Neural Network Programming and Portability

Arumugam Siri Bavan

A thesis submitted for the degree of

Doctor of Philosophy

of the

University of London

Department of Computer Science

University College London

August 1991

ProQuest Number: 10609979

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

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

ProQuest 10609979

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

ABSTRACT

Artificial neural networks, inspired by the neural structure of the brain, is a rapidly expanding field of research based on algorithms to solve a wide spectrum of tasks including speech recognition, image processing, planning, optimisation and other pattern processing tasks. Although a growing number of neural models have been developed to support a variety of applications, neural network programming is still mainly done using conventional languages.

This thesis investigates the problems concerned with the programming of neural network models and their portability. The main goal of this thesis is to propose and develop a programming system that can facilitate the implementation of a range of neural network models on a range of hardware. This led to the design and implementation of a programming system called NPS, and a specialised neural network implementation language called NIL. NIL, which forms the neucleus of the programming system NPS, is a low level, machine independent network specification language designed to map a spectrum of neural models onto a range of architectures and thus supporting portability.

The neural network programming system NPS provides the user with a system consisting of:

- *A programming language*, NIL, to specify network models.
- *A utility*, to save partially trained networks for further training.
- *Libraries of functions and algorithms*, to aid the network construction and the execution of standard models.

The neural network programming language NIL consists of two major components:

- *A network implementation sub-language*, which provides mechanisms for specifying the functions of the nodes and the interconnection topology of the network.
- *A manipulation sub-language*, which provides interactive control and modification facilities for use during the training and the recall phase of the network.

These sub-languages together produce a low level, machine independent network specification language that can be used to port neural network models.

Chapter 1 introduces the thesis and the background concepts, namely, neural networks, and programming systems for neural networks. In chapter 2, a survey of neural network programming systems is presented. In chapter 3, the proposed NPS programming system is presented. In chapter 4, a detailed description of the NIL language is presented. In chapter 5, implementation details of the NPS and NIL is presented. In chapter 6, an assessment of NPS and NIL is presented. Finally in chapter 7, conclusions are drawn and future work is discussed.

To My Parents

Acknowledgements

The completion of this work owes much to many people. Firstly, I would like to express my gratitude to my supervisor - Prof. Philip Treleaven - for his guidance, constructive criticism, and help throughout the duration of this research. Secondly, I would like to thank my second supervisor - Dr. M Lee - for his support and advice during this research.

I also acknowledge my colleagues in the Computer Science Department at University College London for their advice and constructive criticism. Especially, I wish to thank Marco Pacheco, Dr. Andrew Eliasz, and Dr. Chang Wang for reading the earlier versions of this thesis and making positive suggestions.

I am very much indebted to Science and Engineering Research Council (SERC) for their financial support during the first two years of this research project. I am also grateful to my research tutors - Charles Easteal and Dr. Russel Winder - and the Department of Computer Science at University College for the support and generosity shown towards me.

Finally, I wish to thank Vijaya and both my sons, Selvan and Luckshman for allowing this thesis to dominate our lives for so long.

CONTENTS

1. Introduction 1
1.1 Neural Computing 3
1.1.1 Artificial Neural Networks 4
1.1.2 Neural Network Models 7
1.2 Programming Systems for Neural Network 9
1.2.1 Portability 11
1.2.2 Programmability 12
1.3 Aims and Background to Research 13
1.3.1 NPS and NIL 14
1.4 Outline of the Thesis 18

2. Neural Network Programming Systems 19
2.1 Classification of Neural Network Programming Systems 20
2.1.1 Educational systems 20
2.1.2 Research systems 22
2.1.3 Commercial systems 31
2.2 Assessment 36

3. The Neural Network Programming System, NPS 39
3.1 The Implemented System 44

4. The Network Implementation Language, NIL 50
4.1 Motivations and Requirements 50
4.2 Keywords 53
4.3 Brackets and Separators 53
4.4 Constant-Valued Tokens 53
4.5 Syntax and Typing 54
4.6 A Network Implementation Language - NIL 54
4.6.1 The Network Specification Sub-Language 55
4.6.1.1 Link Statements 55
4.6.1.2 Definition of functions 59
4.6.1.3 Output From Compilation 64
4.6.2 Manipulation Sub-Language 64
4.6.2.1 Reading Status and Links 65

4.6.2.2 Reading Inputs from Nodes 66
4.6.2.3 Reading Data from a File 67
4.6.2.4 Creation and Deletion of Links 67
4.6.2.5 Loading Initial Values on to the Input Links 68
4.6.2.6 Outputting Results 68
4.6.2.7 Feeding New Information 69
4.6.2.8 Deleting and Creating Nodes 69
4.6.2.9 Saving and Reloading Network 70
4.6.2.10 Executing the Network 70
4.6.2.11 Stopping and Starting 71
4.6.2.12 Loops and Conditions 71
4.7 Semantic Properties 71

5. Implementation of NPS and NIL 72
5.1 Implementation of NIL 73
5.2 The NIL Translator 73
5.2.1 The Virtual (C-Machine) Machine 74
5.2.2 The Implementation of the Manipulation Part 77
5.3 The Prototype Compiler 78
5.3.1 Data Structures for UCL-Neurocomputer 79
5.4 Implementation of NPS 81

6. Assessment of NPS and NIL 83
6.1 Assessment of NIL 83
6.1.1 Programmability 84
6.1.1.1 Specification of Models. 84
6.1.1.2 Dynamic Properties of NIL 94
6.1.1.3 NIL as an Intermediate Language and its Reusability 94
6.1.1.4 Comparison with another Language in its Class 96
6.1.2 Portability 100
6.1.2.1 Target machine independence 100
6.1.2.2 Occam and NIL 100
6.1.2.3 Translating high level languages into NIL 102
6.2 Assessment of NPS 114
6.2.1 Programmability 114

6.2.1.1 Usability 114
6.2.1.2 Facilities 115
6.2.1.3 Simplicity 116
6.2.2 Portability 116

7. Summary 117
7.1 Contributions 118
7.1.1 Portability 118
7.1.2 Programmability 118
7.2 Future Work 119

REFERENCES 122

APPENDIX A - Syntax Definition of NIL 127

APPENDIX B - Sample NIL Programs 130

APPENDIX C - Comparison of NIL with BIF 147

APPENDIX D - Sample Output from the Compiler 163

APPENDIX E - C Representation of the Virtual Machine 169

APPENDIX F - Published Works 172

LIST OF FIGURES

Figure 1. Idealized View of a Biological Neuron 3

Figure 2. Typical Artificial Neuron 5

Figure 3. Artificial Neural Network 6

Figure 4. A Simplified View of Network Programming 9

Figure 5. A Generalised Neural Network Programming System 10

Figure 6. Neural Network Programming System, NPS 15

Figure 7. A Typical Development Process in ANNE 26

Figure 8. SFINX Environment 28

Figure 9. CONE System 29

Figure 10. Neural Network Programming System, NPS 40

Figure 11. The Implemented System 45

Figure 12. UCL Neurocomputer 47

Figure 13. UCL Neuro-Chip 48

Figure 14. A Simple Network 56

Figure 15. Sequential Replication 56

Figure 16. Parallel Replication 57

Figure 17. A Regular Network 58

Figure 18. The Computational Model 60

Figure 19. NIL Translator 75

Figure 20. The C-Machine 76

Figure 21. A Node 76

Figure 22. An I/O Vector 76

Figure 23. Elements of an I/O Vector 77

Figure 24. The Prototype Compiler 78

Figure 25. Structure of a Node Program 79

Figure 26. Organisation of the Input Data 80

Figure 27. Organisation of the Output Data 81

Figure 28. Hebb/Hopfield Network 85

Figure 29. Exclusive OR Network 90

Figure 30. Kohonen's Feature Map 92

Figure 31. An Object-Oriented Model for a Node 120

LIST OF TABLES

TABLE 1. A Classification of Neural Network Models 7

TABLE 2. Educational NNPSs 21

TABLE 3. Research NNPSs 23

TABLE 4. Commercial NNPSs 32

TABLE 5. Keywords 53

TABLE 6. Initial Weights 88

TABLE 7. Weights for Nodes in Hidden and Output Layers 91

TABLE 8. Model_Names Table 108

TABLE 9. Network Layers for Model1 108

TABLE 10. Node Interface for Type Input 108

TABLE 11. Node Interface for Type Hidden 109

TABLE 12. Node Interface for Type Output 109

TABLE 13. Node_1[1] - Node 1 of Layer 1 109

TABLE 14. Node_1[2] - Node 2 of Layer 1 110

TABLE 15. Node_1[3] - Node 3 of Layer 1 110

TABLE 16. Node_2[1] - Node 1 of Layer 2 110

TABLE 17. Node_2[2] - Node 2 of Layer 2 111

TABLE 18. Node_2[3] - Node 3 of Layer 2 111

TABLE 19. Node_3[1] - Node 1 of Layer 3 112

TABLE 20. Node_3[2] - Node 2 of Layer 3 112

TABLE 21. Node_3[3] - Node 3 of Layer 3 112

TABLE 22. Node_2[1] - Node 1 of Layer 2 113

TABLE 23. Node_2[2] - Node 2 of Layer 2 113

TABLE 24. Node_2[3] - Node 3 of Layer 2 114

TABLE 25. Weights for Nodes in Hidden Layer 160

TABLE 26. Weights for Nodes in Output Layer 161

Chapter 1

This chapter introduces the research work conducted and the important background concepts. These include an introduction to neural computing, and an investigation of programming and portability of neural network models and applications.

1. Introduction

The work described in this thesis represents a precursor of the PYGMALION project [Ange89] funded by ESPRIT II (project 2059) to build a general purpose neurocomputing platform (both programming environment and hardware). PYGMALION is a collaboration of major industrial and academic partners in Europe to produce a number of applications, a programming environment and hardware to promote the exploitation of neural network technology.

Typically a neural network programming system consists of an integrated set of software and hardware tools for specifying and executing neural network models and applications. The major components of a typical system are:

- an algorithms library of common neural network models;
- a special purpose high level language for programming network algorithms and applications;
- a graphic monitor for interactively building and controlling a network.

For programming neural networks, many current neural network programming systems provide a specialised language. These languages are usually very high level, often object oriented and with a C or Pascal syntax [Anza87, Guts88, Ange88] together with data types and functions specifically for neural networks. These high level languages are good for programming neural network applications, but frequently their portability is limited to a few machines due to the complexity of writing translators. In addition, these languages often do not have the explicit constructs or the structural features (such as explicit connectivity, parallel control, and specialised data structures) which are necessary for the efficient mapping of algorithms on parallel hardware.

For portability, certain systems like Neuralworks Professional II [Kolo88] tackles the problem of portability by translating the neural networks specified in high level form into C code. This is a pragmatic approach and is useful for conventional machines as most machines support a C compiler. However, for parallel machines this C code is not

particularly useful since C is a sequential language and hence will require excessive overheads to isolate codes that can be executed in parallel. This is due to the lack of explicit parallel control constructs or explicit parallel structure defining connectivity which are essential for isolating components that can be distributed and executed in parallel.

What we believe is required is a language with the following features:

1. *Programmability*: It must be general enough to be able to capture the features of a range of neural (and semantic) network models. That is, it must be *model independent*. It must be a readable language so that the specification and testing of these models is easy. It must be based on a simple and general syntax and semantics so that complex models can be easily supported.

2. *Portability*: It must be based on simple syntax and semantics so that it can be used as a low level target language for higher level neural programming languages. By this, we mean that it should be almost at the same level as OCCAM to capture the general features of sequential and parallel machines and hence support portability. This also implies that it must be simple to translate into assemblers of a range of machines. This language should either have explicit parallel constructs or features such as explicit connectivity and structure to support parallelism.

To put it more broadly, what we need is a language which is:

1. readable and expressive enough to be a programming language for implementing neural network models and applications;

2. simple and low level enough to be easily translated for a range of hardware;

3. simple and general enough to be a target language for higher level languages.

This thesis investigates the **programmability** and **portability** of neural network models and their applications. Our main goal is to design and implement a neural network implementation language that can be both portable and programmable. In order to demonstrate the feasibility of our solution, that is, to demonstrate the capabilities of the language we use a simplified version of a programming system called NPS. The main purpose in designing and implementing NPS is to use it as a test-bed for assessing the *network implementation language* NIL. Although, NPS, contributes to programmability

by providing a number of support tools, it is NIL which tries to facilitate the programming of a range of neural network algorithms and mapping of these algorithms on a range of hardware.

Having briefly stated the motivations and the goals of this thesis, the next section introduces the reader to neural computing as a prelude to subsequent sections which deal with programming systems for neural networks, and the aims and background of this research.

1.1 Neural Computing

Since the motivation for neural computing is biological neural networks, it is useful to review the properties of biological neural networks before introducing artificial neural networks. The brain deals with pattern matching and pattern manipulation with ease and efficiency that no electronic computer of the present generation can match. The brain performs these tasks not through a more powerful computational device in a sequential fashion but in a parallel fashion using a large network of much slower primitive devices called neurons (or nerve cells). The brain is believed to be organised in a hierarchy with successively higher layers performing more complex and abstract operations on the input data. Each layer performs its processing of the input data before passing the result up to the next layer.

The *neurons* are the "processing elements" of the brain. The human neocortex contains over 10 billion neurons with each neuron connected to thousands of other neurons. The brain contains a vast number of different types of neurons each with slightly differing structure and properties [Shep79].

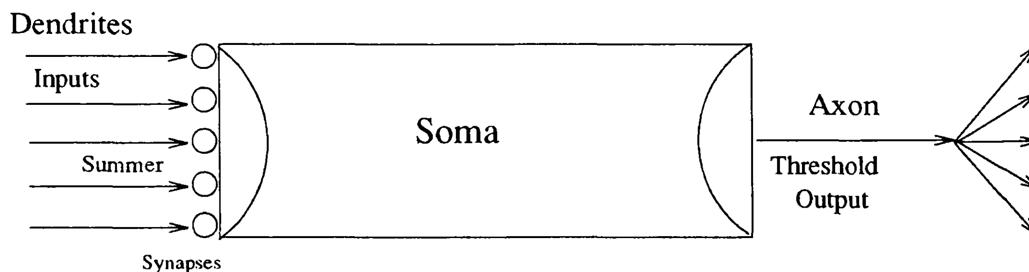

Figure 1. Idealized View of a Biological Neuron

Figure 1 shows a "biological" neuron which has:

- several *dendrites*, which receive input from other neurons;

- a cell body, called the *soma*, which performs some processing (typically a threshold summation) on the information collected by the dendrites;
- a single *axon* which outputs the processed information, usually by the propagation of a "spike" or action potential. The axon splits into various branches that make synapses onto the dendrites and cell bodies of other neurons.

Artificial neural network modelling started in the early 1940s by Professor Warren McCulloch and Dr Walter Pitts [Rum86b]. This initial work on neural network modelling was given a significant encouragement in 1949 when Donald Hebb [Hebb49], published a paper postulating that learning in the brain may be achieved through the changing of the strength of these synaptic junctions. Specifically the more a particular synapse is used or activated, the stronger that connection becomes. This is termed *synaptic facilitation*. As a result, a particular pattern, once established and learnt, can easily be refreshed in the future. It is this general principle that is the basis of artificial neural networks.

In 1957 Rosenblatt created a neural model called the *perceptron* [Rum86b] which showed remarkable promise as a computing device. The perceptron brought many people into the field of neural networks and generated great enthusiasm, until in 1969 Minsky and Papert published a book [Mins69] that showed the inadequacies of the perceptron. They were so convincing that the research into neural computing slumped markedly, until in 1980 Hopfield produced a paper [Hopf82] that showed the potentials of the collective computational abilities of neural networks. Since then research into neural networks has expanded rapidly. During the past few years, there has been a great deal of research into computational models which are inspired by the brain to deal with pattern recognition problems. These models are simply known as neural models and have been used in a number of applications with reasonable degree of success. However, there is a great deal to be learnt about the real neural model in order to be able to produce a more efficient and meaningful model that can be nearly as good as the original one.

In the next section a brief introduction to artificial neural networks is presented which is followed by an overview of the common models and their properties. This is done so in order to give a clear view of the type of processing that is required by a neural network programming system.

1.1.1 Artificial Neural Networks

An artificial neural network is an attempt to solve pattern and image recognition problems by using approaches analogous to the methods adopted by the brain. There are

two reasons why one would want to do this:

1. by using techniques inspired by the brain one hopes to solve pattern and image processing problems much more efficiently than by using the current methods.

2. it is also hoped that these neural models will lead to more powerful and fault tolerant hardware.

Current neural models are still extremely simple when compared with the brain, and use a simple summation and threshold device as the basic processing element in a layered network. A typical artificial neuron has a single output and several inputs, usually one from each neuron in the preceding layer.

Artificial neurons are the fundamental building blocks of neural networks. As shown in Figure 2, a neuron takes a set of inputs X_i which are the equivalent of the excitation or the inhibition signal levels of a neuron.

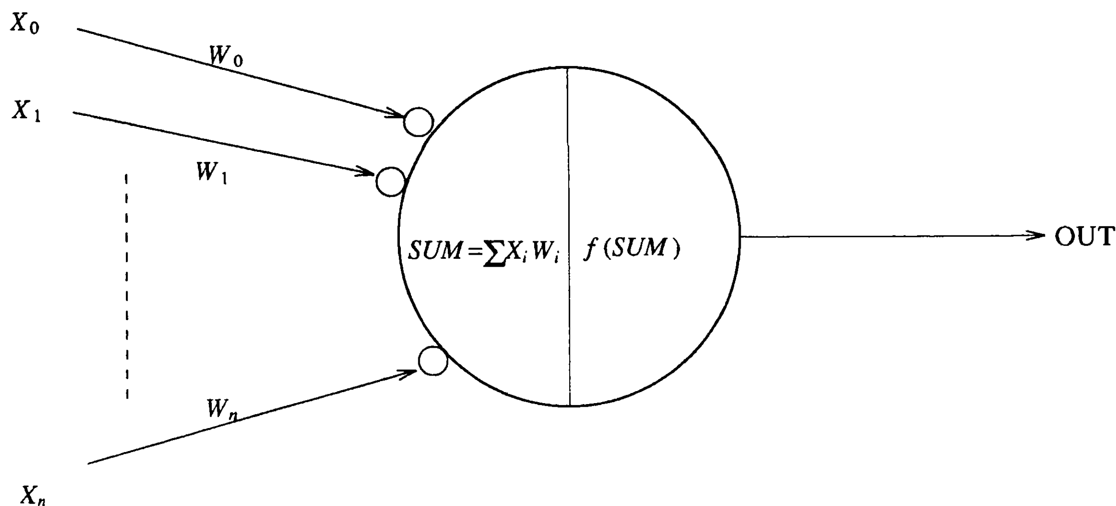

Figure 2. Typical Artificial Neuron

These are then acted upon by a set of associated weights W_i which correspond to the synaptic strengths of a neuron. The weighted sum of these inputs is then compared with a threshold value and an output is delivered depending on the result of thresholding. The weighting factors are analogous to the synaptic strengths.

The artificial neurons are usually connected in a simple layered structure. Most models consist of either two or three layers of neurons since it has been shown that any continuous mappings can be achieved by a three layered system [Hect87]. The network shown in Figure 3 is a multi-layer network where each input X_i to a neuron N_j, in a layer has an associated weight W_{ij}. The outputs from each layer are propagated as inputs to the

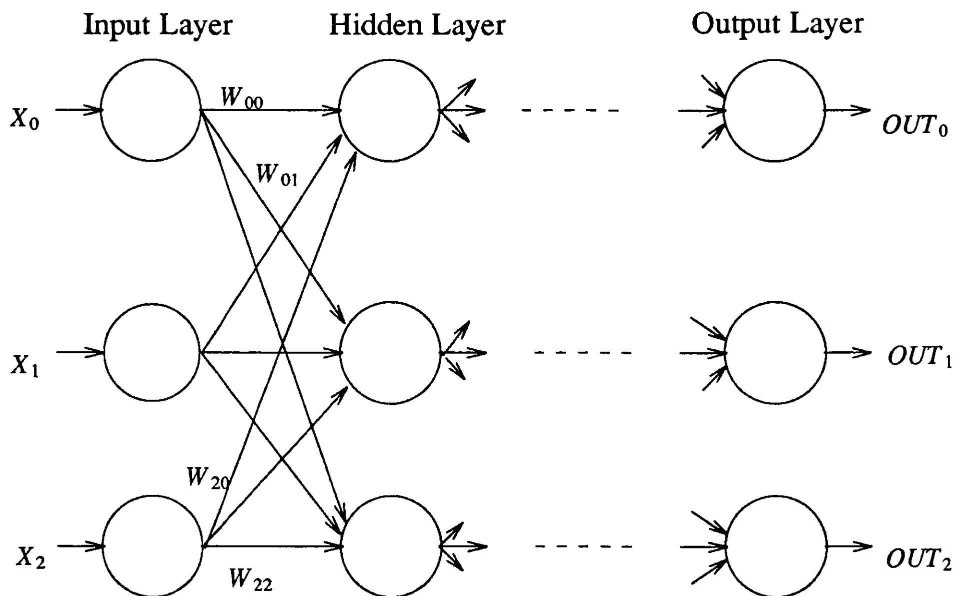

Figure 3. Artificial Neural Network

next layer until a final set of outputs is generated. Multi-layer networks have proved to be more broadly applicable and general than single-layer networks [Wass88] because of their abilities to map any input to output patterns and capture the underlying features of the data space.

These networks can learn to map a specified input pattern to a specified output pattern. Once a mapping has been learned the network will provide the required output pattern when supplied with the appropriate or part of the appropriate input pattern.

The system learns by adjusting the connection strengths between successive layers of neurons [Rum86a]. Different models use different algorithms to determine this adjustment. These models form an N-dimensional energy terrain, where N is the number of connections in the most interconnected neuron [Hopf82]. Given a set of input and output pattern, these models adjust their connection weights in such a way that a local or global energy minima is found by using the input pattern as the entry point to it (ie- the possible output patterns generatable from the current connection weight matrix form local energy minima on the energy surface). When a model seeks a global minima to map its input patterns to output patterns, it adds some noise to the energy space to pull itself out of the local energy minimas in order to avoid being trapped in it without finding its global minima. Once the energy terrain has been learned, an input pattern can be supplied to test the model. This input pattern specifies where on the terrain the search for an output pattern will start. The model then "relaxes" into the nearest minima. No matter what input pattern is supplied the model will always settle on an output pattern, but of course this may not be the desired output pattern. The closer the test input pattern

is to the originally learned input pattern the better the chance that the correct output pattern will be selected.

1.1.2 Neural Network Models

Over the past few years the research efforts in neural networks have produced a number of neural network models. Some of these models namely, Hebb/Hopfield [Hopf82], Boltzmann [Hint85, Hint86], Kohonen's self-organising feature map [Koho84], Adaptive Resonance (ART models) [Gros88] and Back-Propagation [Rum86c] have become popular because of their applicability to practical problems. These neural network models in general can be classified into three major categories. They are *associative memories*, *weak-constraint optimizers*, and *learning systems* as shown in Table 1.

Neural Network Models			
Associative Memories	*Weak-Constraint Optimizers*	*Learning Systems*	
		Supervised	*Unsupervised*
examples: Hopfield Models Kosko's BAM	examples: Boltzmann Machine Hopfield/Tank Model	examples: Back Propagation Boltzmann Machine	examples ART Models Self-organising Map

TABLE 1. A Classification of Neural Network Models

In **associative memories**, the memory is organized as a storage of pattern vectors. Presentation of a part of one of these stored vectors will enable the system to recall that whole vector. These Vector patterns are distributed and stored in the connections of the network. The values stored in these connections are simply referred to as connection weights and are calculated using some fairly simple non-iterative algorithms. Adaptive learning is usually not possible in such a system because all the pattern vectors must be stored at the same time. Later storage of an additional vector will necessitate the process of storing all the vectors again. In this respect, it is rather like a conventional programmable-read-only-memory (PROM). However, the aim here is to build fast-access fault and noise tolerant associative memories using networks of simple processing elements similar to neural networks. The binary Hopfield [Hopf82] model is a recent example of such models.

Optimization models offer good, though approximate, solutions to combinatorial optimisation problems. As no learning processes are involved, the weights are fixed and calculated *a priori*. Such models correspond to weak constraint satisfaction problems. Constraints are embedded in the weights of the connections. The network evolves in such a way as to observe these constraints whilst optimizing some general cost function.

Typical neural network optimisation models include the Boltzmann Machine [Aart86] and the analog Hopfield model [Hopf85].

Learning systems are probably the most commonly used systems in neural networks. There are two major types of learning systems, namely, *supervised* and *unsupervised* learning systems.

In **supervised learning models** the learning process is governed by a set of training pairs. A training pair is formed by an input vector and a desired output vector. The difference between the target output and the output the neural network produced constitutes an error. The learning process seeks to reduce this error by modifying the connection weights in a similar fashion as in a typical relaxation scheme. The multi-layer perceptron with Back error Propagation and the Boltzmann Machine are well known examples of this kind. Their learning rules have their origin in the Hebb's rule of learning (or some variations of it) which states the connection between two nodes that are highly activated at the same time should be strengthened. Basically, the change in the weight of a connection is proportional (with a proportionality constant k -- the so called learning rate) to the product of source and destination neuron activation states:

i.e $W_{ij}(t+1) = W_{ij}(t) + k * A_i * A_j$

In **unsupervised learning models** there is no "specified" target output; the neural network organises itself by applying some rules. This means that the network is only given input data and is expected to organise itself into some useful configuration in response to it. Such models grew out of an analysis of a simpler type of adaptive pattern recognition network, often called *Competitive Learning* [Rume86]. Its development has lead to different models. One of these as defined by Grossberg in [Gros88] corresponds to neural networks that self-organises stable recognition codes in real time in response to arbitrary sequences of input patterns. Rumelhart and Zipser have developed another model in [Rume86b] which is a regularity detection model. Their basic idea is that the set of units is divided into a number of disjoint clusters in which the units compete with one another in order to become active, i.e. to win the competition.

It should be noted that any supervised learning model (e.g. the Boltzmann Machine) can be converted into an unsupervised learning scheme by using the input itself to do the supervision. This kind of models have been used for image processing and speech recognition tasks [Hint84].

In the next section, the general structure of a neural network programming system and their contribution to programmability and portability is discussed.

1.2 Programming Systems for Neural Network

Generally speaking, programming of a neural network can be viewed as a two level process. They are:

- specification of the network topology and the functions of the neuron.
- execution of the network.

The role of the neural network programming system is to provide the user with facilities to accomplish these two tasks. Typically, a neural network programming system is an integrated suite of software tools, and possibly associated hardware, which enable the user to specify the neural network and map it on to the hardware for execution. A simplified view of programming a neural network can be described by the diagram shown in Figure 4.

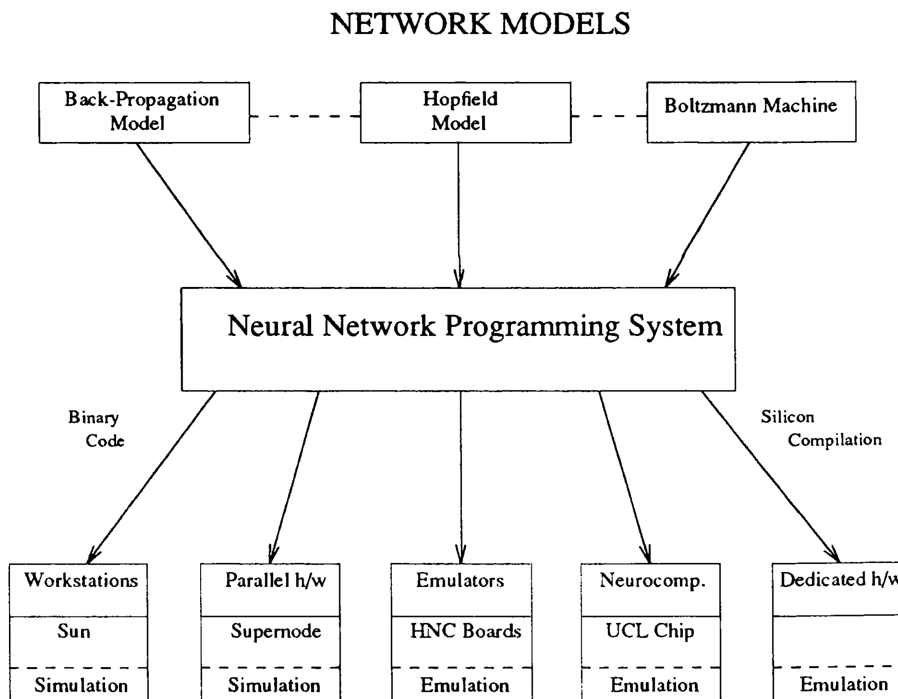

Figure 4. A Simplified View of Network Programming

What this diagram tells us is that a neural network programming system accepts neural network models as input and transforms them into a suitable form (eg:-binary code) and maps them on to the desired hardware for execution.

Typically a neural network programming system consists of four major components (see Figure 5). These are:

- **An algorithms library** - which contains a set of parameterised neural network algorithms such as Hopfield, Boltzmann, Back propagation, and Competitive

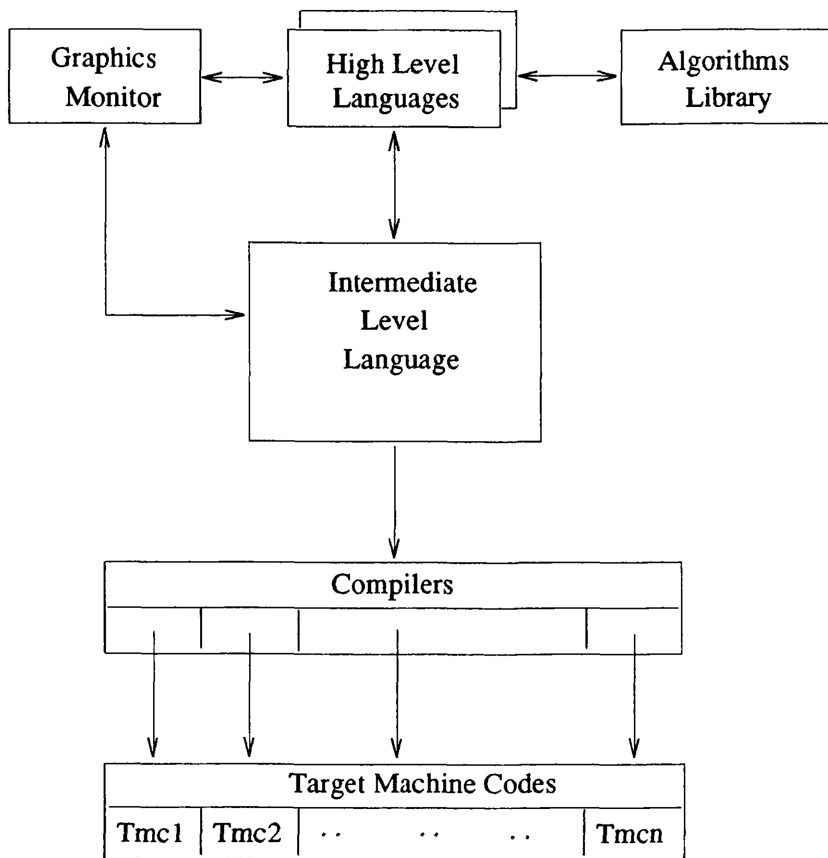

Figure 5. A Generalised Neural Network Programming System

learning. These can be configured for a specific user application by providing the necessary parameters.

- **A graphic monitor** - provides the facilities for the user to express the network models in the form of a network graph which can be then translated into either a high level language program or an intermediate language for further translation and eventual execution. This also provides an easy way of controlling and monitoring the network at run time.

- **A high level language system** - this system (language + compiler) allows the user to specify algorithms in the form of a high level description and compile it into either graphic or intermediate form for eventual execution.

- **An intermediate level language system** - an intermediate level language system (language + compiler) is the converging point of the system in that the high level language programs and the graphic descriptions of the networks are translated into this form first and then only compiled into specific machine language for execution. This form is a low level description of the network model and acts as a common form from which specific target machine codes are generated. An

intermediate language compiler can be dedicated to a particular hardware if the role of this language is to provide a common form of representation for a particular architecture or be a common representation such as a *virtual machine* that can be mapped on a range of hardware architectures.

Of these major components, the algorithms library, high level language, and the graphic monitor can be classified as programming tools. These are there to aid the user to communicate with the hardware system. On the other hand, the intermediate language serves a different purpose, which is **portability**. By being able to represent the network in a simple and general form, it is able to support portability. It must also be said that the high level language also deals with the aspects of expressibility and portability.

Having briefly introduced a general neural network programming system let us now look at two of the major problems associated with neural computing on the whole and the neural network programming system in particular. These are *programmability* and *portability*. A good system should offer the user the facilities for programming a range of neural network models and mapping them on a range of hardware. These two major issues are not uniquely associated with neural computing. These issues concern computing in general. But in the case of neural computing it is more important because of the cost involved in producing application packages which are commercially acceptable. The next two sub sections considers these two issues in greater detail and tries to evaluate their importance.

1.2.1 Portability

Portability is considered to be an important issue because:

1. The implementation of a neural network model consists of two phases namely, training, and recall. It is generally accepted that the training phase consumes an excessive amount of time. This suggests that it is economically wise to train a system on a fast computer and use a low performance machine for the recalling phase. This stresses the need for a portable system.

2. Recent years have seen the emergence of a diverse range of hardware for executing neural network models, each with their own programming systems. This means that there are what one would call "software barriers" to be crossed by the programmer to execute these network models on different machines available to him/her. To overcome this barrier, one needs a programming system that supports portability over a range of

machines.

3. The ultimate aim of neural network modelling is to develop practical applications for industrial use. In this context, an application developer wants to be able to have the constructed application made available across a wide class of users with varying computer platforms. This again demands portability.

1.2.2 Programmability

Programmability of a particular system concerns the provision of tools for easy and efficient programming. In the case of neural computing, major tools which contribute to programmability are the implementation languages (high and low level) and graphic monitors. Of these two major tools, we believe that the languages are more important as they are the ones which offer more freedom in terms of expression and control. In neural computing, we believe that programmability should also include model independence. This leads us to consider the following issues:

1. Conventional languages - Conventional languages like C, C++, and OCCAM are being used to implement neural network models and applications in many programming systems. The main advantage in using these languages to implement neural networks is that most of the conventional machines have compilers available for these languages. The disadvantage in using these languages are that they do not have specialised features for efficiently implementing neural network models. In addition, although they are portable across a wide range of sequential machines their portability across parallel machines is restricted.

2. Specialised high level languages - Recent years have seen the emergence of a number of specialised high level languages for implementing neural network models and applications. These specialised neural network languages, like other application specific languages for other areas of computation, typically offer more specialised features on top of a classical language. They are usually complex in nature and tend to be better at expressing a particular sub class of the problem domain. This naturally encourages the user to choose different language for different application. The major advantages are that these languages provide better expressive power and domain specific features for coding the algorithms. On the other hand these languages are biased towards application building and often lacks explicit parallel constructs or explicit features to aid

parallelism.

3. Specialised Low level languages - First of all, having a specialised low level language which is similar to an assembler may help to port the network models across a range of machines, but it may not contribute to programmability. What would be more suitable is a language that can be both a target level language as well as a programming language. This will offer the user the best of both worlds. In the case of neural network programming, it is useful to have the facility of programming at the intermediate level. These requirements lead us to a specialised low level language which is at a similar level as OCCAM so that programmability can be achieved.

In conclusion, our aims namely, portability and programmability demands a programming system where the user has the choice of a number of high level languages which can be compiled down to a common intermediate representation (intermediate level language) that can be mapped on to a range of hardware. In addition, this language should have the necessary basic features so that it can be used as a low level programming language for implementing a range of neural network models.

Having considered the issues of portability and programmability and identified their importance in neural computing systems, the next section states the goals of this thesis and presents a brief description of the research carried out in order to reach those goals.

1.3 Aims and Background to Research

As mentioned earlier the work described in this thesis represents a precursor of the PYGMALION project [Ange89] funded by ESPRIT II (project 2059) aimed at producing a general purpose neural network programming environment and applications.

The research reported in this thesis on the portability and programmability of neural networks was undertaken in conjunction with a complementary project on neurocomputer architectures [Pach91]. They started in October 1986 and aimed at producing a general purpose neural computing platform that could support:

- a wide range of neural network models and hardware;
- applications based on neural network techniques;
- further research and experimentation in this area of computing.

Firstly, a primitive processing element (PE) which was to be the basic element of a massively parallel computer that supported neural computing [Pach88] was designed. The design of this PE was completed jointly with another student by the latter part of April 1987 and forms the basis of another PhD project in VLSI design [Pach91]. This PE was simulated at the register transfer level and configured in an array to form a network of processors. Then the design and implementation of a neural network programming system to aid the development of various applications based on neural network models was undertaken. This led to the design of a neural network programming system called NPS. Analysis of this design showed that there were basically two major areas to be investigated, namely, the **higher level** and **lower level** of the system. The higher level consists of high level language(s), algorithms library, and the graphics monitor(s) as major components. We believe that these tools falls into the category of human computer interactions and ample attention is being focussed on this area [Hood87]. The lower level is concerned with portability and programmability at the intermediate level. We believe that, portability is the least investigated area in neural network computing and there was a need to address this problem because of the diversity in the available hardware for executing neural networks. On the issue of programmability, what we are looking for is a low level language which is capable of implementing a range of neural network models (i.e - **model independence**) and expressive enough to also be a programming language at the intermediate/low level. Programmability at the intermediate/lower level is an important requirement in neural network computing due to its dynamic nature and computational complexity. This initiated the design and implementation of a specialised low level language whose main aims are to support **portability** and **programmability**.

