NDL

A Network Description Language

Mark Alan Johnson
B.S.E.E Oregon State University, 1980

A thesis submitted to the faculty
of the Oregon Graduate Institute
of Science and Technology
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science & Engineering

June, 1990

The thesis "NDL: A Network Description Language" by Mark Alan Johnson has been examined and approved by the following Examination Committee:

Daniel Hammerstrom, Thesis Research Advisor
Associate Professor
Department of Computer Science and Engineering
Oregon Graduate Institute

Ron Cole
Associate Professor
Department of Computer Science and Engineering
Oregon Graduate Institute

William Bain Jr.
Adjunct Assistant Professor
Intel Scientific Computers, Inc.

TABLE OF CONTENTS

LIST OF FIGURES

APPENDICES

ABSTRACT

NDL
A Network Description Language

Mark Alan Johnson, M.S.
Oregon Graduate Institute, 1990

Supervising Professor: Dan Hammerstrom

NDL is an environment which supports the development of network graphs for massively parallel computer architectures. Formally NDL generates attributed undirected multigraphs. Data, procedural, and control abstractions above those provided by the implementation language C++ are utilized. Aggregates, generalizations, and scopes are used for applying operations on and in the graph. Attributes are used for augmenting the graph components. The attributes are also used as a mechanism for referencing the network components. NDL output is flexible allowing graph structure, attributes, and graph construction data to be output. This flexibility permits users to tailor the output into formats acceptable by other environments.

Chapter 1
Introduction

A new direction in computer research is the study of massively parallel computer architectures. This research is commonly referred to as connectionist or neural network research. As with most research today, computer simulation is essential for the development, test, and demonstration of new ideas. The research involving neural networks is no exception. Part of the process of designing a neural network simulation is the specification of the network's topology. For small regular networks, this is not a problem. However, as the networks get larger with more complex structure, the specification of the topology using conventional programming languages like C and Pascal becomes very tedious. This thesis describes an environment for describing these complex topologies.

Motivation

The Cognitive Architecture Project (CAP) at the Oregon Graduate Institute (OGI) is building a neural network simulation environment. The first step taken in generating a simulation is to create a network description file defining the network's topology. This file is referred to as a BIF

(Beaverton Intermediate Form) file. The other tools in the simulation environment read the BIF file. Currently the BIF files are constructed by hand or by a program specifically written for a particular network topology. These primitive methods precipitated the need for a tool to automatically generate the BIF files from some sort of specification.

Goals

This thesis has four goals. The first and most obvious goal is to produce BIF files. Associated with this goal is the desire to have a tool which allows the user to construct large complex network topologies with relative ease. The mechanisms should be easy to use, but powerful. Control, data, and procedural abstractions should be utilized to abstract away the tedious underlying details. The abstractions should be consistent with the model presented.

A second goal is to allow for extensibility. Many new neural algorithms and models are being developed every day, so the NDL environment should easily support extensions and improvements.

The third goal is to create a network generating tool which is flexible enough so as to not just create BIF files. Some other simulation environment might have different requirements. Flexibility in this dimension also makes NDL easier to maintain as changes occur in the BIF format.

A fourth goal is to investigate graph grammars as a means for generating network graphs.

Neural Networks

A typical neural network model consists of a large number of computing elements called *connection nodes* (CNs) interconnected to each other with *links*. The CNs compute some simple arithmetic or logic function on the inputs they receive from their input links. The result of the computation is communicated to other nodes in the network via output links. The output links of one CN are input links to another. In most neural network models today, all the CNs perform the same simple function. This is, however, not a requirement. A network could consist of a heterogeneous mixture of CN types, each type performing a different function. The function(s) performed by the CNs determine in part the overall behavior of the network.

Each link in the network has a weight associated with it. The values of these weights and the connectivity given by the links add to the behavior of the network. It is often stated that the weights comprise the knowledge of the system.

The generic CN model is shown in figure 1.1. The inputs, x_1 - x_n, are multiplied by their respective weights w_1 - w_n. The summed value is

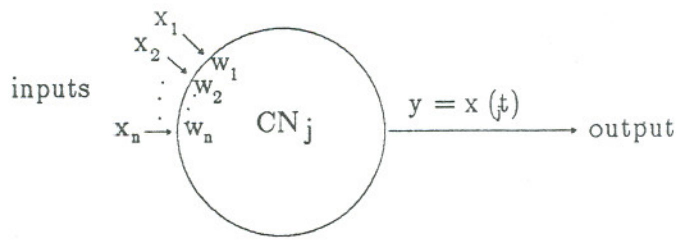

Figure 1.1 - Generic Neural Network CN Model

presented to the CNs function $x_j(t)$, and the result is transmitted to other CNs in the network.

The CAP Simulation Environment

At the Oregon Graduate Institute (OGI), the Cognitive Architecture Project (CAP) group is studying and evaluating several different neural network models. To aid in this endeavor, a neural network simulation environment has been created using the Intel iPSC parallel system as the hardware platform. The purpose of this simulation environment is to provide a short idea-simulation cycle for neural network experiments, where it takes little time to go from an idea to a working simulation model. Another purpose of the environment is to provide feedback on the feasibility of the model in VLSI.

A typical simulation using the tools is divided into four steps. The first step requires the user to define the network graph. NDL performs this function. The user writes a NDL program which is compiled and made into an executable program. By running the program, the desired BIF file is produced.

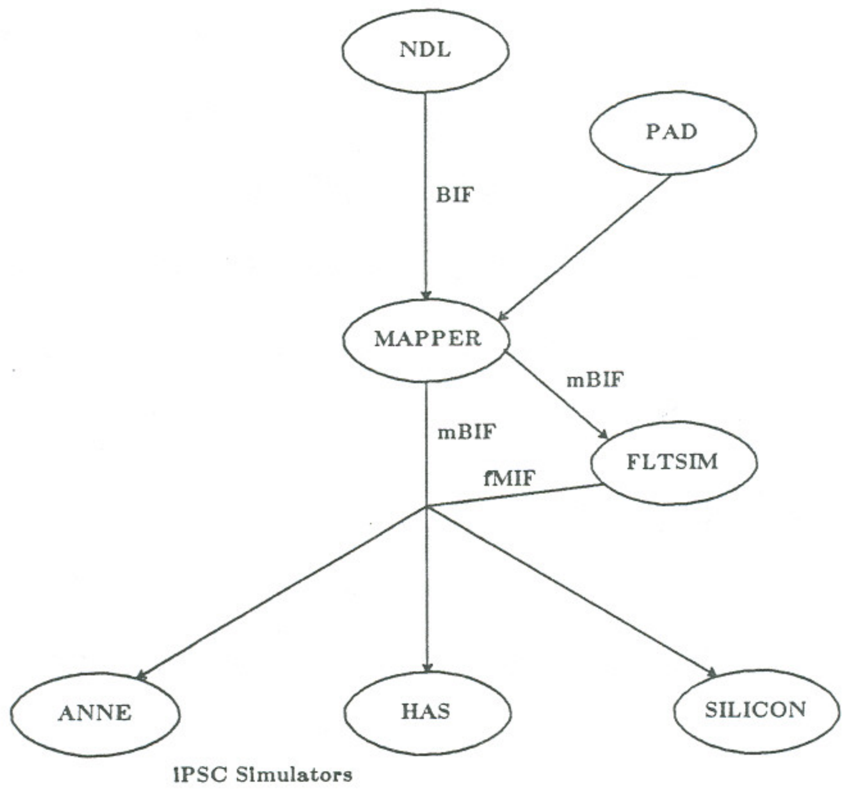

Figure 1.2 - The CAP Simulation Environment

The second step involves a program called the MAPPER [Bai88]. The MAPPER reads the BIF file describing the network's connectivity and another file called the PAD. The PAD file contains architectural configuration information describing the multi-processor hardware being used. The goal of the MAPPER is to efficiently map the CNs of the neural network model being simulated to the hardware nodes of the hypercube. The output file created by MAPPER is called MIF (Mapped Intermediate Form).

The third step introduces errors into the network graph simulating possible VLSI manufacturing faults. The program, which performs this function, is called FLTSIM [May88]. FLTSIM reads the MIF file, a silicon technology file, and a file containing fault parameters. The output file produced by FLTSIM is called a fMIF file.

The last step supplies the network with functions and control, allowing the model to be simulated and analyzed. Two programs perform this function, ANNE (Another Neural Network Emulator)[Bah88], and HAS (Hardware Architecture Simulator)[Jag89]. ANNE is a general purpose simulator which can execute a wide range of neural network models. HAS is a more special purpose simulator which emulates how a model will behave on wafer-scale hardware. Figure 1.2 outlines the CAP simulation environment.

Overview of the Thesis

Chapter two describes some of the related work being done in neural network simulation environments, concentrating on the NDL counterparts within those environments. Chapter three introduces the formal concepts used by NDL. Chapter four describes the NDL functions and how to use them to create a network graph using the concepts introduced in chapter three. Chapter five describes the NDL environment. Compiling a program and application libraries are discussed. Chapter six describes an NDL application environment which generates layer topologies. Chapter seven describes another NDL application environment which generates networks based on a pyriform cortex neural network model. An example is given of how NDL can produce complex connectivity patterns in a network graph. Chapter eight discusses future work. Chapter nine summarizes and concludes the thesis.

Chapter 2

Related Work

There are many neural network simulation environments in existence today. Most have a NDL counterpart, but they have structured its role in their respective environments differently. The method for specifying the topology in each environment can be reduced to one of three approaches. The *graphical* approach has the user construct the network interactively on a graphical display terminal. The *canned* approach has the user specify the architectural parameters of a known topology. The *declarative* approach has the user specifying the topology with some type of language specifically written for generating networks. This is the approach taken by NDL. The rest of this chapter describes some of the NDL "counterparts" in other neural network simulation environments.

P3 and its Plan Language

P3 is the Parallel Network Simulating System introduced in the book Parallel Distributed Processing [RuM86]. Its major constituents are the *plan language*, the *method language*, the *constructor*, and the *simulation environment*. The plan language describes the underlying graph structure in the network. The units are described and the connections between them are specified. In P3, this description is called the "plan". It is the "counterpart" to NDL. The method language describes the functions to be performed by the units. The constructor links the plan and the methods together and forms the program which simulates the network. The simulator environment runs the program and provides feedback to the user for observation and analysis.

The plan language provides three fundamental constructs which enables the user to construct the underlying graph structure of the neural network. The UNIT TYPE construct creates a unit type and describes it. The UNIT construct instantiates either single or sets of UNIT TYPEs and assigns them a name. The CONNECT construct makes the connections, and is utilized within language provided control constructs.

Each unit has two classes of parameters. The *unit parameters* are associated with the function of the unit, and the *terminal parameters* are associated with the connections. The unit parameters are used only by the function performed by the unit, which is described in the method language. The unit parameters constitute the unit's local data accessible only by the unit's function. The terminal parameters are values like the weights assigned to the connections. These data are accessible to functions outside the unit and are therefore considered global data. Both types of parameters are defined when the UNIT TYPE is defined, and changed by the simulator environment during the course of a simulation session.

