
; This routine reads data from the C012 data port, that is connected
; to port 3 of the 87C752, and places it into A.
;     If data present bit of READ INPUT STATUS REGISTER is set then
;     (bit 0)  [refer figure 5.3 on p453.]
;       data is present, so read
C012Read:
        ;Write #0FFH to port 3 so it can read data
        MOV    P3,#0FFH
        ;First test the input status register
        SETB   P0.0          ;notCS = 1
        NOP
        CLR    P0.3          ;RS0  = 0 - Select input status register
        SETB   P0.2          ;RS1  = 1
        SETB   P0.1          ;RnotW = 1
        NOP                  ;allow for settling time
READ_STATUS1:
        SETB   P0.0          ;notCS = 1
        CLR    P0.0          ;notCS = 0 - Latch address selection
        NOP                  ;allow for settling time

        ;Status register contents appear on data bus, so check status bit.
        ;Now test if the Input Ready Flag (bit 0) is set
        JNB    P3.0,READ_STATUS1 ;Data not present so loop until ready
        ;Ready, put C012 into READ mode
READ_DATA:
        SETB   P0.0          ;notCS = 1
        NOP                  ;allow for settling time
```

```
        CLR     P0.3        ;RS0  = 0 - Select read data
        CLR     P0.2            ;RS1  = 0
        CLR     P0.0            ;notCS = 0 - Latch address selection
        NOP                     ;allow for settling time
        NOP                     ;allow for settling time

        ;Data now appears on data bus
        ;Move data from port 3 into Acc
        MOV     A,P3            ;Put port data into A
        SETB    P0.0            ;notCS = 1
        RET
;*****************************************************************************
; C012 Link Adaptor Write Routine
;       This routine writes data contained in A, to the C012 data port.
C012Write:
        ;Write 0ff to port 3 so it can read data
        MOV     P3,#0FFH
        ;First test the output status register
        SETB    P0.0            ;notCS = 1
        NOP
        SETB    P0.3            ;RS0  = 1 - Select output status register
        SETB    P0.2            ;RS1  = 1
        SETB    P0.1            ;RnotW = 1
        NOP                     ;allow for settling time
READ_STATUS2:
        SETB    P0.0            ;notCS = 1
        CLR     P0.0            ;notCS = 0 - Latch address selection
        NOP                     ;allow for settling time
        ;Status register contents appear on data bus; so check status bit.
        ;Now test if the Output Ready Flag (bit 0) is set
        JNB     P3.0,READ_STATUS2       ;Data not present so loop until ready
        ;READY, put C012 into WRITE mode
WRITE_DATA:
        SETB    P0.0            ;notCS = 1
        NOP                     ;allow for settling time
        SETB    P0.3            ;RS0  = 1 - Select write data
        CLR     P0.2            ;RS1  = 0
        CLR     P0.1            ;RnotW = 0
        NOP                     ;allow for settling time
        CLR     P0.0            ;notCS = 0 - Latch address selection
        NOP                     ;allow for settling time
                                ;so send the data
        MOV     P3,A            ;Put A into port
        NOP                     ;allow for settling time
        SETB    P0.0            ;notCS = 1
        RET
```

```
;*************************************************************************
; Micro Synchronisation Routine
;     This routine synchronises communications between the micro-controller
; and the transputer system. This code looks for a sequence of #FFH, #00H.
; Return when synchronisation is established.
MSynch:
NOT_EQ:
        ACALL  C012Read             ;Data returned in A
        CJNE   A,#0FFh,NOT_EQ       ;Wait for #ffh to appear
        ACALL  C012Read             ;Now chech for #00h
        CJNE   A,#00h,NOT_EQ        ;If no, start again
        RET                         ;Synchronised, so return!
;*************************************************************************
; A/D Conversion Routine.
;  This is an alternative version of the A/D routine which
;  starts the conversion and then waits for it to complete before
;  returning. A/D data is returned in the ACC.
;  ACC contains channel number [0..4]
ADConv:
        ORL    A,#28h               ;Add control bits to channel #.
        MOV    ADCON,A              ;Start conversion.
ADC1:  MOV     A,ADCON
        JNB    ACC.4,ADC1           ;Wait for conversion complete.
        MOV    A,ADAT               ;Read A/D.
        RET
;*************************************************************************
        END
```

Appendix D

ALTERA Design File for the Motor Control EPLD.

```
TITLE "Motor Control Logic";

DESIGN IS Motor
BEGIN
      DEVICE "EPM5032";
END;

SUBDESIGN encode
(
   enable, direction, brake      : INPUT;
   gateA, gateB, gateC, gateD    : OUTPUT;
)


BEGIN

TABLE
      enable,     direction,  brake =>  gateA, gateB, gateC, gateD;

        0               0           0       =>    0      0      0
0;    % stop    %
        0               0           1       =>    0      0      0
0;    % stop    %
        0               1           0       =>    0      0      0
0;    % stop    %
        0               1           1       =>    0      0      0
0;    % stop    %
        1               0           0       =>    1      0      1
0;    % forward %
        1               0           1       =>    0      0      1
1;    % brake   %
        1               1           0       =>    0      1      0
1;    % reverse %
        1               1           1       =>    0      0      1
1;    % brake   %
END TABLE;


END;
```