1.3.1 NPS and NIL

Having identified the major aims of the thesis, a full neural network programming system, NPS, was outlined. This system as shown in Figure 6 consists of:

- *high level languages*; a number of high level neural network programming languages (HLL).
- *a graphic monitor* for building and interacting with the running network.
- *a specialised low level language* that can be used as both an intermediate level language to represent the high level language and graphical description of the network in a suitable form for mapping on a range of machines and as a programming language at a lower level offering the user the facility of multi-level programming.

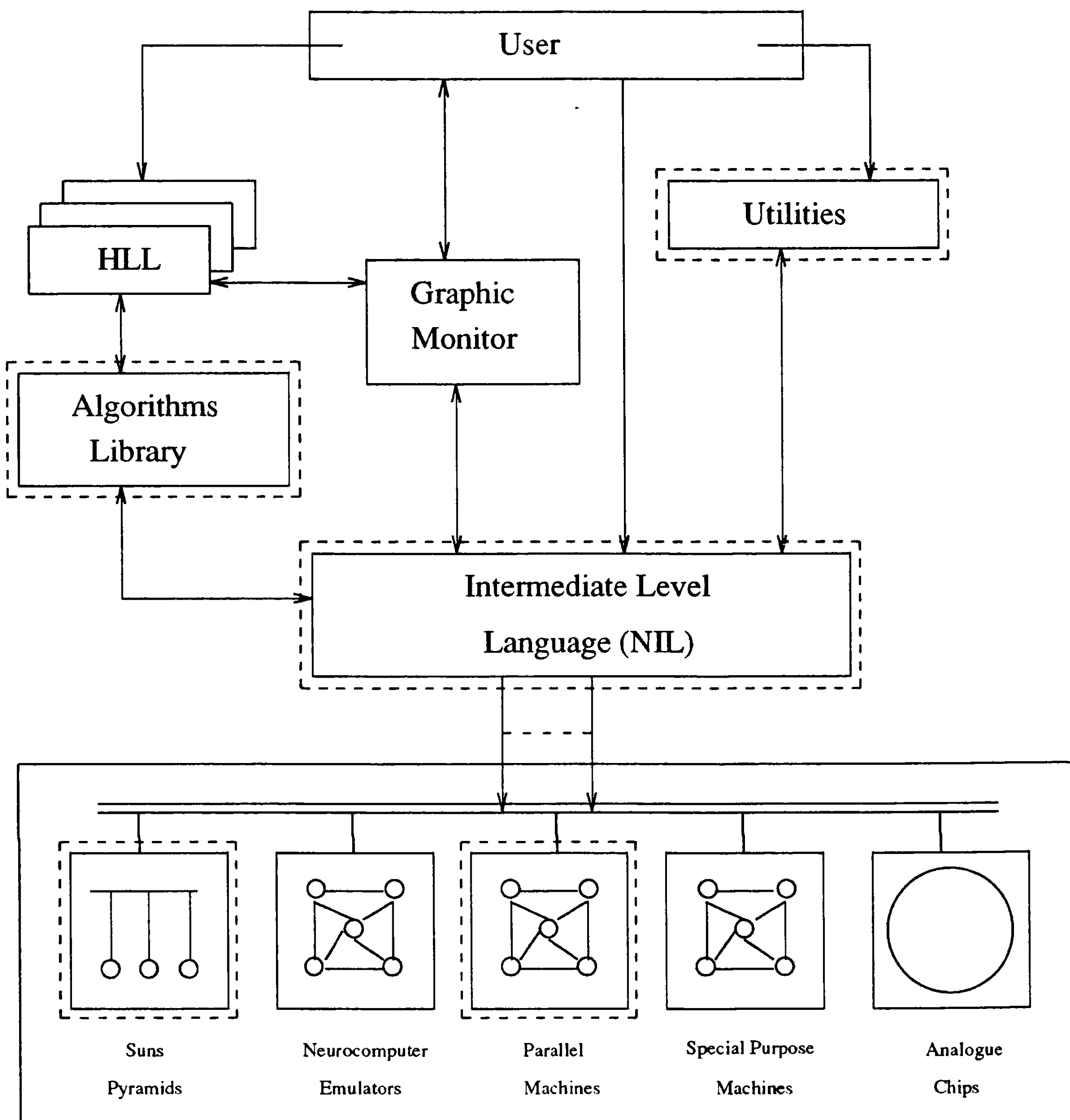

Figure 6. Neural Network Programming System, NPS

- *a utility system*, comprising a set of useful tools such as context editors, debugging aids, and software to save the partially trained networks for further training and recall at a later time.

- *an algorithms library*, consisting of a set of popular models which can be executed by supplying the necessary parameters.

Since the main aim of the thesis is to investigate portability and programmability of neural network models and applications, and a full implementation of NPS is beyond the scope of this thesis, it was decided to implement only the relevant parts of the NPS (indicated by the broken lines in Figure 6) to demonstrate the feasibility of the solutions proposed in this thesis. Thus, the currently implemented system consists of:

- *a low level language*, NIL, to build a spectrum of neural network models and applications and map them on a range of hardware.
- *a utility*, to save partially trained networks for further training and recall at a later time.
- *a library*, consisting of a set of popular models which can be executed by supplying the necessary parameters.
- *a facility*, to generate a range of target machine code for different architectures based on a simple *virtual machine* called the **C-Machine**.
- *a neurocomputer architecture simulator* based on the design specification mentioned in chapter 3. As part of the project, a compiler was implemented to map NIL on to this simulated architecture to test the practical feasibility of mapping NIL on parallel hardware.

The main reason for including the C-Machine (a "C" based virtual machine) is to show that NIL can be mapped on to a range of conventional hardware. This is based on the fact that the majority of the currently available hardware (network of transputers, conventional machines such as Sun workstations, connection machine, and neural network hardware like HNC [Anza87]) support languages which use "C" language as their base.

The basic aim of the neural network programming system NPS as far as this thesis is concerned is to enable us to demonstrate the feasibility of the proposed solutions by serving as a platform for demonstrating the portability and programmability aspects of NIL. This is why the currently implemented systen offers neither a high level language nor a graphic monitor.

NIL consists of two major components:

- *A network implementation sub-language*, which enables a network to be built by providing suitable statements for specifying the functions of the nodes and the topology of the network.
- *A manipulation sub-language*, which provides monitor, control and modification capabilities for the network.

These sub-languages together produce a *low level, machine independent network specification language*.

The primary design aim of the of the network specification language NIL, is to support:

- **portability** by being target machine independent.
- **programmability** by being able to

  a. represent a spectrum of network models including neural networks (model independence).

  b. express the connectivity and computations in a clear and concise manner.

  c. to handle non-determinism. This is achieved by splitting the functions of a node into guarded processes and randomly selecting one of the eligible processes for execution (see chapter 4).

The computational model on which NIL is based is formulated to support a variety of neural network models. NIL tries to provide capabilities such as implicit synchronisation and non-determinism to meet the basic needs of a network system and neural networks in particular.

NIL is believed to be capable of implementing a wide variety of network models in general and neural networks, and semantic networks in particular. We believe this general property is a significant advantage for the following reasons:

1. Being at the early stages of neural network modelling no one knows what the future models are going to be except that they will be based on network principles. So an intermediate network specification language should be general enough to represent network models of any type to cater for the future needs.

2. Building applications based on neural network techniques may involve the use of different network models. This means that a general network language rather than a special purpose neural network language is useful in building such heterogeneous network systems.

In addition to this general property, it is also found that this language is suited to programming a network of transputers. This, we felt was another significant achievement when considering both the current difficulties with the mapping of algorithms on a network of transputers using the existing programming tools and the popularity of the transputer as a building block of parallel computer platforms.

1.4 Outline of the Thesis

This thesis presents the design, implementation and the assessment of the neural network programming system, NPS and the network implementation language, NIL. In chapter 2, a survey of neural network programming systems is presented. This survey is used to form a critical assessment of the existing systems, as background to design a neural network programming system. In chapter 3, the proposed programming system NPS is presented. It also discusses the reasons behind some aspects of the design. In chapter 4, the network implementation language NIL which forms the central part of the proposed programming system is presented. In this chapter, semantics and syntax, and the computational model of the language are discussed. In chapter 5, implementation details of NPS and NIL is presented. This includes the detailed description of the virtual (C-Machine) machine, the data structures and the communication mechanism between the application code and the system. In chapter 6, an assessment of NPS and NIL is presented. This chapter includes sample programs, coded in NIL, to run some well known neural network models, a brief comparison with existing languages in this class, namely Occam and BIF [Bahr87], and a discussion on how to compile programs written in high level languages into NIL. The discussion on translating high level languages into NIL involves the loose specification of a high level language that combines most of the major features found in majority of the high level neural network languages currently in use and showing how it can be translated in to NIL. Finally, in chapter 7, conclusions are drawn and future work is discussed.

Chapter 2

This chapter presents a survey of systems for programming neural networks. This includes the descriptions of some of the well known systems. The main reason behind this survey is to ascertain the current state of research in this area and to serve as a background to our investigation of neural network programming systems.

2. Neural Network Programming Systems

The term "Programming Systems" refers to the collection of software and hardware tools available to a system developer to build software systems. A neural network programming system provides facilities for network specification, testing and execution. These facilities may be provided in the form of a set of stand alone tools or as an integrated software/hardware package. A good programming system offers a systematic path for developing software by providing the necessary tools at every stage under a single environment. It is this approach that is adopted by most of the neural network programming systems for developing software [Kolo88, Hans87, Paik87, Test88]. To specify and control the neural network most of the systems offer either a special purpose high level network specification and control language or a graphical system or both. Most of the debugging, run time control and manipulation is carried out with the aid of the control component of this language and its graphical counter part. Apart from offering these high level tools as basic facilities and a few minor aids, the majority of the existing systems do not offer anything radical in the way of a debugging tool for parallel systems or intermediate/low level programming. Most of these systems are also dedicated to particular hardware. That is, they generate executable code for a specific hardware.

A few of these neural network programming systems have an intermediate level language so that the high level description of a network can be commonly represented in this form for further translation and execution on different machines.

This chapter presents a survey of the existing Neural Network Programming Systems with the view to examine their strengths and weaknesses in order to help us design a neural network programming system that satisfies our intended goals which were stated in the previous chapter.

2.1 Classification of Neural Network Programming Systems

Currently, neural network simulations are mainly done using tools which were developed for traditional computing on conventional machines such as C language on workstations. However, building, analysing, debugging, and maintaining a neural network is quite different from that of a traditional sequential program. This is due to the fact that there is no single point of control that can be traced in a parallel distributed system like the neural network. More specialised tools are needed to aid the developers to specify networks, observe their behaviour, identify faults, correct the problems, and retest the networks.

Although there is a certain degree of commonality among the different types of neural network programming systems, they can be broadly classified according to the specific purposes for which they are developed into three categories.

1. Educational systems
2. Research systems
3. Commercial systems

2.1.1 Educational systems

These are primarily intended for introducing neural computing to the novice users. Some of these are free whilst others are sold as commercial products. The majority of these packages are model dependent (i.e.- restricted to execution of a given set of popular models such as Hopfield, Boltzmann, and Back propagation etc).

NNPSs in this category include Adaptics, Netwurkz, NeuralWorks Explorer, and the PDP Exercise Neural Network Tool. They are briefly summarised in Table 2.

Neural Network - The Course

Neural Network - The Course is developed by Adaptics [Adap88] as a training software, available on MacIntosh and IBM-PC computers. Its main purpose is to introduce the various common neural network models to the novice user. This product is a part of a broad line of support services and products to assist companies to use neural network techniques.

Organisation	Environment	Description
Adaptics	NEURAL NETWORK - THE COURSE	Simple Training Software for NN
Dair Computer Systems	NETWURKZ	Training Software for NN with a low-level network language PL/D
NeuralWare	NEURALWORKS EXPLORER	A cut-down Educational Version of the NEURALWORKS PROFESSIONAL series
Stanford	PDP EXERCISE TOOL	Free Software available with the Rumelhart PDP book vol.3

TABLE 2. Educational NNPSs

Netwurkz

Netwurkz [Netw87] is a neural network simulator, developed by Dair Computer Systems, available on IBM-PC computers. It is intended for people not familiar with neural networks. Associated with Netwurkz is a low-level network language PL/D which is based on list storage and representation. Netwurkz is itself implemented in PL/D. A best-fit pattern recognizer demonstrator program called "Spell" is also included.

NeuralWorks Explorer

NeuralWorks Explorer [Neur89] is a basic programming environment, developed by NeuralWare, available on the IBM PC, XT, AT, PS-2, Sun Microsystems SUN/3 and SUN/4 computers. It is an introductory package with several of the features of a more powerful package *NeuralWorks Professional II* (see later sections) marketed by the same company, but with fewer capabilities. Basically, this package provides a number of parameter driven neural network models and some example applications. A primitive graphical environment is also provided.

PDP Exercise Neural Network Tool

This tool is available on MS DOS or UNIX Operating Systems, and is given free with the Rumelhart neural computing book [Rum86d]. It is produced as a training tool to illustrate the various PDP models covered in the accompanying book. It provides a simple version of each of the basic PDP models. The user is guided through those models by a textual description of the behaviour of each model. The effects of changing the

parameters of these models can also be observed.

Comments

The main strengths of these educational NNPSs are that they are relatively cheap and usually adequate as training and educational tools for novice users. They guide the user through the basic neural network models, which are well described. Moreover the user can learn the effect of slight changes on various components and parameters (e.g. learning rate and threshold function) of a model. One of the best in this respect is the PDP Exercise Neural Network tool. Finally they provide the user with facilities for understanding the exact behaviour of the model (such as running the network in a "slow-motion"), which is otherwise difficult to capture due to the dynamic nature of neural networks.

However, because of the model-dependence of these NNPSs, the user doesn't have the flexibility required to experiment with new and novel neural network models. Moreover, since their purpose is educational and not intended as a tool for building commercial or research products these environments are usually slow in operation. In addition these NNPSs lack a proper user-friendly graphical interface. Finally these NNPS are appropriate only for small models; they are not effective in modeling neural networks with large number of neurons (e.g. counter propagation model). But, as the major aim of these NNPS is educational these restrictions are minor.

2.1.2 Research systems

The group of research NNPS encompasses various experimental research environments. They include NETSIM [Test88,Gart87], the King's College Simulator [Smit87], P3 [Zips86], SNAIL [Hood87], NDL/ANNE [Bahr87], Rochester Connectionist Simulator [Feld88], UCLA SFINX [Paik87], and IBM CONE [Hans87]. Some of them are hardware dependent whilst others are more general purpose and are not designed to run only on a specific target machine, although very often a specific hardware emulator is included. NNPSs under this category are summarised in Table 3.

NETSIM

NETSIM is a specialist parallel neural network simulator. It has been designed for high speed simulation of large systems of networks. There is also an associated VLSI chipset forming the core component of the simulator. But the importance of the ability to

Organisation	Environment	Description
Texas Instruments/ Cambridge Univ.	GRIFFIN	A Parallel Network Simulator based on the TI NETSIM neurocomputer
King's College London	King's College Simulator	A PDP Network Simulator implemented on Multiple-Transputer System
Stanford Univ.	P3	A Lisp-based Windowed Environment with Graphical display, Plan/Method as the Specification Language
University College London	Pygmalion	A Programming System for a range of Connectionist Models, with NC as the Specification Language
Carnegie Mellon University	SNAIL	An Interactive/Graphical Windowed Programming Environment (Textual Description Not Supported)
Oregon Graduate Centre	NDL/ANNE	An Integrated Programming System with NDL (as HLL), BIF (as ILL), H/W mapper, and Intel iPSC Emulator
University of Rochester	ROCHESTER CONNECTIONIST	A C-based Hierarchical Graphical Programming Environment that runs on various Hardware Emulators
UCLA	SFINX	A C-based Interactive Programming Environment with Primitive Graphics, ILL, Assembler (but no HLL yet)
IBM Palo Alto	CONE	An Extensive Environment with Library, GNL (as HLL), NETSPEC (as ILL), XIP Graphical Monitor, and IBM NEP (as Hardware Emulator)

TABLE 3. Research NNPSs

design and debug programs has been recognised in this NNPS. Therefore a software environment is provided with this parallel NN simulator [Test88]. The user interface is based around a multi-window environment which is driven either by the user application program (written in C) or by the interface itself. However, the NNPS is very low level in nature in that it mainly facilitates the user not in the high level construction of models and applications but in the low level efficient utilization of the parallel hardware.

Kings College Simulator

Kings College Simulator [Smit87], corresponds to a PDP network simulator implemented on a multiple Transputer system. The system is implemented via two processes : the user interface and the network simulator. The user interface provides functions to the user to analyse and simulate a PDP network. It supports all the PDP network models, taken from the book by Rummelhart *et al* [Rum86d, Rum86e]. It is probably the first system designed to simulate neural nets on a network of transputers. The major facilities provided by this system includes, facilities to construct networks and map them on the hardware optimally, display network structures and unit status, and display simulation performance.

P3

The P3 system was developed by Zipser and Rabin as a simulation tool to aid the development of parallelly distributed processing models [Zips86, Trel88a]. P3 is a language centred system which is implemented in LISP and runs on a Symbolics 3600. Although it was not originally intended as a neural network development system, it has the necessary facilities and an inherent parallel structure that would map onto suitable neural network hardware. The major components of the P3 system are a *plan language*, a *method language*, a *constructor*, and a *simulation environment*.

The *plan language* describes the collection of nodes (called units in P3) in a network model and specifies the connections between them. To do this, the language uses a small but rich set of statements. The three fundamental constituents of the plan language are a **UNIT**, a **UNIT TYPE**, and a **CONNECT**. The **UNIT TYPE** statement names and describes a kind of unit. The **UNIT** statement instantiates and names actual units. This statement can instantiate either a single unit or a whole array of units of the same type. The **CONNECT** statement makes connections. Associated with each type of unit is a method which defines the computational behaviour of the units in the model program. This method is described in the *method language* which is an extension to LISP. After the plan and the associated methods have been specified the *constructor* generates a distributed data structure. This data structure is loaded into the *simulation environment* for execution. This simulation environment is highly interactive and makes extensive use of the "window" system and the "mouse" pointer of the Symbolics 3600. It offers two layers of debugging tools for testing the code and monitoring its run-time behaviour interactively. The first layer of the debugging allows the user to test and verify the network connections and the second layer allows the user to test the individual units. It allows "strip-chart recorders" to be connected to any of the parameters of a unit so that

the behaviour of that parameter over the time can be measured. The simulator also provides the user with a display of nodes in an orientation of his/her choice.

The environment of the network is handled by an appropriately defined environment unit which handles any input or output connections, and has a suitably defined method. Control over the update sequence of units is also handled in this manner via a control unit. Without this control unit each of the units will be sequentially executed, but through this unit an asynchronous updating process can be simulated.

SNAIL

SNAIL is a an interactive graphical tool for designing and testing neural networks which was developed by scientists at Carnegie Mellon University [Hood87]. A neural network design in the SNAIL system consists of a set of drawings. With these drawings, one can construct a network by selecting appropriate primitives for drawing neurons, synapses, etc. Labels can be attached to lines in order to reduce the number of connecting lines, just as is done in electrical circuit schematics. In SNAIL modifications, deletions and additions of elements of network can be made interactively and the consequences of the change can be seen immediately. States of the network are shown by assigning different colours for different parameters and their current values. Windows can be created with parameter's names and their values, and the user may modify any of these parameters. Parameter windows may also be created for abstract objects, such as neuron types, or a set of global simulation parameters, and the user can also modify any of these parameters. Parameter windows, once displayed will have their values continuously updated while the simulator is running. Neurons firing rate or synaptic strength over time can also be displayed by requesting a chart recorder window.

Just as in designing circuits where it is extremely useful to abstract away from the transistor level, and work in terms of gates or registers, in SNAIL system one can encapsulate portions of a network up into prototype drawings, which can then be used as units in themselves. Once a prototype drawing has been created, an instance of it can be included in a drawing by creating a *block*. In the simplest case, the name of the block is just the name of the prototype drawing which is to be included at that location in the drawing. Connections are made between lines external to the block and lines internal to the block by means of labels. The prototype drawing includes labels on all lines to which external connection can be made. In the drawing containing the block, lines attached to the block are labeled at the point of attachment to indicate the internal lines to which they should be connected. It is also possible to use blocks within prototype drawings, thus permitting the construction of hierarchical network designs.

NDL/ANNE

ANNE (Another Neural Network Emulator) is a general purpose neural network simulation system, developed at the Oregon Graduate Centre for the Intel iPSC [Bahr87, Trel88a, Trel88b]. Although an emulator exists in the form of an Intel iPSC multi-processor, the development environment is not dedicated to any particular hardware architecture (see Figure 7).

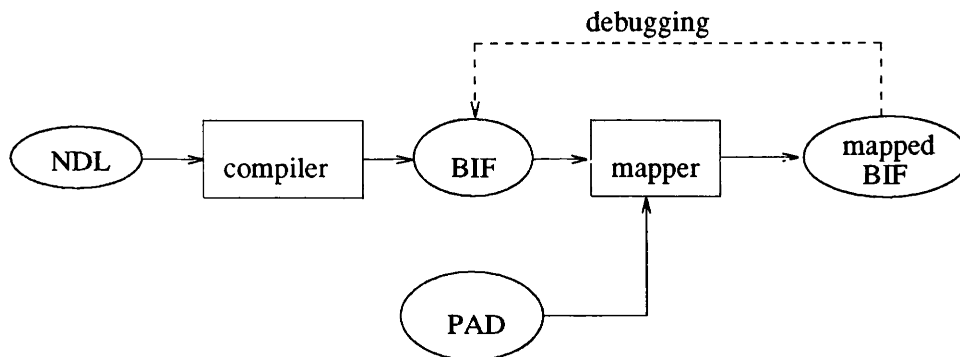

Figure 7. A Typical Development Process in ANNE

The user describes the network using a high level NDL (Network Description Language) which is then compiled into a low level generic BIF (Beaverton Intermediate Form) which is a common specification format designed to express network structures with both generality and compactness. Mapper is a tool for assigning the network structure (by graph partitioning) to a particular target machine architecture. It takes as its inputs a BIF file and a PAD (Physical Architecture Description) file describing a specific target machine architecture (which is the Intel iPSC hypercube in this case). The functions of the nodes for learning and computation are specified by pointers to C procedures. The output of the mapper is then used by ANNE which acts as a test bed and debugger for the neural network model described by BIF. Using ANNE the user has the ability to examine, modify or save pertinent data within the network, including the entire BIF specification of the network at any point in the simulation. It should be noted that, although the design of NDL/ANNE is intended to be hardware independent the only existing version runs on the Intel iPSC.

Rochester Connectionist Simulator

The Rochester Connectionist Simulator (RCS), developed over a long period of time at the University of Rochester [Feld88], is designed to be run on the Unix Operating System. Versions of the simulator have run on a DEC/VAX, a Sun workstation and on the BBN Butterfly Multiprocessor. It appears not to be dedicated to any neural networks

models.

The overall system consists of a user program, a Graphics Interface and the Simulator, which corresponds to a run-time environment. The neural network is built in the simulator via the user program (written in C). The user has to define, via a data structure, each unit, its sites and links (sites at which incoming links are attached). This specification has the ability to give a description at different levels of abstraction. The lowest level corresponds to a single-unit description; i.e. unit, sites and link functions. The next level corresponds to the description of the connectivity pattern; i.e. specifying the links and the group of units. The highest (user-defined language) may be read in and compiled into units and links by "user-supplied" functions. The Simulator and Graphics Interface are independent. The Graphics Interface allows the user to display network information during simulation and aids for the network debugging process. Furthermore the user can examine the network before, during, and after it executes via a "simulation window".

UCLA SFINX

SFINX (Structure and Function In Neural Connections) is a neural network simulator environment, developed at UCLA's Machine Perception Laboratory [Paik87, Trel88a], that allows researchers to investigate the behaviour of various neural networks.

As shown in Figure 8, the SFINX structure is analogous to traditional language based systems. A neural network algorithm is specified in a high level textual language which is compiled into an equivalent low level language. Next this low level language is assembled into a binary data structure (defining the network) and is loaded into the SFINX simulator for interactive execution. In SFINX, network specifications have two basic parts:

- **set of nodes** - a node is a simple computing element, composed of: *memory* storing the state of the nodes, and *functions* defining how signals are processed.
- **interconnections** - defining the connectivity and the flow of data amongst the nodes.

These network specifications are represented by virtual PEs, each comprising:

- function pointer
- output register

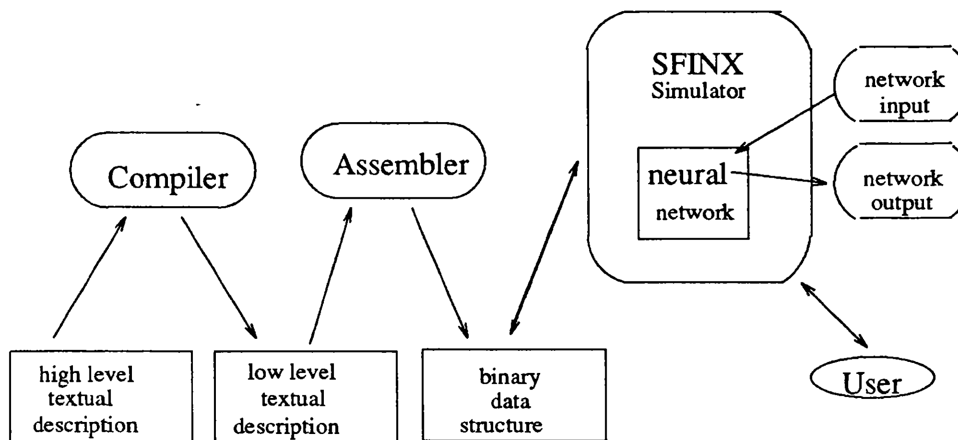

Figure 8. SFINX Environment

- vector of state registers
- vector associated weight/link_address(es)

Lastly, the front-end of the SFINX simulator is a command interpreter, accepting SFINX shell scripts. These shell commands include: *load, save, peek, poke, run, draw and set;* whose meanings should be fairly obvious. Once a network structure is created, these SFINX shell commands can be used to exercise the simulator, displaying and modifying the state of the network.

IBM CONE

The IBM Computational Network Environment (CONE) [Hans87, Trel88a] is based on hierarchical and functional decomposition design methodology and consists of a high level General Network specification Language (GNL), a generic intermediate network specification (called NETSPEC), and an Interactive Execution Program (IXP). Neural network programs are specified in GNL and compiled into a machine independent intermediate form NETSPEC. This intermediate specification is then assembled into an executable form dependent upon the execution engine (see Figure 9).

To describe a network in GNL, the designer uses three fundamental primitives:

- **proc**, which describes the functional processors.
- **path**, which defines the connections between **procs**.

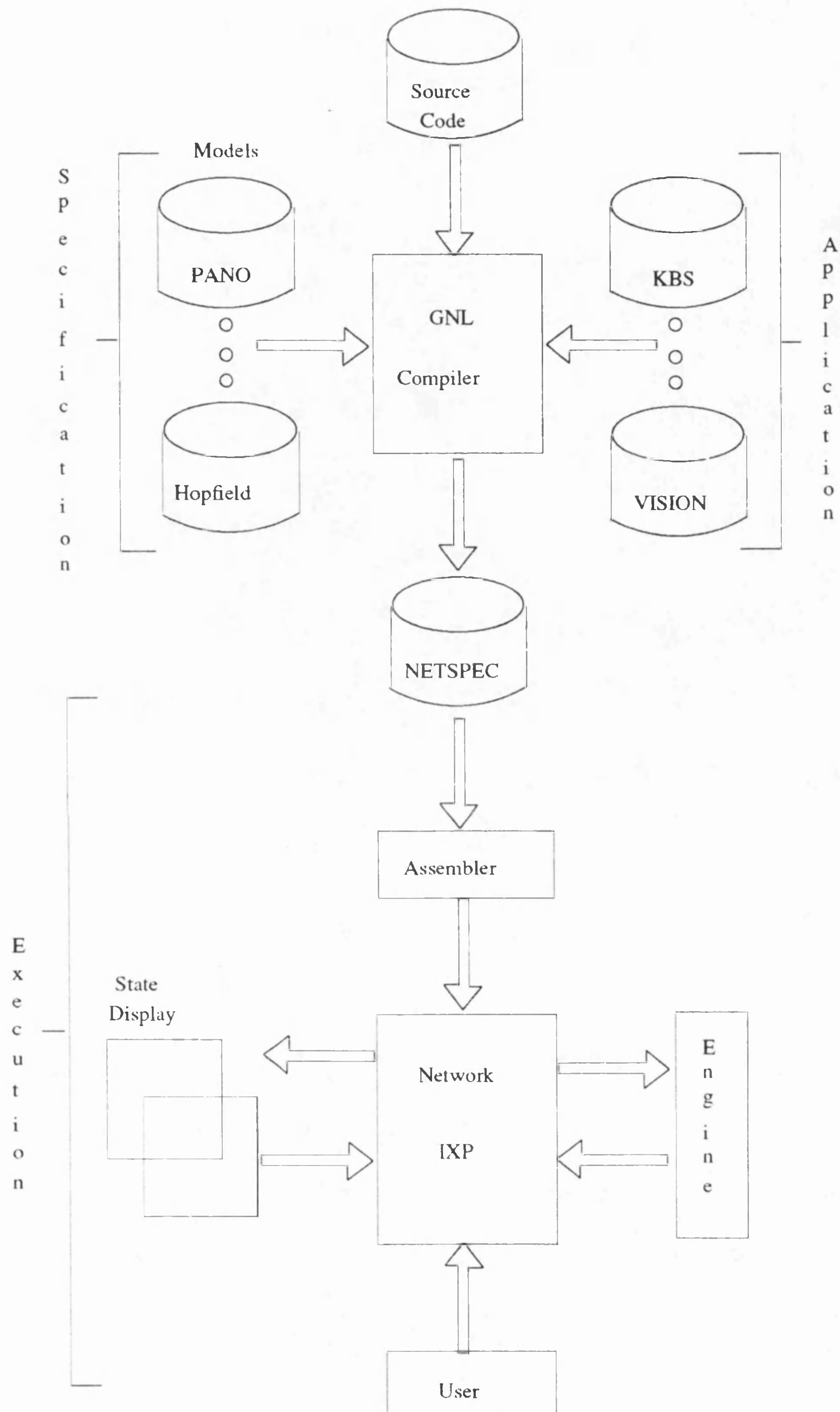

Figure 9. CONE System

- **port**, which specifies the point at which a **path** joins a **proc**.

For any specific application area, the designer can establish a library of **procs** and these are then made available to the compiler in a fashion completely analogous to providing a

scientific library of routines. The NETSPEC description is simply a parts lists of various primitive processors and a listing of the topology.

To run and manipulate the network, CONE provides an execution environment called NET Interactive Execution Program (IXP) that runs on a PC connected to the Network Emulation Processor (NEP). The IXP has three major components:

- a graphics display,
- an operator interface,
- a network engine.

The graphics display enable the operator to view the state of the network by selecting a set of **procs** of interest. The operator can also vary the hierarchical depth for which the network **procs** are to be decomposed. The IXP operator interface is a command shell which controls the execution of the network engine and the display terminal. It has been developed in close association with the hierarchical GNL compiler. Thus, the IXP interface recognises the hierarchical decomposition of the network **procs**. The network engine provides a simple interface to the target hardware. The IXP has isolated the entire engine interface into a couple of low level functions. Thus, the IXP itself can be easily modified to support the inclusion of special purpose hardware for the network updates.

Comments

Most of the NNPS in this category are hardware specific. In terms of execution speed, the advantages of a hardware specific NNPS are obvious. Indeed, if the main purpose is to create a specific end-user application then NNPSs from this group will fit the bill well. They can provide complete solutions to specific problems. But when the aim is to build a general purpose portable NNPS capable of integrating novel applications, then a hardware independent NNPS seems to be a priority.

The obvious disappointment as far as these NNPSs are concerned is the failure to achieve the expected level of performance Hood87, Zips86]. In terms of speed, the dedicated systems reduce the running time by a significant margin. Even for ANNE which runs on a parallel target engine (namely the Intel iPSC), the results in terms of speed and storage efficiency are far from adequate, and in fact are much worse than expected. This problem lies in the fact that at this stage of technology development, no efficient general purpose "neurocomputer hardware" can be designed yet. To increase efficiency, some NNPSs give the user two ways of specifying their model. For instance, in SFINX the user has an explicit and an implicit way of designing his model. The

implicit specification is designed to take advantages of high degrees of regularity in connectivity patterns (as in low-level vision model). Because of the restrictions imposed on the user it increases notably the space and time efficiency. The explicit specification provides a great deal more flexibility to the user by assuming only minimal constraints. But, because of the massive amount of details that is needed in explicitly specifying large irregular models, the user is restricted to specify only small neural networks in reality. Placing too much reliance on graphical facilities can also become a liability for NNPSs. An example is the SNAIL NNPS. Because SNAIL is graphics-dedicated (i.e. specification is by graphics only) the user is restricted to the design of neural network models with only a small amount of neurons and interconnections which can be shown and specified visually.

Flexibility is the main strength of most of the research NNPSs. There are two aspects in this, viz., modelling flexibility and execution flexibility. Modelling flexibility is indicated by the degree of freedom a NNPS gives to the user in expressing a neural network model and application. This freedom depends on the expressive power of the specification languages used (e.g. high-level network primitives, description granulity, pre-imposed network structures and constraints, graphical/textual languages etc.). The Rochester Simulator is a good example of a NNPS that gives the user the power to construct specification at different levels of abstraction. A lowest level provides single-units description; corresponding to units, sites and link functions. The next level describes the pattern of connectivity via a set of functions which specify the links and groups of units. A higher level gives the overall network specification. Execution flexibility is measured by the kind of facilities offered to the user during a simulation session. Regarding the generality, user-friendliness, and integration of tools, the IBM CONE is one of the best example. These tools are highly interactive and provides graphical display at any level of the specification; network as well as individual nodes.

2.1.3 Commercial systems

NNPSs in this categories have an overwhelming commercial orientation. They include ANSE and Mark III/IV, HNC ANZA & AXON [Guts88], SAIC ANSim & ANSpec, Cognitron [Fuku88], NESTOR, NeuralWorks and Professional II. A summary of these NNPSs is given in Table 4.

ANSE and MARK III/IV

ANSE (Artificial Neural System Environment) is a design environment which supports the neural network designer in the areas of neural network definition, network

Organisation	Environment	Description
TRW	ANSE	commercial environment for TRW neurocomputers Mark III, IV and V
Hecht-Nielsen Neurocomputer	ANZA	sophisticated environment with library, object-oriented HLL, for HNC neurocomputers ANZA and ANZA Plus
Cognitive Software	COGNITRON	multi-window icon-driven environment + LISP-like HLL for Macintosh
Nestor	NESTOR DEVELOPMENT	A NNPS for pattern recognition and signal processing applications
NeuralWare	NEURALWORKS PROFESSIONAL II	Graphical environment for IBM PC and SUN, C Converter Available
Science Applications Int. Corp.	SIGMA/ANSpec	commercial environment with library, object-oriented HLL, for SIGMA/DELTA neurocomputer

TABLE 4. Commercial NNPSs

editing, network storage and retrieval, and network implementation [Souc88].

ANSE is machine independent and it is compatible with the family of MARK computers, the most significant of which are the MARK III and MARK IV neurocomputers. The MARK III neurocomputer is a parallel processor implementing neurons as virtual PEs and virtual full-connectivity is supported. The MARK IV neurocomputer is a single high-speed pipelined uniprocessor, also implementing virtual PEs and virtual full-connectivity.

HNC ANZA & AXON

The ANZA package comprises: the ANZA User Interface Subroutine Library, basic Netware packages for the common neural network algorithms, the AXON specification language, and a IBM PC co-processor board for the speeding-up of floating point operations in neural network simulations.

The User Interface Subroutine Library (UISL) is a collection of routines providing access to the ANZA system functions. Examples include: load network, set learning etc. The basis of UISL is common set of data types defining formats for *slabs*, *weights* etc. As with the data types, the UISL routines adhere to a naming convention. The UISL routines names use the same set of nouns as the data types. Likewise, the UISL data files required to implement networks use four types of data: state data, weight data, constant data and network description data.

The Basic Netware Package contains five of the classic neural network algorithms in a parameterised specification that can be configured for a specific user application. These algorithms are: Back propagation, Spaciotemporal (Formal Avalanche), Neocognition, Hopfield (plus Bidirectional Associative Memory) and Counter-Propagation networks. In these networks the interconnection geometry and the transfer equations are already specified. However, the number of PEs, their initial state and weight values, learning rates and time constants, are all user selectable.

Lastly, AXON is a language for describing neural network architectures in a machine-independent form. It is object-oriented and its syntax combines features of Pascal and C, with constructs such as *weight*, *input class* and *slab* as keywords. AXON is based on a *generic* neuron model containing attributes such as output state, transfer function, interconnection class, connection weights, and local data memory.