A program written in the plan language consists of two segments. The first segment defines the unit types using the UNIT TYPE construct. The second segment instantiates and connects the units with the other two programming constructs, UNIT and CONNECT. The CONNECT statements are usually buried deep within nested loops of the implementation language Lisp.

For debugging purposes, the user will want to know that the network is connected in the manner desired. A display is provided by P3 in which the user interacts with the P3 space, enabling him to trace out connections.

Hycon

Hycon is a general purpose connectionist simulator that runs on the Intel iPSC parallel processor system [Pla87]. The model uses a network of *units* connected together by *links*. The network is constructed by organizing the units into groups. A group can consist of one to many units. The units are given an index within a group by using consecutive integer numbers. To specify the interconnection between two groups, three mechanisms are used. *Complete* interconnect links all the units in one group to all the units in another. *Random* interconnect links a certain percentage of units from one group to a percentage of units in another group. *Point to point* interconnect links a specified unit from one group to a specified unit from another (or same) group.

In Hycom, the simulation is run in phases. The temporal sequencing of phases constitute a simulation. The units are referenced in these phases by the group they belong to. A phase could specify that a group of units are to compute their transfer function. The process of constructing the network is therefore tied to the process of executing the network.

GRADSIM

The GRADSIM neural network simulation environment specializes in iterative gradient optimization techniques [Wat88]. It is a collection of C software modules which when compiled and linked together form a general purpose connectionist network simulator. The structure of the network used in GRADSIM is specified by an ASCII file. The file is called the network descriptor file and specifies both connectivity and network characteristics. The files are produced using text editors and Unix shell scripts. This is probably the most primitive form of generating a neural network graph description.

ANSpec

ANSpec provides a virtual processing environment where each element of a neural network is allocated a virtual concurrent processor[Kra]. It is based on Hewitt's Actor Model of concurrent processes. The environment includes a high level simulation environment, and an operating system which is uniquely tailored to implement the Actor Model. There is no separate software module that generates a network structure describing the connectivity. Instead, the structure is contained in the communication patterns between the concurrent processes allocated to each of the processing elements. This form of describing connectivity and network graphs is unique from the other simulators, and requires the user to be familiar with a new programming methodology.

The Rochester Connectionist Simulator

The simulator being used at the University of Rochester is probably the most publicized [Fan86]. It is based on the BBN Butterfly Multiprocessor. The network is built in the simulator by a C program written by the user. The primitive functions needed to construct the network are supplied via a library. The basic components in the model are the link, the site, and the unit. The site is a concentration point for connections on a unit. Associated with each of these components is a data field. Space must be allocated for the units before network construction begins. A *MakeUnit* function creates a unit. The units are assigned integer indices as they are created. A *MakeSite* function creates a site on a specified unit. A *MakeLink* function creates a link between two specified units and their respective sites. The network is constructed by first allocating space, creating the units, creating the sites on the units, and then creating the links between the sites on the units. Creating the links is usually done in tight loops.

The user can construct sets of units for display purposes. Set functions are included allowing sets to be created from existing sets using set theoretic operations.

Chapter 3

The NDL Model

This chapter presents the NDL model. Formal notation is introduced for the fundamental mechanisms used in NDL. By having this notation, formal descriptions can be made of the algorithms that generate network graphs. The notation therefore eliminates potential ambiguity and allows the study of the network generating algorithms as mathematical objects[Dij76].

First the connection node (CN) will be introduced. It represents the fundamental building block of the NDL model. It will be shown how NDL uses a graph as a pictorial representation of a network and ordered rooted trees as a mechanism for assigning labels to the elements in the network. The *components* of the network graph represent the physical elements in the network graph. The *Attribute Base* and the *attribute access list* provide a mechanism for augmenting the network graph components with attributes. The *generalization* is used to perform an operation on a set of components by virtue of the attributes contained in a component. The *aggregate* is used for performing an operation on a collection of components by virtue of their membership in sets. The *scope*, which utilizes both the aggregate and generalization, is discussed last. Most of the names used for the concepts were borrowed from reports about *hypertext* [CaG88][Gar88].

The Connection Node

The connection node, hereby denoted CN, is the fundamental element in an NDL network graph. The body of the CN, shown in figure 3.1, represents the processing part of the node. The sites on the body are connection points for the links and allow the body to differentiate inputs and outputs by virtue of their connection point on the CN. The links make up the connectivity of the network graph. Associated with each link is a

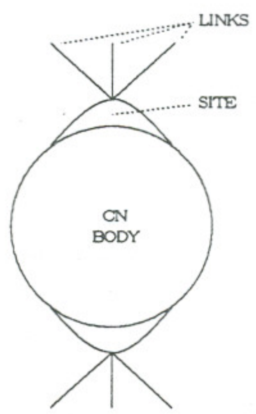

Figure 3.1 - The Connection Node

weight [Fan86]. An ensemble is a disjoint set of CNs.

The Pictorial Network Graph

A network graph is pictorially represented by drawing the CNs and connecting them with links. This allows both the person drawing the graph and the viewer to communicate such things as network topology and connectivity. Phrases like "the CN in the second layer on the far right" and "the site on the bottom of the third CN in the middle layer" would be a way of describing the network components. It would not, however, lend itself well for creating operations on a network graph programmatically or algorithmically. Therefore, the motivation for a systematic method of labeling and representing the network graphs arises.

Representing and Labeling the Network Graph

If a picture of an arbitrary network graph was shown to two people, chances are they would each label the graph differently. Certainly a canonical method could be employed which would get both people to label the graph consistently. But what if another arbitrary network graph was presented? Would the method be appropriate? For example, a method which worked for graphs represented in two dimensions might not work for graphs in three dimensions. Another problem is that slight changes pictori-

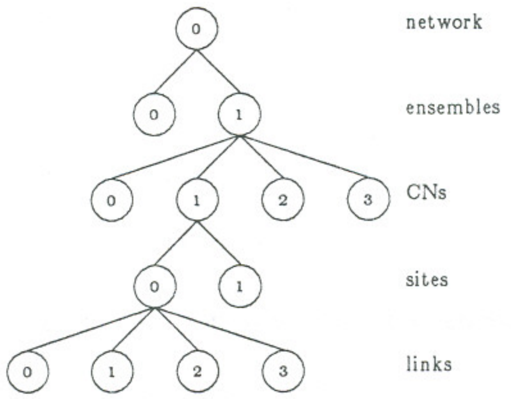

Figure 3.2 - The Network Component Tree

ally could change dramatically the labeling of the elements without changing at all the graph and its connectivity.

In NDL, this problem is addressed by labeling the graph elements in the order they would be drawn. A description is required which details the algorithm for drawing the graph. For example, suppose a network graph consists of two layers of nodes. The algorithm for drawing the graph might be to draw the bottom layer first, then the top layer. In both cases, the nodes are drawn left to right. Note that this is not the only algorithm for drawing the example network. Somebody else might prefer to see the top layer drawn first and done right to left. The important concept here is that to relate the network description generated by NDL to a pictorial representation, this drawing algorithm must be known.

As each element is added to the graph, it is given a unique label. In addition, as each element is added to the network graph, it is entered into an ordered rooted tree. The elements can therefore be uniquely identified by the labeling scheme shown in figure 3.2. This labeling scheme is referred to as the *universal address system.* The ordered rooted tree in NDL is called the Network Component Tree. To uniquely identify an ensemble, only one index is required. To uniquely identify a CN in the component tree, two index references would be required. At the bottom of the network component tree are the links, which take four index references to uniquely identify. The four indices are simply the path elements encountered as the tree is traversed from the root to the leaf (link). The network node is left out since it can be implied for all references.

A one-to-one mapping does not not exist for a pictorial network graph and its Network Component Tree. The mapping comes from the order of construction of the pictorial network graph. By knowing the order of construction of the pictorial network graph, leverage can be applied using tree and graph transformations as operations on the network component tree. In other words, it is much easier to write algorithms involving an ordered rooted tree than an arbitrary graph. These operations are typically the creation of the links in a network graph. Figure 3.3 shows a network graph and its corresponding network component tree. The network node represents the root in the component tree. Beneath the network node is the ensemble node. The CNs were added to the graph left to right, bottom to top. The sites were added to the CNs bottom to top. The links were added from the bottom up, left to right.

The Network Graph Components

The physical structure of a neural network is given by its underlying network graph. In NDL, the network graph consists of four types of objects. They are the ensemble, the CN, the site, and the link. They are, and have previously been, referred to as the network components.

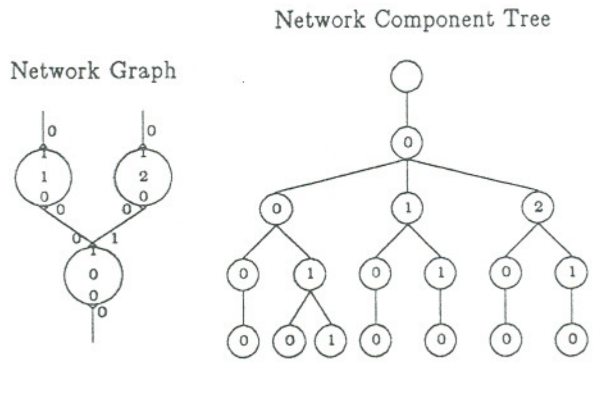

Figure 3.3 - Network Component Tree vs. Network Graph

The Ensemble

In neural network models, groups of CNs representing related concepts or performing similar functions typically have a high degree of connectivity. The network graph therefore has a direct correlation with the function of the different constituents making up the network [GhH88]. In NDL, the *ensemble* was developed to capture these concepts.

An NDL network graph consists of an ordered and indexed set of ensembles. The ensembles are assigned natural number indices starting with zero and incrementing consecutively. Each ensemble is a subgraph of the entire network graph. Each ensemble is also disjoint from the other ensembles. That is, any CN existing in an ensemble will not exist in any other ensemble. A special situation exists when there is only one ensemble, in which case the ensemble contains all the CNs in the network graph. An ensemble is denoted and referenced by

$$\text{ens}[\text{i}]$$

where i is the index of the ensemble in the network graph. Associated with each ensemble is an attribute list and a set of CNs. In the network component tree, the ensembles are represented by the the direct children of the root network node (second layer nodes), and are numbered from left to right. A set of n ensembles is denoted by

$$\text{ENS} = \{\text{ens}[0],\ \text{ens}[1],...,\text{ens}[\text{n-1}]\}$$

In NDL, the network is a set of ensembles.

The CN

Each ensemble in the network graph consists of an ordered and indexed set of *CNs*. The CNs are assigned natural number indices starting with zero and incrementing consecutively. A CN is denoted and referenced by

cn[i,j]

where *i* is the index of the ensemble the CN belongs to in the network graph and *j* is the index of the CN in the ensemble. Associated with each CN is an attribute list and a set of sites. In the network component tree, the CNs are represented by the direct children of the ensemble nodes, and are numbered from left to right.

The Site

Each CN in the network graph consists of an ordered and indexed set of *sites*. The sites are assigned natural number indices starting with zero and incrementing consecutively. A site is denoted by

site[i,j,k]

where *i* is the index of an ensemble in the network graph, *j* is the index of a CN in the ensemble, and *k* is the index of the site in the CN. Associated with each site is an attribute list and a set of links. In the network component tree, the sites are represented by the direct children of the CN nodes, and are numbered from left to right.