The structure of a specific network is defined by four sections.

1. The *network parameter section* which defines the load-time and run-time constants.

2. The *network data declaration section* which declares the processing elements and their attributes.

3. The *network construction and connection section* which specifies interconnections.

4. The *network execution section* which schedules the updating and also defines the transfer functions.

SAIC ANSim & ANSpec

Scientific Applications International Corp. (SAIC) market a series of sophisticated tools and co-processor boards collectively known as *ANSkit* [SAIC88] for developing neural networks. The kit is designed for an IBM PC/XT/AT environment.

The main utilities of the kit are the *ANSim* simulator and the *ANSpec* programming language. ANSim comprises a Microsoft Windows operating environment, interfaces to dBase III and Lotus 1-2-3, together with files of the popular algorithms. The windows environment provides pull-down menus to select and change I/O format, network architecture, network learning algorithm, network training and execution, and displays of activations, weights etc.

The popular algorithms provided are:

1. Back propagation (with/without momentum, shared weights, recurrent networks),
2. Hopfield,
3. Boltzmann (learning, machine, I/O),
4. Kohonen feature map,
5. Adaptive resonance (ART 1, ART 2),
6. Enhanced counter propagation,
7. Bi-directional associative memory, and
8. Hamming net.

Networks can be loaded and saved, using data creation/load/save/modify commands, and neurodynamic equations specified using simulation, activation/transfer and learning functions.

ANSpec is a concurrent specification language for defining and simulating neural networks, and extending the ANSim environment. ANSpec is object-oriented allowing code developed for different applications including ANSim networks to be integrated into more extensive systems. Applications can include mixtures of ANS networks and other non ANS processes such as image processing, signal processing, and data base management. The ANSpec specifications can either be simulated in a virtual processing environment of thousands of concurrent processors or used for native code generation for one or more Delta floating point co-processors, also available from SAIC.

Cognitron

Cognitron is a MacIntosh-based neural network simulation system. It is designed as an advanced network model simulator based on the principles of parallel distributed

processing, and utilizing multi-window displays [Fuku88]. A main feature of the system is the provision of "Modelling Windows" which allow the user to design a network and its components in a hierarchical and graphical manner. There is also a textually-oriented *Creator/Editor* primarily used to program the functionality of units and initial weights. The simulation engine is represented by a specific simulation window. This window provides the user with an implicit way of start, pause and resume a simulation, as well as graphical displaying simulation status. Finally, there are facilities for time-charting the output behaviour of any unit in a network. Input/Output handling is done via normal file handling, and is compatible with standard graphing, statistics, and spread sheet programs. The user interface has an appearance of a common MacIntosh software package in that it is intuitive and heavily graphical-oriented. The system appears capable of supporting most currently imaginable neural network models.

NESTOR

NESTOR Inc. has produced a number of standard products. Their first product is the *Nestor Writer* which allowed a computer user to read handwritten text into a typical data-processing system. The system is implemented using a standard IBM PC. A follow-on product is the *Nestor Decision Learning System* for the financial-services market. Finally the *Nestor Development System* (NDS) provides access to the *Nestor Learning System* for the development of solutions to pattern recognition or signal processing applications. Therefore NESTOR corresponds to range of different products serving the needs of different markets.

NeuralWorks Professional II

This is a fully fledged version (superseding an earlier release--NeuralWorks Professional I) of an educational system called Neuralworks Explorer mentioned previously, and is available on most PC's and workstations (such as SUNs).

It is supposed to be a neural network environment for the "neural computing professionals". In this system, in addition to a number of manufacturer-supplied standard neural network models and applications, the user can define any network topology (provided that it can be structured in terms of layers of neurons) and functionalities. A package called the *Designer Pack* is also available for the conversion of a network design into a subset of standard "C" code for portability and easy integration with other conventional systems. Network manipulations are mostly icon-driven. There is also a library of learning rules, activation rules, summation functions, and transfer functions. A simulation run can be controlled in a number of ways, e.g. continuous, counted, or single

step. Input/Output compatibility with Lotus 1-2-3 and dBase III is supported as well as user defined I/O. It seems partially executed networks can be saved and restarted through a *check-point* facility.

Comments

It seems that most of these commercial packages tend to place more emphasis on performance. This inevitably leads to a certain amount of loss of flexibility in terms of modelling and executing of neural networks in order to make them commercially attractive(i.e. better performance and less flexibility). The *Nestor Learning System* is a typical example in that producing a pattern recognition application is feasible but observing the features of different learning schemes is quite impossible. On the other hand some commercial NNPSs (e.g. ANZA) contain their own model specification languages (e.g. AXON) which are flexible enough for specifying most of the existing neural network models as well as new ones. Another common problem is the lack of flexibility in the investigation of the behaviour of models during a simulation run, and in the precise control of the system during a simulation. However, these NNPSs usually come with a bundle of popular models and some of them even contain libraries of common applications. The result is that the user can start using the technology in a very short time. This factor undoubtedly has commercial appeal.

2.2 Assessment

A good NNPS should provide facilities for building and simulating a variety of neural network models. This means that it should have the ability to deal with different topologies, activation functions, and learning scheme. This demonstrates that a NNPS (whether it is a commercial or a research system) needs to be able to specify and run any type of neural network model. This is a crucial requirement when investigating novel neural network models. In order to have such flexibility, at the specification level one needs a high level language which should be heavily oriented towards the process of model and application construction. This means that it should be very powerful and flexible in its descriptive power. Efficiency considerations should not be a major concern in this language. Indeed, efficiency concerns should be more appropriately dealt within a low level language. Moreover, NNPSs have to be able to deal with heterogeneous neural network models. These are models formed by a combination of different basic models (shown in the previous section). They can be combined in parallel, in series or in hierarchies. For example, Josin [Josi87] has demonstrated an interesting model composed by the combination of an optimisation model and a self-organising model. Such a model has been designed for a robot's arm control application. There are no general rules for

combining neural network. However, some sort of hierarchy concept (or level) appears to be useful.

NNPSs have to be able to accommodate a wide range of neural network models including some possible new ones, both from the point of view of the design and the execution. Thus there is a need for a target engine independent NNPS. Indeed, if the purpose is an end-use application there is no doubt that a dedicated hardware NNPS is the best choice since it gives a complete solution to a specific problem. But when the aim is to build a general-purpose NNPS then a target-engine independent NNPS seems a priority. On the other hand we should bear in mind the efficiency of execution. But it is premature to seek a general purpose neurocomputer given the current state of the technology in parallel architectures for neural networks [Feld88]. Even if today's technology moves toward parallel systems, it is not necessarily a move towards massively parallel "neural-style" machine. Therefore a flexible NNPS should contain an intermediate level language able to be mapped onto any future hardware.

Another important point is the flexibility given to the user in terms of execution. Indeed, neural networks are dynamic systems. Thus it is quite difficult to understand all the processes involved at the same time. Graphics interface has proven to be indispensable for displaying information during simulation and for aiding the network debugging. Indeed, with a good display of run-time system behaviour, one can quickly determine if a network is working properly and if not where the problem lies. Furthermore NNPSs should allow the user to focus on a certain part of his model, that is to stop the execution and "look" at certain components of the network. These components can, for example be either a particular neuron or a particular layer etc. They should also allow the isolated execution of only a part of the network; which is useful for heterogeneous models and for general debugging. Much of the power of this type of interfaces lies in their dynamic properties, which are not shown in a static pictures. For example, with dynamical graphic display, the user can catch oscillations which can penalise a network. Thus good graphical facilities seem a condition "sine-qua-non" for a NNPS, given the useful information that these graphical displays can convey.

A large number of neural network models have been developed. Generality requires the NNPS to be able to implement any known neural network models. Indeed, generally one has to understand the existing models in order either to improve them or to present a new type of model. Moreover, a NNPS has to give the user a reusability of previous models. Thus, the need for a library is demonstrated. This library should supply the user with a complete set of neural network models, each of those is equivalent to a parameterised module and the user can use any particular instance of a model (either as a

complete model or as a part of a more complex model) by providing the modules with parameters.

Current research in programming systems is concentrating more and more in the use of graphical aids in these environments. Also efforts are being made in producing a single programming system that can provide most of the useful facilities which are available in the existing programming systems together with appropriate tools for developing parallel programs, especially tools designed to deal with network programming. This thesis is one such attempt at providing a neural network programming system which tries to address a few of these problems, namely, portability and programmability.

Chapter 3

This chapter presents the proposed neural network programming system, NPS, and then goes on to present the subset which was implemented for the purpose of this thesis. It also includes the descriptions of each component of the system and discusses the design motivations.

3. The Neural Network Programming System, NPS

The neural network programming system, NPS, was primarily designed to support **portability** and **programmability** [Bava90a, Bava90b]. That is - a system which can take algorithms specified in a range of high level languages as input and map it on a range of hardware. The NPS as shown in Figure 10 (which is same as the Figure 6 seen earlier in chapter 1), consists of:

- a number of high level languages for specifying neural network algorithms and controlling (monitoring) the execution of the network;
- a graphic monitor for graphically representing and interacting with the network;
- an intermediate level neural network implementation language system that can represent the graphical and high level language specifications of the various neural network models;
- a library of popular algorithms for the user to run standard models by supplying parameters;
- a set of utilities for assisting the user in the programming tasks;
- a wide range of hardware.

In this system, neural network models specified either in one of the high level languages or in graphical form is translated into an intermediate language(NIL). The intermediate language compiler translates this into appropriate target machine code and passes it on to an appropriate mapping system for mapping it on to a specific hardware for execution. The monitor and the control part of the high level programs can be executed in an interactive mode in conjunction with the graphics system to display and modify the network at run time. In this case each command is translated into its intermediate counter part and communicated to the hardware through the appropriate communication module. This run time translation is performed by a sub system of the

respective language and graphics systems. Apart from this the user can execute any of the available models in the algorithms library by supplying the necessary parameters.

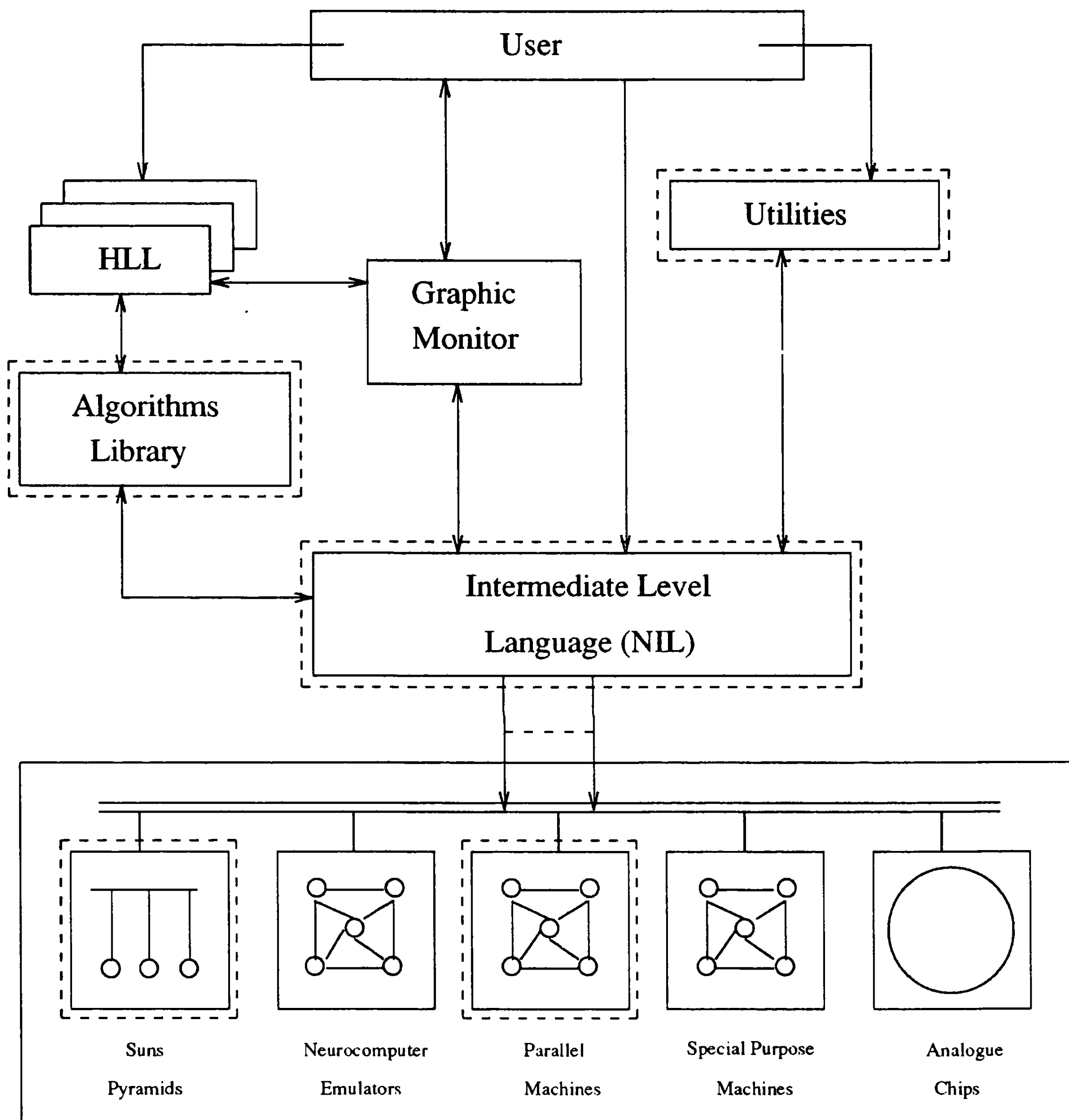

Figure 10. Neural Network Programming System, NPS

The high level languages (HLL): A developer needs the flexibility to specify different topology, activation functions, and learning schemes for building a variety of models and applications. In order to have such a flexibility, at specification level one needs a high level language which is highly expressive in terms of modelling and application construction. This language does not have to be efficient as long as it is powerful and flexible in its descriptive role, since the efficiency factor can be dealt by the intermediate level language. Since different high level languages offer different facilities, a good

programming system should have more than one high level language for the user to choose from so that a suitable language can be chosen according to the special needs of a particular application. This will no doubt enhance the programmability of the system.

Graphic Monitor: When it comes to interacting with the network, the user prefers a method that would present the information in a form which is easily understood. It is quite difficult to observe all the processes and make alteration to individual components in a massively parallel and dynamic system like a neural network. This can only be possible, if an overall picture of the system is available and sub components can be accessed. Displaying the network using graphics and indirectly accessing sub components via graphic displays for making alterations has proved to be a useful technique in network programming and debugging. With a good display of the run-time system behaviour, one can quickly determine if a network is working properly and if not where problem lies. Furthermore a good Neural Network Programming System should allow the user to focus on a certain part of the network and stop execution and examine it. These components can be a neuron, a link, a layer or a sub network. Thus the provision of good graphical facilities is an important requirement for a neural network programming system.

The Intermediate Level Language: First of all let us answer the question "Why do we need an intermediate language system ?". The answer to this question is that we want a common representation of the models we are trying to implement so that we can achieve portability. Generally, an intermediate language can be used to achieve three different kinds of mapping.

One to many mapping - In this mapping, a single high level language is mapped on to a range of machines.

Many to one mapping - In this case, a number of high level languages are mapped on to a single machine.

Many to many mapping - In this case, number of high level languages are mapped on to a range of hardware.

These three systems require different design approaches to be efficient. The first form, one to many mapping, requires an intermediate language that can both capture the essential properties of a particular high level language and represent the algorithm expressed in this language as a simple virtual machine that can be easily mapped on to any hardware. The second form, many to one mapping, has to capture the essential properties of all the high level languages and be capable of exploiting the unique characteristics of the particular hardware on to which it is trying to map these high level

language programs. This can be achieved by having a low level language which is closer to the assembly language of that machine but slightly at a higher level so that the essential features of most if not all the high level languages can be captured. On the other hand the third form, many to many mapping, is more complex. It must combine the properties of the previous two forms of representations to be a good system. That is, the intermediate language should not only has to capture the essential properties of a broad spectrum of high level languages but also should have the ability to exploit the desirable features of a range of hardware through a common scheme of representation. A possible solution to this is to have an intermediate language that represents the algorithms specified in these high level languages in the form of virtual machine which is simple and very general. This demands a high level description similar to "C" language or Occam in order to be of any practical use. The major problem with this approach is that it does not lend itself to producing optimised code for all the target machines.

These three forms of mappings are only a rough generalisation and further classification at specific levels can also be possible. One such case is the problem at hand, which is mapping a particular class of algorithms (models) specified in a range of high level languages on to a range of hardware. That is, we want to map neural network models specified in a number of high level languages on to a range of hardware. Here we feel the problem is slightly simplified because of the existence of a set of basic features which is inherent in all the neural network models. For example, a neural network model must consist of a description of its network topology, descriptions of the different types of nodes in the network, the learning and recall procedures, and the training and test data. Using these basic features as the back bone of the language, one can design an intermediate level language which is simple and general enough to represent these models specified in various high level languages in the form of a simple virtual machine so that it can be mapped on to a range of hardware. In here we see the NIL language as a set of notations for representing such a virtual machine rather than a language for specifying network models. This is substantiated by the fact that we are able to generate "C" code from the NIL translator which is similar to a NIL program. This is the approach the author has taken to tackle the problem of portability and programmability in this piece of research.

So, what we want is a neural network programming system which is independent of the execution hardware to take advantage of the advances in technology and one that is capable of accommodating a wide range of neural network models, from the point of view of the design, execution and interaction.

Combining these two basic requirements with the need for efficiency creates the need for an intermediate language which is model independent and capable of representing the high level descriptions in a simple and efficient form (*a virtual machine*). The reason for requiring a simple form is because a simple form of network representation is considered to be easier to map on to different hardware architectures. Other important characteristics needed in an intermediate language in a neural network programming system is that it should provide facilities for observing and modifying the network during run time. This particular characteristic is in a way an important one to network programming as a whole in the same way it is important in an Object Oriented Language System. Because it is the one that makes it possible for experimenting and debugging since there are no other known tools for debugging a massively parallel system.

The other important issue as far as a network language at an intermediate level is concerned is "whether one should have primitives to specify parallelism explicitly or the language should implicitly express it without the use of such primitives?". We believe a neural network language (especially at an intermediate level) must be implicit in its expression of parallelism like object-oriented languages. The main reason behind this belief is that a network algorithm is generally considered to be inherently parallel. If this assumption is adopted then the designer of the language should aim to design the language in such a way that it expresses parallelism in a natural way.

Algorithms Library: A programming system for neural network should provide facilities for building applications and testing the suitability of an existing model to solve a particular problem in hand. One way of aiding the user in this task is to provide a library of models and functions which can be executed by supplying the necessary parameters. This approach also promotes reusability. This library should contain a complete set of existing neural network models and associated functions. These are equivalent to a set of parameterised modules and the user can use any particular instance of them (either as a complete model or as a part of a more complex model) by selecting the appropriate module and suplying the parameters.

Utilities: A good programming system should provide a number of tools for assisting the programmer in the development of software. These may include a good editor (in our case a context editor would be of great help), a screen dump program, a file maintenance system appropriate for the particular type of software being developed, a program for saving partially run programs etc.

Mapper: The software that maps an executable network on to a particular piece of hardware must be efficient. This implies that it must be a dedicated system to exploit the advantages offered by that particular hardware. This leads to a decision to have dedicated mapping software for each target machine. This approach may be considered inefficient for building software systems but it has the advantage of producing a run time system which is efficient. Also depending on the architecture, physical placement of processes can be arranged in such a way as to reduce the possibility of deadlocks due to message passing.

3.1 The Implemented System

Since the main aims of this thesis is to investigate portability of neural network models over a range of hardware and programmability (at a low level) of a range of models, only the components marked by the broken lines in Figure 10 were implemented. The main reason for deciding on this particular approach was to demonstrate the capabilities of the intermediate level language and to show that the basic concepts behind the proposed solution is a practical possibility. In this system (see Figure 11) the input commands are interpreted by a command interpreter and appropriate functions or sub systems are called to perform the required task. The user may issue commands for compiling and executing a NIL program, running an algorithm from the library, save a partially run network using the utility or abort a running network etc. This implemented system as shown in Figure 11 consists of:

1. the intermediate network implementation language (NIL) system;
2. an algorithms library;
3. a utility for saving partially trained network for further training and recall;
4. a library of general purpose functions specified in NIL for implementing some well known node functions;
5. a main module comprising a command interpreter and a set of mappers;
6. a neurocomputer architecture simulator based on a primitive processing element.

NIL System - This sub system consists of a NIL compiler and a translator, and a library of useful function definitions for nodes. NIL programs can be either translated into the virtual machine (C-Machine) based on "C" language and then compiled into object code using a standard "C" compiler or compiled into target machine code for UCL

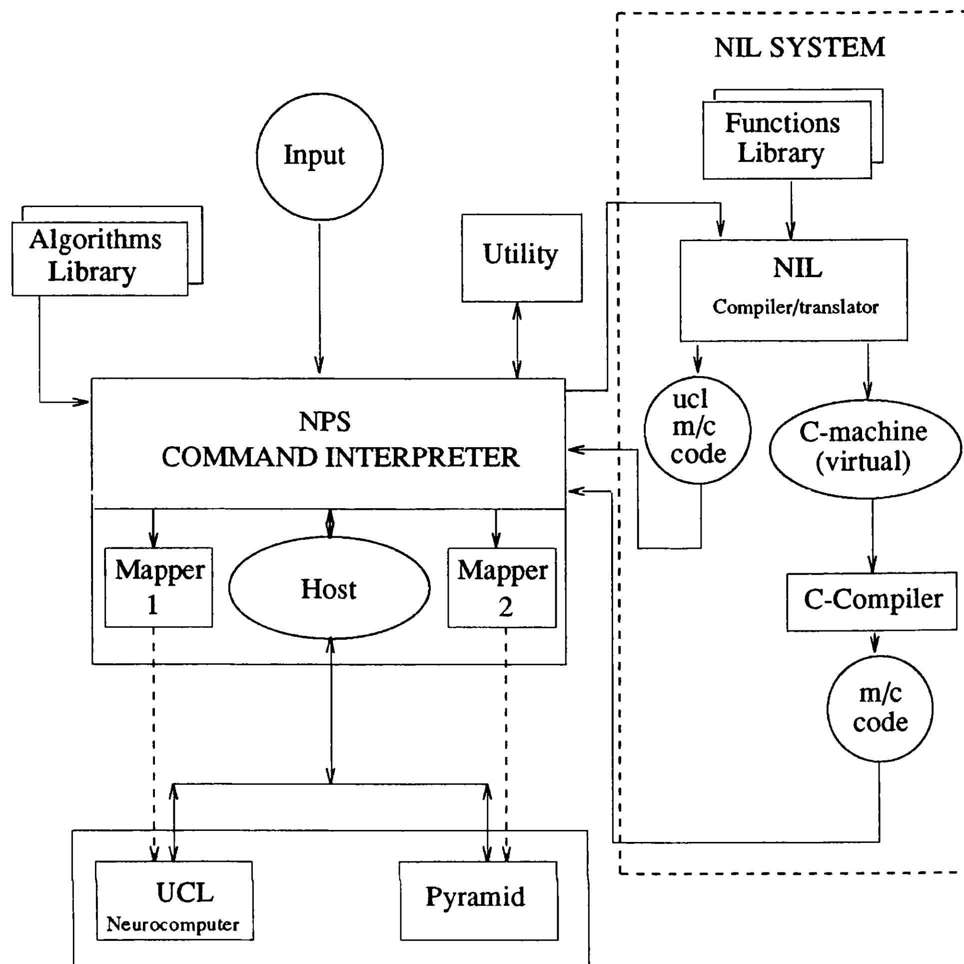

Figure 11. The Implemented System

Neurocomputer Architecture simulator. The command interpreter takes these machine codes and maps them on the appropriate hardware for running. In the case of the virtual machine, it is loaded on a Unix machine (Pyramid/Sun). Since the C-Machine is virtually similar to the NIL program in terms of its constructs, we envisage no difficulty in generating target machine code for the UCL Neurocomputer architecture simulator from the C-machine. The current system uses a NIL compiler (which was originally implemented for mapping NIL and testing the UCL neurocomputer architecture) to generate code for UCL neurocomputer simulator, and a separate translator for generating the virtual machine based on the "C" language. It must also be pointed out that NIL can be implemented in other languages and may be considered as a notation for representing neural network models rather than a language. This is found to be true in the case of the C-machine and during our experience with the hand translation of NIL into OCCAM.

Algorithms Library - The algorithm library consists of a set of popular models such as Hopfield model, Back-Propagation model, Boltzmann machine, etc. These can be

executed by providing the appropriate parameters. Most algorithms in this library require the user to specify the number of nodes in each layer, set of input patterns, a set of output patterns, and tolerance value.

Utility - This is a single function utility package containing software for saving a partially run network for further training at a later time. This is invoked by a "save" command which is available both at the command level and at the manipulation level of the intermediate language.

Library of functions - Since the functions which define the nodes in each layer of a particular neural network algorithm can be specified as a general purpose function in NIL, we found it useful to have a library of node functions readily available for use in a NIL program. This is similar to the "C" language library, where the programmer is provided with a set of functions for performing some common tasks.

Command interpreter - The command interpreter in the implemented system is like a Unix shell. It accepts commands from the user and interprets it and calls the appropriate function/sub system. This means that it shares some of its commands with the NIL system. That is - some of the commands can be used within a NIL program.

There are all together seven commands available to user at this level. They are

1. **load** *-m file_name* - for loading a partially run network for execution on a particular hardware by specifying the hardware option in the parameter "m".

2. **save** *file_name* - to save a partially run program for running later. This command can also be used in the manipulation part of a NIL program.

3. **stop** - to stop a running network. Again this command can be used in the manipulation part of a NIL program.

4. **go** - to run a loaded network.

5. **run** *-m file_name* - to compile, load and run a source program file with options to execute in a particular machine.

6. **exec** *-m model_name* - to execute one of the standard models available in the model library. When executed, this command will initiate a series of questions regarding the size of each layer, tolerance value, and input and output patterns etc. The user responds by typing the required data. Once

all the information is available, the system builds the appropriate network and proceeds with the compilation and subsequent execution process.

7. **abort** - to abandon the execution of the network. This command is also available in the NIL language.

Ucl Neurocomputer - At University College London we have designed and are currently implementing in CMOS, a primitive processing element for building a parallel MIMD neurocomputer, configured from an array of these elements [Pach88, Pach91]. The main goal of the neurocomputer is to support a range of connectionist algorithms spanning both neural network models and semantic network models.

The overall structure of the system, as shown by Figure 12, consists of a set of arrays of processing elements(PE) connected to a host computer that controls the activities of the network.

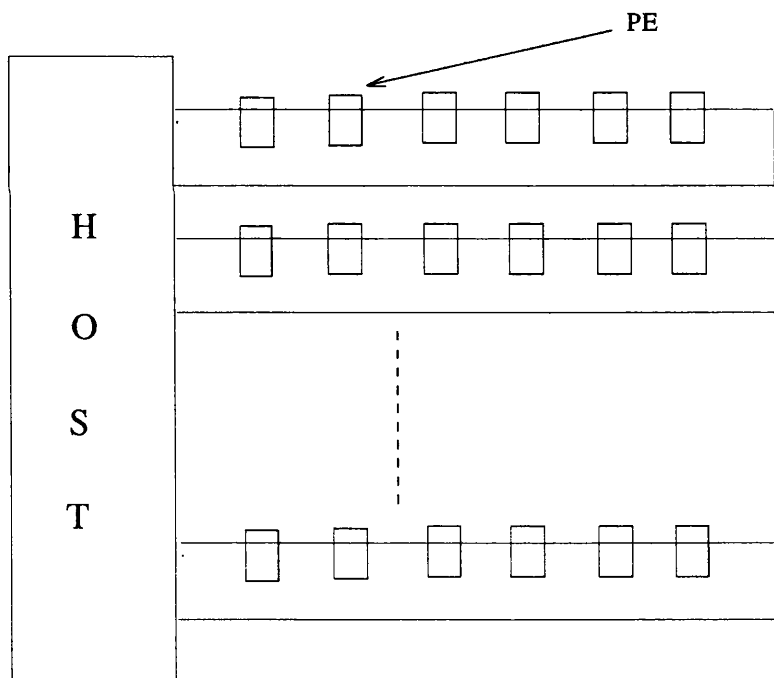

Figure 12. UCL Neurocomputer

A PE in an array is linked by a bi-directional, point-to-point connection to its two neighbours and communicates by sending message packets (see Figure 13). Each message consists of three fields: a destination address (ie-the identity of the destination PE.), a logical input channel number, and a data value. When a PE receives a packet, it compares the address field with its own address and if it matches then the processor is interrupted and the packet is passed to the application process. If the address is different then it is passed to its other neighbour. In this system both the host and the individual PE

can broadcast messages by placing a special destination address and input number in their packets. In the case of a broadcast packet, each PE takes a copy of the packet and passes the original to its other neighbour.

Each PE as shown in Figure 13, consists of three units: a communication unit, a processor and a local memory. The communication unit has two I/O **Buffers** and a **name** register to hold its address.

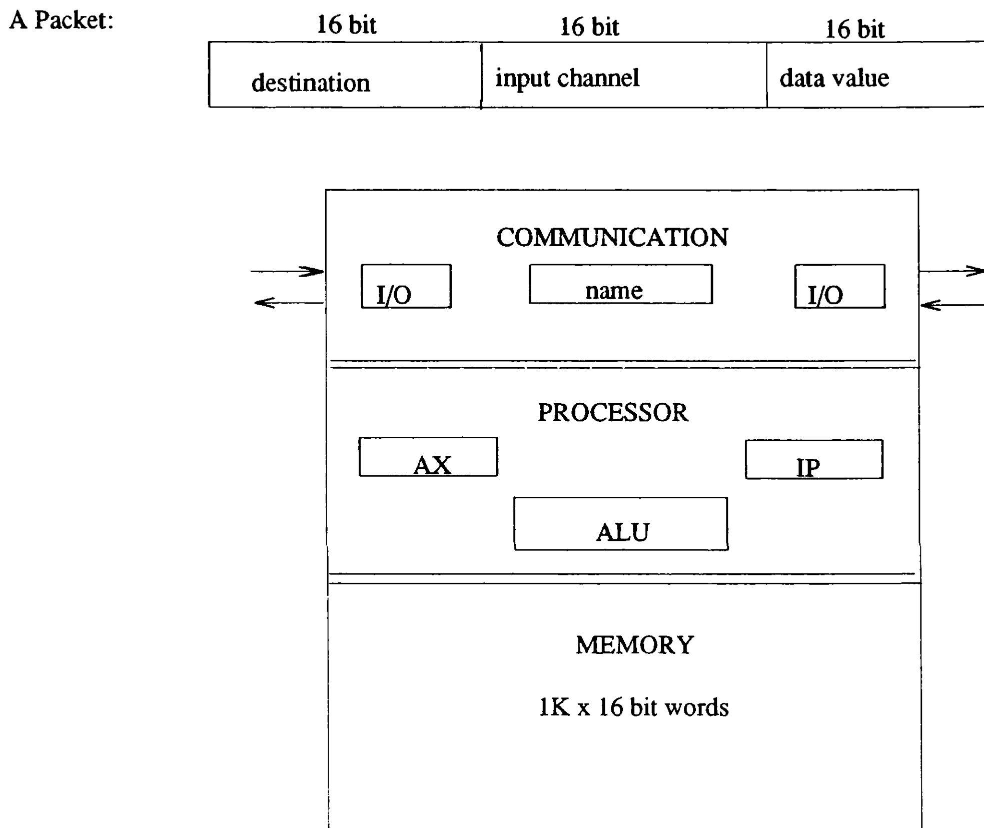

Figure 13. UCL Neuro-Chip

The processor consists of a primitive ALU, supporting a reduced instruction set of 16 instructions. This instruction set enables the processor to perform all the necessary functions required by a network program by providing instructions such as *load, store, add, sub, mult, and, xor*, etc. There is only one working register, the accumulator AX, and a very few memory mapped registers. All the data addresses and instructions are 16-bits long. Each instruction consists of a 4-bit opcode and 12-bit data address. The size of the local memory can be up to 1K X 16-bit words long which is found to be adequate enough to support a range of neural network node functions.

The neurocomputer is configured by initialising each PE with a unique address and then loading them with appropriate codes. These codes can be either identical or

different for each PE depending on the overall task. Each PE also has a local operating system to control and manage its activities. Currently, a simulation of the architecture at the register level is running on a UNIX machine and a CMOS implementation is under way.

Chapter 4

This chapter describes the intermediate level language, NIL for mapping and manipulating neural networks on a range of hardware. The language incorporates the concept of guarded process and combines this with the philosophies behind network languages to build networks and manipulate them.

4. The Network Implementation Language, NIL

4.1 Motivations and Requirements

The main motivations behind the design of NIL is to produce a low level language for specifying neural network algorithms with the following features :

1. **machine independence** - The language must be independent of any hardware, especially neurocomputers so that it can be used as an intermediate language for porting neural network models specified in a range of high level languages across a range of hardware.
2. **neural network independence** - The language must be capable of representing a wide range of neural network models and applications so that it can be used as a programming language, especially at a low level during the development phase as well as during runtime.
3. **simple and general** - It must be simple enough to be an intermediate level language as well as general enough to be translated into high level languages like Concurrent Pascal, "C", etc and low level languages like Occam and assemblers.

Using these motivations as a basis, a list of desirable properties for an intermediate level, machine independent neural network language is arrived at [Ange88, Bahr87, Chol88, Guts88, Kohl88, MayD87]. These are:

- *Small and specialised language:*
  The language should be dedicated to neural network algorithms. This language should offer specialised features on top of a more classical language, providing the user with useful primitives and tools to deal with the unique aspect of the problem domain. But these must be kept to a minimum in order to keep the language as small as possible, thus avoiding costly compilation.

- *Simplicity:*

  The language should be readable. Generally, specialised languages are considered to be less simple to understand than more general purpose ones, because they are used by specialists, and less attention is given to their readability.

- *Generality:*

  Any connectionist algorithm should be programmable in the language. The expressive power of the language should not be a limiting factor. This is a very ambitious requirement because no one knows what the models of tomorrow will require. For example, some of the recent research papers suggests the need for

  - Primitives for the dynamic evolution of the network structure - Some models allow the creation and deletion of links and nodes, and primitives for doing so should be provided in the language.

  - Expression of randomness - Some models make use of random connection patterns, and/or probabilistic cell activation. Primitives for implementing these functions should be provided in a language.

- *Parallelism:*

  An intermediate level language for neural network must have parallelism built into it in a natural way, that is, it should reflect the parallel nature of the algorithm in the way it represents it without explicitly stating it. It must also encourage the programmer to isolate components in the algorithm which can be processed in parallel.

- *Reusability:*

  The language should incorporate a mechanism for making use of already written and validated network components. One possible approach is to create a library of common components and models, written in the most general and parameterised form so that, it can be used by giving values to parameters.

- *Support for graphical environment*

  A network programming language benefits from having the facilities to display its structure as a graph wholly and partly during run time. Graphic facility should also have the capability to express the behavioural characteristics during run time. As there is no easier way to monitor the progress and debug the network other than graphically displaying the network and making alteration through the graph, it is important that the language should provide a mechanism for graphic interface.

This brings us to the question "why not use an existing language like "Occam" as an intermediate language for implementing neural networks?". The reasons for not following this approach are:

1. To manipulate a neural network, we need special commands, especially when dealing with dynamic networks which require facilities for creating and deleting links and nodes. If we are to use Occam, we need to synthesise new functions to deal with these tasks. This will require complicated coding and will only result in an inefficient system.

2. Additions needed to make Occam a neural network intermediate language will make the language larger and less simple.

3. Supporting a graphical environment using an Occam based language requires unacceptable level of overheads. This is mainly due to the difficulties in extracting the topological informations and the informations contained in each nodes. This will make the whole system unacceptably slow.

4. So far, no one has shown that Occam can be efficiently mapped on other parallel hardware (i.e - other than the transputer).

These considerations led to the design and implementation of NIL. NIL is intended to be a machine independent low level language for neural network specification. NIL has

1. a basic part that builds the network representation of the algorithm and specifies the functionality of the nodes in the network.

2. a manipulation part that provides the capabilities for controlling and modifying the network.

In the network specification part, nodes are connected using statements which implement a series of mapping of inputs to outputs. The statements which describe the functional behaviour of a node consists of control flow statements which are found in popular procedural languages.

Having stated the motivations and requirements of NIL, section 4.2 lists all the keywords used in NIL. Section 4.3 explains the use of brackets and separators in NIL. This is followed by an explanation of the meanings and the use of constant values. Section 4.5 describe the syntactic and typing convention used in formally describing the language. Section 4.6 gives a detailed description of NIL. Finally, section 4.7 describes

the semantic properties of the language.