The Link

Each site in the network graph consists of an ordered and indexed set of *links*. The links are assigned natural number indices starting with zero and incrementing consecutively. A link is denoted by

link[i,j,k,l]

where i is the index of an ensemble in the network graph, j is the index of a CN in the ensemble, k is the index of a site in the CN, and l is the index of the link on the site. Associated with each link is an attribute list, weight, and connection information. The connection information references the other end of the link. A link can be either single-ended or double-ended. A single-ended link is an edge in the network graph with one endpoint undefined. There would be no connection information in this case. A double-ended link is an edge in a graph with both endpoints defined. The connection information is the other endpoint in this case. The double ended link is actually two links in the network component tree, each pointing to the other. In the network component tree, the links are represented by the the direct children of the site nodes, and are numbered from left to right.

Aggregates

An *aggregate* is an mechanism and abstraction by which a collection of objects can be referenced by a single identifier. Once the aggregate is defined, an operation can be applied to all elements of the aggregate. An aggregate in NDL is a set of network component indices. A network component index is its index under its parent node in the network component tree. It should not be confused with a component's reference. A component's reference uniquely identifies it from the other components. The index does not. An example should make this concept a little easier to grasp. A site is referenced by an ensemble index, a CN index, and a site index. The reference uniquely identifies one site in the tree. Now consider that the site is referenced by an ensemble index, an aggregate of CN indices, and a site index. Suppose that the aggregate of CNs was defined as the set

$$\{0,2,4,6\}.$$

The site reference now identifies 4 sites. Each site is on a different CN, the CNs specified in the set, but in the same ensemble. This concept can be taken even further by making the ensemble constituent of the site reference an aggregate. In fact all three constituents of the site reference can be made an aggregate. It is also valid to use an aggregate previously used as the CN constituent in a reference as the ensemble constituent of a reference. The aggregate is simply a list of integer indices. These concepts can also be applied to link, CN, and ensemble references.

The Attribute Base

The *Attribute Base* is a hierarchically organized system for storing attributes which are to be assigned to the network components. To create a BIF file, a specific attribute base is constructed containing the attributes used by the BIF model. The hierarchy consists of *groups*, *lists*, and *records*. Figure 3.4 shows the attribute hierarchy. The attribute base is constructed top to bottom. The reason for having the attributes is that they can give a component some semantic meaning, allowing each to take on a set of unique properties. The attributes also provide a mechanism and abstraction above a component's reference identity by which to reference it. This means that a component can be referenced by virtue of its attribute content.

NDL's attribute base was implemented using a three level hierarchy. A n-level hierarchy would have been better and less restrictive, but much harder to implement.

The *attribute record* is the fundamental unit in the Attribute Base. It is the information carrying entity. In figure 3.4, the attribute records are the terminal nodes in the Attribute Base tree. An attribute record consists of its name and a value. The value can be either a character string,

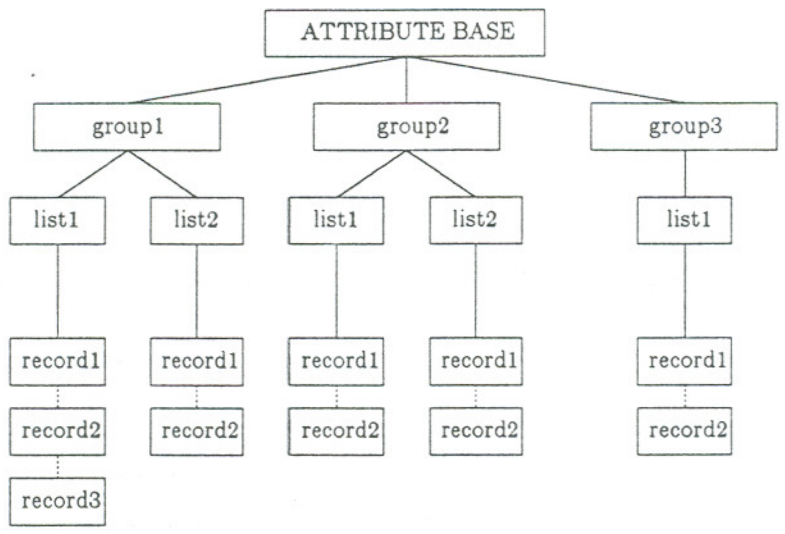

Figure 3.4 - The Attribute Base

integer, or floating point number.

The *attribute list* is a collection of attribute records. It provides a way of organizing attribute records. An attribute list consists of its name and the records it contains.

The *attribute group* is a collection of attribute lists. It provides a way of organizing the attribute lists. An attribute group consists of its name and the lists it contains.

The Attribute Reference

The purpose of the *attribute reference* is to provide a mechanism for which to reference the records. An attribute reference is a 3-tuple, and is denoted as

$$\text{ref} = (\text{group},\text{list},\text{record})$$

Attribute references can point to an attribute group, an attribute list, or an attribute record. If the record element is omitted, then the records referenced are all those contained in the list specified by the group and list elements. If both the record and list components are omitted, then the records referenced are all those contained in all the lists contained in the group specified. If all three elements are omitted from the reference, then it is assumed that all the records in the attribute base are referenced.

The Attribute Access List

The *Attribute Access List* (AAL) is a list of attribute references. It provides a mechanism for specifying an arbitrary collection of attributes from the Attribute Base. It should not be confused with the attribute list, which is a collection of attribute records. The attribute access list can be denoted as

$$\text{AAL} = (\text{ref}_0,\ \text{ref}_1,\ \text{ref}_2,\ldots,\text{ref}_{i-1})$$

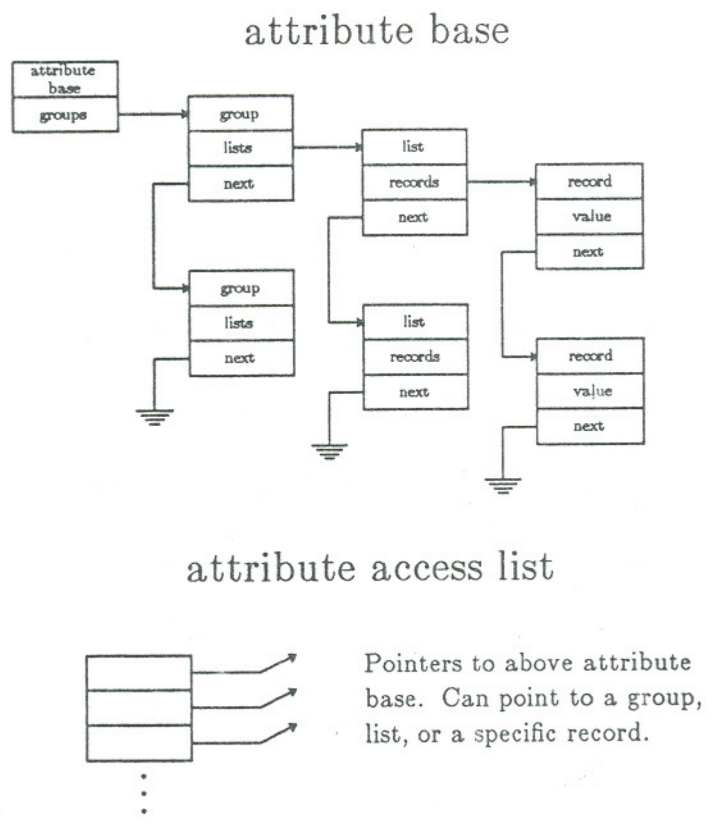

Figure 3.5 - The Attribute Base Data Structures

The attribute access lists are assigned to the network components as they are created.

Generalizations

A *generalization* is a mechanism and an abstraction by which a collection of objects with a similarity in attribute content can be referenced by a single identifier. A generalization in NDL is an attribute access list used for referencing the network components. A component is said to satisfy a generalization if it contains all the attribute records the generalization specifies. For example, all the CNs in a graph with a particular set of attribute records can be referred to with a special identifier assigned to them. Whenever the identifier is used in an operation, only those components which satisfy the generalizations predicate are included in the operation. The generalization can be denoted as

$$\text{GEN} = (\text{ref}_0, \text{ref}_1, \text{ref}_2, \ldots, \text{ref}_{i-1})$$

The generalization is used in much the same manner as the aggregate was used in the network component reference. As in the aggregate, an example should make this concept a little easier to grasp. A site is referenced by an ensemble index, a CN index, and a site index. The reference uniquely identifies one site in the tree. Now consider that the site is referenced by an ensemble index, a CN generalization, and a site index. The site reference now identifies a site on each CN in the ensemble specified which satisfies the CN generalization.

Scopes

A *scope* is a mechanism for combining the functionality provided by both the aggregate and the generalization. It takes an aggregate, specifying a set of component indices, and performs an operation on only those indices which satisfy a generalization. It is a way of saying "connect a CN to all the CNs in aggregate A which satisfy the generalization B." A scope can be denoted as a 2-tuple.

$$\text{scope} = (\text{aggregate,generalization})$$

Again the same example is used as in the aggregate and the generalization. This time the site reference consists of an ensemble index, a scope specifying which CNs to use, and a site index. The reference now identifies a site on each CN in the ensemble specified which both belongs to the aggregate specified in the scope and satisfies the generalization specified in the scope.

Chapter 4

Creating A Network Graph

In the previous chapter, the NDL model was presented. The purpose of presenting such a model was to formalize the concepts used in NDL. A major goal in developing NDL was to create a programming environment composed of abstractions consistent with the model. Terry Winograd stated that the fundamental use of a programming language should not be in creating sequences of instructions for accomplishing tasks, but in expressing and manipulating descriptions of operations and the objects which they are carried out [Win79]. In keeping with the spirit of that statement, the abstractions should help support a declarative environment.

To generate a network graph in NDL, a C++ program is written using functions from the NDL primitive function library. These functions provide the level of abstraction described above consistent with the NDL model. C++ was particularly useful since function overloading could be utilized, allowing same name functions with different types of argument lists. In high-level programming languages, three types of abstraction mechanisms are most often recognized: control, data, and procedural [Isn82]. All three are utilized by the NDL primitive function library and features supported by C++.

The rest of this chapter describes the functions which create the network graph. An attempt was made to present them in a sequence as they might be found in a program.

Initializing NDL

Before any other NDL function is called, the NDL environment must first be initialized. The NDL function *init_ndl* performs this initialization. It creates the internal NDL data structures needed to support all other NDL functions.

As links are created in the network, weights are assigned to them. Control can be exercised over what values these weights assume. In most cases, random weights are desired. The function *set_weight_limits* sets the limits on the range of random weights generated. The function is overloaded allowing either integer or floating point values. Another function, *set_weight_seed*, sets the random seed used. Different sequences of random weights can therefore be generated for a given network graph.

Creating the Network Components

After NDL is initialized, the network components can be created. The components are created in the order they would appear in the network component tree, starting with the ensemble nodes. The network can be constructed in any order, it is only necessary that for any component created, there must already exist a path to it in the component tree. For example, before a site can be created, the CN it belongs to and the ensemble the CN belongs to must have previously been created.