4.2 Keywords

The following list(see Table 5) of tokens represent keywords and hence may not be used as identifiers.

add	all	and	begin
construct	delete	do	end
exp	fi	fun	get
getwt	go	if	input
int	ival	join	ldconst
load	nde	not	od
or	output	read	readst
real	rep	rmv	rnd
run_net	save	skip	stop

TABLE 5. Keywords

4.3 Brackets and Separators

The following symbols are used for grouping or separating objects:- (), [], { }, ->, =>, semicolon ';', colon ':' and comma ','. In general, round brackets are used for grouping and precedence, square brackets are used to denote array elements, and curly brackets are used to enclose processes and blocks of codes. The arrow is also used as a mapping operator in the link statement. The imply symbol '=>' is used to separate the input condition from the associated block of statements in a guarded process. The semicolon is generally used as a terminator of assignment statements. The colon is used as a special separator to distinguish weight vectors from ordinary input vectors in the parameter lists and separating subscripts in an array. Finally, the comma is used as a general purpose separator for syntactic convenience (e.g. for separating parameters etc.).

4.4 Constant-Valued Tokens

Certain tokens have priori values associated with them. These are characters, strings, integers, and real numbers. It should be noted that maximum and minimum values for numbers will be machine dependent. A character is enclosed in single quotes (e.g. 'c').

Integer numbers are represented using decimal numbers(0-9). The real numbers are represented using decimal point notation. A string is any sequence of characters enclosed within double quotes(e.g. "string").

4.5 Syntax and Typing

The syntactic elements in NIL may be subdivided into five groups: link statements, function definitions, guarded processes, expressions, and statements(see APPENDIX A - for further details). Throughout this document a Backus-Naur Form (BNF) notation will be used for describing NIL. The following conventions are adopted:

italics = nonterminal symbol
lower_case = terminal symbol
{ symbol } = optional symbol
{ symbol }* = repetition of zero or more symbols
{ symbol }+ = repetition of one or more symbols

Alternative categories of a syntax rule are separated by a vertical bar.

4.6 A Network Implementation Language - NIL

NIL is a machine independent, low level neural network programming language for mapping and manipulating neural network algorithms on a range of parallel and non-parallel hardware.

NIL consists of two sub-languages :

- **a network specification sub-language** which specifies the connections between nodes and the functions performed by the nodes in the network.
- **a manipulation sub-language** which allows the user to observe the behaviour and modify the network during run time.

a program written in this language has the following syntax.

```
begin
network_specification_part
begin
manipulation_part
end
end
```

4.6.1 The Network Specification Sub-Language

The network specification sub-language consists of:

- **link statements** for linking nodes in the network to specify the desired topology;
- **function definitions** for specifying the computational behaviour of the nodes in the network.

4.6.1.1 Link Statements

In NIL the network is built using three types of link statements. These statements implement a series of mappings using appropriate functions with specified lists of input and output parameters. They are

1. a **link** statement, which is the most primitive statement that specifies the mapping of a set of input to a set of output using a particular function which represents the computational behaviour of a node.
2. a **rep** statement, which replicates nodes and their connections both in a series and in parallel.
3. a **construct** statement, which builds a complete network structure with regular connections.

link statement - a link statement has the following syntax.

name(*input_list* {: *wt_list*}) -> (*output_list*)

The *name* refers to the specific function performed by that node. The *input_lists* and *output_lists* represent the set of input and output list associated with that node. The *wt_list* represents the list of initial values such as initial weights, status status values and any other local data for that node. These data are private properties of that node and are not accessesd by any other nodes in the system except the host for controlling the network. To illustrate the use of the link statement, consider three nodes A, B and C linked in a network as shown in Figure 14.

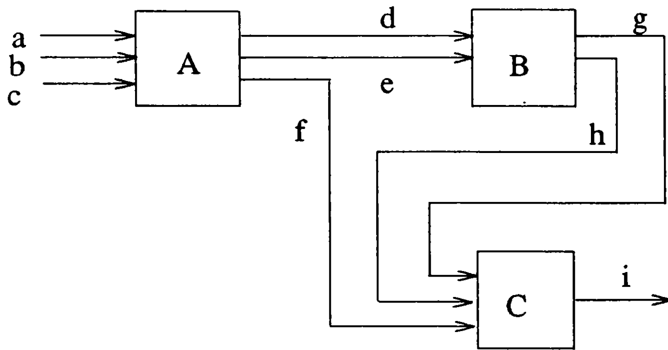

Figure 14. A Simple Network

Where A and B perform the function defined by *f1* and C performs function *f2* to transform their inputs into outputs. The linking of these three nodes are achieved by the following link statements.

f1([a, b, c] :[1, 3, 5]) -> ([d, e, f])

f1([d, e] : [7, 9]) -> ([g, h])

f2([f, g, h]) -> ([i])

When this network is executed, appropriate connections are established, weights vectors are initialised and the system will wait until input channels a, b, and c are loaded with appropriate values. This will initiate the firing of node A which in turn will output values d, e, and f and provide input for other nodes.

rep statement - There are two forms of **rep** statements and these are similar to those found in Occam [MayD87]. A **rep** statement of the form

rep[3] ff([X[I], Y[I]]) -> ([X[I+1], Y[I+1]])

would replicate the function *ff* three times in a series as shown in Figure 15.

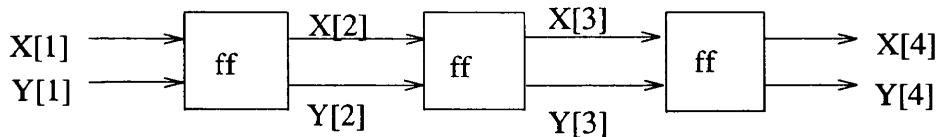

Figure 15. Sequential Replication

A **rep** statement of the following form

rep[2] ff([X[I], Y[I]]) -> ([A[I], B[I]])

would replicate the node and its I/Os in parallel as shown in Figure 16.

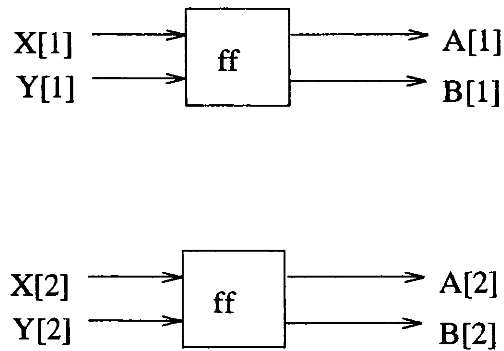

Figure 16. Parallel Replication

construct statement - This allows the programmer to build a complete network structure with regular connections. For example, the following **construct** statement

construct ([i = 2] in(iv[i], [sig[1]]) -> ([ou[i]])

:[p = 1] hi(ou[i], [erbk[1]]) -> (out[p], sig[p])

:[k = 1] op(ou[i], out[p], [eop[1]], [rcl[1]]) -> (erbk[k], result[k]))

is equivalent to the following set of statements.

rep[2] in(iv[i], sig[1]) -> (ou[i])
hi([ou[1],ou[2]], [erbk[1]]) -> ([out[1]], [sig[1]])
op([ou[1], ou[2]], [out[1]], [eop[1]], [rcl[1]]) -> ([erbk[1]], [result[1]])

These statements implements the network shown in Figure 17.
The construct statement has the following syntax.

construct(

[*rep_par*] *name*(*input_par*) -> (*output_par*)

{: [*rep_par*] name(*input_par*) -> (*output_par*)}+

)

where

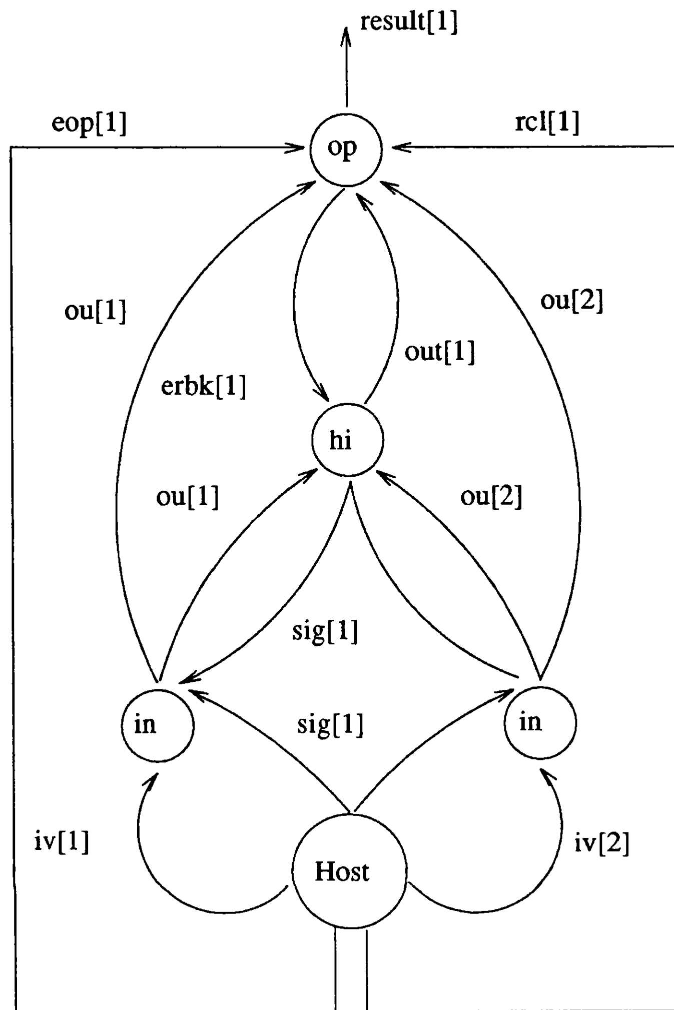

Figure 17. A Regular Network

rep_par ::= *index_var* = number of replications

input_par ::= *c_i_list* {: *c_wt_list* }

output_par ::= *c_o_list*

c_i_list ::= *in_var_vector* | *array_range*

c_o_list ::= *var_name*[*index_var*]

{ ,*var_name*[*index_var*] }*

in_var_vector ::= *var_name*[*subscript_variable*]

array_range ::= *var_name*[*index_range*]

index_range ::= *range* {, *range*}*

range ::= *sub_range* | *set_range*

sub_range ::= *integer* .. *integer*

set_range ::= *integer* {, *integer* }*

c_wt_list ::= see later

This can produce statements such as

```
construct( [i = 5] f1(a[i], [b[1,2, 4..7, 9], d[1..4]]) -> (c[i])
        : [j = 14] f2(c[i]) -> (b[j])
        : [k = 12] f3(c[1,4,5], b[j]) -> (d[k]) )
```

The rules governing the synthesis of a construct statement is as follows

1. The value of the *index_var* indicates the number of repetition of the associated link statement.

2. The *subscript_variable* in the *in_var_vector* must be either the current or a previous *index_var*. If it is an *index_var* of a previous link component then all the elements specified by that range are repeated for each replication. If only a selection is required then it must be specified as an *array_range*.

3. Any *index_var* that has not yet appeared must not be used.

4. Links from nodes that are not part of the construct statement can be included in the *c_i_list*.

5. Any feed back loop must be specified using an *array_range*.

Weights can be introduced into these statements using a standard random function as follows.

rep[3] ff(x[i], y[i] : 2*rnd[0-1]) -> (a[i], b[i])

This will generate two random values in the range of 0 to 1 for each link statement. Fixed array of weights also can be used

```
----- ... :[2.0,3.0, 4.0]....
              or
          :[i, i+1, i-1]
                    where "i" is known index_var.
```

Only simple expressions involving +, - and * can be used in this way.

4.6.1.2 Definition of functions

Once the topology of the network is specified, the computational behaviour of each node(or a group of nodes in the case of a number of nodes performing the same type of computation) is described in the form of function definitions. A function that represents a node in this language is based on a generalised model of a biological neuron.

The Computational Model - A computational model of a function that represents a neuron should reflect the overall properties of a neuron. A popular view is that a neuron only starts firing when the inputs that arrive at the dendrites satisfy certain conditions and subsequently produces a new set of outputs which may or may not be different from the previous set of outputs. This is then followed by a change of state and waiting until the arrival of a new set of inputs that satisfy the input condition for firing and outputting new values again. This process is repeated until the network becomes stabilized. This generalisation led to the computational model shown in Figure 18.

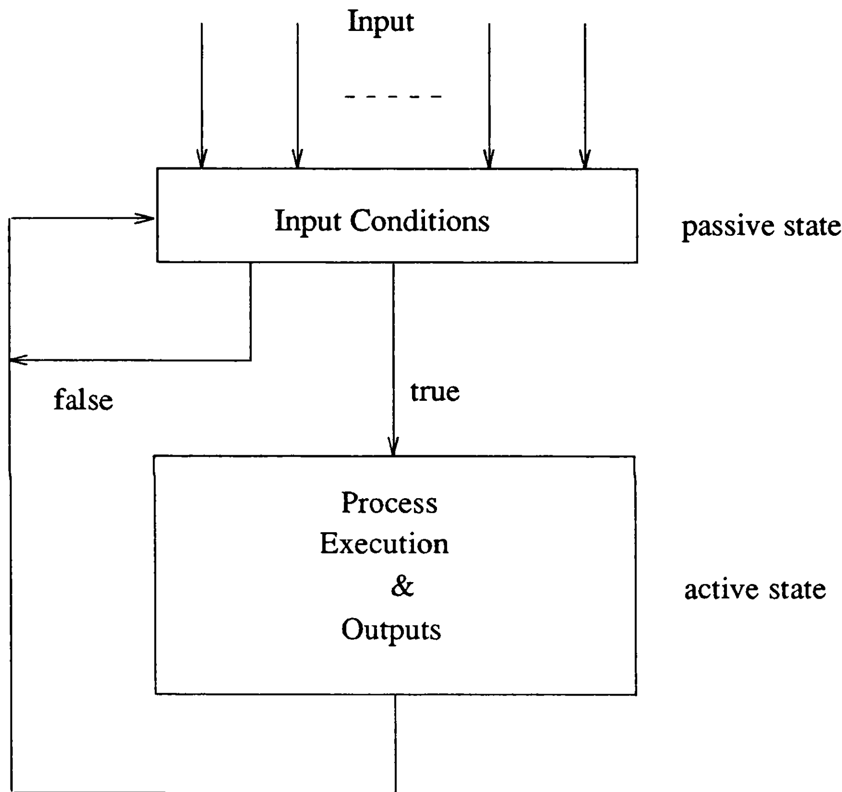

Figure 18. The Computational Model

We believe this model is flexible and general enough to accommodate a wide range of artificial neurons. This model also makes it possible to define functions in a general way so that it can be used again. A function which implements this model has the following form.

```
fun name (input_vectors {ival:wt_vectors } )

                    -> (output_vectors)

        "{"

           body

        "}"
```

A function definition, although very much similar to a function definition in Pascal, differs in many respects.

1- A function in NIL is considered to be a mapping function (i.e. - it maps inputs onto outputs). This means that it can act upon any set that belongs to the input space. This makes the function more general purpose and can be re-used again with a different set of I/O parameters.

2- The parameter list consists of three types of parameters, they are input, output and initialisation vectors. The initialisation vectors are sets of constant values which are used to initialise variables within the functions at the start of the system and they represent the state of computation in a node at any given time. These are not carried by any links and are planted into the node during compilation and are the sole property of that node.

3- All the parameter vectors denote a group(array) of variables.

4- Each element of the input and output vectors represent a set of *virtual* input and output channels through which the data is sent and received by nodes.

5- The length of each I/O vector parameter is only determined by the link statements which use this function to build the neural network(ie- the individual components of the parameter lists are only specified at the linking stage of the network).

6- The dimension component of the input, output and weight vectors are used to hold the lengths of these vectors when a link statement is activated(see example for explanation).

7- The body of the function consists of one or more *GUARDED PROCESSES* with boolean input conditions as its head. A guarded process has the

following construct.

input_condition => "{" process_description "}"

8- A guarded process becomes eligible for execution when the corresponding input condition becomes true.

9- Only one of the eligible guarded processes is executed during a *node cycle* where a node cycle consists of the execution of a guarded process and dispatching the output.

A general structure of a function definition and a link statement is as follows.

```
sum([a, b, c] : [2, 5]) -> ([k, l, m])

fun sum ( x[n1] ival: w[n2]) -> (y[n1])
"{"

   input_condition_1 =>"{" process_1 "}"

   input_condition_2 =>"{" process_2 "}"

      :

   input_condition_n =>"{" process_n "}"

"}"
```

In the above definition *sum* is defined to be a function that takes an input vector x and a weight vector w each with n1 and n2 elements respectively and produces an output vector y. When sum is called n1 takes a value 3 and n2 a value 2 along with initialisation x[1] <- a, x[2] <- b, x[3] <- c, and w[1] <- 2, w[2] <- 5 and produces outputs y[1], y[2], y[3] which are passed to other nodes in the network in k, l and m. This mechanism makes it possible for the user to build general purpose functions that can be used by a number of nodes wishing to perform the same type of computation but with different numbers of input elements, which is an essential feature of a neural network.

Guarded Process

A guarded process becomes eligible for execution when the corresponding input condition becomes true. The guarded process in this language is similar to the guarded process described by Hoare [Hoar85] except that it provides a compact and elegant notation for representing it.

Input Condition - The input condition is represented by an array with an **index-range**. If new input values are present in all the elements of the input array as specified by the index range then a true value is realised and the corresponding process is executed. This input condition can be expressed as an "ALT" statement in Occam [MayD87]. An input condition has the following syntax.

*vector_1[index_range] {,vector_i[index_range] }**

The above syntax should produce a family of input condition that would cater for every eventuality as illustrated below.

x[3]	one specific channel input.
x[1..n], y[1..m]	all input channels from x and y.
x[k..l]	sub range.
x[k1..m1,k2..m2]	set of sub ranges.
x[1,12], y[2,5,9]	subset of x and y.

Statements - The statements available for describing a process consists of "if", "do", "assignment" and "skip".

Output Statements - There is no special statement to perform output in this sub-language. This is achieved by assigning values to the elements of the output vectors.

The following example describes a general function to perform summation and thresholding. This function can be used anywhere in a network whenever a summation and thresholding unit is needed.

```
fun lay2(x[n]) -> (y[i])
int     i, n, j;
real    x[10], y[10], netj;
{
 x[1..n] => { j := 1;
            netj := 0.0;
            do (j<=n) -> netj := netj + x[j];
                   j := j + 1;
                   od
             if (netj>=0) -> y[1] := 1.0;
             ->-> y[1] := 0.0;
             fi
              }
}
```

4.6.1.3 Output From Compilation A compilation of a program consisting of definitions and link statements would output one of the following two outputs.

1- A list of errors and diagnostics if the compilation fails.

2- If the compilation is successful then it produces a list of labeled nodes, name of the functions performed, input list, weight values and an output list(see Appendix D).

The production of the second list gives all the relevant information so that the user is familiar with the network he/she has built and can refer to each node uniquely by the node label provided in the list during manipulation.

4.6.2 Manipulation Sub-Language

Once a network is built one would require a MANIPULATION language to retrieve and deduce information from the network and modify it. Commands given in this language can be expressed by the following syntax.

begin *control_comnds* **end**

control_comnds ::= { *control_statement* }*

This language should ideally have the following capabilities.

1- Stopping and starting the network.

2- Feeding and receiving data from individual nodes.

3- Looking at the current inputs and outputs of a node or nodes.

4- Inspection and modification of state variables of a node(s).

5- Creating and deleting links and nodes.

6- Saving and reloading an incomplete process(network).

7- Facilities for conditional and repeated execution of the above mentioned functions(ie-loops and conditions).

Instead of having very descriptive constructs for each tasks mentioned above, it was decided to have very simple commands to reduce the overheads in interpreting these commands during run time. This led to the inclusion of the following statements in the manipulation sub-language.

4.6.2.1 Reading Status and Links A single construct is used to read the status of nodes as well as the links.

readst *parameters*

parameters ::= **all** | *node_id* { , *node_id* }* ;

Where a *node_id* refers to a particular node and is referenced using the following convention.

nde[*unit_number*]

nde[2]

denotes the second unit in the network. The construct above should produce the following set of statements.

readst all

readst nde[2], nde[5];

The first statement reads the status of all the nodes in the network. The second statement reads the status of nodes 2 and 5. This reads the state variables(ie- elements of the weight vectors) and the values of the I/O channels.

When using the "readst" statement , all the specified state and I/O variable values are made available to the programmer. Access to these values are made by referencing them by their link I/O variable or referencing the I/O vector element of the particular node. For example - Vector element t[2] of the I/O vector for the node 5 is accessed by the following identifier.

nde[5].t[2]

4.6.2.2 Reading Inputs from Nodes To read any messages sent to host by nodes in the network, a separate read statement is provided. This is similar to the one provided in Occam [MayD87]. The reason for adopting this is to provide the host the ability to make use of the data as soon as it arrives and respond to it as well as synchronize the activity of the network. This is a very important statement in the language and helps to avoid overwriting inputs sent from the host and bring about an orderly computational system.

get *var_name1* = *linkname*
{ , *var_name2* = *linkname* }* ;

When this statement is encountered, the linkname is checked repeatedly until a new

datum is present in the named link variable and is placed in var_name1. If the weight/status variable of a particular node is to be read instead of the link variable then the following command is used.

getwt *var_name1 = Weightname*
{ , *var_name2 = weightname* }* ;

In this case the value of that variable is simply read into the named variable (i.e. var_name1, say).

4.6.2.3 Reading Data from a File To read input data from a file, the following statement is used.

read(*filename*, *data_type*) *data_list*

Where the file name is a string or a variable which contains the name of the source file. The data_type is an integer value which denotes the type of data to be read. This value range from 0 to 3, where 0 stands for integer, 1 for real, 2 for character, and 3 for strings. The data_list is a list of variables into which the data is to be placed. Some examples of this statement are as follows.

```
read(data.p, 1) x, y, z;
read(file1, 3) head_line;
```

4.6.2.4 Creation and Deletion of Links To delete and create links, the following construct is used.

operation link_name(join_par | *delete_par)*

operation ::= **delete** | **join**

join_par ::= *node_id.formal_input_vector_element*

delete_par ::= *,node_id* | *node_id*

This construct will enable the deletion and creation of links. For example -

delete *link_one*(,nde[2])

or

delete *link_two*(nde[4])

The first delete statement will delete the link at the destination node. The second statement will delete the link at the origin node.

join *link_one*(nde[2].x[3])

In this case output channel, link_one from a node will be linked to node 2 with the formal parameter name x[3]. This means that the node 2 must have an input vector called 'x' with two elements already in place.

4.6.2.5 Loading Initial Values on to the Input Links Loading new(initial) values on to those input links which require input from outside can be achieved by the following construct.

input *link_name = value*

{ , *linkname = value* }* ;

4.6.2.6 Outputting Results Outputting values to the standard output device is achieved by the following construct.

output *results*

results ::= op_list | read_status

op_list ::= element { *,element* }*;

element ::= strings or variables or values

Using the above syntax statements, like

output x,y,z;

output "The result is ", result;

output link_name1;

can be constructed. These three statements output values of the variables x, y, z, a string, result and the value of the link named "link_name1".

4.6.2.7 Feeding New Information To feed in new information and alter the behaviour of the network a "ldconst" statement is introduced. This statement enables us to introduce new weights to an existing node before or during run.

ldconst *node_id.state_variable* = value_1,
..,
node_id.state_variable = value_n;

The construct above produces the statements such as

ldconst nde[3].wt[5] = 5,
nde[3].zt[3] = index;

This statement changes the weights of an existing node. The changes made by control commands are permanent. This capability is of great value to a system of this nature. This will allow modifications based on the current state of the system to be introduced at run time and to perform experiments with the network.

4.6.2.8 Deleting and Creating Nodes Only node types which already exists in the network can be added. This is achieved by copying an existing node of the desired type and adjusting its i/o vector and parameters and loading into a free processor(virtual in the case of C implementation).

add *name*([a,b,c]:[1.0,2.0,3.0]) -> ([d,e])

To delete a node from the network the remove command is used as follows.

rmv *node_id*;

For example

rmv nde[5];

will remove node five from the network.

4.6.2.9 Saving and Reloading Network Commands for saving a partially trained network and reloading it for further training is (in fact) belong to the system. That is, it can be executed outside the scope of the manipulation program. But we decided to allow the manipulation program to be able execute it also. The main reason for this decision is to allow the network to run in batch mode without any user interaction. So to save and reload network programs we have the following commands.

save *file_name*;

for saving a partially run network and

load *-m file_name*;

for loading that network for running from where it was stopped. The user specifies the target hardware by replacing "m" with the appropriate integer value. Currently only one value is given(which is 1) as the target machine is Pyramid in the implemented programming system.

4.6.2.10 Executing the Network Initially the control is in the hands of the host. Once the necessary data is loaded on the links, and the weights are placed in each node, the command

run_net;

must be issued to run the network. This will execute each node in the system twice before returning the control to the host processor.

4.6.2.11 Stopping and Starting For stopping and restarting the network, the following commands are used.

stop

go

4.6.2.12 Loops and Conditions To be able to perform the above mentioned tasks in a loop or perform them conditionally requires constructs like the guarded "do", "if" and the "assignment" statements encountered in section 4.6.1 of this chapter. So it was decided to use them for this purpose also.

4.7 Semantic Properties

- only control flow execution takes place in a node.
- Deleting a link means loading a zero on it.
- Only nodes with existing descriptions can be added to an active network.
- stopping a running network implies waiting until the end of current process in each node and passing the control to an idle process for a fixed time before freezing it.
- all input/output and weight values are taken to be real values.

Chapter 5

This chapter describes the implementation of the NPS and the NIL compiler/translator. First it describes the main components of the translator that generates the virtual machine called the C-Machine which is based on "C" data structure that can be compiled by a C compiler and executed on a Unix machine. Then it describes the prototype compiler and the data structure originally developed for generating machine code for execution on the UCL neurocomputer simulator. Finally it describes the implementation of the NPS and its interfaces.

5. Implementation of NPS and NIL

As previously mentioned, only a subset of the NPS design was implemented. This subset consists of the following.

1. A NIL language translator and a compiler.
2. An algorithms library comprising a set of well known neural network algorithms which can be executed by supplying the necessary parameters.
3. A utility to save a partially run network for further training/recall at a later time.
4. A library of general purpose node functions which can be used in a NIL program in place of user written node functions.
5. A command interpreter that interprets the user commands and calls the necessary sub system to execute it.
6. A neurocomputer simulator (which was jointly implemented with another student) based on the design features described in chapter 3.

The major part of the implementation effort went into the implementation of the intermediate language. The first version of the NIL was implemented as a compiler that generated machine code for the UCL Neurocomputer simulator. A translator was implemented to generate a virtual machine based on "C" language which can be compiled using a "C" compiler for execution on a Pyramid technology machine. This translator is the one that is used to in the Network Programming System. The implementation of the translator is described first followed by a description of the implementation of the prototype compiler. In the description of the translator, prominence is given to the description of the virtual machine because of the important

role it plays in demonstrating that NIL is in fact a computational model for processing neural networks and can be implemented in any language like the *paralation model* [Sabo88] which tries to model parallelism in general.

5.1 Implementation of NIL

When NIL was designed, implementation efforts were mainly concentrated on generating code for the UCL neurocomputer architecture in order to test the suitability of the language for implementing neural network algorithms on a massively parallel machine. Having successfully demonstrated this, the next step was to implement it on a sequential machine. This led to the design of the virtual machine which is a "C" representation of the NIL. This approach is in a way similar to the *paralation model* [Sabo88] approach. That is, we have a very general and simple neural network implementation model in the form of NIL which can, not only be implemented in NIL specification language but is also implementable in other languages. This implementation language can be "C", Pascal, Lisp or Occam. To prove this very point "C" was chosen to represent this model. This also helped to demonstrate the fact that our intermediate level language is not only hardware independent but also language independent (ie- easy to translate into high level languages as well as into low level languages).

The subsequent sub-sections describe the NIL translator and the data structures used in generating the C-Machine followed by a description of the original prototype compiler for generating code for UCL Neurocomputer simulator and the data structure used for implementing a computational node in this system. Section 5.4 of this chapter describes the implementation of the Network Programming System.

5.2 The NIL Translator

The present NIL translator produces "C" code in the form of a virtual machine for execution on a Unix based sequential machine. The main reason for generating "C" code for execution on a sequential machine is two-fold:

1. Firstly to show that the intermediate language is target machine independent;
2. Secondly to demonstrate that the simple model represented by NIL can be implemented in other languages and mapped on a range of hardware whether it be a sequential one or a parallel one.

The translator is a two-pass translator and slightly different from a typical one for obvious reasons. During the first pass all the **construct** and **replicate** statements are

converted into their equivalent set of simple link statements and an intermediate (Expanded) source file is generated. This source file is used as input to the second pass for the final translation. During the first pass only the construct and replicate statements are checked for lexical and syntactical error before expanding these statements into a set of link statements. The rest of the program is ignored. During the second pass the expanded source is checked for lexical and syntactic error. While performing syntactic check, some semantic checks are also made before building the **virtual machine** (C-Machine) based on "C" structures. The structure of the translator is as shown in Figure 19. Once the C-Machine is generated, depending on the target machine an appropriate path can be taken. Currently it is executed on a sequential machine, so the "C" compiler is activated and the C-machine is compiled and executed.

5.2.1 The Virtual (C-Machine) Machine

The C-machine is the simplest that one could design for representing an artificial neural network. The idea stemmed from the simple view that a neural network is a network of automatons and each of which are associated with a set of

- inputs;
- weights and state parameters;
- outputs.

This led to the synthesis of a data structure which is simple and easy to manipulate (see Appendix E for the full representation of this data structure in "C"). This data structure as shown in figure 20 consists of a list of nodes (see Figure 21). Each node consists of two data fields

1. a node identification.
2. name of the function which specifies the computational method of that node.

and four pointers-

1. pointer to a list of input vector nodes.
2. pointer to a list of weight vector nodes.
3. pointer to a list of output vector nodes.
4. pointer to the next node in the network.

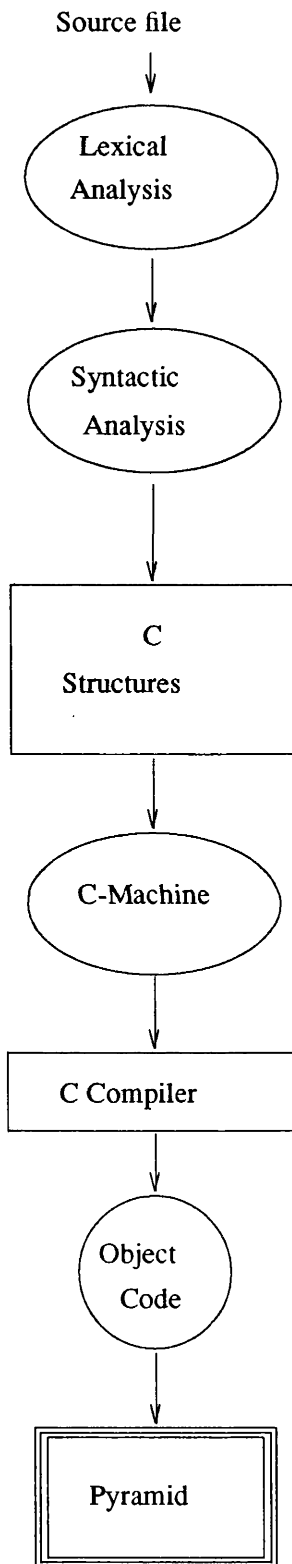

Figure 19. NIL Translator

These nodes are not arranged in any order. The topology of the network, that is the way the nodes are connected to each other is only determined by names of I/O link variables which are included in the description of the elements of the I/O vectors of each

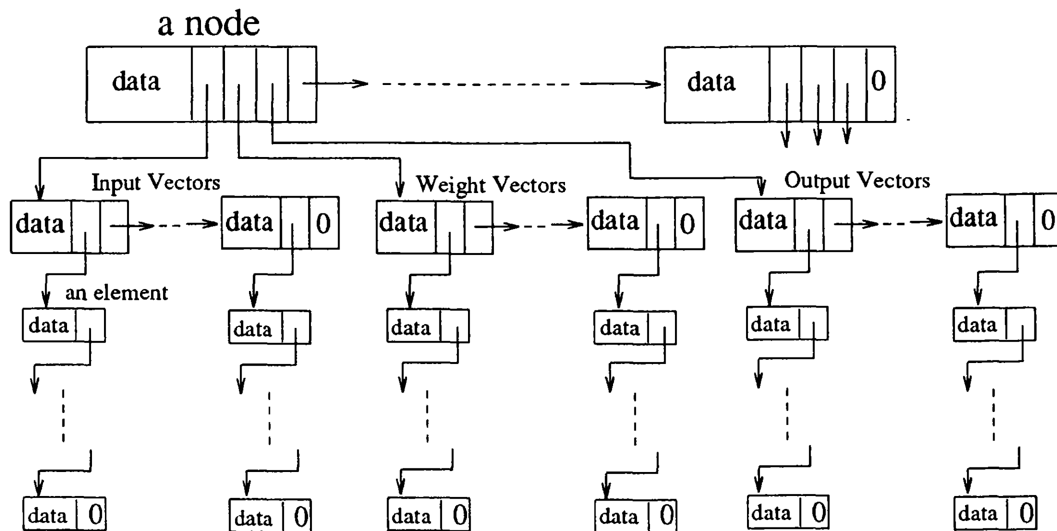

Figure 20. The C-Machine

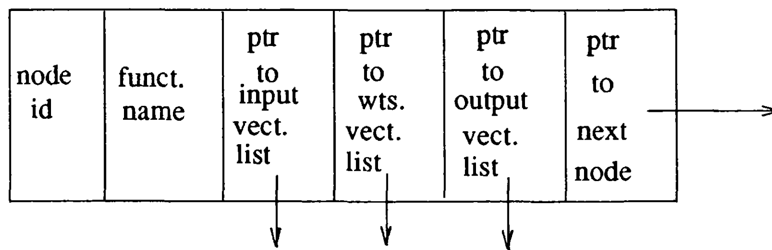

Figure 21. A Node

node(see Figure 22 and 23). Each input/output/weight vector node contains information about a particular input vector of a function definition and two pointers, one for the associated list of vector elements and the other for the next input vector node as illustrated in the Figure 22 and Figure 23.

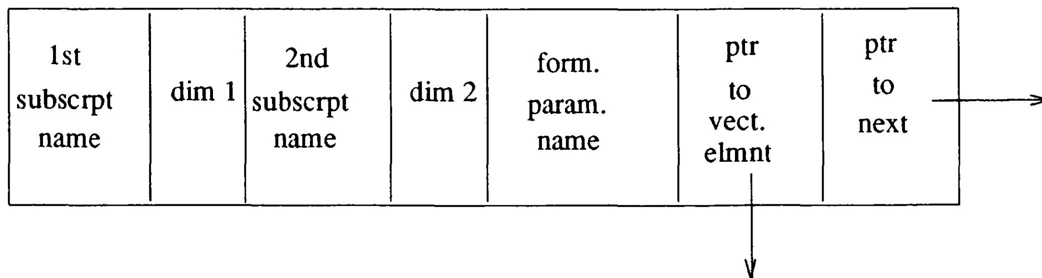

Figure 22. An I/O Vector

Each element of the I/O vector contains the name of the link (actual parameter) variable, index values of the I/O vector element (formal parameter), the status of the I/O value, the current or the last I/O value and a pointer to the next element in the list. The status value of an input element indicates whether this value is a fresh one (ie- not yet used) or an old

value (ie-an output has been generated by processing the input). In the case of the output value it indicates whether this value has been distributed to all the relevant destination nodes or not (see Figure 23).

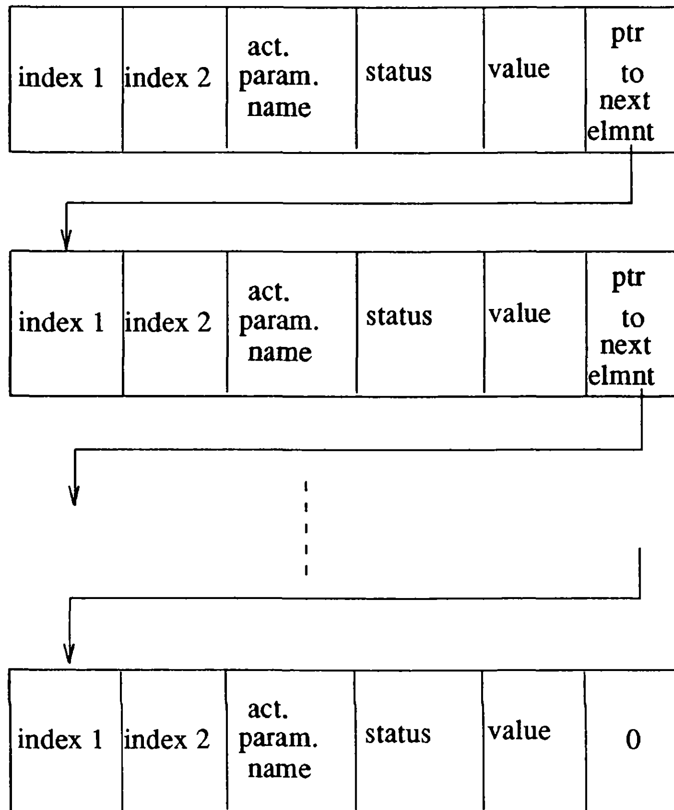

Figure 23. Elements of an I/O Vector

The element of the weight vector is slightly different from the I/O elements in that it does not have the name of the actual parameter name for there is none.

This virtual machine can then be compiled and executed on the pyramid. The main advantage of having the network in this form is that it can easily be translated into other target languages especially for execution on parallel hardware. It fully describes the processing activity of a node and its I/O data space. This also makes it easier for converting it into object-oriented programs such as C++ and Objective C.