Creating the Ensemble

Ensembles are created and added to the network by the following function:

create_ensemble(n)

Multiple ensembles can be created with the optional n argument to the function. The default is one. The index of the ensemble created is returned by the function. If multiple ensembles are created, the index of the first is returned. Subsequent operations regarding the ensemble reference it by its index. The first ensemble created is assigned an index of 0. Subsequently created ensembles receive consecutive integer indices.

Creating the CN

CNs are created and added to an ensemble by the following function:

create_cn(ens, n)

The *ens* argument specifies in what ensembles to create the CNs. Multiple

CNs are created in each ensemble specified with the optional *n* argument. The default is one. This ensemble specification can be one of the following :

(1) ensemble index

(2) aggregate

(3) generalization

(4) scope

(5) null

If the ensemble specification is an index, then the CN is created in the ensemble referenced by the index. If the ensemble specification is an aggregate, then a CN is created in each of the ensembles referenced in the aggregate. If the ensemble specification is a generalization, then a CN is created in each ensemble that satisfies the generalization. This applies to all existing ensembles in the current network graph. If the ensemble specification is a scope, a CN is created in each ensemble which both belongs to the aggregate specified in the scope and satisfies the generalization specified in the scope. If the ensemble specification is *null* (zero address), a CN is created in all ensembles in the network graph.

The first CN created in an ensemble is assigned a 0. All subsequent CNs are assigned consecutive integer indices. Each ensemble in the network has CNs indexed from 0 to however many CNs belong to it. The ensemble index and the CNs index are used in subsequent operations regarding the CN.

Creating the Site

Sites are created and added to a CN by the following function:

create_site(ens, cn, n)

The *ens* argument specifies in what ensembles to create the sites. The *cn* argument specifies in what CNs to create the sites. Multiple sites can be created with the optional n argument. The ensemble and CN specifications work like the ensemble specification in the create_cn function. The difference is that a site is created on each CN specified in the CN specification for each ensemble specified in the ensemble specification. For programmers, this might be best thought of as two nested loops. The outside loop consists of the ensemble specification, and the inside loop the CN specification. If the n argument is specified, then n sites will be created on each CN specified.

The first site created in a CN is assigned a 0. All subsequent sites are assigned consecutive integer indices. Each CN in each ensemble has sites indexed from 0 to however many sites belong to it. The ensemble index, the CN index, and the sites index are used in subsequent operations regarding the site.

Creating Links

Links are created and added to a site by the following two functions:

create_site(*ens, cn, n*)

create_site(*ens1, cn1, site1, ens2, cn2, site2*)

The first creates a single-ended link, and the second a double-ended link. The single-ended links are created in the same manner that CNs and sites were created. In the argument list is an ensemble, a CN, and a site specification. A link is created for each site specified for each CN specified for each ensemble specified. For programmers, this might be thought of as three nested loops. The outside loop represents the ensemble specification, the middle loop represents the CN specification, and the inside loop represents the site specification.

The double-ended loop is created using two sets of specifications, six arguments in all. The first three specify a site to create a link on, and the second three specify another site to create a link on. A double-ended link is created between each pair of sites specified. This is probably best thought of again in programmer's terms. It is simply the single-ended links case with another three nested loops placed inside its inside loop for a total of six nested loops.

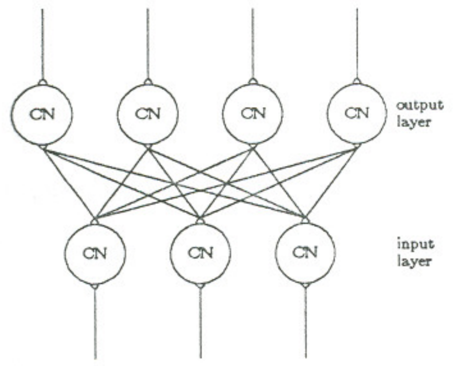

Figure 4.1 - Example: Network Components

Network Component Example

An example will demonstrate the generation of a simple network graph. The network graph to be generated is shown in figure 4.1. The program generating the graph is shown in figure 4.2. The example is generated using only single index specifications in the create functions. The graph consists of an input layer and an output layer. The input layer has three CNs and the output layer has four. Each CN in the graph has two sites. The site on the bottom will be assigned the index of 0 (created first) and the one on top the index of 1 (created second). The links connecting the input layer and the output layer are double-ended links. The links going into the input layer and coming out of the output layer are single-ended links.

```
create_components()
{
 // create the ensemble

    int ens = create_ensemble();

 // create the CNs

    create_cn(ens,7);

 // create 2 sites on each CN

    for(cn=0; cn<7; cn++)
      create_site(ens,cn,2);

 // create input to output links

    for(int cn1=0;cn1<3;cn1++)
      for(int cn2=3; cn2<7; cn2++)
        create_link(ens,cn1,1,ens,cn2,0);

 // create input links

    for(cn=0; cn<2; cn++)
      create_link(ens,cn,0);

 // create output links

    for(cn=3; cn<7; cn++)
      create_link(ens,cn,1);
}
```

Figure 4.2 - Creating Network Components

The Aggregate

Aggregates are created by the following functions:

create_range_aggregate(*start,finish*)

create_list_aggregate()

A handle to the new aggregate is returned by the functions. The *range* aggregate consists of a contiguous set of component indices ranging from *start* to *finish.* The *list* aggregate consists of a list of non-contiguous component indices. Aggregates are removed from NDL with the *remove_aggregate* function. The result of a set operation on two aggregates can be appended to a third aggregate with the *and_aggregate* and *or_aggregate* functions.

The network graph shown in figure 4.1 is generated using aggregates by the program shown in figure 4.3. Note the difference between the program generating the network graph using single index specifications versus the program in figure 4.3 using aggregate specifications. The single most important difference is the absence of loops. The **AGGR** in the program specifies the aggregate data type.

The Attribute Base

The Attribute Base contains the attributes that are assigned to the network components. The attributes are arranged in a hierarchy consisting

```
create_components()
{
  // create the ensemble

      int ens = create_ensemble();

  // create the CNs

      create_cn(ens,7);

  // create the layer aggregates

      AGGR* input  = create_range_aggregate(0,2);
      AGGR* output = create_range_aggregate(3,6);
      AGGR* all = create_range_aggregate(0,6);

  // create 2 sites on each CN

      create_site(ens,all,2);

  // create double-ended links between input and output layers

      create_link(ens,input,1,ens,output,0);

  // create input and output single-ended links

      create_link(ens,input,0);
      create_link(ens,output,1);
}
```

Figure 4.3 - Creating Network Components Using Aggregates

of *groups*, *lists*, and *records*. Reference lists into the attribute base are then constructed. These reference lists are called Attribute Access Lists. A typical program will build the Attribute Base before the network components are created. Then as the components are created, their associated attributes can be assigned to them.

Attribute Groups

The attribute base is divided into attribute groups. They allow the user to create a set of attributes particular to some domain. For example, the attributes for links might be grouped into a class of attributes particular to links. The same for the sites, CNs, and the ensembles. Attribute groups are created and added to the Attribute Base by the function

create_attribute_group(*group*)

The *group* character string argument is assigned to the group is used to reference the group and all the lists and records it contains.

Attribute Lists

An attribute group can be further divided into attribute lists. The lists allow the user to create sub-domains within a domain. For example, there might exist an attribute group for link attributes. This group can then be divided into lists which could contain the attributes for input links and output links. Attribute lists are created and added to the Attribute Base by the function

create_attribute_list(*group,list*)

The *group* argument specifies which group to add the list to. The *list* argument specifies the name assigned to the list. To reference the list in subsequent operations, both the group and list names must be provided.

Attribute Record

Each attribute list contains a set of attribute records. The record contains the actual information contained in a single attribute. A record can be either a floating point number, integer number, or a character string. Attribute records are created and added to the Attribute Base by the function

create_attribute_record(*group,list,record,value*)

The *group* and *list* arguments specify where to add the record. The *record* argument specifies the name assigned to the record. The *value* argument can be one of three types : integer, float, or character string. To reference the record in subsequent operations, the group, the list, and the record name must be specified.

Attribute Access List

An Attribute Access List (AAL) is created by the function *create_AAL*. The handle to the AAL is returned. An AAl can be created by specifying either another AAL or an attribute reference as an argument.

If another AAL is specified, then the new AAL contains the attribute references of the one it was created with. If an attribute reference is specified, then the new AAL contains the attribute reference specified. For example,

create_AAL("*links*")

creates an attribute access list starting with the reference to the attribute group *links*. An AAL with no references can be created by not specifying either an AAL or an attribute reference on function invocation.

Once an AAL has been created, additional attributes can be appended to it by the *add_to_AAL* function. Another AAL can be appended, resulting in a new list consisting of the original plus the references from the appended list. In addition, single attribute references can be appended to an AAL. The *remove_AAL* function removes the AAL from NDL.

Assigning Attributes to the Network Components

Attributes are assigned to the network components when the components are created. There is a global variable designated for ensembles, CNs, sites, and links. As each component is created, it is assigned its respective AAL. The global variables may be changed so as to assign the same type of component a different set of attributes. For example, input sites might have a different set of attributes than the output sites. The functions *set_link_AALs*, *set_site_AAL*, *set_cn_AAL*, and *set_ens_AAL* set their respective global variables.

Creating the Generalization

A generalization is created by the function

create_generalization(*ar*)

The *ar* argument specifies either an attribute reference or an attribute access list for which to assign the generalization. The handle to the generalization is returned. The generalization is much like the AAL, only it is used in specifying which network components to include in an operation. The functions regarding the generalization are all like those concerning the AAL's, but with syntax indicating a generalization is involved. In figure 4.4, an example program is shown. It creates the same network graph the previous examples did. In the example program, it is assumed that the attribute base has been built. In that attribute base, there is an attribute group named "input" and an attribute group named "output". Notice that the generalizations are created first. Next notice that the input and output CNs are created with the *input* and *output* attributes respectively. Finally, notice that the *create_link* functions utilize the generalizations in creating the links.

```
create_components()
{
 // create the generalizations

    GEN *input  = create_generalization("input");
    GEN *output = create_generalization("output");

 // create the ensemble

    int ens = create_ensemble();

 // create the Input and Output CNs

    set_cn_aal(input);
    create_cn(ens,3);
    set_cn_aal(output);
    create_cn(ens,4);

 // create 2 sites on each CN

    create_site(ens,NULL,2);

 // create double-ended links between input and output layers

    create_link(ens,input,1,ens,output,0);

 // create input and output single-ended links

    create_link(ens,input,0);
    create_link(ens,output,1);
}
```

Figure 4.4 - Creating Network Components Using Generalizations

Scopes

A scope is created by the function

create_scope(*aggr*, *gen*)

The *aggr* argument specifies an aggregate to assign to the scope and the *gen* argument specifies a generalization to assign to the scope. The handle to the new scope is returned. The function *set_scope* sets the appropriate constituent. The function *remove_scope* removes the specified scope from NDL.

```
create_components()
{
 // create aggregates - generalizations - scopes

     AGGR* inputs  = create_range_aggregate(0,1);
     AGGR* outputs = create_range_aggregate(2,5);
     GEN*  left = create_generalization(leftOutCN);
     GEN* right = create_generalization(rightOutCN);
     SCOPE* l_out = create_scope(outputs,left);
     SCOPE* r_out = create_scope(outputs,right);

 // create double-ended links between input and output layers

     create_link(ens,inputs,1,ens,l_out,0);
     create_link(ens,inputs,2,ens,r_out,0);

 // create input and output single-ended links

     create_link(ens,inputs,0);
     create_link(ens,outputs,1);
}
```

Figure 4.5 - Creating Network Components Using Scopes

An example program is shown in figure 4.6 which generates the network graph shown in figure 4.7. The example is a bit contrived, but illustrates the use of a scope. The example program assumes that the attribute base has already been built. Three types of CNs were created with different attribute access lists. Those CN types are associated with the three attribute access lists *inputCN*, *leftOutCN*, and *rightOutCN*. The first *create_link* function call utilizes an aggregate specifying the input CNs and a scope specifying the left output CNs. The second *create_link* function call utilizes an aggregate specifying the input CNs and a scope specifying the right output CNs. The scopes specify the left and right output CNs by combining the fact that they belong to the aggregate *outputs* and contain their respective attributes specified by the generalizations *leftOutCN* and *rightOutCN*. The final two "create link function" calls create the input and output single-ended links by using their respective aggregates.