5.2.2 The Implementation of the Manipulation Part

The manipulation part is run on an interpretation mode using a code interleaving strategy. That is, whenever a **get** command is encountered all the node processes are executed once in a loop before checking to see if the inputs expected by the get command are available. This is repeated until the expected values are available to the host before executing the next statement in the manipulation program. In the case of the

UCL Neurocomputer implementation this checking is done after the execution of each node to be consistent with the assumption that Host and the nodes are parallel processing units.

5.3 The Prototype Compiler

The prototype compiler for generating code for UCL Neurocomputer was generated using *lex* [Lesk75] and *yacc* [John75] and has the similar structure as the translator down to the syntactic and semantic analysis stage. Then it differs from the translator by producing the symbol tables and generating assembler codes like any other compiler. In this compiler, no intermediate code is generated. A straight forward code generation is applied. To convert the assembler code into executable machine code, an assembler-compiler was implemented and is used to produce the machine code for the neurocomputer as shown in Figure 24.

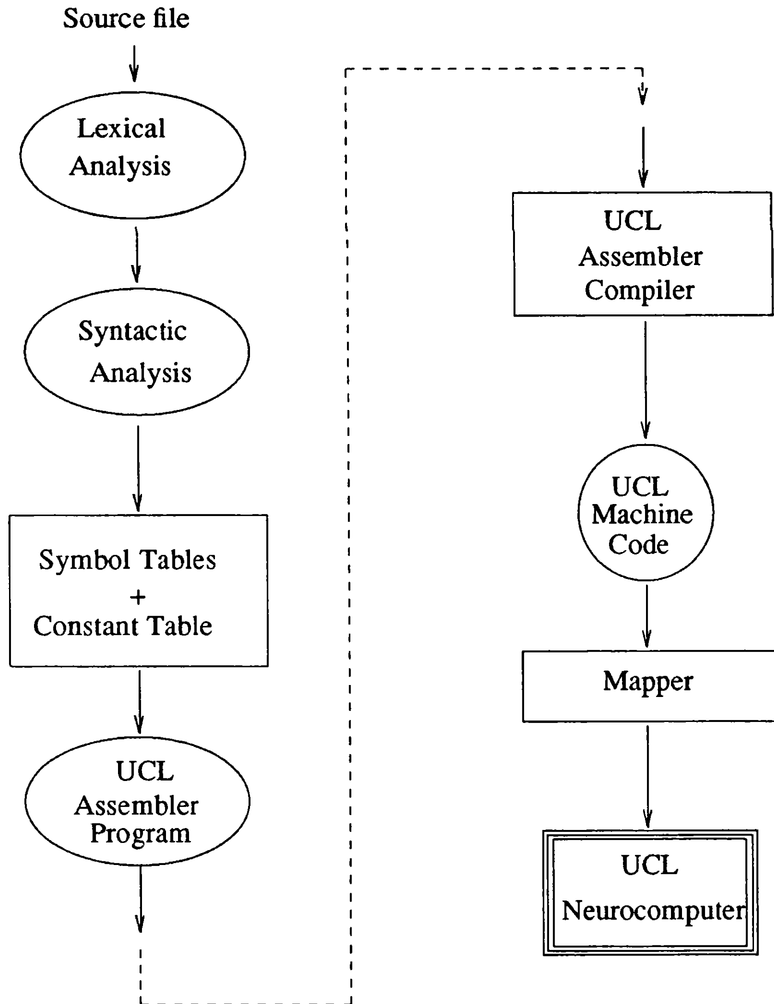

Figure 24. The Prototype Compiler

The loader and mapper used was developed during the implementation of the simulator and is a very simple one which merely loads each virtual node into a simulated processor in a sequence.

5.3.1 Data Structures for UCL-Neurocomputer

The implementation on the UCL-Neurocomputer simulator was much easier than expected. This could be due to the fact that the computational model of a node in the language is in a way very similar to the model of a processing element. The overall structure of the program in each node is as shown in Figure 25.

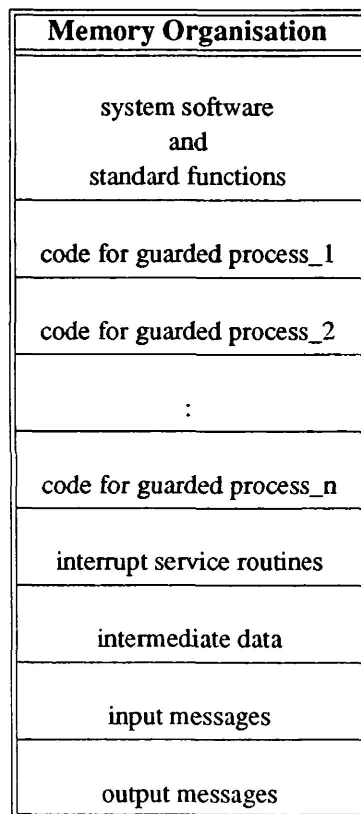

Figure 25. Structure of a Node Program

The control flow structure of the application code can be explained by the following pseudo-code.

```
while TRUE do
begin
    if (input_condition_1_is_true)  then
            execute guarded_process_1
            call update_I/O_status_flags
            call send_output
            goto START
            fi
            :
            :
    if (input_condition_n_is_true) then
            execute guarded_process_n
            call update_I/O_status_flags
            call send_output
            goto START
            fi
            end
```

When an interrupt occurs (ie- when a packet arrives) the control is transferred to the service routine for transferring the input packet to the data area allocated for placing input messages and appropriate updatings are done before returning control to the application software. The main feature of this implementation is in the way the input and output data are organised. First of all let us look at the organisation and handling of the input data.

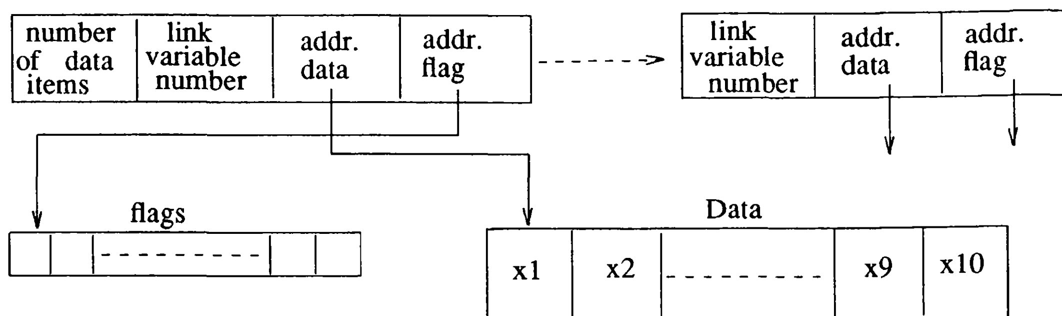

Figure 26. Organisation of the Input Data

The input messages are organised in contiguous blocks of memory (see Figure 26). At the head of these blocks is a single cell which contains the number of input data blocks. This number includes the actual number in use plus ten free blocks for further use during run time (for example:- when creating new links). Each input block consists of a

number which represents the link (input) variable, a pointer to the formal parameter variable into which it is to be copied and a pointer to a flag field which represents the status of the current data.

When an input arrives, an interrupt transfers the control to a routine which determines the source and services the interrupt. The service routine copies the newly arrived packet into a temporary buffer and then the appropriate block which contains information about this input (ie-link variable/input number) is used to copy the data into its formal parameter variable and update its status.

The output data is again controlled by a set of control blocks placed contiguously (see Figure 27). As with the input control blocks, at the head of the output blocks a single memory location contains the number of output data. Each output block consists of a link variable number as before, number of destination nodes (m say) for which this data is to be sent, a list (m) of destination node identification numbers and ten spare slots for further use.

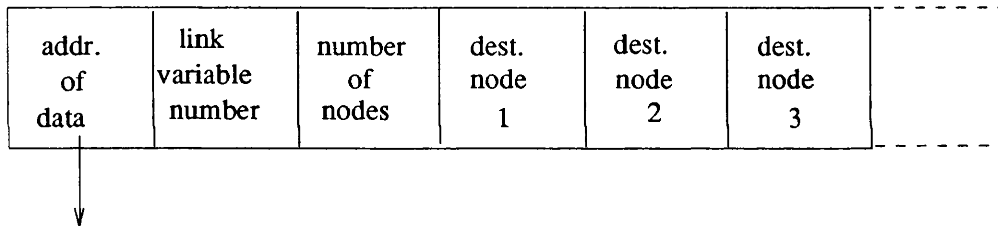

Figure 27. Organisation of the Output Data

5.4 Implementation of NPS

The NPS is implemented in the form of a command interpreter where each command is interpreted and appropriate action is taken. In the case of sequential execution NPS behaves almost like a UNIX shell where commands are accepted and appropriate programs are executed and the control is returned to the user with one exception. When a network is being executed the user can gain control in order to stop and save the network by typing a "CONTROL X" . This character input is trapped by a function and the control is given to the user after the execution of each node in the list. The command interpreter has the following form :-

```
While not EOF do
begin
   get command
   execute command
   end
```

The "execute command" consists of interpreting the commands and calling the appropriate function to act. For instance when the run command is issued, the file name is passed to the NIL translator for translation and subsequent execution.

The commands that can be executed by the NPS are

- **run** *-m file_name* - which takes the source file called file_name and compile, load and execute it on the desired machine specified by "m".
- **load** *-m file_name* - takes a partially run network in file_name and loads it on to the desired system specified by the value given to "m".
- **go** - start executing the network.
- **stop** - stops a running network.
- save *file_name* - saves the current configuration of the network for later execution in the named file.
- **abort** - aborts the current network.
- exec *model_name* - initiate the execution of a network package available in the library. This will be followed by a number of request for parameters such as input values, number of layers, nodes per layers etc. Once the system has got all the information, it will proceed with the compilation, loading and execution of the network.

Chapter 6

This chapter presents an assessment of NIL and the NPS. The assessment of NIL consists of testing its portability and model independence, and showing how programs written in high level languages can be compiled into NIL. The assessment of the NPS centers around the execution of the system's commands and the use of library functions to demonstrate its programmability.

6. Assessment of NPS and NIL

To assess a programming system where the specification language forms the major part is to demonstrate that it fulfills its intended role efficiently and easily. In the case of NIL it must demonstrate that it is portable and contributes to programmability of the system. This means that it must demonstrate that:

1. It is independent of any particular hardware;
2. It can represent a range of neural network models;
3. It is suitable as an intermediate level language for neural network programming;
4. It supports reusability.

In the case of NPS, assessment is made with respect to its programmability. This means that it must possess the following properties:

1. **usability** - how easy it is to code and execute neural network algorithms on a particular machine?
2. **facilities** - the main facility, library of functions and models, how helpful are they and how can they be improved?
3. **simplicity** - how simple is the design? is it easy to maintain and upgrade?

6.1 Assessment of NIL

This section assesses the capabilities of NIL. As mentioned earlier the assessment falls under two categories. They are

1. **Programmability**: In this category, NIL is assessed for its

a. specification capability by coding three different algorithms, namely, Hopfield, Back-error propagation and Kohonen's feature map algorithms in NIL and executing it to show that it can satisfy the programmability requirement.

b. dynamic properties to show its general applicability.

c. suitability as an intermediate language and its reusability.

d. overall strengths and weaknesses by comparing it with another intermediate network language called BIF.

2. **Portability**: In this category, NIL is assessed for its

a. target machine independence to show that it satisfies its portability requirement.

b. ability to deal with parallel computations by comparing with Occam.

c. suitability as a target code by showing generally how an algorithm specified in a high level language can be compiled into NIL.

6.1.1 Programmability

6.1.1.1 Specification of Models. The specification capabilities of NIL is demonstrated by coding three different models, namely, Hebb/Hopfield, Back-error propagation, and Kohonen's feature map in NIL and executing them. To look at the capabilities of the language and how it is used, let us first consider the following program written in NIL to implement a Hebb/Hopfield algorithm that learns three patterns and recalls them. In this example each input vector consists of five elements and there are two layers, an input layer which merely propagates the input patterns to the next layer and an output layer that learns and recalls the patterns. A model with five input units and five output units to implement an auto-associator was coded in NIL. Three patterns were used as input and the system was trained and recalled as described in [Psal87]. The network that describe this model is as shown in Figure 28. This network was specified using six connection statements as follows.

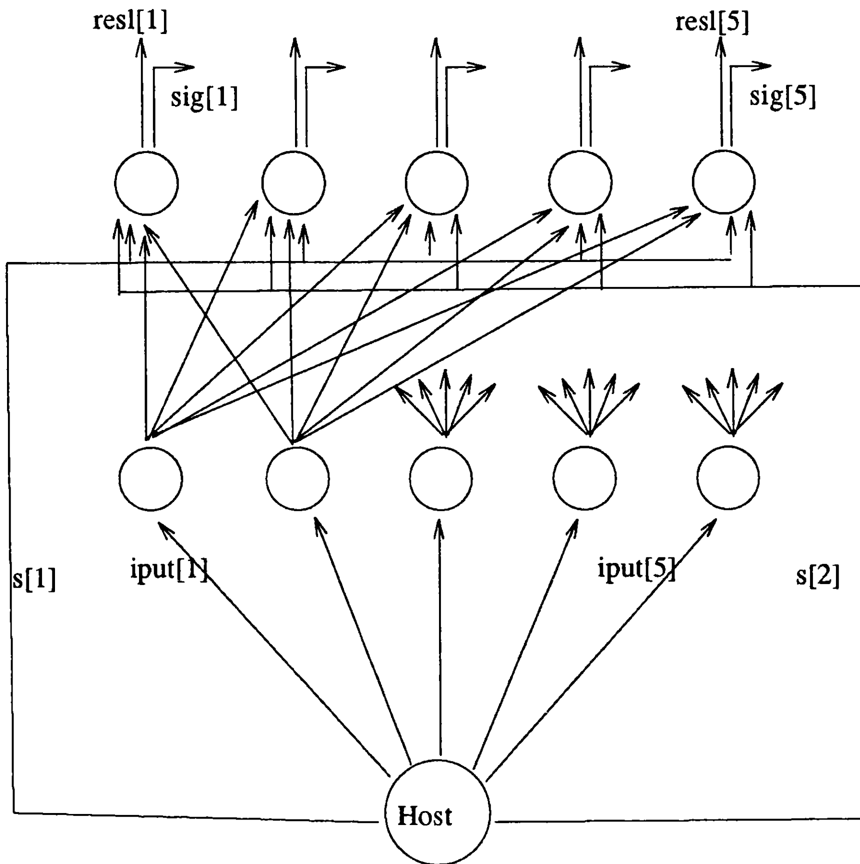

Figure 28. Hebb/Hopfield Network

```
begin
/* a rep statement specifying the input layer */
rep[5] lay1(iput[i]) -> (out[i])
/* five link statements specifying the output layer */
lay2([out[1],out[2],out[3],out[4],out[5]],[s[1],s[2]] :
        [0.0,0.0,0.0,0.0,0.0],[1.0]) -> ([sig[1]],[resl[1]])
lay2([out[1],out[2],out[3],out[4],out[5]],[s[1],s[2]] :
        [0.0,0.0,0.0,0.0,0.0],[2.0]) -> ([sig[2]],[resl[2]])
lay2([out[1],out[2],out[3],out[4],out[5]],[s[1],s[2]] :
        [0.0,0.0,0.0,0.0,0.0],[3.0]) -> ([sig[3]],[resl[3]])
lay2([out[1],out[2],out[3],out[4],out[5]],[s[1],s[2]] :
        [0.0,0.0,0.0,0.0,0.0],[4.0]) -> ([sig[4]],[resl[4]])
lay2([out[1],out[2],out[3],out[4],out[5]],[s[1],s[2]] :
        [0.0,0.0,0.0,0.0,0.0],[5.0]) -> ([sig[5]],[resl[5]])
/* specification of the function for nodes in the input layer*/
fun lay1(x[i]) -> (y[i])
int i;
```

```
real x[2],y[2];
{
 x[i] => {y[i] := x[i];} /*sends inputs to output layer */
}
/* specification of the function for nodes in the output layer*/
fun lay2(y[i], lr[j] ival:w[i], index[m]) ->(signal[m],result[m])
int i,j,m,k,ii;
real y[7],lr[4],index[2],signal[2],result[2],w[10],net;
{
y[1..i], lr[1] => {
  /* learns the input pattern by adjusting the weights */
               }
y[1..i], lr[2] => {
  /* recalls the input pattern */
               }
}
/* specification of the manipulation part (host)*/
begin
real a, b, p, q, r, s, t;
a :=1.0; b:=-1.0;
/* loading initial weights for nodes 6 and 8 */
ldconst nde[6].w[1] = 0.11, nde[6].w[2] = 0.21, nde[6].w[3] = 0.12,
      nde[6].w[4] = 0.04, nde[6].w[5] = 0.13;
ldconst nde[8].w[1] = 0.15, nde[8].w[2] = 0.06, nde[8].w[3] = 0.17,
      nde[8].w[4] = 0.02, nde[8].w[5] = 0.11;
/* inputting patterns for learning */
input iput[1] = b,iput[2] = a,iput[3] = a,iput[4] = a,iput[5] = b,s[1] = a;
run_net;
/* has all the 2nd layer nodes learnt the pattern? */
get p = sig[1], q = sig[2], r = sig[3], s = sig[4], t = sig[5];
/* yes - input the next pattern for learning */
input iput[1] = b,iput[2] = b,iput[3] = a,iput[4] = a,iput[5] = a,s[1] = a;
run_net;
get p = sig[1], q = sig[2], r = sig[3], s = sig[4], t = sig[5];
input iput[1] = a,iput[2] = a,iput[3] = b,iput[4] = a,iput[5] = a,s[1] = a;
run_net;
get p = sig[1], q = sig[2], r = sig[3], s = sig[4], t = sig[5];
```

```
/*RECALL THE FIRST PATTERN*/
input iput[1] = b,iput[2] = a,iput[3] = a,iput[4] = a,iput[5] = b, s[2] = a;
run_net;
/* is the output ready? */
get p = resl[1], q = resl[2], r = resl[3], s = resl[4], t = resl[5];
/* yes - display the results */
output "result = ",p,q,r,s,t;
    :
end
end
```

The first "rep" statement describes the input layer and the subsequent five statements describe the second layer and its connections(see Appendix B). The processing activity of an input unit is described by the function lay1(), which merely sends the input sent by the host (represented by the manipulation program) to all the output units in the next layer. In this case the first layer is not really needed and can be excluded from the network. The reasons for including it is to show how a simple two layered network can be built and the use of the parallel "rep" statement. The computational behaviour of the output units are described by the function lay2(). This function consists of two guarded processes, one for learning and the other for recalling. The link statement associated with these nodes have two input vectors, [out[1],..,out[5]], and [s[1],s[2]]. The first input vector represents the input pattern and the second input vector represents the learn(s[1]) and recall(s[2]) signals to the nodes. The two weight vectors of each link statements are mapped into the weight vectors w[i] and index[m] of the output nodes respectively. The w[i] vector represents the actual weights on the links(which are all zeros initially) and the index[m] represents the node index (i.e. 1,2,..,5) which enable each node to determine its position in the output layer. The output of each link statements again consists of two vectors, [sig[]], and [resl[]]. The first set carries a signal from each node([sig[1]],..,[sig[5]]) to the host to inform that the current pattern has been learnt and the second set of vectors([resl[1]],..[resl[5]]) carries the output generated by each node when a recall is initiated.

The manipulation part begins with some initialisation statements followed by two "ldconst" statements which changes the weights associated with node 6 and 8. These two statements are mainly included to show how new weights can be loaded on to the links. This is then followed by an "input" command to load the first input pattern and the signal for learning(s[1]). After loading the inputs, the network is executed using the "run_net" command. Then the "get" command is used to read the completion signals(sig[]) before submitting the next pattern for learning. Once all the patterns have been learned, "input"

command is used again to input a pattern and the recall signal(s[2]) for recalling that pattern. Finally "output" command is used to print the results.

When this algorithm was executed both on the UCL's simulated neurocomputer and on the pyramid technology machine, it learnt the patterns and recalled them correctly. The initial weights are as given in table 6.

Weights	Node 6	Node 7	Node 8	Node 9	Node 10
w1	0.1100	0.0000	0.1500	0.0000	0.0000
w2	0.2100	0.0000	0.0600	0.0000	0.0000
w3	0.1200	0.0000	0.1700	0.0000	0.0000
w4	0.0400	0.0000	0.0200	0.0000	0.0000
w5	0.1300	0.0000	0.1100	0.0000	0.0000

TABLE 6. Initial Weights

The node numbering is done by the compiler by assigning the values in ascending order starting with the first node associated with the first link statement and so on. The final listing of the weights and the output results produced by running the program is as given below.

```
WEIGHTS FOR NODE 6 /*i.e- the first output node */
w1 = 0.000000 w2 = 1.210000 w3 = -2.880000 w4 = -0.960000 w5 = 1.130000
WEIGHTS FOR NODE 7 /* the second output node */
w1 = 1.000000 w2 = 0.000000 w3 = -1.000000 w4 = 1.000000 w5 = -1.000000
WEIGHTS FOR NODE 8
w1 = -2.850000 w2 = -0.940000 w3 = 0.000000 w4 = 1.020000 w5 = -0.890000
WEIGHTS FOR NODE 9
w1 = -1.000000 w2 = 1.000000 w3 = 1.000000 w4 = 0.000000 w5 = 1.000000
WEIGHTS FOR NODE 10
w1 = 1.000000 w2 = -1.000000 w3 = -1.000000 w4 = 1.000000 w5 = 0.000000

  INPUT --> OUTPUT
 -1.000000 --> -1.000000
 1.000000 --> 1.000000
```

```
1.000000 -->  1.000000
1.000000 -->  1.000000
-1.000000 --> -1.000000

  INPUT -->  OUTPUT
-1.000000 --> -1.000000
-1.000000 --> -1.000000
1.000000 -->  1.000000
1.000000 -->  1.000000
1.000000 -->  1.000000

  INPUT -->  OUTPUT
1.000000 -->  1.000000
1.000000 -->  1.000000
-1.000000 --> -1.000000
1.000000 -->  1.000000
1.000000 -->  1.000000
```

full listing of the program can be found in Appendix B.

Back-Propagation Model

Back-propagation algorithm [Rum86c] is a fairly popular model and goes a long way towards guaranteeing a global minimum. Apart from this, it can also succeed in mapping any function using one or more layers of hidden units. Our reason for including this algorithm in this assessment is to test the ability of the language in dealing with feed-back loops. To test this basic property, the exclusive **or** (XOR) problem was coded using two input units, one hidden unit and an output unit. Again the host executes the manipulation sub-program. The problem is represented by the network topology shown in Figure 29. In this example, the complete network is built using the powerful **construct** statement to demonstrate its capabilities, especially its ability to deal with feed-back loops.

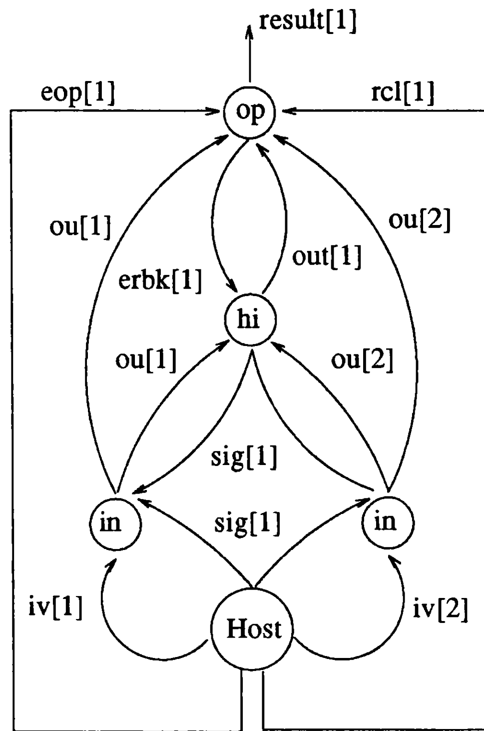

Figure 29. Exclusive OR Network

```
begin

construct ( [i=2] in(iv[i], sig[1]) -> (ou[i])
        :[p=1] hi(ou[i], erbk[1] : 3*rnd[0-1], 1*rnd[0-1], 1*rnd[0-1],
                    1*rnd[0-1],1*rnd[0-1] ) -> (out[p], sig[p])
        :[k=1] op(ou[i], out[p], eop[1], rcl[1] :3*rnd[0-1],1*rnd[0-1],
                        1*rnd[0-1]) -> (erb[k], result[k])
:
:
end
```

Even though, slightly complex for an intermediate level language, it is an essential facility that greatly reduces the number of link statements.

The random function provided by the language was found to be very useful when initialising the weights etc. Various other models based on this algorithms were tested during trials and found to be satisfactory (see Appendix B for full coding, test data and results). This example also demonstrate the power and value of a separate manipulation sub-language to control and monitor the network using the control commands in "do" and "if" statements.

The initial weights and the output of the program which includes the final weights, and the output values produced when recalled is as given in Table 7.

Weights	Node 3(Hidden)	Node 4(Output)
w1	3.10000	1.20000
w2	2.10000	2.60000
w3	-	1.27000
theta	2.50000	1.70000

TABLE 7. Weights for Nodes in Hidden and Output Layers

```
FINAL WEIGHTS FOR NODE 3 /* the hidden node */
W1 =  5.596046
W2 =  5.521795
theta =  -2.134852

FINAL WEIGHTS FOR NODE 4 /* the output node */
W1 =  -3.271263
W2 =  7.620885
W3 =  -3.245167
theta =  -2.529501

  input     input ==>   output
0.000000  0.000000 ==>  0.151414
0.000000  1.000000 ==>  0.831594
1.000000  0.000000 ==>  0.830358
1.000000  1.000000 ==>  0.193722
```

A full listing of the program can be found in Appendix B.

Kohonen's Feature Map Model

This model is basically a pattern classifier and serves as a good test bench for any neural network language. The main reason is that it is a system where the neighbourhood of a node changes during every cycle. To implement such a system in a truly parallel hardware one requires a language capable of handling a heterogeneous network. We

found that NIL seems to handle this aspect fairly easily with the help of an extra node to determine the neighbourhood nodes at the beginning of each cycle. This also demonstrates that NIL is able to deal with non-homogeneous neural network models.

A simple version of the model without lateral connections consisting of three input nodes, twenty five feature map nodes and an output node (see Figure 30) was coded for this assessment [Lipp87].

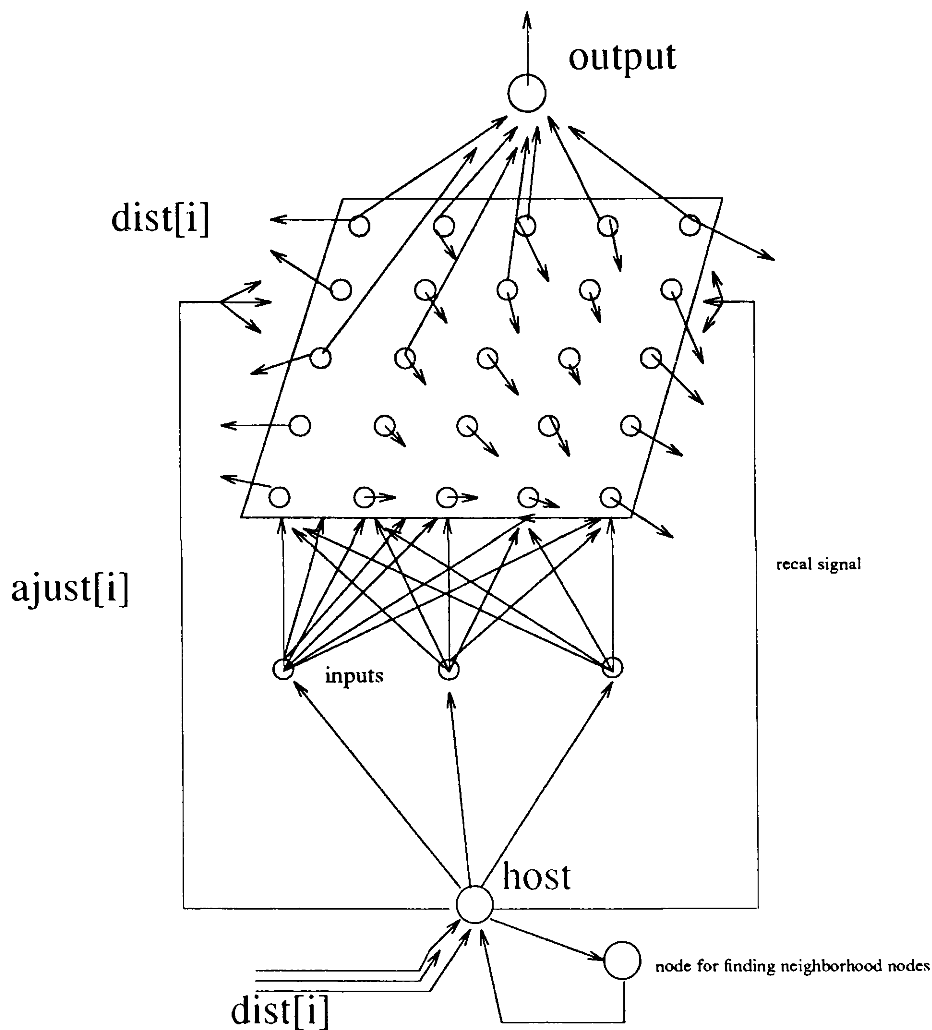

Figure 30. Kohonen's Feature Map

Originally the connections were specified using a "rep" statement and twenty six simple link statements(25 for each node in the feature map and one for the output node). Then the network was specified using the "construct" statement to show the advantage of using this statement. In both cases the results were the same.

CODE USING SIMPLE LINK STATEMENTS ONLY

```
begin
rep[3] inode(iput[i])->(invec[i])
onode([invec[1],invec[2],invec[3]],[learn],[ajust[1]],[si],[recal]
        :[0.2, 0.1, 0.05],[0.0])->([dist[1]],[otp1])
onode([invec[1],invec[2],invec[3]],[learn],[ajust[2]],[si],[recal]
        :[0.2, 0.1, 0.05],[0.0])->([dist[2]],[otp2])
onode([invec[1],invec[2],invec[3]],[learn],[ajust[3]],[si],[recal]
        :[0.2, 0.1, 0.05],[0.0])->([dist[3]],[otp3])
onode([invec[1],invec[2],invec[3]],[learn],[ajust[4]],[si],[recal]
        :[0.2, 0.1, 0.05],[0.0])->([dist[4]],[otp4])
        :[0.2, 0.1, 0.05],[0.0])->([dist[21]],[otp21])
                :
onode([invec[1],invec[2],invec[3]],[learn],[ajust[22]],[si],[recal]
        :[0.2, 0.1, 0.05],[0.0])->([dist[22]],[otp22])
onode([invec[1],invec[2],invec[3]],[learn],[ajust[23]],[si],[recal]
        :[0.2, 0.1, 0.05],[0.0])->([dist[23]],[otp23])
onode([invec[1],invec[2],invec[3]],[learn],[ajust[24]],[si],[recal]
        :[0.2, 0.1, 0.05],[0.0])->([dist[24]],[otp24])
onode([invec[1],invec[2],invec[3]],[learn],[ajust[25]],[si],[recal]
        :[0.2, 0.1, 0.05],[0.0])->([dist[25]],[otp25])
resnode([otp1,otp2,otp3,otp4,otp5,otp6,otp7,otp8,otp9,otp10,otp11,
otp12,otp13,otp14,otp15,otp16,otp17,otp18,otp19,otp20,otp21,otp22,
otp23,otp24,otp25]) -> ([clout])
:
end
```

CODE USING CONSTRUCT STATEMENT

```
construct([i=3] inode(iput[i]) -> (invec[i])
      :[j=25] onode(invec[i],learn[1],ajust[j],si[1],recal[1]
: 3*rnd[0-1], [0.0]) -> (dist[j], otp[j])
      :[k=1] resnode(otp[j]) -> (clout[k]) )
            :
end
end
```

As expected the execution of this model was very slow on the neuro-computer architecture simulator. The heavy load of messages between the host and the nodes of the network reduced the speed of execution. On the other hand, the "C" version was comparable with a three layered back propagation model with ten nodes in each layer. Again the construct statement showed its power in terms of its compactness and expressibility. The manipulation part seems a bit long, but given the required amount of interaction with network it was tolerable. A full listing of the program with nine feature nodes, input values, and the output from the program can be found in Appendix B.

6.1.1.2 Dynamic Properties of NIL The dynamic properties of the language was tested by creating a simple network consisting three layers, two input nodes, two hidden nodes, and an output node. The test consisted of deleting, creating links and nodes to see if the network behaves as expected (see example in Appendix B for more details and the execution results). This proved to be satisfactory however there were limitations such as

1. when adding node types which are not already present in the network.
2. in the neurocomputer implementation, adding nodes with different number of I/O links other than the one already present needed major modifications to the code and was found to be a complex and time consuming process.
3. in the case of creating a new link in a network, it was found that some extra processing is needed to adjust the I/O channel indexes in the C-Machine version.

6.1.1.3 NIL as an Intermediate Language and its Reusability Assessing the suitability of NIL as an intermediate language depends on what we use as basic criteria for classifying a language as an intermediate level language. We believe an intermediate language should have the following properties.

1. It should not have any high level functions.
2. A program written in this language should ideally represent a virtual machine.
3. Intermediate state of this machine can be saved and the execution can be resumed from where it was stopped.
4. Translating to target machine code for execution on a particular hardware should not involve too many overheads nor should it be a multi-step process.

5. A high level language should compile down to this language and not up to it.

6. In the case of network language, an intermediate language should clearly define its topology and allow interaction at the intermediate level(ie- multi-level interaction).

7. A network language at intermediate level also should have the facilities for interfacing with other tools such as graphics, debuggers and other support tools.

The first property, which is that the intermediate level language should be free of any high level functions is generally considered to be an unnecessary restriction given the current diverse nature of computing. Especially in the case of an intermediate level neural network language that is supposed to facilitate interaction at this level. So this requirement is not strictly met by NIL. However, the language was deliberately designed to be at a similar level as PARLE[McCa88] and Occam[MayD87] which are generally considered to be typical intermediate languages.

The second property, that an intermediate level language should represent a virtual machine is an important one and we believe that this requirement is fully satisfied. This is achieved by designing the language as a representation of a network of automatons where each node is based on a very general computational model which changes its state according to its input. This is further proved by the fact that a NIL program can be translated to a C-Machine which is almost similar to the NIL program in structure.

The third property is also believed to be satisfied by the successful demonstration of the "save" and "load" commands. The existence of a compact prototype compiler for mapping NIL programs on a simulated general purpose neurocomputer architecture seems to prove that the fourth requirement is also satisfied. The fifth requirement can be safely interpreted as meaning that at least no extra data structures must be generated during translation from a high level language. This seems to be true with NIL as there are no data structures and the language is primarily built on primitive constructs.

The sixth requirement, which is clear definition of the topology and intermediate level interaction, is believed to be satisfied. Firstly, on the issue of clear definition of the topology, the language scores well in using the three types of connection statements namely, link, rep and construct. Especially the "rep" and "construct" statements gives a clear and concise view of the network in a compact form. Secondly, on the issue of

intermediate level interaction, the manipulation sub-language have demonstrated its ability beyond reasonable doubt.

Finally, satisfying the last requirement is not seen as a problem. Especially, because of the primitive nature of the connection statements. NIL is ideally suited for translating into graphic form and back to it. The question of a debugger for NIL is made redundant to an extent by the existence of the manipulation sub-language. It is also anticipated that there will be no problem in calling other tools in the system.

In all the above mentioned cases NIL seems to score well except in its definition of functions which seems to be slightly at a higher level. But when compared with BIF (the only intermediate language in its class) it compares favourably.

The reusability issue in the case of NIL is an important one in that the language lends itself to reusing a function definition as long as the link statement which uses it correctly specifies the actual parameters. This makes it possible for building function definitions of general type, for example, a hidden unit in a back-propagation model or an output unit in Kohonen's feature map model can be built and used again and again. This is why a set of general functions are also included in the functions library.

6.1.1.4 Comparison with another Language in its Class One of the very few languages in this class (ie- intermediate level network languages) is BIF (Beaverton Intermediate Form) in ANNE [Bahr87]. This is based on the "C" language syntax and describes the network structure in a standardized network format using a "C" data structure. In BIF the basic network object is called a Connection Node (CN). Each CN may have one or more sites, and each site has one or more links. A CN is, thus, made up of three sub-parts. The main CN field, a site, which groups the links and hold the values from those links. At the terminal end of each link is the address of its other end in the node which is connected to it.

A BIF file has two parts. The first contains a listing of the CN groups, each of which consists of a unique group index, a string name, and two initialisation values for CNs belonging to that group. Each CN carries an index corresponding to the group to which the CN belongs. The group name allows the user to address groups of CNs symbolically. Each BIF file must have two special CN groups named "input" and "output". These groups designate the CNs used for global I/O operations.

The second part consists of individual CN records. These records are composed of a hierarchy of CNs, sites, and links. Sites nest within CNs, and links nests within sites. Input or output sites are not listed in any order. Neither the sites nor the links are

explicitly indexed in a BIF file. Sites happened to be specified in BIF as being either input or output. The network procedure at node level consists of functions for performing sub tasks which collectively implement the activity of a node. The procedure at host level is similar to the manipulation part in NIL except that NIL is much richer in terms of network manipulation commands. The rest of this sub section compares NIL with BIF systematically using the following criteria.