Outputting Network Graph Information

Outputting the network graph is the user's responsibility once the graph has been created. The information which can be output is the graph structure information, Attribute Base information, aggregates, generalizations, and scopes.

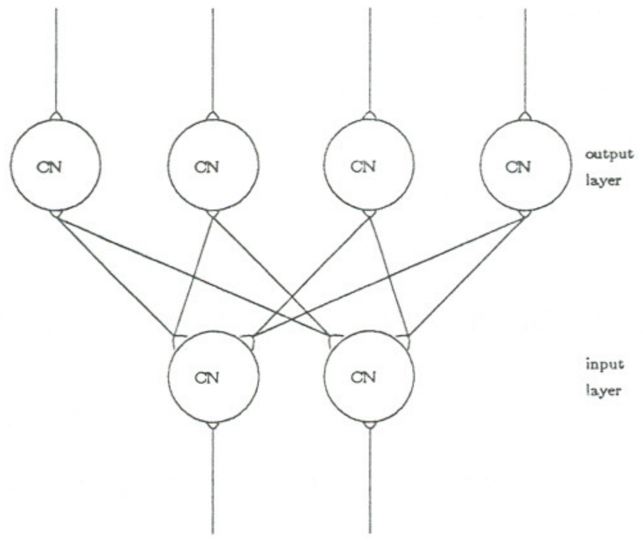

Figure 4.6 - Network Graph for Scope Example

The *print_network* function outputs the entire network graph. It starts at the top of the network component tree and performs a pre-order traversal of the tree, outputting each component and its attributes as they are encountered. The *print_ensemble* function outputs the information for just the designated ensemble. It starts at a designated ensemble node in the network component tree and performs a pre-order traversal from that node down. Each component and its attributes are output as they are encountered. The *print_cn* and *print_site* functions operate in the same manner as the ensemble and CN print functions do. The *print_link* function simply outputs the information about a specified link. That output consists of its connection data, weight data, and any attributes assigned to it.

The attribute base can be output using three functions. The *print_record* function prints the name and the value for the specified attribute. The *print_list* function prints the name and value of each record it contains. The *print_group* function prints the name and value of each record it contains. This includes all the records in each list the group contains.

The *print_aggregate* function simply outputs the indices it contains. The *print_generalization* outputs the records it contains by virtue of the attribute base references it contains. The *print_scope* function outputs both the aggregate information and the generalization information it contains. The *print_AAL* function outputs the records it contains by virtue of the attribute base references it contains.

Chapter 5

The NDL Environment

This chapter describes the NDL environment and the process of developing a NDL executable program. The development process and the cooperating constituents are shown in figure 5.1. The NDL include file defines the data abstractions used. The user interface library contains the primitive NDL functions. The application specific library contains functions specific to a particular neural network topology. The user program contains the user's program with embedded NDL user interface and application specific functions. The C++ compiler creates the executable user program which in turn will generate an output file describing the network graph generated. Optional output filter programs will convert the output to a form acceptable by some other program. The steps in generating a network graph can therefore be outlined as follows:

(1) write user source program

(2) compile source user program into executable program

(3) run the executable program

(4) run output through desired output filter program

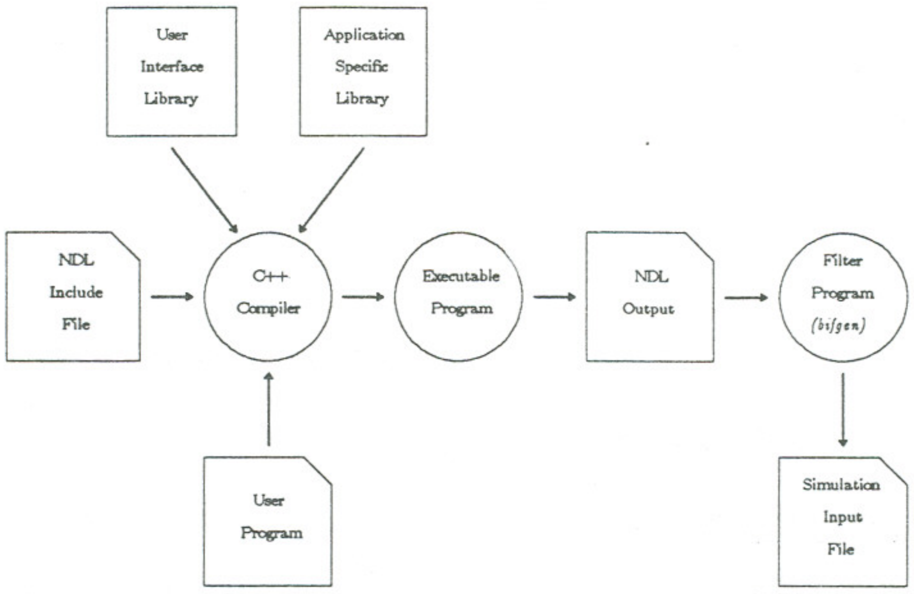

Figure 5.1 - The NDL Program Development Environment

The NDL Include File

The NDL include file contains the data abstractions used by the NDL user interface functions and the application. Although not in the spirit of good C++ object oriented programming, a few global data structures are defined so as to relieve the user of having to include them in every function call made.

The NDL User Interface Library

The NDL User Interface Library contains all the primitive NDL functions. The functions provide a stable interface to the user. It is likely that any future additions will be at a higher level consisting of lower level function calls from the library.

Application Specific Libraries

An application specific library contains functions specific to a class of topologies. For example, layered network topologies are used extensively in neural network research. The *layered network* application library consists of functions built on top of the NDL User Interface Library functions and provide a level of abstraction above the details of constructing a layered network using the primitive functions.

A very important application specific library to the users at OGI is the BIFLIB library. It contains all the functions necessary to build a network which can be output as a BIF file. The main functionality it provides is the construction of the attribute base and the output capability. An appendix is dedicated to describing the BIF environment.

The User Program

The user program is a C++ program utilizing the functions provided by the User Interface and Application Specific libraries. It was intended that the user would simply call NDL functions and not have to deal with the C++ programming paradigm.

Output Filters

The only output filter to date is *bifgen.* It takes output from a NDL program and converts it into a BIF file. Since this is the only form currently desired, no others were written.

Chapter 6

Layered Networks

The majority of the neural networks being studied today consist of multiple feed-forward layers. This motivated the need for an application specific function library dedicated to constructing layered networks. This application library is not part of the primitive NDL functions discussed in chapter four. It is an example of the extensibility of NDL and how higher levels of abstractions can be achieved.

The Layer Model

Each layered network is created in a single ensemble. There are three types of layers one can create in the layer model: the *input* layer, the *hidden* layer, and the *output* layer. Each layer consists of an arbitrary number of CNs. No constraint is placed on the number of layers created. The only requirement is that network input data enter through an input layer and network output data go through an output layer. Each CN in the model consists of only two sites. This is not a restriction of NDL, which allows an arbitrary number of sites, but a constraint imposed by the model. One site is dedicated for input from either another layer or network input.

The second site is dedicated for output, either to another layer or network output.

When an input layer is created, a specified number of CNs will be created. Each CN created will have an input site with an index of zero and an output site with an index of one. Each input site will have a single-ended link assigned to it. These input links are the input into the network. An input layer is created by the function *create_in_layer.* A *name* argument specifies the name to assign the layer created. A *size* argument specifies how many CNs to create in the layer.

When the hidden layer is created, a specified number of CNs will be created. As with the input layer CNs, each CN in a hidden layer will have an input site with an index of zero and an output site with an index of one. However, since this is an internal CN, no single-ended links are created. The sites will normally be connected with double-ended links to other sites in input, hidden, and output layer CNs. A hidden layer is created by the function *create_hid_layer.* A *name* argument specifies the name to assign the layer created. A *size* argument specifies how many CNs to create in the layer.

When the output layer is created, a specified number of CNs will be created. As with the input and hidden layer CNs, each CN in an output layer will have an input site with an index of zero and an output site with an index of one. Each output site will have a single-ended link assigned to it. These output links are the output of the network. An output layer is created by the function *create_out_layer*. A *name* argument specifies the name to assign the layer created. A *size* argument specifies how many CNs to create in the layer. The network in figure 6.1 was generated by the following sequence

```
create_in_layer("input",4);
create_hid_layer("hidden",4);
create_out_layer("output",4);
```

The network is incomplete in that it has no connections (double-ended links) between the layers of CNs. To perform that function, four intercon-

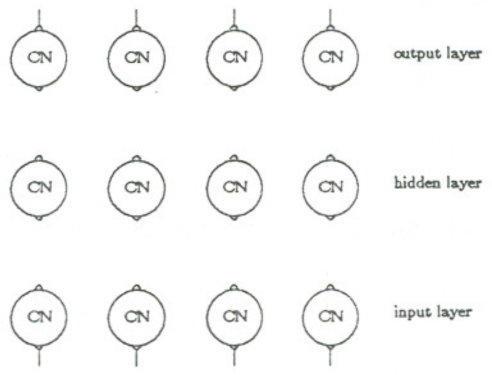

Figure 6.1 - Types of layers in the Layer Environment

nection schemes have been included which to create the links.

Interconnection Schemes

Many network models consist of layers which are fully connected between two consecutive layers. That is, all the CNs in one layer are connected to all the CNs in the next layer. To accommodate this type of interconnection, *full* interconnect is provided which creates a link between each output site in one layer to each input site in a second layer. The function *connect_full* performs this connection operation. Two arguments to the function specify the names of the layers to connect. Figure 6.2 shows an example of two layers fully connected.

Because neural structures tend to be randomly connected, some network models use random connections between layers. To facilitate this in the layer model, *random* interconnect is used. The function *random_connect* performs this connection operation. Its arguments include the names of the two layers and a number specifying the probability that a link will be created which would normally be created using full interconnect. In figure 6.2, the random network was created using a .50 probability of connection.

Some neural network models use a concept known as receptive fields. This is best described using a two dimensional image. A receptive field

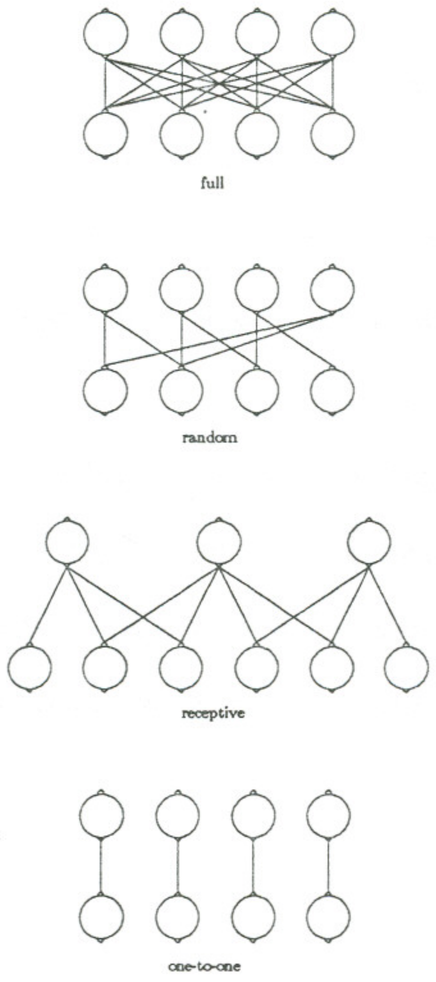

Figure 6.2 - Interconnection Schemes between Layers

would cover a small contiguous area of the image and maybe be tuned such that it would be sensitive to a particular feature. Receptive fields can and usually do overlap each other. This third type of connectivity is referred to as the *receptive field.* Each CN in a layer is connected to a contiguous field of CNs in the layer beneath it. The receptive field size is specified and the function determines a balanced way to connect the two layers. The function *connect_receptive* performs this connection operation. Its arguments include the names of the layers and the size of the receptive field. Figures 6.2 shows an example of receptive interconnections between two layers.

Sometimes it is desirable to just connect two layers together in a straight through fashion. That is, one CN in one layer is connected to one CN in the next layer. This fourth type of connectivity is called *one-to-one.* The function *connect_one_to_one* performs this connection operation. The output sites of the first layer are connected to the input sites of next layer. Figures 6.2 shows an example of one-to-one interconnections between two layers.

Layer Example

The program in figure 6.3 creates a network utilizing the full, one to one, and receptive fields connection schemes. The network consists of five layers which can be broken down into a two layer receptive field network feeding a three layer fully connected network. The network graph created by this program is shown in figure 6.4.

```
main()
{
 init_ndl();
 layer_environment();
 create_in_layer("layer1",8);
 create_hid_layer("layer2",6);
 create_hid_layer("layer3",6);
 create_hid_layer("layer4",6);
 create_out_layer("layer5",4);
 connect_receptive("layer1","layer2",3);
 connect_one_to_one("layer2","layer3");
 connect_full("layer3","layer4");
 connect_full("layer4","layer5");
}
```

Figure 6.3 - A Layer Environment Program Example

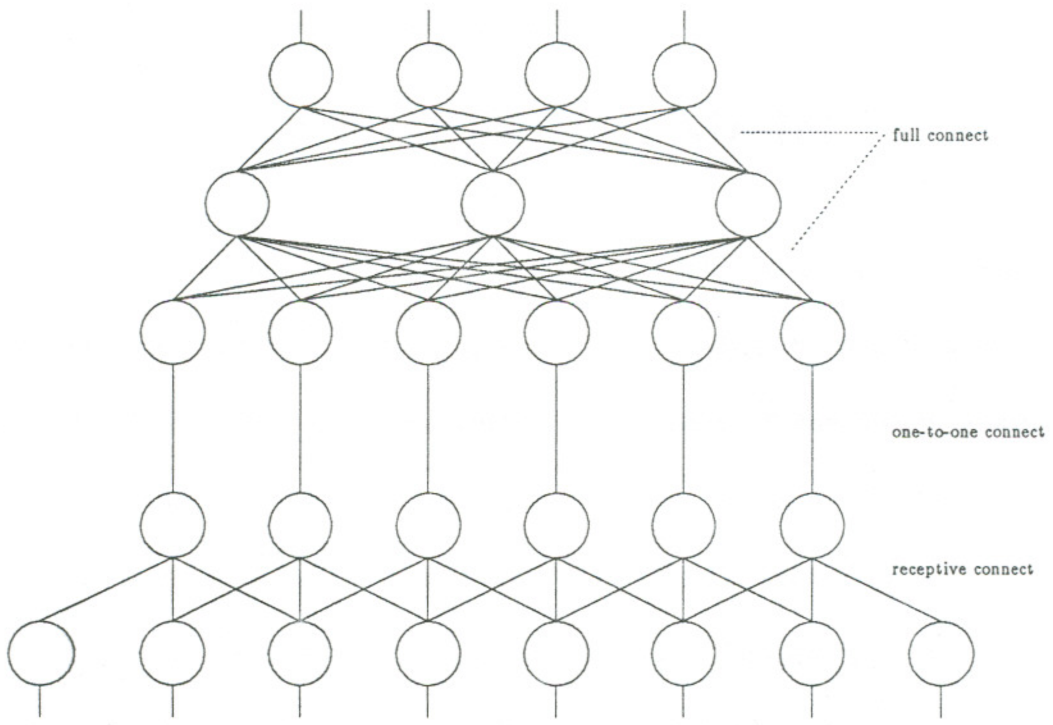

Figure 6.4 - A Layer Network Example

The first two function calls initialize both NDL and the layer model environment respectively. Extra data structures were required to facilitate referencing layers by name. The following five function calls create the layers in a bottom up fashion. That is, the bottom (input) layer first, the first hidden layer second, and so on. The last four function calls create the connectivity between the layers. This is also done bottom up. Remember that the input and output single-ended links were created when their respective layers were created.

Chapter 7

A Pyriform Cortex Model

Gary Lynch, Richard Granger, and their colleagues at the University of California at Irvine are developing a model simulating the olfactory pyriform cortex[GrL]. The olfactory cortex is the part of the brain which preprocesses the information coming from the nasal receptors in the nose via the olfactory bulb. The structure of the olfactory cortex is relatively simple when compared to the other types of cortex. The nasal receptors are directly connected to the neurons in the olfactory bulb, which is directly connected to the layer II neurons in the pyriform cortex, which supply input into the hippocampus and frontal cortex. These direct connections, called monosynaptic connections, indicate that information flows in one direction. There is a feedback loop involving a structure called the anterior olfactory nucleus (AON), but it is ignored in this model. It also means that the cortex is only two to three synapses from the sense organ. These features make simulating the olfactory pyriform cortex particularly attractive to neural researchers.

One of the long range goals of the CAP group at OGI is to develop advanced neural hardware out of silicon. The neural hardware will be implemented using wafer-scale integrated chip architecture. It is intended that the architecture be based on biological neural networks. Because of the relatively regular structure of pyriform cortex, it has been selected as the most likely structure on which to base the design. The Lynch-Granger model is one of many computer simulated models involving pyriform cortex. It is however the most functional and has the most precise definition. For these reasons, the Lynch-Granger model of the pyriform cortex has been chosen for the research at OGI.

The NDL Model

As an extension to NDL, an application specific library was created which builds a model of the pyriform cortex. It is based on the Lynch-Granger model. The model consists of two basic components: The Lateral Olfactory Tract (LOT) and the patch. A diagram of the model is shown in figure 7.1. Information in the figure flows left to right.

The LOT

The LOT consists of outputs from the LOT CNs. The input patterns seen on the LOT CN inputs would be those resulting from a smell presented to the nasal receptors in the nose. In the real brain, the size of the LOT is relatively constant from input to output. Although not evident from figure

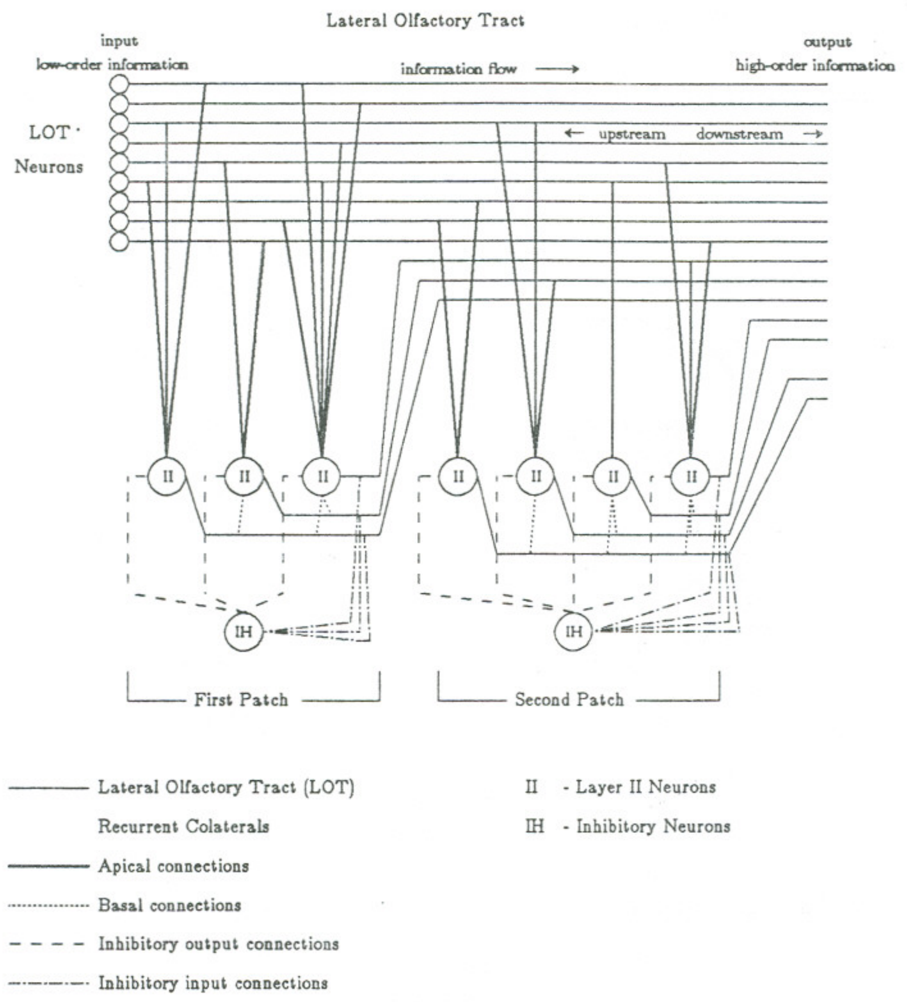

Figure 7.1 - A Olfactory Pyriform Cortex Model

7.1, the original outputs from the LOT CNs eventually disappear and are replaced by the recurrent colateral outputs from the Layer II neurons. Because the recurrent colateral outputs eventually dominate the LOT, it would appear that the LOT must represent some higher order information downstream than what was presented at the LOT CNs. The LOT CN consists of an input site and an output site. Refer to figure 7.2.