1. representation of network topology.
2. representation (holding) of data values.
3. representation of a node (ie- function representation) and description of computation in a node.
4. control structure for controlling and manipulating the network during run time.

Comparison of NIL with BIF is based on the programs given in Appendix C. These two programs implement a three layer Back-propagation model consisting of five nodes in each layer. Each aspect mentioned above is considered one by one.

Representation of network topology - As mentioned earlier, the network topology in BIF is described in two parts using "C" data structures. In the first part, the nodes are grouped according to their types. The second part describes the individual nodes as a set of hierarchical records. These records contain informations such as its group identification, number of sites, number of links per I/O values etc. These are accessible to the user and can be viewed. This form of representation does not give a clear picture of the network topology. On top of this, the input and output sites are not listed in any order nor are they explicitly indexed. These make it even more difficult for the user to understand the topology. On the other hand, NIL expresses its network topology in a neat and simple way by using its simple link statement or rep and construct statements. In this particular case, it uses the construct statement to give a compact view of the network topology. A NIL's link statement completely describes the input, output links, the weights, and other state values associated with each node in a single statement. When a construct statement is used to define the topology, the topology of the network becomes even more clearer to the user. This is not so clear in the case of BIF [see Appendix C] because most of the informations concerning the network topology is burried into the "C" data structure. In NIL the topology can be clearly worked out by looking at the link statements. This leads us to conclude that NIL's representation of the network topology is preferable to that of BIF.

Representation of data values - By this we mean, how the data such as weights, status values, and the I/O values are held by the intermediate language. In the case of BIF, these are kept as a part of the data structure. In the case of NIL, these are part of the program constructs. For example, the weights are a part of the parameter list of a node function and the link statement associated with it, and can be displayed by using the "readst" or the "get" command. We are of the opinion that it is preferable to hold the data in the language construct than in a data structure at this level (i.e. - intermediate level). The main reason for this is that it requires less overheads to access these data. The second reason is that it gives a clear view of the algorithm and the data in a combined form.

Representation of a node - In BIF, a node procedure is described as a collection of separate functions, where each function performs a sub task. This means that a node activation triggers a series of calls to all or a sub set of these functions depending on the purpose behind the activation (eg- learning or recalling). In NIL, a node is described as a complete entity in the form of a function. This function consists of a header that specifies the input, output, and weight parameters, a body that specify the computational tasks of the node in the form of a set of guarded processes (eg- for learning and recalling). A node function is a general class of function in NIL. Using this general class approach, one can specify the basic classes such as input, output, and hidden nodes and then specific nodes (units) which can be generated from these by connecting the appropriate I/O links. This is where the important differences between NIL and other languages in its class comes to the surface. First of all, on the issue of clarity, NIL fairs well because it is able to describe a node completely (ie- its computation and data). Secondly, it shows that mapping of neural network algorithms as a network of nodes on a parallel as well as a sequential hardware is easier with NIL. On the other hand, a faithful implementation of a network model using BIF will require excessive amount of resources and so it is mainly used as a simulation language. This is further supported by the fact that the current implementation of BIF on a hyper-cube system only runs on a simulation mode. This lead us to conclude that NIL is better suited to program the type of massively parallel machines which are being developed for executing neural network models. On the other hand, NIL can also be used as a simulation language by splitting a node, that is- using each guarded process as a virtual process.

Control structures and manipulation commands - The control structures in BIF are composed of purely "C" statements such as "if", "for", "return", function calls, etc. These are very familiar constructs and easy to learn and use. The manipulation and other commands consists of a number of system calls. These includes

- Send_node_output(cn_index, site_index); which sends the output from a single CN along all the output links belonging to the named site.

- Send_net_output(filename); which is used specifically to get the host to write the output vector to the named file.

- Update_node_weights(cn_index, site_index); which is used to transmit the new weights from one end of a bidirectional link to the other. The CN and site indices in the parameter list name the group of links that transmit their weights.

In addition to these system functions, there are a set of run-time/user commands such as

- buildnet; for constructing and initialising the network and the auxiliary data structures.

- newrun; to begin a simulation run.

- stopnet; to suspend a network simulation.

- savenet; to save current network structure in a new BIF file, which can be used later for a new simulation.

- show; to display the state of the "local" simulation parameters and list the currently active traces.

- quit; to exit from the simulator.

In NIL, the control statements consists of "if" , "do", and "goto". The syntax of the "if" and "do" are made simpler. Even though the presence of "goto" statement gives the impression that it is not a proper structured language, it can be argued that it is a necessity. On the other hand, the system call/user commands in NIL are much more simple and geared towards parallel network execution mode and are part and parcel of the language. This means, NIL is independent of any simulation environment and does not require any other software other than its compiler for it to be implemented on a different computer. For example, if weights are to be updated for a set of binary links, these will be done by the user written code in each node autonomously. Building the network is the task of the compiler in NIL and is not seen as a systems task as in BIF. Apart from these, NIL has a richer command set for manipulating and controlling the network. Especially, the delete and join commands for links and nodes in NIL makes it more powerful and makes it eligible for dealing with connectionist models as a whole.

In conclusion the BIF programs do not clearly show the structure of the network as NIL does. Secondly, it does not give a complete picture of a node and its activity in a

compact form as in NIL. Translation into different target machine code and mapping of the code on to the target machine requires more computational effort than NIL would. Mapping a BIF program onto a neural network architecture like the UCL neurocomputer would only result in an inefficient system. It is more geared towards a simulation mode especially with its timer mechanism than a true neural network language. Even this is debatable when one considers the simplicity of the C-Machine produced by the NIL translator in terms of the procedural code for nodes and the host, and its simple data structure which describes the network.

6.1.2 Portability

6.1.2.1 Target machine independence The target machine independence of the language is demonstrated by successfully mapping the NIL programs on to a sequential machine (Pyramid) and a simulated parallel architecture (UCL neurocomputer). The execution of these models on the UCL simulated architecture led us to conclude that

1. The language is well suited to this particular architecture.
2. Parallel implementation of neural network algorithms using NIL does not require any extra overheads.
3. It can be implemented and executed on both parallel and sequential hardware.

Execution of these models both on a sequential machine and the parallel simulator also confirmed that the underlying virtual machine(C-Machine) is a suitable vehicle for porting network models over a range of machines. This view is further enforced by the fact that "C" language programs can be mapped on a range of hardware(i.e. parallel and sequential machines). In fact, the main reason for implementing a translator to translate NIL into "C-Machine" is to prove that NIL can be mapped on a wide range of hardware.

6.1.2.2 Occam and NIL As already mentioned, NIL has a lot in common with the language Occam. In a way NIL can be considered as a possible alternative to Occam in programming a network of transputers. In Occam, the computation falls into two categories, parallel, and sequential. This means that the programmer has to explicitly specify the mode of computation using the "PAR" and "SEQ" constructs provided by the language. In NIL, there are no such explicit constructs to specify the mode of computations. These are done in a natural way by splitting the parallel components into virtual nodes and allowing sequential execution within nodes. The pattern of connections are specified by the "link" statements. This has the obvious advantage of being able to express each node (which is the equivalent of a procedure in Occam) as a generic

function. The other advantage is that it expresses the whole process as a set of related but independent functions and specify how they are related to each other in terms of data channels. In Occam, communications between processes are established via data channels in similar manner as in NIL except that these channels are specified explicitly at the procedure level. The "rep" and "seq" statements in NIL have their equivalents in Occam.

For example, the statement

```
rep[N] A(X[i], Y[i]) -> (X[i+1], Y[i+1])
```

is equivalent to the following Occam code

```
[N+1] CHAN X, Y :
PAR
  ASTART (X[1], Y[1])
  PAR i = 1 FOR N
    A(X[i], Y[i], X[i+1], Y[i+1])
  ASTOP (X[N+1], Y[N+1])
```

and the statement

```
rep[N]  A(S[i], T[i]) -> (U[i], V[i])
```

can be expressed in Occam as

```
[N] CHAN S, T, U, V :
PAR i = 1 FOR N
   A(S[i], T[i], U[i], V[i])
```

The "construct" statement in NIL on the other hand is more powerful and will need many more statements in Occam.

Receiving and sending data is treated in different ways in both languages. In Occam, "?" and "!" notations are used to receive and send data and they are synchronised. In NIL, an input is accessed by referencing the appropriate input channel of a node. There is no explicit form of outputting data on a channel in NIL. This is done automatically by the run-time system in NIL. Most importantly, the I/O transfers are not synchronised. The body of a function in NIL consists of one or more guarded processes. These can be constructed using the ALT statement in Occam. The input condition associated with a guarded process also can be coded in Occam very easily as follows.

```
X[1..N] => {PROCESS_1}
```

can be coded in Occam as

```
M := 1
SEQ i = 1 FOR N
  B[i] := TRUE
WHILE M < N
```

```
ALT i = 1 FOR N
  C[i] ? X[i] && B[i]
   SEQ
     B[i] := FALSE
     M := M + 1
  TRUE
    SKIP
PROCESS_1
```

All these lead us to conclude that NIL can be an alternative to Occam and that it offers some compact notations to specify computations on a network of transputers. There are also few weaknesses in NIL such as the lack of facilities for configuring standard I/O devices, and a TIMER mechanism. It is also felt that provision of high level connection statements and richer input conditions will make NIL a serious candidate for programming transputers.

6.1.2.3 Translating high level languages into NIL In order to show that neural network programs written in high level languages can be easily translated into NIL programs, we decided to specify a high level language in a pseudo language form that will possess all the basic properties which are common to all neural network languages such as SLOGAN [Ange88], CONDELA [Kohl88], NEURAL [Chol88], AXON [Guts88]. These basic properties are

1. application modelling- most of the high level language in this field tend to act as application builders rather than algorithm implementors. This means they describe the network as a collection of sub networks cooperating to perform some task. In this, each sub network is described by specifying their layers, connections and the nodes.

2. connection statements- which allow full connection between layers of nodes, connection between individual nodes, and between sub networks.

3. node interfaces- definitions of node functions used in these systems may belong to a library or can be described in the program. These are specified as general purpose functions with variable lists of input, output and weight parameters. In order to configure these nodes in a given network, their I/O list's parameters(eg- length of I/O vectors) must be specified. This is usually done in node interface section of the language.

4. node definition - which specifies the computational behaviour of the nodes.

5. network control routine- which controls and manipulates the network.

A program written in this language has the following form:

```
network consists of sub_nets {
/* this part describes the network as a collection of sub networks */
                    model_name_1,
                    model_name_2,
                       :
                    model_name_k;
                    } network_name;
definition of model_name_1 :
/*This part defines each sub network by specifying the number of layers, number of
nodes in each layer and their types */
    { layers = no_of_layers ->
      {
/* the nodes and their types in each layer */
      layer[1] consists of  no_of_nodes of type node_name_1;
      layer[2] consists of  no_of_nodes of type node_name_2;
                 :
      layer[no_of_layers] consists of no_of_nodes of type node_name_n;
      }
node interface {
/* interface to each node type- ie- number of I/O channels and the weight vectors
associated with the prototype of the particular node type. The information about the base
type of this prototype is available to the programmer from a library */
node type node_name_1 has
      input_name_1[n], input_name_2[p] : inchan,
      output_name_1[m]: outchan,
      wt_name1[i], wt_name2[j] : local;
          :
node type node_name_n has
          :
          :
}

connections ->
{
/* This part consists of connection statements which link nodes in the sub network. These
```

connections can be layer to layer connections or single connections between nodes. */

connection statements for model_name_1;

}

} **end** model_name_1;

:

:

definition of model_name_k :

{

:

:

} **end** model_name_k;

sub_nets connections ->

{

/* this part describes the connections between the sub nets, again using similar connection statements used in connecting the nodes within a sub net. This part is optional and is only included if there are more than one sub network in the network */

connection statements

}

host()

{

/*this part is executed by the host processor and consists of commands for controlling the network */

}

Connection statements used to link nodes within a sub network take the following forms :
To send all output from one layer to all the nodes in another layer.

connect all output_1 **of** layer[1] **to** input_1 **of** layer[2]

This is referred to as "all to all" connection.
To send output from each node in one layer to input channels called input_1 of each node in another layer.

connect output_1 **of** layer[1] **to** input_1 **of** layer[2]

This is referred to as "one to one" connection.
To send a particular output from a node to a particular input of another node.

connect output_1 **of** layer[1][2] **to** input_1 **of** layer[2][5]

To send output from a layer to host.

send all output_2 **of** layer[3] **to host**

The connection statements used to connect sub networks are similar to the above statements except further qualification is added to distinguish layers and nodes of one sub network from another. For example, to connect all outputs from layer[3] of model_1 to all the inputs of layer[1] of model_2 the following statement is used.

connect all output of model_1.layer[3] **to** input of model_2.layer[1]

A node in this language is defined as follows:

```
node type output ( input[n],target[1] : inchan,
                output[1], error[1] : outchan,
                w[n], t[j] : local )
{
 switch mode {
 learn => if test(input[1..n], target[1]) then {
             ..
           else
           if test...

           fi
 recall =>    ..
          ..
  } end switch;
 }
```

The node/function body and the host use control flow statements similar to the ones found in the "C" language. The special commands available for controlling the networks are

ldvals - to load the links with data,

ldwts - to load weight variables with data,

save - to save a trained network,

test - to test the presence of data on input channels,

show - to display the weights and I/O values of a node (or nodes),

set mode - to select the mode of activation (ie- learning or recalling),
read - to read data,
file - to assign file names to file pointers.

A three layer back propagation model with three nodes in each layer can be specified in this language as follows.

```
Network consists of sub_nets {
                    model1;
                    } backprop;
definition of model1:
layers = 3 ->
{
layer[1] with 3 nodes of type input;
layer[2] with 3 nodes of type hidden;
layer[3] with 3 nodes of type output;
}
node interface {
node type input has
     input[1] : inchan,
     output[1]: outchan;
node type hidden has
     input[3], in_error[3] : inchan,
     output[1] : outchan,
     si[1], theta[1], w[3] : local;
node type output has
     input[3], target[1] : inchan,
     output[1], error[1] : outchan,
     si[1], theta[1],w[3] : local;
 }
connections ->
{connect all output of layer[1] to input of layer[2]
connect all output of layer[2] to input of layer[3]
connect all error of layer[3] to in_error of layer[2]
send all output of layer[3] to host
}
end model1;
host()
```

```
{
    file f1 = data_file;
    set mode = learn;
  for i = 1 to 3 {
   ldwts layer[2][i].si = read(f1,num);
   ldwts layer[2][i].theta = read(f1,num);
   ldwts layer[3][i].si = read(f1,num);
   ldwts layer[3][i].theta = read(f1,num);
   for j = 1 to 3 {
   ldwts layer[2][i].w[j] = read(f1,num);
   ldwts layer[3][i].w[j] := read(f1,num);
   }
   }
   t = 1;
   while (t <= 2000 ) {
   for p = 1 to 3 {
   for i = 1 to 3 {
   ldvals layer[1][i].input[1] = read(f1,x);
   }

   evaluate layer[1];
   evaluate layer[2];
   evaluate layer[3];
   evaluate layer[2];
   }
   t = t + 1;
   }
  for i = 1 to 3
   {
    print (layer[2][i].si);
    print (layer[2][i].theta);
    print (layer[3][i].si);
    print (layer[3][i].theta);
   for j = 1 to 3
    {
    print (layer[2][i].w[j],layer[3][i].w[j] );
    }
```

```
}
save backprop; } end host
```

Translating into NIL

In order to implement a network model in NIL, first we need the informations on the topology of the network. The network definition part is used to construct a table of models as shown in Table 8.

Names of Models
model_1
model_2

TABLE 8. Model_Names Table

After this, the definitions of each model in the network is scanned to generate informations about

1. the layer structure of the particular model,
2. the node interfaces for each type of nodes in the model.

These informations are used to build the Tables 9, 10, 11, and 12 where "vn" stands for vector number of a particular I/O channels.

Layer	Nodes	types
1	3	input
2	3	hidden
3	3	output

TABLE 9. Network Layers for Model1

In_Channels	vn	Out_Channels	vn	Weights	vn
input	1	output	1		0

TABLE 10. Node Interface for Type Input

In_Channels	vn	Out_Channels	vn	Weights	vn
input	1	output	1	si	1
input	1			theta	2
input	1			w	3
in_error	2			w	3
in_error	2			w	3
in_error	2				

TABLE 11. Node Interface for Type Hidden

In_Channels	vn	Out_Channels	vn	Weights	vn
input	1	output	1	si	1
input	1	error	2	theta	2
input	1			w	3
target	2			w	3
				w	3

TABLE 12. Node Interface for Type Output

After this, specific tables of input, output, and weight parameters have been constructed by giving pseudo-names for each I/O channels associated with each node in the first layer as as shown in Tables 13, 14, and 15.

In_Vectors	vn	Out_Vectors	vn	Weights	vn
input_1[1]	1	output_1[1]	1		0

TABLE 13. Node_1[1] - Node 1 of Layer 1

Now if we scan the first connect statement and produce the three node tables for layer 2 using the node interface table for the hidden unit type. At this stage we may or may not have a complete table. If we fail to have a complete table at this stage, we produce the partially updated tables for each node in this layer and go on to scan the next

In_Vectors	vn	Out_Vectors	vn	Weights	vn
input_1[2]	1	output_1[2]	1		0

TABLE 14. Node_1[2] - Node 2 of Layer 1

In_Vectors	vn	Out_Vectors	vn	Weights	vn
input_1[3]	1	output_1[3]	1		0

TABLE 15. Node_1[3] - Node 3 of Layer 1

connect statement and come back and complete these tables as and when we encounter the rest of the informations required to complete these tables. So in this case, we have three partial tables (Tables 16, 17, and 18).

In_Vectors	vn	Out_Vectors	vn	Weights	vn
output_1[1]	1	output_2[1]	1	si	1
output_1[2]	1			theta	2
output_1[3]	1			w	3
in_error	2			w	3
in_error	2			w	3
in_error	2				

TABLE 16. Node_2[1] - Node 1 of Layer 2

Having produced the partial node tables for nodes in layer 2, we proceed with the scanning of the next connect statement and the production of the node tables(Tables 19, 20, and 21) for layer 3.

Scanning of the last connect statement will enable us to complete the node tables for the layer 2 as shown in Tables 22, 23, and 24.

While building these node tables, a list of channel names vs pseudo names are also created for further reference while compiling the code for the host. These node tables are further updated by replacing the weight variables with the appropriate weight values given in the host part of the program. At the end of this stage, link statements are generated for the whole network by extracting informations from these tables one after

In_Vectors	vn	Out_Vectors	vn	Weights	vn
output_1[1]	1	output_2[2]	1	si	1
output_1[2]	1			theta	2
output_1[3]	1			w	3
in_error	2			w	3
in_error	2			w	3
in_error	2				

TABLE 17. Node_2[2] - Node 2 of Layer 2

In_Vectors	vn	Out_Vectors	vn	Weights	vn
output_1[1]	1	output_2[3]	1	si	1
output_1[2]	1			theta	2
output_1[3]	1			w	3
in_error	2			w	3
in_error	2			w	3
in_error	2				

TABLE 18. Node_2[3] - Node 3 of Layer 2

the other. This should produce the following link statements.

```
input([input_1[1]]) -> ([output_1[1]])
input([input_1[2]]) -> ([output_1[2]])
input([input_1[3]]) -> ([output_1[3]])
hidden([output_1[1],output_1[2],output_1[3]],[error[1],error[2],error[3]]:
        [#si],[#theta],[#w1,#w2,#w3] ) -> ([output_2[1]])
hidden([output_1[1],output_1[2],output_1[3]],[error[1],error[2],error[3]]:
        [#si],[#theta],[#w1,#w2,#w3] ) -> ([output_2[2]])
hidden([output_1[1],output_1[2],output_1[3]],[error[1],error[2],error[3]]:
        [#si],[#theta],[#w1,#w2,#w3] ) -> ([output_2[3]])
```

In_Vectors	vn	Out_Vectors	vn	Weights	vn
output_2[1]	1	output[1]	1	si	1
output_2[2]	1	error[1]	2	theta	2
output_2[3]	1			w	3
target_[1]	2			w	3
				w	3

TABLE 19. Node_3[1] - Node 1 of Layer 3

In_Vectors	vn	Out_Vectors	vn	Weights	vn
output_2[1]	1	output[2]	1	si	1
output_2[2]	1	error[2]	2	theta	2
output_2[3]	1			w	3
target_[2]	2			w	3
				w	3

TABLE 20. Node_3[2] - Node 2 of Layer 3

In_Vectors	vn	Out_Vectors	vn	Weights	vn
output_2[1]	1	output[3]	1	si	1
output_2[2]	1	error[3]	2	theta	2
output_2[3]	1			w	3
target_[3]	2			w	3
				w	3

TABLE 21. Node_3[3] - Node 3 of Layer 3

```
output([output_2[1],output_2[2],output_2[3]],[target_[1]]:
    [#si],[#theta],[#w1,#w2,#w3] ) -> ([output[1]],[error[1]])
output([output_2[1],output_2[2],output_2[3]],[target_[2]]:
    [#si],[#theta],[#w1,#w2,#w3] ) -> ([output[2]],[error[2]])
```

In_Vectors	vn	Out_Vectors	vn	Weights	vn
output_1[1]	1	output_2[1]	1	si	1
output_1[2]	1			theta	2
output_1[3]	1			w	3
error[1]	2			w	3
error[2]	2			w	3
error[3]	2				

TABLE 22. Node_2[1] - Node 1 of Layer 2

In_Vectors	vn	Out_Vectors	vn	Weights	vn
output_1[1]	1	output_2[2]	1	si	1
output_1[2]	1			theta	2
output_1[3]	1			w	3
error[1]	2			w	3
error[2]	2			w	3
error[3]	2				

TABLE 23. Node_2[2] - Node 2 of Layer 2

```
output([output_2[1],output_2[2],output_2[3]],[target_[3]]:
        [#si],[#theta],[#w1,#w2,#w3] ) -> ([output[3]],[error[3]])
```

Where #name stands for the numerical values assigned to the relevant weight variables by the "ldwts" commands in the host program. The rest of the host program is scanned and equivalent NIL program is generated and kept in a temporary file. This is then followed by the translation of the node functions used in the network. This is done by converting each "if test() then are compiled into NIL, the NIL version of the host program is copied at the end of link statements and functions.

In_Vectors	vn	Out_Vectors	vn	Weights	vn
output_1[1]	1	output_2[3]	1	si	1
output_1[2]	1			theta	2
output_1[3]	1			w	3
error[1]	2			w	3
error[2]	2			w	3
error[3]	2				

TABLE 24. Node_2[3] - Node 3 of Layer 2

The difficult part of translating any high level neural network language into NIL is the production of the node tables to specify the network topology. Once this is achieved, we see no major problems in completing the rest of the translation.

6.2 Assessment of NPS

Assessment of the Network Programming System, NPS, also involves the two main aspects of this thesis, programmability and portability.

6.2.1 Programmability

In the case of programmability, NPS was evaluated for presence of the following properties.

1. **usability** - how easy it is to code and execute neural network algorithms on a particular machine?
2. **facilities** - the main facility, library of functions and models, how helpful are they and how can they be improved?
3. **simplicity** - how simple is the design? is it easy to maintain and upgrade?

6.2.1.1 Usability Firstly, number of models were coded in NIL and executed and found that the overall activity of coding, compiling and executing models required no more efforts than compiling an ADA program in a UNIX environment. Considering the complexity of a neural network model in terms of its connectivity among nodes and the special control mechanism required by the manipulation part, we found it usable to an

acceptable degree. The existing facilities are found to be adequate for executing algorithms and conducting experiments except when it comes to examining the structure and state of the network on the screen. This is mainly due to the lack of a graphical display system and this proved to be a major handicap in this respect. However, it must be noted that we can only form a valid opinion when it is assessed with a graphic system. Unfortunately, it was not possible to do so because of building such a graphic system and interfacing with NIL is beyond the scope of this project.

6.2.1.2 Facilities Assessing the facilities provided by the NPS involved the execution of the basic commands available at this level. These are

1. **load** *-m file_name* - for loading a partially run network for execution on a particular hardware by specifying the hardware option in the parameter "m".

2. save *file_name* - to save a partially run program for running later. This command can also be used in the manipulation part of a NIL program.

3. **stop** - to stop a running network. Again this command can be used in the manipulation part of a NIL program.

4. go - to run a loaded network.

5. **run** *-m file_name* - to compile, load and run a source program file with options to execute in a particular machine.

6. exec *-m model_name* - to execute one of the standard models available in the model library. When executed, this command will initiate a series of questions regarding the size of each layer, tolerance value, and input and output patterns etc. The user responds by typing the required data. Once all the information is available, the system builds the appropriate network and proceeds with the compilation and subsequent execution process.

7. **abort** - to abandon the execution of the network. This command is also available in the NIL language.

and the use of already written functions stored in the functions library in new programs.

Execution of the basic commands produced the expected result The exec command for executing library models proved to be very restrictive. This is due to three reasons:

1. The overall topologies of the models are fixed. This means that the user cannot have a different topology other than the one provided except that the number of nodes in each layers can be varied;

2. Smoothing, threshold and other functions are also fixed and cannot be changed;

3. The format for outputting the data to the screen could not be altered since it was part of the manipulation program which is not accessible to the user.

The use of existing node functions in the library made programming easier and the program more compact. The save command for partially trained network proved to be valuable during the training of a Boltzmann Machine. The conclusion was that more flexibility and options must be provided when executing standard models from the library. The provision of the function library which consists of a collection of commonly used functions that describes various types of nodes proved to be valuable. Strictly speaking it does not belong to the Network Programming System and it is part of the NIL translator.

6.2.1.3 Simplicity It is believed that the NPS is simple and easy to maintain due to its design simplicity. Its amenability to changes in terms of upgrading is considered to be satisfactory because the programming system is held together by a simple command interpreter which calls the relevant subsystems to execute the commands.

6.2.2 Portability

This has been achieved by having NIL as its intermediate language system. This is further demonstrated by generating "C" code from NIL compiler to show that NIL can be mapped on a range of machines such as a network of Transputers and Sun workstations. This leads us to safely conclude that NPS offers the user the facility for porting neural network models and applications through its intermediate system.

Chapter 7

This chapter presents some concluding remarks. First, a summary of the thesis is presented. This is followed by a statement of the contributions made by this thesis to neural network computing research. Finally, a short list of ongoing work and potential future work is presented.

7. Summary

The main goals of this work has been the design and implementation of a neural network programming system which supports portability over a range of hardware, and programmability by facilitating the implementation of a range of neural network models. The system comprises:

- An intermediate Language - NIL - Which is a machine independent intermediate level neural network language that can implement a range of neural network models. In addition it is possible to implement semantic network and other network problems using this language.
- A Virtual Machine - Which encapsulate the machine independent model of the intermediate language in a simple and directly executable form(C-Machine).
- An Algorithms Library - Which contains a set of parameter driven popular neural network models which can be executed by the user by supplying the necessary parameters.
- A functions Library - Which contains a set of generalised node functions which can be used to build network models.
- A Neurocomputer Architecture - Which is based on a primitive processing element for executing neural network models and applications.

This thesis has mainly concentrated on the specification and the implementation of the intermediate language, NIL. The motivation and the design approach taken were already stated in chapter 4 and 5. The implementation strategies (both parallel and sequential) were discussed in chapter 5.

7.1 Contributions

In pursuing this research described in this thesis, a prime consideration has been to provide answers to the following questions:

- Is it possible to design a programming system that can support a range of target hardware?
- If so, can a range of neural network models be efficiently mapped and executed on this system?

It is believed that these questions have been answered positively. In the following sections, the contributions of this work to neural network computing research are summarised.

7.1.1 Portability

Since neural network computing is at its infancy, it is virtually impossible at this stage to identify the ideal form of computing system that can enhance portability in a general way. The reasons for this is that there is no obvious candidate architecture for neural network computing. The other equally important point is that the existing neural network algorithms are not ideally suited for building practical applications. This is because they take so long to learn (the reliable ones) and their adaptive capabilities are very very limited. What this thesis tried to do is to provide a software machinery capable of accommodating any progress in both of these directions(ie- hardware and algorithms) in the form of an intermediate language, NIL. What this language tries to do is, while representing neural network problems in a basic form it avoids explicitly specifying the mode of execution either in part or whole. It represents the network in such a way one can choose to execute sequentially or in parallel at the time of execution rather than at the time of specification. Again, the parallel execution can be either large grained parallelism or coarse grained parallelism depending whether each node is going to occupy a single processor or each guarded process going to occupy a single processor. The language also tried to tackle the problem of selective control over the network by providing simple commands for controlling and synchronizing.

7.1.2 Programmability

The model independent capability (i.e. programmability) of the system (in particular the language) has been demonstrated within the framework of the existing models. Again, it is impossible to judge its capability in this volatile period of research and it is even more difficult to judge its ability in dealing with the future models. But one

can safely assume that the system is model independent given the present state of research in this area. The features which give positive encouragement in this direction are in its ability to deal with semantic networks and its general computing ability.

7.2 Future Work

There are currently two mini projects underway at the Polytechnic of North London based on the intermediate language. The first one is the production of object oriented intermediate code from NIL by adding Objective C type of extensions to it. An exploratory study has been done on the design and implementation of an object oriented language and is found to be feasible. We also found that the underlying features of neural network has become more clear in object oriented form. It is also anticipated that the extensive graphics libraries available in Objective C would greatly simplify the building of a good user interface. There are two basic types of objects in this form, namely, nodes and links. A link can be generated in three basic form:

- **Single Linking** - which generates a single connection between any two nodes.
- **Layer Linking** - which connects all the outputs from nodes of a particular layer to all the nodes in another layer.
- **Random Linking** - which connects two layers of nodes randomly.

When translating a NIL program to an objective "C" program the output links associated with a given process are **pointers** to destination nodes. We generate links with the help of a **dictionary object class** which maintains a set of associations as ordered pairs of node identifiers and node pointers. These dictionaries, together with tables (derived from the NIL program) describing the network topology can then be used to create collections of destination node pointers to be associated with each guarded process. This approach works for the case involving *point to point* or *layer to layer* connections. With random connections the dictionaries are used together with a suitable random number generator. In the object oriented version each node contains a set of **synaptic input objects**. These may **pre-process** the input before writing it to the **data slot** associated with that synapse within that node. In addition each node contains a set of **guarded processes** and only one of these will be able to fire at any one time as in NIL. When such process fires it will send the appropriate messages to the synapses it knows about. Some basic components of a node object and their relationships are shown in the figure 31 below.

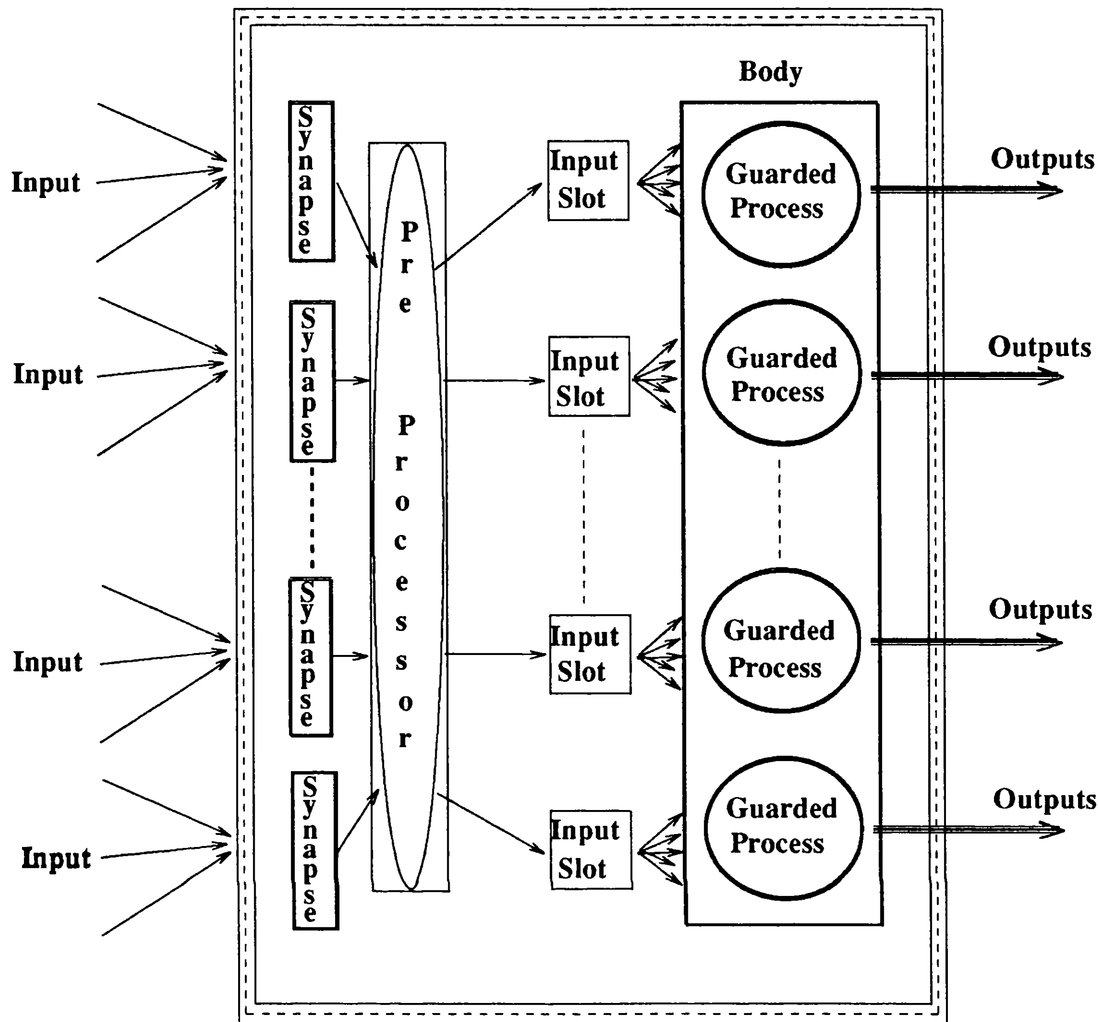

Figure 31. An Object-Oriented Model for a Node

The beauty of this approach is that it allows us to implement **functional links** as proposed by Sobajic [Soba88] in a direct and elegant way. Thus we have the potential to build complex systems for process control purposes.

In a particular neural network application the number of different classes of neurons (differing in terms of the different guarded processes they contain) is fairly small. Neurons from these various classes are connected together into functional networks. The process of translating from NIL to objective C (or to any other object oriented programming language) consists of creating the code for the individual classes of neurons and their associated synapses. The network specification part of NIL is then used to create and link these collections of objects together in the desired configuration. By associating suitable **display methods** with the objects in the system it is possible to construct user interfaces much more rapidly than with conventional programming methods.

The second project is concerned with the mapping of NIL on a network of transputers. This project is being supported by the Science and Engineering Research Council as part of their Transputer Loan Initiative (Project Number :- HB0001B). The aims of the project are as follows:

1. To translate NIL into Occam and map it on the transputer network. This seems to be very feasible in that NIL has similar features found in Occam and the task of translating merely consists of producing a set of macros.

2. To generate Transputer assembler code directly and develop an efficient mapping software to optimally map the network.

This work is currently being carried out at the Polytechnic of North London, where the author is a full time member of the academic staff.

Apart from this, NIL has been used in building communication network management systems using neural network techniques to control and manage networks [Elia89, Elia90]. In this work, various network management models were implemented to assess the potential of using self organising networks in this area. As part of this work, few models based on fuzzy logic were also coded in NIL. This work is still continuing and we hope to exploit the dynamic capabilities of NIL to produce a practical network management system eventually.

REFERENCES

[Aart86] E. Aarts and J. Korst, Combinatorial Optimization on a Boltzmann Machine, Submitted for publication in "Journal of Parallel and Distributed Computing".

[Adap88] "Adaptive Solutions for Tomorrow's Problems --- Today", Adaptics, 16776 Bernardo Centre Drive, Suite 110B, San Diego, CA 92128.

[Ange88] Angeniol. B, Texier. J, Mateu. J, "SLOGAN : An Object-Oriented Language for Neural Network Specification", nEuro'88, France, June,1988.

[Ange89] Angeniol. B, Treleaven "PYGMALION : Neural Network Programming & Applications", ESPRIT Conference, 1989(to appear).

[Anza87] ANZA User's Guide, Hect-Nielsen Corporation, Release 1.00, 1987.

[Bahr87] C. Bahr and D. Hammerstrom, ANNE - Another Neural Network Emulator

[Bava89] A. S. Bavan, "NIL-PLUS: A Neural Network Implementation Language", Proc. First Neural Computing Meeting, London, April 1989, The Institute of Physics.

[Bava90a] A. S. Bavan, "A Programming System for Implementing Neural Nets", - XI Sitges Conference on Neural Networks, 4 June - 7 June 1990, Barcelona, Spain.

[Bava90b] A. S. Bavan, "NPS: A Neural Network Programming System", in Proc. International Joint Conference on Neural Networks, June 17-21, 1990, Sandiego, California, pp143-148.

[Chol88] Chol. Ph and Muntean. T, "NEURAL : Towards an Occam Extension for Neurocomputers", nEuro'88, France, June88.

[Cruz87] C. Cruz, W. Hanson, J. Tam, "Neural Network Emulation Hardware Design Considerations", Proceeding of the First IEEE International Conference on Neural Networks, pp.III-427-434. 1987.

[Elia89] A. Eliasz S. Bavan J. Crowcroft, "Adaptive Network Management Using Neural Computing", - Third Race TMN Workshop, 30 August - 1 September 1989, London.

[Elia90] A. W. Eliasz, A. S. Bavan, and J. Crowcroft, "Approaches to Using Neural Computing Methods to Develop Adaptive Distributed Routing Algorithms", - Proc. Fourth Race TMN Conference, 14 - 16 November 1990, Dublin, pp218-232.

[Feld88] J. Feldman M. Fanty N. Goddard, Computing with Structured Neural Networks, IEEE March 1988.

[Fuku88] K. Fukushima, "A Neural Network for Visual Pattern Recognition", IEEE Computer, pp. 65-75, March, 1988.

[Gart87] S. Garth, "A Dedicated Computer for Simulation of Large Systems of Neural Nets", Texas Instruments Ltd., Manton Lane, Bedford, England.(Draft).

[Guts88] T. Gutshow, "Axon: The Researchers Neural Network Language", Presented in INNS'88, September88.

[Gros88] S. Grossberg and G. Carpenter, "The ART of Adaptive Pattern Recognition by a Self-Organizing Neural Network", IEEE March 1988.

[Hood87] G. Hood, "SNAIL : A Graphical Design System for Neural Networks", Proceeding Volume of the first IEEE International Conference on Neural Networks, 1987.

[Hopf82] J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities.", Proceedings of the National Academy of Science, USA. 1982.

[Hopf85] J. Hopfield and D. W. Tank, "Neural" Computation of Decisions in Optimization Problems, in Biological Cybernetics, Vol 5 pp 141 - 152, 1987.

[Hans87] W. Hanson , C. Cruz , J. Tam, "CONE - Computational Network Environment", Proceeding of the first IEEE International Conference on Neural Networks, 1987. Vol3 pp 531-538

[Hebb49] D. O. Hebb, "The Organisation of Behaviour"., Willey, New York. 1949.

[Hect87] R. Hect-Nielsen, "Kolmogorov's Mapping Neural Network Existence Theorem", Proceeding of the first IEEE International Conference on Neural Networks, 1987. Vol3 pp 11-14.

[Hint84] G. Hinton, T. Sejnowski, and D. Ackley, "Boltzmann Machine: Constraint Satisfaction Networks that Learn", Technical Report, CMU-CS-84-119.

[Hint85] G. Hinton, D. Ackley, T. Sejnowski, "A Learning Algorithms for Boltzmann Machines", Cognitive Science 9, pp147-169, 1985.

[Hint86] G. E. Hinton, and T. J. Sejnowski, "Learning and relearning in Boltzmann machines". In "Parallel distributed processing", Vol. 1, pp. 282-317. Cambridge, MA:MIT press. 1986.

[Hoar85] Hoare C A R, "Communicating Sequential Processes", Prentice-Hall international UK, LTD, 1985.

[Hech88] R. Hecht-Nielsen, "Neurocomputing : picking the human brain", IEEE Spectrum March 1988.

[John75] S.C. Johnson, "Yacc - Yet Another Compiler Compiler", Comp. Sc. Tech. Rep. 32, AT&T Bell Labs., Murray Hill, N.J., 1975.

[Josi87] G. Josin, "Combinations of Neural Systems for Particular Application Situations", Proceeding of the first IEEE International Conference on Neural Networks, 1987. Vol4 pp 517-524.

[Koho84] T. Kohonen, "Self-Organization and Associative Memory", Springer-Verlag, Berlin. 1984.

[Kohl88] Monika Kohle and Franzi Schonbauer, "CONDELA - A Language for Neural Networks", nEuro'88, France, June88.

[Kolo88] M. Kolonay Klimmasauskas "NeuralWorks Professional ; User's Guide", NeuralWorks Professional, NeuralWare Incorporated.

[Lesk75] M. E. Lesk, "Lex - a Lexical Analyser Generator", Comp. Sc. Tech. Rep. 39, AT&T Bell Labs., Murray Hill, N.J., 1975.

[Lipp87] Lippmann R, "An Introduction to Computing with Neural Nets", IEE ASSP Magazine, April 1987.

[McCa88] McCabe S. C., "A Kernel System for Parallel Numeric and Symbolic Computing", PhD Thesis, Dept. of Computer Science, Univ. of London, July 1988.

[MayD87] May D, "Occam2 language definition", INMOS, February 1987.

[Mins69] M. Minsky and S. Papert, "Perceptrons", MIT Press. 1969.

[Netw87] Netwurkz, Dair Computer.

[Pach88] Pacheco M, Bavan S, Lee M & Treleaven P, "A Simple VLSI Architecture for Neurocomputing", Proceedings of the International Neural Network Society, First Annual Meeting, Boston, Massachusetts., September 1988, pp 398.

[Pach91] Pacheco M., "A "Neural-RISC" Processor and Parallel Architecture for Neural Networks" , PhD Thesis to be submitted, Dept. of Computer Science, Univ. of London, 1991.

[Paik87] E. Paik, E. Gungner, J. Skrzypek, "UCLA SFINX - A Neural Network Simulation Environment", Proceeding of the first IEEE International Conference on Neural Networks, 1987. Vol 3 pp367-376

[Psal87] D. Psaltis, K. Wagner, and D. Brady, "Learning in Optical Neural Computers", IEEE: First International Conference on Neural Networks, Vol. 3, pp. 549-555. 1987.

[Rum86a] D. Rumelhart, G. Hinton, and J. McClelland, "Chapter 2: A General Framework for Parallel Distributed Processing.", pp. 45-76 in Parallel Distributed Processing , Explorations in the Microstructure of Cognition Volume 1: Foundations, ed. D. Rumelhart & J. McClelland, MIT Press. 1986.

[Rum86b] D. Rumelhart, D. Zipser, "Chapter 5: Feature Discovery by Competitive Learning"., pp. 151-193 in Parallel Distributed Processing. Explorations in the Microstructure of Cognition Volume 1: Foundations, ed. D. Rumelhart & J. McClelland, MIT Press. 1986.

[Rum86c] D. Rumelhart, G. Hinton, and R. Williams, " Chapter 8: Learning Internal Representations by Error Propagation.", pp. 318-362 in Parallel Distributed Processing. Explorations in the Microstructure of Cognition Volume 1: Foundations, ed. D. Rumelhart & J. McClelland, MIT Press. 1986.

[Rum86d] D. Rumelhart J. McClelland, "Parallel Distributed Processing , Explorations in the Microstructure of Cognition : Foundations", MIT Press Vol.1 and 2.

[Rum86e] "Parallel Distributed Processing, Explorations in the Microstructure of Cognition : Foundations", MIT Press Vol.3.

[Sabo88] G. Sabot, "The Paralation Model": Architecture-Independent Parallel Programming. The MIT Press, Cambridge, Massachusetts. 1988.

[SAIC88] "DELTA/SIGMA/ANSim", editorial, Journal of Neurocomputers, Vol 2, Number 1, 1988.

[Shep79] G. M. Shepherd, "The Synaptic Organisation of the Brain", Oxford University Press. 1979.

[Smit87] A. Smith, A Parallel PDP Network Simulator, Internal King's college London Report, June 1987.

[Soba88] D. Sobajic, "Neural Nets for Control Power Systems", PhD. Thesis, Computer Science Department, Case Western Reserve University, Cleveland, Ohio, USA. 1988.

[Souc88] B. Soucek and M. Soucek, "Chapter 12: Artificial Neural Systems or Neurocomputers" pp. 245-276, in Neural and Massively Parallel Computers, John Wiley & Sons, 1988.

[Test88] C. Testa D. Pike S. Garth, "NETSIM : Software Environment For a Parallel Neural Network Simulator", Internal Note, Cambridge.1988

[Trel88a] P. C. Treleaven, M. Recce, "Programming Languages for Neural Computers", European Seminar on Neural Computing, London(IBC), Feb. 1988.

[Trel88b] P. C. Treleaven, "Neurocomputers: invited tutorial", Proc. Neuro-Nimes, 1988.

[Wass88] P. Wasserman, T. Schwartz, "Neural Networks Part2 : What are they and why everybody so interested in them now ?", IEEE Expert Systems, 1988.

[Zips86] D. Zipser, D. Rabin, "P3 : A Parallel Network Simulating System", in Parallel Distributed Processing by D.E. Rumellhart, *et al*, Vol 1 Chapter 13, 1986.

APPENDIX A - Syntax Definition of NIL

```
        prog ::= begin netblock funcblock manp end

------------------------------------------------------------------------
========================================================================
netblock ::= linkstat { linkstat }*
linkstat ::= slink | replink | structlink
slink ::= fun_name ( {input_list} {:wt_list})
                         -> ( out_list )
replink ::= rep [integer] fun_name (vec_list
                            {:wt_vec})
                          -> (vec_list)
vec_list ::= vec {,vec}*
vec ::= ident [ ident ]
wt_vec ::= int * rnd(low - high)
low ::= high ::= int
input_list ::= var_list { , var_list }*
out_list ::= var_list { , var_list }*
var_list ::= [ list_v { ; list_v }* ]
list_v ::= ident {, ident }*
wt_list ::= :val_list {, val_list}*
val_list ::= [ list_c {; list_c }* ]
list_c ::= constant {, constant }*
structlink ::= construct(
        [rep_par name(input_par) -> (output_par)
        {:[rep_par name(input_par) -> (output_par)}
rep_par ::= index_var = number of replications
------------------------------------------------------------------------
========================================================================
funcblock ::= func { func }*
func ::= header [declarations] "{" body "}"
header ::= fun fun_name ( input_par ) -> ( output_par)
input_par ::= in_array_list {ival: wt_array_list}
out_par ::= out_array_list
in_array_list ::= array_list
wt_array_list ::= array_list
out_array_list ::= array_list
array_list ::= ident [dim]{, ident[dim]}*
dim ::= ident { , ident }
body ::= guarded_process { guarded_process }*
guarded_process ::= input_condition => "{" statements "}"
input_condition ::= ident[index_range
                          {, ident index_range }*
index_range ::= [ range_comp {, range_comp }]
range_comp ::= index .. index | index
index ::= int | ident
```

```
statements ::= stmnt { stmnt }*
stmnt ::= if | do | assignment | skip ;
if ::=  if (boolean_expression) -> statements
                                { ->-> statements } fi
do ::=  do (boolean_expression) -> statements  od
assignment ::= variable  := expression ;
boolean_expression ::= expression rel_op expression
                  | boolean_expression { logic_op
                                boolean_expression }
expression ::=  term aop expression | term
term ::= factor mop term | factor
factor ::= variable | unsigned_const | ( expression )
variable ::= ident | ident [subscrpt]
subscrpt ::= elemnt | elemnt : elemnt
elemnt ::=  int | ident
rel_op ::= > | >= | < | <= | <> | =
logic_op ::=  and | or |  xor | not
aop ::= + | -
mop ::= * | /
-----------------------------------------------------------------------
==============================================================================
manp ::= begin declarations mstatblock end
mstatblock ::= ctstmnt { ctstmnt }*
ctstmnt ::= if | do | assignment | skip;
            | readstat | lnkop | input
            | output |get | getwt
            | ldconst | save | load
            | rmv | run_net; | stop; | go
readstat  ::=  readst par
par ::= all | nodelabel {, nodelabel}*;
lnkop ::=  operation lnkname (operation_par)
operation ::= delete | join
operation_par ::= del_par | join_par
del_par ::= nodelabel | ,nodelabel
join_par ::= nodelabel.input_array_elmnt
input ::=  input lnkname = value
                                {,lnkname = value}*;
get ::= get variable = linkname
                              {,variable = linkname}*;
getwt ::= get variable = wt_array_elmnt
                        {,variable = wt_array_elmnt}*;
ldconst ::= ldconst wt_array_elmnt = value
                        {,wt_array_elmnt = value}*;
output ::= output {string}variable {,variable}*;
string ::= "letter|digit {letter|digit}*
rmv ::= rmv nodelabel;
elmnt ::= variable name | strings
save ::= save filename;
load ::= load -m filename;
```

m ::= 1 | 2
sourcenode ::= *nodelabel*
destnode ::= *nodelabel*
nodelabel ::= nde[*int*|*ident*]
declarations ::= {*type variable_list*}+
type ::= int | real
variable_list ::= *variable* {,*variable*}*;
integer ::= integer number
ident ::= name

Heb/Hopfield model

```
begin
rep[5] lay1(iput[i]) -> (out[i]) /*link statements */
lay2([out[1],out[2],out[3],out[4],out[5]],[s[1],s[2]] :
          [0.0,0.0,0.0,0.0,0.0],[1.0]) -> ([sig[1]],[resl[1]])
lay2([out[1],out[2],out[3],out[4],out[5]],[s[1],s[2]] :
          [0.0,0.0,0.0,0.0,0.0],[2.0]) -> ([sig[2]],[resl[2]])
lay2([out[1],out[2],out[3],out[4],out[5]],[s[1],s[2]] :
          [0.0,0.0,0.0,0.0,0.0],[3.0]) -> ([sig[3]],[resl[3]])
lay2([out[1],out[2],out[3],out[4],out[5]],[s[1],s[2]] :
          [0.0,0.0,0.0,0.0,0.0],[4.0]) -> ([sig[4]],[resl[4]])
lay2([out[1],out[2],out[3],out[4],out[5]],[s[1],s[2]] :
          [0.0,0.0,0.0,0.0,0.0],[5.0]) -> ([sig[5]],[resl[5]])

fun lay1(x[i]) -> (y[i])  /*node function to input layer */
int i;
real x[2],y[2];
{
 x[i] => {y[i] := x[i];}
}
/*node function for output layer */
fun lay2(y[i], lr[j] ival:w[i], index[m]) ->(signal[m],result[m])
int i,j,m,k,ii;
real y[7],lr[4],index[2],signal[2],result[2],w[10],net;
{
y[1..i], lr[1] => {          /*learn*/
            k:= index[1];
            ii:=1;
            do (ii<=i) ->
               if(ii<>k) ->
                  w[ii] := w[ii] + y[k]*y[ii];
                 ->->
                    w[ii] := 0.0;
                    fi
                    ii:=ii+1;
                    od
                  signal[1] := 1.0;
             }
y[1..i], lr[2] => {          /*recall*/
            k:= index[1]; ii:=1;
            net := 0.0;
            do (ii<=i) ->
            net := net + w[ii]*y[ii];
            ii:= ii+1;
            od
```

```
            if (net>= 0.0) -> result[1] := 1.0;
               ->-> result[1] := 0.0-1.0;
               fi
            }
}
begin     /*manipulation part*/
real a, b;
real p,q,r,s,t;
a :=1.0;
b:=0.0-1.0;
ldconst nde[6].w[1] = 0.11,  /*load weights for node 6*/
      nde[6].w[2] = 0.21,
      nde[6].w[3] = 0.12,
      nde[6].w[4] = 0.04,
      nde[6].w[5] = 0.13;

ldconst nde[8].w[1] = 0.15,
      nde[8].w[2] = 0.06,
      nde[8].w[3] = 0.17,
      nde[8].w[4] = 0.02,
      nde[8].w[5] = 0.11;

input iput[1] = b,        /*put values on input channels*/
    iput[2] = a,
    iput[3] = a,
    iput[4] = a,
    iput[5] = b,
    s[1] = a;           /*signal for learning */
run_net;                /* run the network */
get p = sig[1],         /*has it learnt */
   q = sig[2],
   r = sig[3],
   s = sig[4],
   t = sig[5];
                  /*yes input the next vector*/
input iput[1] = b,
    iput[2] = b,
    iput[3] = a,
    iput[4] = a,
    iput[5] = a,
    s[1] = a;
run_net;
get p = sig[1],
   q = sig[2],
   r = sig[3],
   s = sig[4],
   t = sig[5];

input iput[1] = a,
```

```
     iput[2] = a,
     iput[3] = b,
     iput[4] = a,
     iput[5] = a,
     s[1] = a;
run_net;
get p = sig[1],
   q = sig[2],
   s = sig[3],
   r = sig[4],
   t = sig[5];


output " ";

getwt p = nde[6].w[1],     /*read the weights */
     q = nde[6].w[2],
     r = nde[6].w[3],
     s = nde[6].w[4],
     t = nde[6].w[5];

output "WEIGHTS FOR NODE 6";
output "w1 =",p, " w2 =",q," w3 =",r," w4 =",s," w5 =",t;


getwt p = nde[7].w[1],
     q = nde[7].w[2],
     r = nde[7].w[3],
     s = nde[7].w[4],
     t = nde[7].w[5];

output "WEIGHTS FOR NODE 7";
output "w1 =",p, " w2 =",q," w3 =",r," w4 =",s," w5 =",t;


getwt p = nde[8].w[1],
     q = nde[8].w[2],
     r = nde[8].w[3],
     s = nde[8].w[4],
     t = nde[8].w[5];

output "WEIGHTS FOR NODE 8";
output "w1 =",p, " w2 =",q," w3 =",r," w4 =",s," w5 =",t;


getwt p = nde[9].w[1],
     q = nde[9].w[2],
     r = nde[9].w[3],
     s = nde[9].w[4],
```

```
    t = nde[9].w[5];


output "WEIGHTS FOR NODE 9";
output "w1 =",p, " w2 =",q," w3 =",r," w4 =",s," w5 =",t;

getwt p = nde[10].w[1],
    q = nde[10].w[2],
    r = nde[10].w[3],
    s = nde[10].w[4],
    t = nde[10].w[5];

output "WEIGHTS FOR NODE 10";
output "w1 =",p, " w2 =",q," w3 =",r," w4 =",s," w5 =",t;

              /* Recall starts */
input iput[1] = b,
    iput[2] = a,
    iput[3] = a,
    iput[4] = a,
    iput[5] = b,
    s[2] = a;      /*recall signal*/
run_net;
get p = resl[1],
   q = resl[2],
   r = resl[3],
   s = resl[4],
   t = resl[5];
output "   ";
output "INPUT  -->   OUTPUT";
output b, "--> ",p;
output a, "--> ",q;
output a, "--> ",r;
output a, "--> ",s;
output b, "--> ",t;


input iput[1] = b,
    iput[2] = b,
    iput[3] = a,
    iput[4] = a,
    iput[5] = a,
    s[2] = a;
run_net;
get p = resl[1],
   q = resl[2],
   r = resl[3],
   s = resl[4],
   t = resl[5];
```

```
output "INPUT -->  OUTPUT";
output b, "--> ",p;
output b, "--> ",q;
output a, "--> ",r;
output a, "--> ",s;
output a, "--> ",t;


input iput[1] = a,
    iput[2] = a,
    iput[3] = b,
    iput[4] = a,
    iput[5] = a,
    s[2] = a;
run_net;
get p = resl[1],
   q = resl[2],
   r = resl[3],
   s = resl[4],
   t = resl[5];

output "INPUT -->  OUTPUT";
output a, "--> ",p;
output a, "--> ",q;
output b, "--> ",r;
output a, "--> ",s;
output a, "--> ",t;


end
end
```

```
begin

inlayer([iv[1]], [sig[1]]) -> ([ou[1]]) /*link statements*/
inlayer([iv[2]], [sig[1]]) -> ([ou[2]])

hlayer([ou[1],ou[2]], [erbk[1]]
:[3.1,2.1],[0.0,0.0],[0.0],[2.5],[0.1])
                        -> ([out[1]], [sig[1]])

olayer([iv[1],iv[2]], [out[1]], [e_op[1]],
[rcl[1]]:[1.2,2.6,1.27],[1.7],[0.1])
                        -> ([erbk[1]], [result[1]])



fun inlayer(in[i], sigl[i]) -> (ip[i])  /*input layer node*/
int i;
real in[3],sigl[2],ip[2];
{
in[i], sigl[i] => {
               ip[1] := in[1];
              }
}
 /* hidden layer node */
fun hlayer(in[m], err[n] ival: w1[m], l_in[m], l_op[n], theta[n],si[n])
                           -> (op[n], rsig[n])
int m, n;
real in[4], err[2],w1[5],l_in[4],l_op[2],theta[2],
si[2],op[2],rsig[2],this_er,temp;
{
in[1,2] => {    /*calculate output*/
           temp := w1[1] * in[1] + w1[2] * in[2];
           l_in[1] := in[1];
           l_in[2] := in[2];
           l_op[1] := 1/(1+exp(0- temp - theta[1]));
           op[1] := l_op[1];
           }
err[1] => {    /*adjust weights etc */
          this_er := l_op[1] *(1-l_op[1]) * err[1];
          theta[1] := theta[1] + (si[1] * this_er);
          w1[1] := w1[1] + (si[1] * this_er * l_in[1]);
          w1[2] := w1[2] + (si[1] * this_er * l_in[2]);
          rsig[1] := 1;
         }
}
```

```
fun olayer(in[l], hi[m], e_op[m], recall[m] ival: w2[n], theta[m], si[m]) ->
                        (errbk[m], out[m])
int l, n, m;
real in[3],hi[2],e_op[2],recall[2],w2[4],theta[2],
    si[2],errbk[2],out[2],temp, c_op,err;
{
 in[1..l],hi[m], e_op[m] => {
                /*calc output,adjust wts, send back error val*/
         temp := w2[1] * in[1] + w2[2] * hi[1] + w2[3] * in[2];
         c_op := 1/(1+exp(0-temp - theta[1]));

          err := c_op * (1-c_op)*(e_op[1] - c_op);
         theta[1] := theta[1] + (si[1] * err);

          w2[1] := w2[1] + (si[1] * err * in[1]);
          w2[2] := w2[2] + (si[1] * err * hi[1]);

          w2[3] := w2[3] + (si[1] * err * in[2]);
          errbk[1] := err * w2[2];
          }

 in[1..l], hi[m], recall[m] =>  { /*recall-output */
         temp := w2[1] * in[1] + w2[2] * hi[1] + w2[3] * in[2];
         out[1] := 1/(1+exp(0-temp - theta[1]));
         }
}
begin     /*manipulation part*/
int p,i;
real w1, w2, w3, theta;
real data[5:5],ex_op[5],y;
data[1:1] := 0.0; data[1:2] := 0.0; ex_op[1] := 0.0;
data[2:1] := 0.0; data[2:2] := 1.0; ex_op[2] := 1.0;
data[3:1] := 1.0; data[3:2] := 0.0; ex_op[3] := 1.0;
data[4:1] := 1.0; data[4:2] := 1.0; ex_op[4] := 0.0;
input sig[1] = 1.0;
i := 1;
do (i<=4000) ->
          p := 1;
          do (p <= 4) ->
                input iv[1] = data[p:1],
                    iv[2] = data[p:2],
                    e_op[1] = ex_op[p];
                    run_net;
                p := p + 1;
                od
                i := i + 1;
                od
```

```
        getwt w1 = nde[3].w1[1],
            w2 = nde[3].w1[2],
            theta = nde[3].theta[1];
        output " ";
        output "RESULTS";
        output "W1 = ", w1;
        output "W2 = ", w2;
        output "theta = ", theta;
        output " ";

        getwt w1 = nde[4].w2[1],
            w2 = nde[4].w2[2],
            w3 = nde[4].w2[3],
            theta = nde[4].theta[1];
        output " ";
        output "RESULTS";
        output "W1 = ", w1;
        output "W2 = ", w2;
        output "W3 = ", w3;
        output "theta = ", theta;
        output " ";

        output "input   input  ==>  output  ";
        p := 1;
        do (p <= 4) ->
              input iv[1] = data[p:1],
                        iv[2] = data[p:2],
                         rcl[1] = 1.0;
              run_net;
              get y = result[1];
              output  data[p:1],"   ", data[p:2], " ==> ", y;
              input sig[1] = 1.0;
              p := p + 1;
              od
end
end.
```

Kohonen feature map for a 3x3 map

```
begin          /* links for feature map nodes */
onode([invec[1],invec[2],invec[3]],[learn],[ajust1],[recal]
:[0.926, 0.908, 0.188] ,[0.0],[0.4])->([dist[1]],[otp1])

onode([invec[1],invec[2],invec[3]],[learn],[ajust1],[recal]
:[0.918, 0.953, 0.071],[0.0],[0.4])->([dist[2]],[otp2])

onode([invec[1],invec[2],invec[3]],[learn],[ajust1],[recal]
:[0.617,0.340,0.404],[0.0],[0.4])->([dist[3]],[otp3])

onode([invec[1],invec[2],invec[3]],[learn],[ajust1],[recal]
:[0.242,0.645, 0.991],[0.0],[0.4])->([dist[4]],[otp4])

onode([invec[1],invec[2],invec[3]],[learn],[ajust1],[recal]
:[0.633,0.379, 0.492],[0.0],[0.4])->([dist[5]],[otp5])

onode([invec[1],invec[2],invec[3]],[learn],[ajust1],[recal]
:[0.724,0.099, 0.296],[0.0],[0.4])->([dist[6]],[otp6])

onode([invec[1],invec[2],invec[3]],[learn],[ajust1],[recal]
:[0.923,0.983, 0.168],[0.0],[0.4])->([dist[7]],[otp7])

onode([invec[1],invec[2],invec[3]],[learn],[ajust1],[recal]
:[0.956,0.570, 0.876],[0.0],[0.4])->([dist[8]],[otp8])

onode([invec[1],invec[2],invec[3]],[learn],[ajust1],[recal]
:[0.816,0.132, 0.206],[0.0],[0.4])->([dist[9]],[otp9])


/* result node */
resnode([otp1,otp2,otp3,otp4,otp5,otp6,otp7,otp8,otp9]) -> ([clout])




fun onode(iput[n],lrn[m],adjwt[m],rcl[m] ival: w[n],ss[m],
si[m])
               -> (d[m], op[m])
int    n,m,i;
real iput[5],lrn[2],adjwt[2],rcl[2],w[7],ss[2],si[2],d[2],op[2];
{

lrn[1],iput[1..n] => {      /*learn*/
          d[1] := 0.0;
```

```
            i := 1;
            do (i<=n) ->
                  ss[1] := iput[i] - w[i];
                  d[1] := d[1] + (ss[1]*ss[1]);
                  i:=i+1;
                  od
            }
adjwt[1] => {           /*adjust weights for neiborhood*/
            if(adjwt[1] > 0) ->
            i:=1;
            do (i<=n) ->
                  w[i] := w[i] + si[1] * ss[1];
                  i:=i+1;
                  od
            fi
                  si[1] := si[1] - 0.1;
            }
rcl[1],iput[1..n] => {    /* recall */

                  i:=1;
                  op[1] := 0.0;
                  do (i<=n) ->
                  op[1] := op[1] + iput[i] * w[i];
                  i := i+1;
                  od
                  }

}
fun resnode(iin[nn]) -> (ooput[m])
int    nn,m,i;
real   iin[12], ooput[2];
{
iin[1..nn] => {      /* output */
            ooput[1] := 0;
            i:=1;
            do (i<=nn) ->
            ooput[1] := ooput[1] + iin[i];
            i:=i+1;
            od
            }
}
begin
int n,jj,i,learn, kk,p, xx,r,r1,r2,c,c1,c2,j,jmn;
real d[12], data[4:4],x,a[10],yes;

data[1:1] := 1.0; data[1:2] := 1.0; data[1:3] := 0.0;
data[2:1] := 0.0; data[2:2] := 0.0; data[2:3] := 1.0;
data[3:1] := 1.0; data[3:2] := 0.0; data[3:3] := 0.0;
```

```
learn := 1;
kk := 3;

do (learn = 1) ->
 kk := kk - 1;
 p := 1;
 do (p <= 3) ->
input invec[1] = data[p:1],
    invec[2] = data[p:2],
    invec[3] = data[p:3],
    learn = 1.0;
run_net;

i:=1;

      get d[1] = dist[1];
      get d[2] = dist[2];
      get d[3] = dist[3];
      get d[4] = dist[4];
      get d[5] = dist[5];
      get d[6] = dist[6];
      get d[7] = dist[7];
      get d[8] = dist[8];
      get d[9] = dist[9];

     n := 9;

     do (i<= 9) ->

      output "dj  =", d[i];
      i:=i+1;
      od
     j := 1; jmn := 1;
     do (j <= n - 1) ->
       jj := j + 1;   /* calculate distance */
       if (d[jmn] > d[jj]) -> jmn := j + 1; fi
       j := j + 1;

      od
        j:=1;
      do (j <= 9) ->
      x := jmn * 1.00; /* determine neiborhood nodes */
      x := x/3.0;
      if (x <= 1.0) -> r := 1; fi
      if (x > 1.0) -> if (x <= 2.0) -> r := 2; fi fi
      if (x > 2.0) -> if (x <= 3.0) -> r := 3; fi fi

       xx := jmn - (r - 1)*3;
       if (xx = 1) -> c := 1;fi
```

```
     if (xx = 2)-> c := 2;fi
     if (xx = 3)-> c := 3;fi


     r1 := r;
     c1 := c;

    x := j * 1.00;
    x := x/3.0;
    if (x <= 1.0) -> r := 1; fi
    if (x > 1.0) -> if (x <= 2.0) -> r := 2; fi fi
    if (x > 2.0) -> if (x <= 3.0) -> r := 3; fi fi

     xx := j - (r - 1)*3;
     if (xx = 1) -> c := 1; fi
     if (xx = 2) -> c := 2; fi
     if (xx = 3) -> c := 3; fi


     r2 := r;
     c2 := c;

   r := r1 - r2;
   c := c1 - c2;
   if (r < 0) -> r := 0 - r; fi
   if (c < 0) -> c := 0 - c; fi
   yes := 0.0;
   if (r <= kk)-> yes := 1.0; fi
   if (c <= kk)-> yes := yes * 1.0; ->-> yes := 0.0;fi
   a[j] := yes;
   j := j + 1;
   od
   input ajust1 = a[1], /*adjust weights */
       ajust2 = a[2],
       ajust3 = a[3],
       ajust4 = a[4],
       ajust5 = a[5],
       ajust6 = a[6],
       ajust7 = a[7],
       ajust8 = a[8],
       ajust9 = a[9];

run_net;

  p := p + 1;
  od
  if (kk = 0) -> learn := 0; fi
  od
```

```
p := 1;
do (p <= 3) ->          /* recall */
   input recal = 1.0,
   invec[1] = data[p:1],
   invec[2] = data[p:2],
   invec[3] = data[p:3];

   run_net;


   get d[1] = clout;

   output " ";
   output " result ",d[1];

  p := p + 1;

 od
end
end
```

Inputs

1.000000 1.000000 0.000000
0.000000 0.000000 1.000000
1.000000 0.000000 0.000000

Initial Weights

w1	w2	w3
0.926	0.908	0.188
0.918	0.953	0.071
0.617	0.340	0.404
0.242	0.645	0.991
0.633	0.379	0.492
0.724	0.099	0.296
0.923	0.983	0.168
0.956	0.570	0.876

0.816 0.132 0.206

Actual output.....

```
        WEIGHTS
   w1------->w2------->w3

 1.065421 1.047421 0.327421
 1.131965 1.166965 0.284964
 0.618804 0.341804 0.405804
 -0.130186 0.272814 0.618814
 0.578737 0.324737 0.437737
 0.794612 0.169612 0.366612
 1.075164 1.135164 0.320164
 0.657083 0.271083 0.577083
 0.943953 0.259953 0.333953


 result  11.725105

 result  3.672553

 result  6.735553
```

Dynamic Properties of the NIL

The program given below illustrates the dynamic properties of NIL by implementing a three layer network and deleting links,removing a node, and creating a link(join). The first layer consists of two input nodes which receives inputs(in[1] and in[2]) from host and send these inputs to all the two nodes in the second layer(out1[1] and out1[2]). The nodes in the second layer takes these inputs from the first layer and sum them before sending them(out2[1] and out2[2]) to a single node in the third layer. This node in the third layer sums these inputs and produces it as output(res[1]). The nodes 1 and 2 of the first layer use the function "one". The nodes 3 and 4 of the second layer use the function called "two". The node 5 in the third layer uses the function called three.

```
begin

one([in[1]])->([out1[1]])
one([in[2]])->([out1[2]])
two([out1[1],out1[2]]) -> ([out2[1]])
two([out1[1],out1[2]]) -> ([out2[2]])
three([out2[1],out2[2]]) -> ([res[1]])

fun one(x[i]) -> (y[i])
int i;
real x[2], y[2];
{
  x[i] => {
         y[1] := x[1];
         }
}


fun two(p[j]) -> (q[i])
int i,j, m;
real p[4], q[4], t;
{
  p[1..j] => {
          m := 1;
          t := 0.0;
         do (m <= j) ->
           t := t + p[m];
           m := m + 1;
           od
         q[1] := t;
         }
}


fun three(r[j]) -> (s[i])
```

```
int i,j, m;
real r[4], s[4], t;
{
   r[1..j] => {
            m := 1;
            t := 0.0;
          do (m <= j) ->
            t := t + r[m];
            m := m + 1;
            od
          s[1] := t;
          }
}

begin
real z;

input in[1] = 2.0,    /*load inputs*/
    in[2] = 3.0;

run_net;            /*run the network*/
               /*get the result*/
get z = res[1];
output "  ";
output "result is ", z;  /*print it */

input in[1] = 2.0,    /*load inputs again */
    in[2] = 3.0;

delete out1[2](,nde[3]); /*delete link from node 2 to 3 */
run_net;              /*run the network */
get z = res[1];
output "result after deletion of out1[2] is ", z;  /*print the result*/

join out1[1](nde[5],r);  /*join link out1[1] to node 5*/

input in[1] = 2.0,    /*load inputs again */
    in[2] = 3.0;

run_net;
get z = res[1];
output "result after joining of out1[1] to node 5 is ", z;
                                             /*print the result*/
input in[1] = 2.0,      /*load inputs a gain */
    in[2] = 3.0;
rmv nde[1];            /*remove first input node */
run_net;              /*run the network */
get z = res[1];
output "result after removing node 3 is ", z;  /*print result*/
```

```
end
end.
```