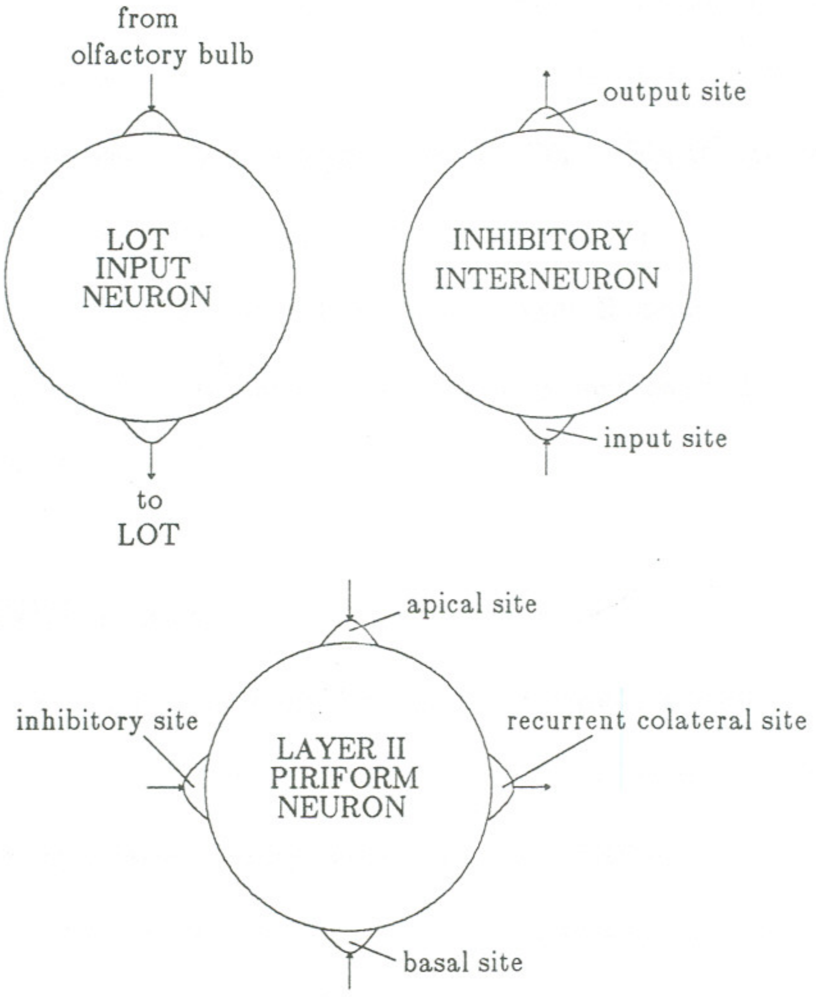

Figure 7.2 - The Pyriform Cortex Model CNs

The Patch

A patch consists of two types of CNs, the Layer II CNs and the inhibitory CNs. The Layer II CNs are the most complex CNs in the model. They receive inputs from the LOT (called apical connections), from the recurrent colateral output of the other Layer II CNs in its patch, and the inhibitory CN within the patch. The output from the layer II CN, called the recurrent colateral, is sent to other layer II CNs within the patch, and are called basal connections. The recurrent colaterals also join the LOT. The inhibitory CN receives input from all the layer II neurons within its patch and outputs to all of them. The inhibitory and Layer II CNs are shown in figure 7.2.

Using the NDL Model

To generate a olfactory pyriform cortex network, a NDL program with the functions specific to the model needs to be compiled with the Lynch-Granger application library. There are six functions in this library. These coupled with the normal NDL initialization function are all that is required to generate the network. These functions are described next.

Environment Initialization

There are some house keeping duties to be performed before the pyriform network is generated. They are performed by the following function:

pyriform_environment()

It must be called before any other pyriform function and after the NDL primitive function *init_ndl*. The house keeping involves creating an instance of the PYRIFORM class data structure from which to build the network.

Setting the LOT Parameters

There are two parameters which describe the structure of the LOT. Both are specified in the following function :

set_lot(*size, die*)

The *size* argument specifies the constant width of the LOT. The *die* argument specifies what percentage of the LOT is replaced by recurrent colaterals. For example, suppose the size of a LOT is 100 and its die rate is 10. What this means is that after every patch, 10 percent of the LOT is replaced by recurrent colateral outputs from the patch. In this example, that means 10 LOT lines would be replaced by 10 recurrent colateral outputs.

Setting the Patch Parameters

To keep the model as faithful as possible to the real biological hardware, random variables are introduced so an irregular network is produced. There are three parameters which can be specified describing the structure of the patch: number of layer II CNs, number of LOT connections, and the interconnection of the layer II CNs to each other. All three are specified by the following function :

set_patch(*patchlo,patchhi,lotconprob,basalprob*)

The *patchlo* and *patchhi* arguments specify the range of how many Layer II CNs will be generated per patch. For example, if a range of 5-9 were specified, then an integer number would be randomly selected between 5 and 9 each time a patch is created, and that number would be how many layer II neurons would be created for the patch. The endpoints 5 and 9 are included.

The *lotconprob* specifies the probability that each LOT line has of being connected to a Layer II CN. For example, if the probability parameter is 10, then each LOT line has a 10 percent chance of being connected to a Layer II CN. Put another way, if the lot size is 20, then there would be on the average two connections to the LOT per Layer II CN.

The *basalprob* argument specifies the probability that a downstream layer II CN will be connected to the recurrent collateral of an upstream layer II CN within a patch. The layer II CNs in the patches go from top to bottom (upstream to downstream). An upstream CN is a CN to the left. A downstream CN is a CN to the right. A value of 65 would mean that a downstream CN would have a 65 percent chance of being connected to any one of the upstream CNs within its patch.

Creating the Patches

Once the LOT and patch parameters have been set, the network can be constructed. The following function creates a specified number of patches using the LOT and patch parameters:

create_patches(n)

The argument n specifies how many patches to add to the network. The patch parameters can be changed between patches. This gives the ability to alter patch characteristics between patches.

Outputting A Pyriform Network

The output of the pyriform model consists of three parts. The first part describes the connectivity of the LOT neurons. The second the connectivity of the patches. The third part describes the final form of the LOT. The following function outputs the pyriform network :

output_pyriform()

```
main()
{
 // initialize the NDL environment

    init_ndl();

 // initialize the Pyriform Model environment

    pyriform_environment();

 // set the LOT parameters

    set_lot(100,5);

 // set the network patch parameters

    set_patch(5,10,20,75);

 // create the Lynch-Granger network

    create_patches(15);

 // output the Lynch-Granger network

    output_pyriform();
}
```

Figure 7.3 - An Example Pyriform Model Program

Compiling The Program

The following command will compile the example shown in figure 7.3.

```
CC -o lg main.c -lNDL -lLG
```

The executable program *lg* will output to *standard out* the final pyriform network.

The Pyriform Model Implementation

The pyriform model is implemented by introducing three new data structures not part of NDL proper. The *PYRIFORM* data structure stores the pyriform model parameters and pointers to the other two new data structures. The *LOT* data structure stores the site reference for each LOT axon. This includes the ensemble, the CN, and the site. The *PATCH* data structure stores the ensemble references representing the patches. Each new patch is dedicated a new ensemble and added to the linked list of PATCH's in the PYRIFORM data structure. Each patch ensemble contains the Layer II neurons for the patch and the inhibitory neuron for the patch. Figure 7.4 shows the relationships of the data structures. Each PYRIFORM data structure consists of one LOT data structure and n PATCH data structures, n being the number of patches in the model being built.

The function *pyriform_environment()* creates the PYRIFORM data structure and initializes it. A global variable *pyriform* is assigned the address of the new PYRIFORM structure.

The *set_lot* function sets the LOT size and LOT die model parameters in the PYRIFORM data structure for the other pyriform functions to use. In addition, it creates the input lot CNs in a dedicated ensemble. Each input LOT CN is created with two sites, site 0 for input from the olfactory bulb, and site 1 for output to the LOT. Finally the set_lot func-

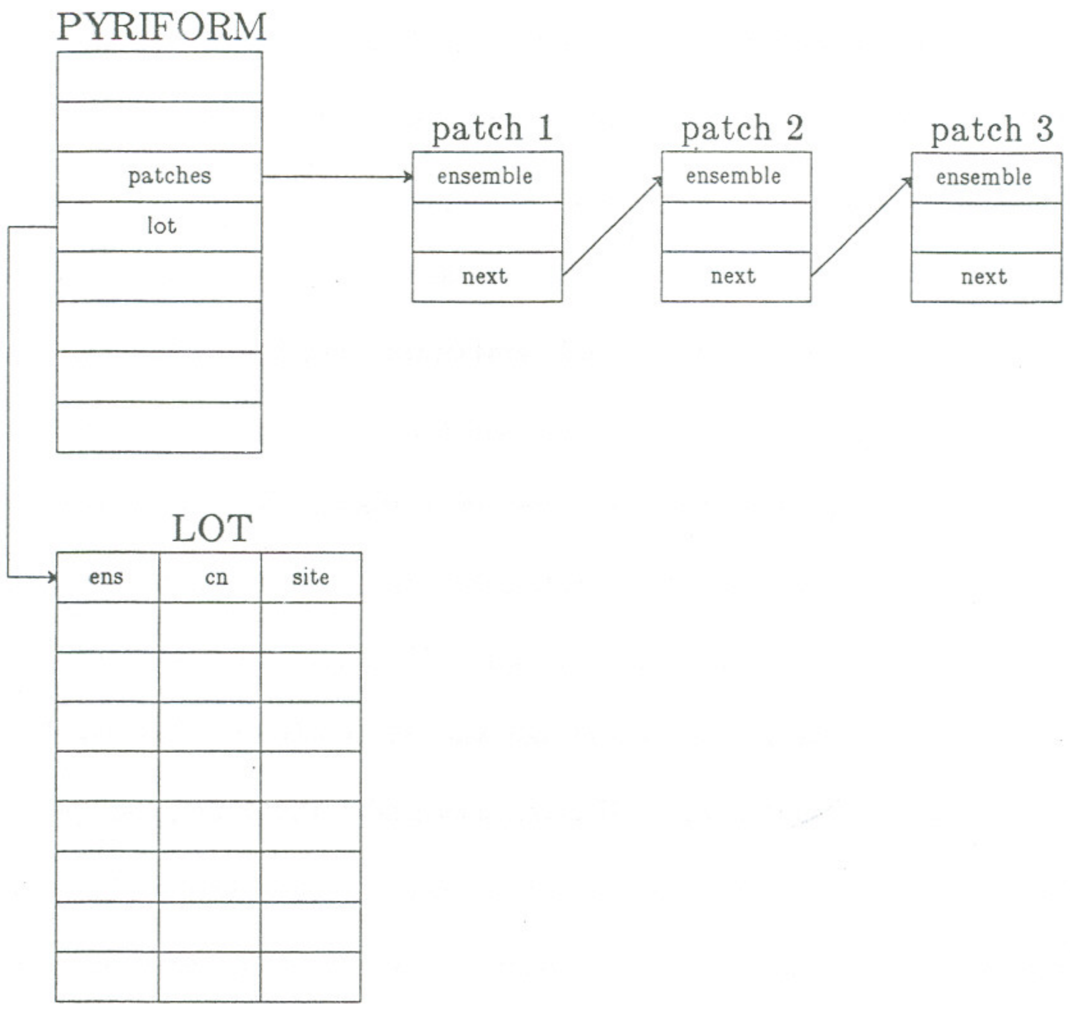

Figure 7.4 - The Pyriform Model Data Structure Diagram

tion creates the LOT data structure. Each input lot CN is entered into the LOT data structure. Eventually the recurrent colateral outputs from the Layer II CNs in the patches will replace these input LOT CNs in the LOT data structure.