The output from the program is :-

result is 10.000000

result after deletion of out1[2] is 7.000000

result after joining of out1[1] to node 5 is 9.000000

result after removing node 3 is 7.000000

APPENDIX C - Comparison of NIL with BIF

A Back-propagation Network Model in BIF

```
#include "userfx.h"
#include <math.h>
#define TOP     1
#define BOTTOM  0

/* cnlists are lists of the cn indices present on the local HN */
static cnlist *cns_in, *cns_hid, *cns_out;
/* user's network functions */
void Input_site_fx(), Other_site_fx(), Assign_to_outsite(),
    Squash();
void Calculate_output_error(), Calculate_other_error();
void Init_user_fx1();
short Activate();
/* simulation parameters accessed by the user code */
extern cycle_params cp;
/* buffer for iPSC log messages */
char sb[80];

/*****************************************************/
/* Init_user_fx1: Called once by ANNE to init user data */
/*****************************************************/
void Init_user_fx1()
{
      int i;

      cns_in = Get_cnlist("input");
      cns_hid = Get_cnlist("hidden");
      cns_out = Get_cnlist("output");
      Update_group_weights("hidden", TOP);
      Update_group_weights("hidden", BOTTOM);
      /* no fault simulation */
      faulting = OFF;
}
/*****************************************************/
/* User_fx1: for back-prop network                   */
/* This procedure describes the "script" modelling the */
/* network's behaviour for a single network cycle.   */
/*****************************************************/
void User_fx1()
{
      int  cnx, i, l;
      float ferr, upd, fout;

      /** FORWARD PASS **/
```

```
    /* get an input vector from the host */
    Input_site_fx();
    Send_group_output("input", TOP);


    /* sum weighted inputs at the hidden layer, activate,
                                and send output */
    Other_site_fx(cns_hid, BOTTOM, 1);
    Squash(cns_hid, BOTTOM);
    Assign_to_outsite(cns_hid, TOP);
    Send_group_output("hidden", TOP);

    /* sum weighted inputs at output layer, activate, send
                                      to host */
    Other_site_fx(cns_out, BOTTOM, 1);
    Squash(cns_out, BOTTOM);
    Assign_to_outsite(cns_out, TOP);
    Send_net_output();
    /** BACKWARD PASS **/
    Calculate_output_error(cns_out);
    for(i = 0; i < cns_out->numcns; i++) {
      cnx = cns_out->cns[i];
      SITEVALUE(cnx, BOTTOM) = ERROR(cnx);
    }
    Send_group_output("output", BOTTOM);

    /* received error from output layer */
    /* sum weighted error signals */
    Other_site_fx(cns_hid, TOP, 0);
    /* calculate this layer's error and send down */
    Calculate_other_error(cns_hid);
    for (i = 0; cns_hid->numcns; i++) {
      cnx = cns_hid->cns[i];
      SITEVALUE(cnx, BOTTOM) = ERROR(cnx);
    }
    Send_group_output("hidden", BOTTOM);

    /* send new weights to other end of links */
    Update_group_weights("hidden", TOP);

    Other_site_fx(cns_in, TOP, 0);
    Calculate_other_error(cns_in);
    /* send new weights to other end of links */
    Update_group_weights("input", TOP);
} /* end user_fx1 () */
/*******************************************************/
/* Calculate_other_error                              */
/*******************************************************/
void Calculate_other_error(cn1)
```

```
cnlist *cn1;
{
    int cnx;
    short i, local_error;
    float ferr, fout;

    for (i = 0; i < cn1_numcns; i++) {
     cnx = cn1->cns[i];
     fout = SHORT_TO_FLOAT(OUTPUT(cnx));
     ferr = SHORT_TO_FLOAT(SITEVALUE(cnx, TOP)) * DERIV(fout);
     ERROR(cnx) = FLOAT_TO_SHORT(ferr);
    }
}
/******************************************************/
/* Calculate_output_error                              */
/******************************************************/
void Calculate_output_error(cn1)
cnlist *cn1;
{
    int cnx;
    short i, local_error;
    float ferr, fout;

    for (i = 0; i < cn1->numcns; i++) {
     cnx = cn1->cns[i];
     local_error = targetvals[cnx] - OUTPUT(cnx);
     fout = SHORT_TO_FLOAT(OUTPUT(cnx));
     ferr = SHORT_TO_FLOAT(local_error) * DERIV(fout);
     ERROR(cnx) = FLOAT_TO_SHORT(ferr);
    }
}
/******************************************************/
/* Input_site_fx                                       */
/******************************************************/
void Input_site_fx()
{
    int cnx;
    short i, siteval;

    /* global inputs are ready and weighting in site value */
    if (cns_in != (cnlist *)NULL) {
     for (i = 0; i < cns_in->numcns; i++) {
        cnx = cns_in->cns[i];
        /* output function is identity */
        OUTPUT(cnx) = SITEVALUE(cnx, 0);
        if (faulting) (void) flt_cn(cnx, &OUTPUT(cnx));
     }
    }
}
```

```
/********************************************************/
/* Other_site_fx                               */
/********************************************************/
void Other_site_fx(cn1,site_index,direc)
cnlist *cn1;
short site_index;
int direc;
{
        int cnx, siteval, i;

        if (cn1 != (cnlist *)NULL) {
          for (i = 0; i < cn1->numcns; i++) {
               cnx = (int)cn1->cns[i];
          if(direc != 1) { /* make weight change while error in inval */
                 Weight_change(cnx, site_index);
                }
                /* weight and sum inputs be they error or output */
                siteval = Sum_inputs(cnx, site_index);
               if (faulting) {
                 (void) flt_site(cnx, site_index, (short *)&siteval);
                }
                SITEVALUE(cnx, site_index) = siteval;
          }
        }
}
/********************************************************/
/* Squash: applies the activation function to the        */
/* output and assigns the result to the output site      */
/********************************************************/
void Squash(cn1, site_index)
cnlist *cn1;
short site_index;
{
        int cnx, i;

        if (cn1 != (cnlist *)NULL) {
          for (i = 0; i < cn1->numcns; i++) {
               cnx = cn1->cns[i];
               OUTPUT(cnx) = Activate(cnx, SITEVALUE(cnx, site_index));
               if (faulting) (void) flt_cn(cnx, &OUTPUT(cnx));
          }
        }
}
/********************************************************/
/* Assign_to_outsite: assign output to output site      */
/********************************************************/
void Assign_to_outsite(cn1, site_index)
cnlist *cn1;
short site_index;
```

```
{
    int cnx;
    int i;

    if (cn1 != (cnlist *)NULL) {
      for (i = 0; i < cn1->numcns; i++) {
          cnx = cn1->cns[i];
          SITEVALUE(cnx, site_index) = OUTPUT(cnx);
      }
    }
}
/*****************************************************/
/* Activate: CN activation function                  */
/*****************************************************/
short Activate(cnx, siteval)
int cnx;
short siteval;
{
    double dblval;
    char s[80];

    dblval = SHORT_TO_DOUBLE(siteval);
    /* don't blow up exp() */
    if (dblval < -30) {
      return(0);
    }
    if (dblval > 30) {
      return(500);
    }
    dblval = 1.0/(1.0 + exp(-1.0 *dblval));
    return(DOUBLE_TO_SHORT(dblval));
} /* end Activate() */
/*****************************************************/
/* CONVERGENCE PROCEDURE : HOST LEVEL                */
/* ie- the convergence procedure that runs in the host */
/*****************************************************/

#include "convergence.h"
FILE *fpcyc; /* print out cycle data to this file */
/* used to convert standard BIF vectors to floating point */
double dbl_err[NCNS], dbl_out[NCNS], dbl_targ[NSNS];
static int total_cycles = 0;
char alf[16] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
           'I', 'j', 'K', 'L', 'M', 'N', 'O', 'P'};
/* alf is for use in character recognition network */
/*****************************************************/
/* Convergence1: for back-prop nets.                 */
/* return 1 if converged, 0 if not                   */
/*****************************************************/
```

```
int Convergence1()
{
    int i, mark, ok = 1, maxind;
    double ferr, maxout;

    /* detect if each node is within defined limits */
    fprint(stderr, "0ARGET: ");
    for(i = 0; i < numoutputs; i++) {
     dbl_targ[i] = INT_TO_DOUBLE(targetvec[i]*SCALE);
     dbl_out[i] = INT_TO_DOUBLE(outputvec[i]);
     dbl_err[i] = dbl_targ[i] - dbl_out[i];
     errorvec[i] = DOUBLE_TO_INT(dbl_err[i]);
     /* fabs() didn't fx properly */
     ferr = dbl_err[i];
     if (ferr < 0.0) ferr = -1 * dbl_err[i];
     if (ferr > cp.err_factor) { /* then too much error */
        ok = 0;
     }
     fprintf(stderr, "%2.2f ",dbl_targ[i]);
    }
    /* print output vector to screen at each synch point */
    fprint(stderr, "576UTPUT: ");
    for (i = 0; i < numoutputs; i++) {
     fprintf(stderr, "%2.2f ", dbl_out[i]);
    }
    if (ok) {
     maxout = dbl_out[0];
     maxind = 0;
     for (i = 1; i < numoutputs; i++) {
         if (dbl_out[i] > maxout) {
          maxout = dbl_out[i];
          maxind = i;
          }
     }
     total_cycles += cp.numcycles;

fprintf(stderr,"CONVERGED on %c in %d cycles, total cycles = %d0,
                    alf[maxind], cp.numcycles, total_cycles);
     fpcyc = fopen("cycles", "a");
fprintf(fpcyc,"CONVERGED on %c in %d cycles, total cycles = %d0,
                    alf[maxind], cp.numcycles, total_cycles);
     fclose(fpcyc);
}

HEADER FILES FOR USER PROCEDURES
#if VAX
#include <stdio.h>
#endif
#include "nglob.h"
```

```
/* ANN system calls */
extern void Send_node_output(), Send_group_output(),
     Send_net_output();
extern void Update_node_weights(), Update_group_weights();
extern cnlist *Get_cnlist();
/* local CN table */
extern CNentry CN[MAX_CNS];
/* used for target vector, if any */
extern short targetvals[MAX_CNS];
/* structure holding user-controlled simulation parameters */
extern cycle_params cp;
/* faulting flag */
extern int faulting;

/* USER MACROS */
/* for simplified access to CN table fields */
#define DELAY(cn) CN[cn].C->delay
#define HISTORY(cn) CN[cn].C->history
#define RESTPOT(cn) CN[cn].C->restpot
#define POT(cn) CN[cn].C->pot
#define STATE(cn) CN[cn].C->state
#define OUTPUT(cn) CN[cn].C->output
#define ERROR(cn) CN[cn].C->error
#define SD(cn) CN[cn].C->sd
#define SITEVALUE(cn,s) CN[cn].sites[s].value
#define SITENLINKS(cn,s) CN[cn].sites[s].nlinks
#define LINKPTR(cn,s,1) &CN[cn].sites[s].links[1]
#define LINKVEC(cn,s,1) CN[cn].sites[s].links[1].lnkvec
#define LINKHISTORY(cn,s,1) CN[cn].sites[s].links[1].history
#define LINKWEIGHT(cn,s,1) CN[cn].sites[s].links[1].weight
#define LINKINVAL(cn,s,1) CN[cn].sites[s].links[1].inval
#define SETWTUP(cn,s,1) CN[cn].sites[s].links[1].lnkvec |=LV_WTUP

/* used to convert int fields to float and vice versa */
#define SHORT_TO_FLOAT(s) ((((float)s)/(float)SCALE))
#define FLOAT_TO_SHORT(f) ((short) (f * (float)SCALE))
#define SHORT_TO_DOUBLE(s) ((((double)s)/(double)SCALE))
#define DOUBLE_TO_SHORT(d) ((short)(d * (double)SCALE))
#define DERIV(f) (f * (1 - f))
```

User Accessible Data Structures

In writing the network procedure the user has access to local data structures, including those holding the CNs. These structures are modelled closely after the BIF format. The following data structures can be accessed by the user:

```
/* entry in local cube node CN table */
typedef struct {
     unsigned char hn;      /* hypercube node index where CN lives */
```

```
    SFWL *sites;         /* array of sites belonging to this CN */
    CFWN *C;             /* pointer to a CN structure   */
}CNentry;

CNentry CN[MAX_CNS];         /* table of local CNs      */

/* a link */
typedef struct {
    unsigned char lnkvec;  /* bit vector for this link  */
    char history;          /* recent history of link    */
    BYT4 cn;               /* CN this link goes to      */
    short site;            /* site this link goes to    */
    short link;            /* link this link goes to    */
    float weight;          /* weight value for link     */
    short inval;           /* input value to this link  */
}LFWI;

/* a site */
typedef struct {
    short value;           /* result of site function   */
    unsigned char sitevec; /* bit vector for this site  */
    short nlinks;          /* number of links attached  */
    LFWI *links;           /* pointer to links array    */
}SFWL;

/* a CN */
typedef struct {
    char group;            /* group this CN belongs to  */
    BYT4 index;            /* unique CN index           */
    short procid;          /* cube processor for CN     */
    short delay;           /* delay of output message   */
    unsigned char bitvec;  /* bit vector for this CN    */
    char history;          /* recent history of CN      */
    short restpot;         /* resting potential         */
    short pot;             /* potential                 */
    char state;            /* current state of CN       */
    short output;          /* current output value      */
    short error;           /* error value               */
    short sd;              /* statistical deviation     */
    short nsites;          /* number of sites on CN     */
}CFWN;

/* struct passed to user from Get_cnlist(groupname)          */
/* recommended that a cnlist be allocated statically         */
typedef struct {
    int numcns;            /* number of CNs in list     */
    BYT4 *cns;             /* list of CN indices        */
}cnlist;
```

```
/* simulator's synchronization parameters              */
/* shared by both the host and nodes                   */
/* recommended that these not be assigned              */
typedef struct {
 short global_clock,    /* global simulation clock (host) */
       local_clock,     /* local simulation clock (nodes) */
       msg_window,      /* window for valid cn<->cn msgs  */
       synch_count,     /* # of local clock cycles to run */
       synch_point,     /* global_clock + synch_count - 1 */
       checkpoint;      /* synch_point to break at (host) */
} cycle_params;

cycle_params cp;

/* node vectors only available in the host              */
int *targetvec,      /* target vector length=numoutputs  */
   *outputvec,       /* output vector length=numoutputs  */
   *errorvec,        /* error vector length=numoutputs   */
   *inputvec;        /* input vector length=numinputs    */

int numoutputs,       /* number of output CNs          */
   numinputs;         /* number of input CNs           */
```

```
begin

inlayer([iv[1]], [sg[1],sg[2],sg[3]]) -> ([ou[1]])
inlayer([iv[2]], [sg[1],sg[2],sg[3]]) -> ([ou[2]])
inlayer([iv[3]], [sg[1],sg[2],sg[3]]) -> ([ou[3]])

hlayer([ou[1],ou[2],ou[3]], [erbk[11],erbk[21],erbk[31]]
:[0.3,0.01,0.2],[0.0,0.0,0.0],[0.0],[2.5],[0.1])
                    -> ([out[1]], [sg[1]])

hlayer([ou[1],ou[2],ou[3]], [erbk[12],erbk[22],erbk[32]]
:[0.23,0.4,0.02],[0.0,0.0,0.0],[0.0],[0.75],[0.1])
                    -> ([out[2]], [sg[2]])

hlayer([ou[1],ou[2],ou[3]], [erbk[13],erbk[23],erbk[33]]
:[0.21,0.31,0.02],[0.0,0.0,0.0],[0.0],[1.5],[0.1])
                    -> ([out[3]], [sg[3]])

olayer([out[1],out[2],out[1]], [e_op[1]],
[rcl[1]]:[0.03,0.02,0.25],[1.05],[0.1])
               -> ([erbk[11],erbk[12],erbk[13]], [result[1]])

olayer([out[1],out[2],out[3]], [e_op[2]],
[rcl[1]]:[0.23,0.24,0.06],[2.65],[0.1])
               -> ([erbk[21],erbk[22],erbk[23]], [result[2]])

olayer([out[1],out[2],out[3]], [e_op[3]],
[rcl[1]]:[0.29,0.01,0.22],[0.15],[0.1])
               -> ([erbk[31],erbk[32],erbk[33]], [result[3]])

fun inlayer(in[i], sigl[m]) -> (ip[i])
int i,m;
real in[2],sigl[4],ip[2];
{
in[i], sigl[1..m] => {
            ip[1] := in[1];
           }
}

fun hlayer(in[n], err[n] ival: w1[n], l_in[n], l_op[m], theta[m],si[m])
                         -> (op[m], rsig[m])
int m, n;
real in[4], err[4],w1[5],l_in[4],l_op[2],theta[2],
si[2],op[2],rsig[2],this_er,temp;
```

```
{
in[1..3] => {
        temp := w1[1] * in[1] + w1[2] * in[2] + w1[3]*in[3];
        l_in[1] := in[1];
        l_in[2] := in[2];
        l_in[3] := in[3];
        l_op[1] := 1/(1+exp(0- temp - theta[1]));
        op[1] := l_op[1];
        }
err[1..3] => {
        temp := err[1] + err[2] + err[3];
        this_er := l_op[1] *(1-l_op[1]) * temp;
        theta[1] := theta[1] + (si[1] * this_er);
        w1[1] := w1[1] + (si[1] * this_er * l_in[1]);
        w1[2] := w1[2] + (si[1] * this_er * l_in[2]);
        w1[3] := w1[3] + (si[1] * this_er * l_in[3]);
        rsig[1] := 1;
        }
}

fun olayer(in[n], exp_op[m], recall[m] ival: w2[n], theta[m], si[m]) ->
                        (errbk[n], out[m])
int n, m;
real in[4],exp_op[2],recall[2],w2[4],theta[2],
    si[2],errbk[4],out[2],temp, c_op,err;
{
 in[1..n], exp_op[m] => {
        temp := w2[1] * in[1] + w2[2] * in[2] + w2[3] * in[3];
        c_op := 1/(1+exp(0-temp - theta[1]));

         err := c_op * (1-c_op)*(exp_op[1] - c_op);
        theta[1] := theta[1] + (si[1] * err);

          w2[1] := w2[1] + (si[1] * err * in[1]);
          w2[2] := w2[2] + (si[1] * err * in[2]);
          w2[3] := w2[3] + (si[1] * err * in[3]);
          errbk[1] := err * w2[1];
          errbk[2] := err * w2[2];
          errbk[3] := err * w2[3];

          }

 in[1..n], recall[1] =>  {
        temp := w2[1] * in[1] + w2[2] * in[2] + w2[3] * in[3];
        out[1] := 1/(1+exp(0-temp - theta[1]));
        }
}
begin
int p,i;
```

```
real w1,w2,w3,theta;
real data[5:5],ex_op[5:5],sg[4], rcl[2],y1,y2,y3;
data[1:1] := 1.0; data[1:2] := 1.0; data[1:3] := 1.0;
ex_op[1:1] := 1.0; ex_op[1:2] := 0.0; ex_op[1:3] := 1.0;

data[2:1] := 0.0; data[2:2] := 1.0; data[2:3] := 1.0;
ex_op[2:1] := 1.0; ex_op[2:2] := 1.0; ex_op[2:3] := 0.0;

data[3:1] := 1.0; data[3:2] := 0.0; data[3:3] := 1.0;
ex_op[3:1] := 0.0; ex_op[3:2] := 1.0; ex_op[3:3] := 1.0;

input sg[1] = 1.0,
    sg[2] = 1.0,
    sg[3] = 1.0;
i := 1;
do (i<=3000) ->
          p := 1;
          do (p <= 3) ->
                input iv[1] = data[p:1],
                    iv[2] = data[p:2],
                    iv[3] = data[p:3],
                    e_op[1] = ex_op[p:1],
                    e_op[2] = ex_op[p:2],
                    e_op[3] = ex_op[p:3];
                    run_net;

                p := p + 1;
                od
                i := i + 1;
                od

output " ";
output "RESULTS";

output "NODE = 4";
getwt w1 = nde[4].w1[1],
    w2 = nde[4].w1[2],
    w3 = nde[4].w1[3],
    theta = nde[4].theta[1];
output " w1  =  ", w1;
output " w2  =  ", w2;
output " w3  =  ", w3;
output " theta  =  ", theta;


output "NODE = 5";
getwt w1 = nde[5].w1[1],
    w2 = nde[5].w1[2],
    w3 = nde[5].w1[3],
```

```
    theta = nde[5].theta[1];
output " w1  =  ", w1;
output " w2  =  ", w2;
output " w3  =  ", w3;
output " theta  =  ", theta;


output "NODE = 6";
getwt w1 = nde[6].w1[1],
    w2 = nde[6].w1[2],
    w3 = nde[6].w1[3],
    theta = nde[6].theta[1];
output " w1  =  ", w1;
output " w2  =  ", w2;
output " w3  =  ", w3;
output " theta  =  ", theta;


output "NODE = 7";
getwt w1 = nde[7].w2[1],
    w2 = nde[7].w2[2],
    w3 = nde[7].w2[3],
    theta = nde[7].theta[1];
output " w1  =  ", w1;
output " w2  =  ", w2;
output " w3  =  ", w3;
output " theta  =  ", theta;


output "NODE = 8";
getwt w1 = nde[8].w2[1],
    w2 = nde[8].w2[2],
    w3 = nde[8].w2[3],
    theta = nde[8].theta[1];
output " w1  =  ", w1;
output " w2  =  ", w2;
output " w3  =  ", w3;
output " theta  =  ", theta;


output "NODE = 9";
getwt w1 = nde[9].w2[1],
    w2 = nde[9].w2[2],
    w3 = nde[9].w2[3],
    theta = nde[9].theta[1];
output " w1  =  ", w1;
output " w2  =  ", w2;
output " w3  =  ", w3;
output " theta  =  ", theta;
```

```
        p := 1;
do (p <= 3) ->
        input iv[1] = data[p:1],
        iv[2] = data[p:2],
        iv[3] = data[p:3],
        rcl[1] = 1.0;
        run_net;
        get y1 = result[1],
           y2 = result[2],
           y3 = result[3];

        output " ";
        output data[p:1], " :: ",ex_op[p:1], " ==> ", y1;
        output data[p:2], " :: ",ex_op[p:2], " ==> ", y2;
        output data[p:3], " :: ",ex_op[p:3], " ==> ", y3;
        output " ";

        input sg[1] = 1.0,
            sg[2] = 1.0,
            sg[3] = 1.0;
        p := p + 1;
        od
end
end.
```

The initial weights(Tables 25 and 26) and the output produced by the program :-

Weights	Node 4	Node 5	Node 6
w1	0.30000	0.23000	0.21000
w2	0.01000	0.40000	0.31000
w3	0.20000	0.02000	0.02000
theta	2.50000	0.75000	1.50000

TABLE 25. Weights for Nodes in Hidden Layer

Weights	Node 7	Node 8	Node 9
w1	0.03000	0.23000	0.29000
w2	0.02000	0.24000	0.01000
w3	0.25000	0.06000	0.22000
theta	1.05000	2.65000	0.15000

TABLE 26. Weights for Nodes in Output Layer

RESULTS

NODE = 4
w1 = 1.524359
w2 = -2.262244
w3 = -1.061634
theta = 1.238360

NODE = 5
w1 = -0.494916
w2 = 5.112295
w3 = -1.363389
theta = -0.633388

NODE = 6
w1 = 5.409690
w2 = -0.850357
w3 = -1.619321
theta = -0.139319

NODE = 7
w1 = -2.103633
w2 = 4.363748
w3 = -1.883630
theta = 0.319483

NODE = 8

w1 = 0.959008

w2 = -3.634852

w3 = -4.151572

theta = 5.411319

NODE = 9

w1 = 1.955741

w2 = -2.039612

w3 = 4.647953

theta = -0.855688

INPUT		EXPECTED		RESULT
1.000000	::	1.000000	==>	0.949730
1.000000	::	0.000000	==>	0.176193
1.000000	::	1.000000	==>	0.911751
0.000000	::	1.000000	==>	0.982998
1.000000	::	1.000000	==>	0.852229
1.000000	::	0.000000	==>	0.093310
1.000000	::	0.000000	==>	0.061869
0.000000	::	1.000000	==>	0.869661
1.000000	::	1.000000	==>	0.994362

APPENDIX D - Sample Output from the Compiler

```
########C - OUTPUT FROM COMPILER . - ##########

==========================================================

==========================================================

nodeid = 1 nodename = inlayer

name1 = * index1 = 1  name2 = i index2 = 1 forminpar = in
index1 = 1 index2 = 1 aclinpar =iv[1] status =3 value =0.000000
name1 = * index1 = 1  name2 = m index2 = 3 forminpar = sigl
index1 = 1 index2 = 1 aclinpar =sg[1] status =3 value =0.000000
index1 = 1 index2 = 2 aclinpar =sg[2] status =3 value =0.000000
index1 = 1 index2 = 3 aclinpar =sg[3] status =3 value =0.000000
name1 = * index1 = 1 name2= i index2 = 1 formoupar = ip
index1 = 1 index2 = 1 acloupar =ou[1] status =3 value =0.000000

nodeid = 2 nodename = inlayer

name1 = * index1 = 1  name2 = i index2 = 1 forminpar = in
index1 = 1 index2 = 1 aclinpar =iv[2] status =3 value =0.000000
name1 = * index1 = 1  name2 = m index2 = 3 forminpar = sigl
index1 = 1 index2 = 1 aclinpar =sg[1] status =3 value =0.000000
index1 = 1 index2 = 2 aclinpar =sg[2] status =3 value =0.000000
index1 = 1 index2 = 3 aclinpar =sg[3] status =3 value =0.000000
name1 = * index1 = 1 name2= i index2 = 1 formoupar = ip
index1 = 1 index2 = 1 acloupar =ou[2] status =3 value =0.000000

nodeid = 3 nodename = inlayer

name1 = * index1 = 1  name2 = i index2 = 1 forminpar = in
index1 = 1 index2 = 1 aclinpar =iv[3] status =3 value =0.000000
name1 = * index1 = 1  name2 = m index2 = 3 forminpar = sigl
index1 = 1 index2 = 1 aclinpar =sg[1] status =3 value =0.000000
index1 = 1 index2 = 2 aclinpar =sg[2] status =3 value =0.000000
```

index1 = 1 index2 = 3 aclinpar =sg[3] status =3 value =0.000000
name1 = * index1 = 1 name2= i index2 = 1 formoupar = ip
index1 = 1 index2 = 1 acloupar =ou[3] status =3 value =0.000000

nodeid = 4 nodename = hlayer

name1 = * index1 = 1 name2 = n index2 = 3 forminpar = in
index1 = 1 index2 = 1 aclinpar =ou[1] status =3 value =0.000000
index1 = 1 index2 = 2 aclinpar =ou[2] status =3 value =0.000000
index1 = 1 index2 = 3 aclinpar =ou[3] status =3 value =0.000000
name1 = * index1 = 1 name2 = n index2 = 3 forminpar = err
index1 = 1 index2 = 1 aclinpar =erbk[11] status =3 value =0.000000
index1 = 1 index2 = 2 aclinpar =erbk[21] status =3 value =0.000000
index1 = 1 index2 = 3 aclinpar =erbk[31] status =3 value =0.000000
name1= * index1 = 1 name2=n index2 = 3 wts_indx->formwtpar = w1
index1 = 1 index2 = 1 value =0.300000
index1 = 1 index2 = 2 value =0.010000
index1 = 1 index2 = 3 value =0.200000
name1= * index1 = 1 name2=n index2 = 3 wts_indx->formwtpar = l_in
index1 = 1 index2 = 1 value =0.000000
index1 = 1 index2 = 2 value =0.000000
index1 = 1 index2 = 3 value =0.000000
name1= * index1 = 1 name2=m index2 = 1 wts_indx->formwtpar = l_op
index1 = 1 index2 = 1 value =0.000000
name1= * index1 = 1 name2=m index2 = 1 wts_indx->formwtpar = theta
index1 = 1 index2 = 1 value =2.500000
name1= * index1 = 1 name2=m index2 = 1 wts_indx->formwtpar = si
index1 = 1 index2 = 1 value =0.100000
name1 = * index1 = 1 name2= m index2 = 1 formoupar = op
index1 = 1 index2 = 1 acloupar =out[1] status =3 value =0.000000
name1 = * index1 = 1 name2= m index2 = 1 formoupar = rsig
index1 = 1 index2 = 1 acloupar =sg[1] status =3 value =0.000000

nodeid = 5 nodename = hlayer

name1 = * index1 = 1 name2 = n index2 = 3 forminpar = in
index1 = 1 index2 = 1 aclinpar =ou[1] status =3 value =0.000000

```
index1 = 1 index2 = 2 aclinpar =ou[2] status =3 value =0.000000
index1 = 1 index2 = 3 aclinpar =ou[3] status =3 value =0.000000
name1 = * index1 = 1  name2 = n index2 = 3 forminpar = err
index1 = 1 index2 = 1 aclinpar =erbk[12] status =3 value =0.000000
index1 = 1 index2 = 2 aclinpar =erbk[22] status =3 value =0.000000
index1 = 1 index2 = 3 aclinpar =erbk[32] status =3 value =0.000000
name1= * index1 = 1  name2=n index2 = 3 wts_indx->formwtpar = w1
index1 = 1  index2 = 1 value =0.230000
index1 = 1  index2 = 2 value =0.400000
index1 = 1  index2 = 3 value =0.020000
name1= * index1 = 1  name2=n index2 = 3 wts_indx->formwtpar = l_in
index1 = 1  index2 = 1 value =0.000000
index1 = 1  index2 = 2 value =0.000000
index1 = 1  index2 = 3 value =0.000000
name1= * index1 = 1  name2=m index2 = 1 wts_indx->formwtpar = l_op
index1 = 1  index2 = 1 value =0.000000
name1= * index1 = 1  name2=m index2 = 1 wts_indx->formwtpar = theta
index1 = 1  index2 = 1 value =0.750000
name1= * index1 = 1  name2=m index2 = 1 wts_indx->formwtpar = si
index1 = 1  index2 = 1 value =0.100000
name1 = * index1 = 1 name2= m index2 = 1 formoupar = op
index1 = 1 index2 = 1 acloupar =out[2] status =3 value =0.000000
name1 = * index1 = 1 name2= m index2 = 1 formoupar = rsig
index1 = 1 index2 = 1 acloupar =sg[2] status =3 value =0.000000

nodeid = 6 nodename = hlayer

name1 = * index1 = 1  name2 = n index2 = 3 forminpar = in
index1 = 1 index2 = 1 aclinpar =ou[1] status =3 value =0.000000
index1 = 1 index2 = 2 aclinpar =ou[2] status =3 value =0.000000
index1 = 1 index2 = 3 aclinpar =ou[3] status =3 value =0.000000
name1 = * index1 = 1  name2 = n index2 = 3 forminpar = err
index1 = 1 index2 = 1 aclinpar =erbk[13] status =3 value =0.000000
index1 = 1 index2 = 2 aclinpar =erbk[23] status =3 value =0.000000
index1 = 1 index2 = 3 aclinpar =erbk[33] status =3 value =0.000000
name1= * index1 = 1  name2=n index2 = 3 wts_indx->formwtpar = w1
index1 = 1  index2 = 1 value =0.210000
```

```
:x1 = 1  index2 = 2 value =0.310000
:x1 = 1  index2 = 3 value =0.020000
ıe1= * index1 = 1  name2=n index2 = 3 wts_indx->formwtpar = l_in
:x1 = 1  index2 = 1 value =0.000000
:x1 = 1  index2 = 2 value =0.000000
:x1 = 1  index2 = 3 value =0.000000
ıe1= * index1 = 1  name2=m index2 = 1 wts_indx->formwtpar = l_op
:x1 = 1  index2 = 1 value =0.000000
ıe1= * index1 = 1  name2=m index2 = 1 wts_indx->formwtpar = theta
:x1 = 1  index2 = 1 value =1.500000
ıe1= * index1 = 1  name2=m index2 = 1 wts_indx->formwtpar = si
:x1 = 1  index2 = 1 value =0.100000
ıe1 = * index1 = 1 name2= m index2 = 1 formoupar = op
:x1 = 1 index2 = 1 acloupar =out[3] status =3 value =0.000000
ıe1 = * index1 = 1 name2= m index2 = 1 formoupar = rsig
:x1 = 1 index2 = 1 acloupar =sg[3] status =3 value =0.000000

eid = 7 nodename = olayer

ıe1 = * index1 = 1  name2 = n index2 = 3 forminpar = in
:x1 = 1 index2 = 1 aclinpar =out[1] status =3 value =0.000000
:x1 = 1 index2 = 2 aclinpar =out[2] status =3 value =0.000000
:x1 = 1 index2 = 3 aclinpar =out[1] status =3 value =0.000000
ıe1 = * index1 = 1  name2 = m index2 = 1 forminpar = exp_op
:x1 = 1 index2 = 1 aclinpar =e_op[1] status =3 value =0.000000
ıe1 = * index1 = 1  name2 = m index2 = 1 forminpar = recall
:x1 = 1 index2 = 1 aclinpar =rcl[1] status =3 value =0.000000
ıe1= * index1 = 1  name2=n index2 = 3 wts_indx->formwtpar = w2
:x1 = 1  index2 = 1 value =0.030000
:x1 = 1  index2 = 2 value =0.020000
:x1 = 1  index2 = 3 value =0.250000
ıe1= * index1 = 1  name2=m index2 = 1 wts_indx->formwtpar = theta
:x1 = 1  index2 = 1 value =1.050000
ıe1= * index1 = 1  name2=m index2 = 1 wts_indx->formwtpar = si
:x1 = 1  index2 = 1 value =0.100000
ıe1 = * index1 = 1 name2= n index2 = 3 formoupar = errbk
:x1 = 1 index2 = 1 acloupar =erbk[11] status =3 value =0.000000
```

```
index1 = 1 index2 = 2 acloupar =erbk[12] status =3 value =0.000000
index1 = 1 index2 = 3 acloupar =erbk[13] status =3 value =0.000000
name1 = * index1 = 1 name2= m index2 = 1 formoupar = out
index1 = 1 index2 = 1 acloupar =result[1] status =3 value =0.000000

nodeid = 8 nodename = olayer

name1 = * index1 = 1  name2 = n index2 = 3 forminpar = in
index1 = 1 index2 = 1 aclinpar =out[1] status =3 value =0.000000
index1 = 1 index2 = 2 aclinpar =out[2] status =3 value =0.000000
index1 = 1 index2 = 3 aclinpar =out[3] status =3 value =0.000000
name1 = * index1 = 1  name2 = m index2 = 1 forminpar = exp_op
index1 = 1 index2 = 1 aclinpar =e_op[2] status =3 value =0.000000
name1 = * index1 = 1  name2 = m index2 = 1 forminpar = recall
index1 = 1 index2 = 1 aclinpar =rcl[1] status =3 value =0.000000
name1= * index1 = 1  name2=n index2 = 3 wts_indx->formwtpar = w2
index1 = 1  index2 = 1 value =0.230000
index1 = 1  index2 = 2 value =0.240000
index1 = 1  index2 = 3 value =0.060000
name1= * index1 = 1  name2=m index2 = 1 wts_indx->formwtpar = theta
index1 = 1  index2 = 1 value =2.650000
name1= * index1 = 1  name2=m index2 = 1 wts_indx->formwtpar = si
index1 = 1  index2 = 1 value =0.100000
name1 = * index1 = 1 name2= n index2 = 3 formoupar = errbk
index1 = 1 index2 = 1 acloupar =erbk[21] status =3 value =0.000000
index1 = 1 index2 = 2 acloupar =erbk[22] status =3 value =0.000000
index1 = 1 index2 = 3 acloupar =erbk[23] status =3 value =0.000000
name1 = * index1 = 1 name2= m index2 = 1 formoupar = out
index1 = 1 index2 = 1 acloupar =result[2] status =3 value =0.000000

nodeid = 9 nodename = olayer

name1 = * index1 = 1  name2 = n index2 = 3 forminpar = in
index1 = 1 index2 = 1 aclinpar =out[1] status =3 value =0.000000
index1 = 1 index2 = 2 aclinpar =out[2] status =3 value =0.000000
index1 = 1 index2 = 3 aclinpar =out[3] status =3 value =0.000000
name1 = * index1 = 1  name2 = m index2 = 1 forminpar = exp_op
```

```
index1 = 1 index2 = 1 aclinpar =e_op[3] status =3 value =0.000000
name1 = * index1 = 1  name2 = m index2 = 1 forminpar = recall
index1 = 1 index2 = 1 aclinpar =rcl[1] status =3 value =0.000000
name1= * index1 = 1  name2=n index2 = 3 wts_indx->formwtpar = w2
index1 = 1  index2 = 1 value =0.290000
index1 = 1  index2 = 2 value =0.010000
index1 = 1  index2 = 3 value =0.220000
name1= * index1 = 1  name2=m index2 = 1 wts_indx->formwtpar = theta
index1 = 1  index2 = 1 value =0.150000
name1= * index1 = 1  name2=m index2 = 1 wts_indx->formwtpar = si
index1 = 1  index2 = 1 value =0.100000
name1 = * index1 = 1 name2= n index2 = 3 formoupar = errbk
index1 = 1 index2 = 1 acloupar =erbk[31] status =3 value =0.000000
index1 = 1 index2 = 2 acloupar =erbk[32] status =3 value =0.000000
index1 = 1 index2 = 3 acloupar =erbk[33] status =3 value =0.000000
name1 = * index1 = 1 name2= m index2 = 1 formoupar = out
index1 = 1 index2 = 1 acloupar =result[3] status =3 value =0.000000

DONE
```

```
        /* input vector */

typedef struct invect {
      int    iaindx1;       /*row */
      int    iaindx2;       /*col*/
      char    *actlinpar;  /* actual parameter name */
      int    status;
      float   value;
      struct  invect  *nxtiput;
      } invect;

/* output vector */

typedef struct ouvect {
      int    oaindx1;
      int    oaindx2;
      char    *actloupar;
      int    status;
      float   value;
      struct  ouvect  *nxtoput;
      } ouvect;

/* weight vector */

typedef struct wtvect {
      int    waindx1;
      int    waindx2;
      float   value;
      struct  wtvect  *nxtwt;
      } wtvect;
```

```
/* input vector list */

typedef struct inp_indx {
      char   *dimname1;  /* subscript name */
      int    invl_num1;  /* index value */
      char   *dimname2;
      int    invl_num2;
      char   *forminpar;  /* formal para name */
      invect  *inlst;
      struct inp_indx  *nxt_i_indx;
      } inp_indx;

/* weight vector list */

typedef struct wts_indx {
      char   *dimname1;
      int    wtvl_num1;
      char   *dimname2;
      int    wtvl_num2;
      char   *formwtpar;
      wtvect  *wtlst;
      struct wts_indx  *nxt_w_indx;
      } wts_indx;

/* output vector list */

typedef struct oup_indx {
      char   *dimname1;
      int    ouvl_num1;
      char   *dimname2;
      int    ouvl_num2;
      char   *formoupar;
      ouvect  *oulst;
      struct oup_indx  *nxt_o_indx;
      } oup_indx;
```

```
/* a node */

typedef struct nodei_o {
      int     nodeid;          /* node id */
      char    *fname;          /* function name */
      inp_indx  *inv_ptr;      /* ptr to input vector list */
      wts_indx  *wts_ptr;      /* ptr to weight vector list */
      oup_indx  *oup_ptr;      /* ptr to output vector list */
      struct  nodei_o  *nxtnde;  /* ptr to next node */
      } nodei_o;
```

APPENDIX F - Published Works

1. M. Pacheco, S. Bavan, M. Lee and P. Treleaven, "A Simple VLSI Architecture for Neurocomputing", in INNS - International Neural Network Society, First Annual Meeting. September 6-10, 1988, Boston, Massachusetts, pp398.

2. A. S. Bavan, "NIL-PLUS: A Neural Network Implementation Language", Proc. First Neural Computing Meeting, The Institute of Physics, London, April 1989, pp171-178.

3. A. Eliasz, S. Bavan, and J. Crowcroft, "Adaptive Network Management Using Neural Computing", - Proc. Third Race TMN Conference, 30 August - 1 September 1989, London.

4. A. S. Bavan, "A Programming System for Implementing Neural Nets", - XI Sitges Conference on Neural Networks, 4 June - 7 June 1990, Barcelona, Spain.

5. A. S. Bavan, "NPS: A Neural Network Programming System", in Proc. International Joint Conference on Neural Networks, June 17-21, 1990, Sandiego, California, pp143-148.

6. A. W. Eliasz, A. S. Bavan, and J. Crowcroft, "Approaches to Using Neural Computing Methods to Develop Adaptive Distributed Routing Algorithms", - Proc. Fourth Race TMN Conference, 14 - 16 November 1990, Dublin, pp218-232.