The *create_patch* function is the workhorse function of the model. First it creates a new PATCH data structure and creates its ensemble. To keep referencing simple, it was decided that an ensemble would be dedicated for each patch. For example, ensemble 4 would correspond to patch 4. The function then creates a random number of Layer II CNs for the patch, keeping within the patch parameters. Each of these CNs is created with 4 sites. Site 0 is the apical input site, site 1 the inhibitory input site, site 2 the basal input site, and site 3 the recurrent colateral output site. Once the Layer II Cns are created, they are connected to the LOT according to the lot connection probability. Then the recurrent colaterals are added to the LOT according to the model *die* parameter. The basal connections are created next, connecting the patch Layer II CNs together. Finally the inhibitory CN is created for the patch, creating the inhibitory input and output connections to the Layer II CNs recurrent colateral and inhibitory input sites, respectively.

Chapter 8

Future Work

As neural network research advances, it will become more and more important to be able to generate the underlying network structures quickly and easily. NDL is a step in the right direction for three reasons. The first is that it provides the fundamental building blocks needed to specify an arbitrary network. The second is that application specific libraries exist on top of the fundamental building blocks providing a level of abstraction which relieves the user from having to deal with tedious details of constructing networks component by component. The third is that it is built on an existing language, so it can be used via that language and extended easily.

Where NDL falls short is that it still requires a textual description of a network, the program. The author believes that the fundamental constructs provided by NDL will always be required, but a front end graphical interface will be necessary to relieve the neural network researcher the tedious task of programming.

One of the goals of this thesis was to investigate the use of graph-grammars as a way of generating network graphs. Unfortunately graph-grammars are hard to realize textually, so they were not included as part of NDL. Graph-grammars however do provide an elegant solution assuming they can be utilized in a graphical fashion. To incorporate the graph-grammar research into this thesis, they are included in this chapter on future work, since they would be so useful in an interactive graphical interface, which is where the future of network graph generators lie.

Graph-Grammars

Simply stated, graph-grammars are a means of specifying sets of graphs. They are to graphs as formal string theory is to strings. They are very useful in specifying families of graphs and recursive structures. Applications include pattern recognition, software specification and development, VLSI layout, incremental compilers, databases, computer animation, complexity theory, developmental biology, parallel computer architectures, and many others. Graph-Grammar theory is still a long way from the theory of formal string languages, but steady progress is being made. The mathematical mechanisms used to deal with 3-dimensional structures like graphs are intrinsically more difficult than that of 1-dimensional strings.

Recognizers vs. Generators

There are two basic models in graph-grammar theory. A graph grammar *recognizer* is an analysis program which would determine if a certain graph belonged to a given grammar. In other words it determines if a graph could be generated from the productions of a grammar. The counterpart in string theory would be the parser in the front end of a programming language compiler. The parser checks to make sure that the program can be generated by the set of grammar rules which describe the language.

A graph grammer *generator* is a synthesis program which generates graphs from a given grammar. A desired graph can be generated by specifying a sequence of productions. The synthesis of graphs from a grammar is what would be useful in generating neural network graphs.

A Typical Graph Grammar Paradigm

Much of the current work in graph-grammars involves changing an existing host graph through some transformation. This transformation is usually a production which consists of two subgraphs, a host graph, and an embedding transformation. When the production is executed, all occurances of an old subgraph in the host graph are replaced by a new subgraph. The new subgraph is inserted into the host graph by following the interconnect specification of the embedding transformation. A classical graph grammar is shown in figure 8.1. It shows how a binary tree can be derived from one simple graph production. In the top half of the figure, the production is shown. In the production, it is shown that the application of

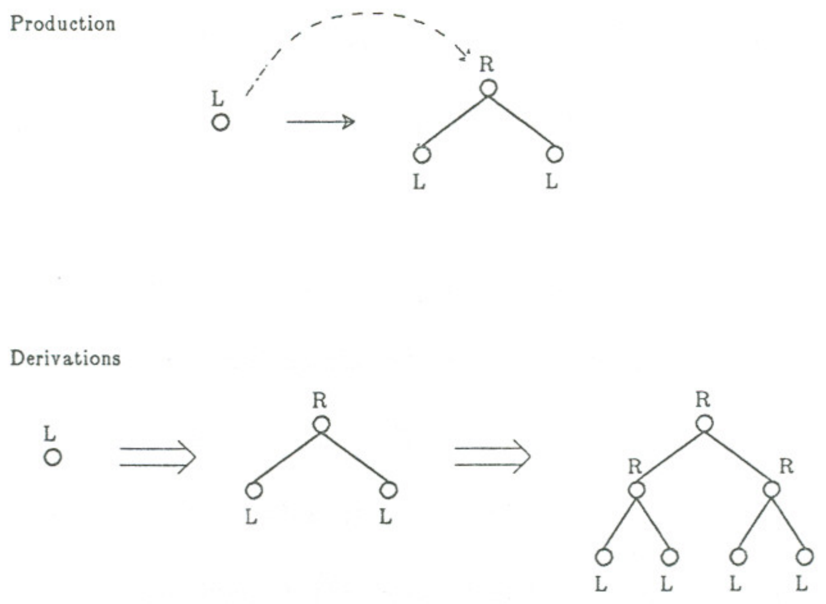

Figure 8.1 - A Binary Tree Graph-Grammar

it replaces each node labeled *L* with the sub-graph shown on the right side. In the bottom half of the figure, it is shown how applying the production two times to a single node graph results in the binary tree on the right. Each application of the production is called a derivation step.

A Graph Editor Using Graph Grammars

In a paper by Cuny and Bailey [BaC87], they describe a *graph editor* which allows the user to specify communication structures graphically. In that environment, the graph operations are based on aggregate rewriting graph grammars. Using this type of graph grammar allows them to perform, in parallel, graph rewrite operations on aggregates of nodes whose labels are logically related. Biases are supported which grew from the identification of common characteristics of statically programmed communication structures.

There are four elements to an aggregate rewriting production. The *mother* graph is a sub-graph in the current host graph. The *daughter* graph is the replacement sub-graph. The daughter sub-graph replaces all instances of the mother sub-graphs. The *inheritance function* maps the edges, which were incident on the mother graph, to the daughter graph. A final function defines the domain of the inheritance function such that for each replacement performed, there is a surjective mapping of the edges. The production is applied to a host graph and yields an image graph. The production is performed in two steps. The first is to remove all instances of the mother graph from the host graph leaving the *rest* graph. The second step is to insert the daughter graph into the vacancies left from the first step by using the inheritance function.

Graph Grammars and NDL

A graph in NDL consists of ensembles, CNs, sites, and links. A graph in a typical graph grammar paradigm consists of nodes and edges. In a graph grammar, there is no distinction between the nodes in the existing graph. In a NDL graph, a distinction needs to be made between the different types of nodes. Take for an example a replacement operation in which it was desirable to replace all occurances of a particular instance of a sub-graph with another sub_graph. For a graph grammar, it would simply search the entire graph for all instances of the mother graph. For a NDL operation, the type of nodes would also play an important aspect in the searching algorithm. You would not want to replace a CN node in the graph with a site node. These are just a few of the problems which will have to be overcome to make a viable graph grammar neural network generator. It is probably more conceivable that a graphical network generator would incorporate graph grammer functionality, but would be combined with other techniques.

Chapter 9

Summary and Conclusions

NDL's attributes provide a unique and powerful mechanism. Besides the ability to create CNs with different characteristics, the attributes provide a powerful connection method. Coupled with NDL's concept of the aggregate, complex connection schemes can be accomplished.

There were four goals set forth at the beginning of this thesis. The first goal, to generate BIF files, has been accomplished. By using the BIF attribute base and the BIFGEN output filter, BIF files are produced.

The second goal, to provide extensibility, is provided. The framework for creating NDL application libraries was presented. Two application libraries, the layer model and the Pyriform Cortex Model were discussed in detail.

The fact that NDL generates BIF files is because of the BIF Attribute Base constructed for it and the manner in which the network is output. Other formats can be constructed by altering the Attribute Base and the manner in which the network is output. These facts support the third goal, which was to provide a tool which created not just BIF files, but a general output which could be used by other simulation environments.

The final goal was to investigate Graph Grammars as a means of generating neural network graphs. Although this work was not integrated into the NDL environment, due to time constraints, directions for the future were made. The work by Cuny and Bailey has made specifying families of regularly structured graphs a straight forward operation using graph grammar productions. A very attractive feature of their work is that very large regular networks can be specified as easily as a very small network of the same structure. The graphs generated by these graph-grammars create regular structures such a binary trees, cubes, and multistage permuting networks. Unfortunately, the structure of these regular networks are not those typically used by neural networks.

An important contribution of NDL is that it is decoupled from the simulation control philosophy. In other neural network simulation environments, the construction of the network is intimately tied to the way the network is executed. In NDL, there are no explicit or implicit ties to the control mechanisms of the network simulator. It does not mean, however, that NDL cannot output information that could be utilized by the simulator. Aggregates can be output which could group CNs into sets which could be used in sequencing the simulator. In addition, NDL can output information which could aid in the mapping of the neural network to a parallel computer simulation environment. Again, this could be done using aggregates.

Another advantage of having NDL decoupled from the control part of the simulation is the ability to have overlapping sets of network components. The fact that a CN could belong to different sets could be of some value in a control philosophy. Suppose a layer of CNs were to be updated at a particular time in a simulation sequence. Then at another time in the sequence, only a subset of the CNs in the layer were to be updated. Only with overlapping sets could this be achieved in a clean manner.

Creating the layer and pyriform model application specific libraries turned out to be a fairly simple task. The pyriform model is one of the more complex networks being studied today, and as can be seen in appendix C, not that much code was required. It should be noted that the data abstractions provided by NDL were more useful than the control abstractions.

The experience using C++ as the implementation language was the author's first exposure to an object oriented language. Because of this inexperience, NDL's implementation was not as elegant as it could have been. Data abstraction was used extensively, but type hierarchy was not utilized as it should have been.

In summary, the main contribution that NDL has given in neural network construction is the ability to abstract away complicated, nested control constructs in generating the links in a network. These abstractions may seem a bit trivial when constructing simple layered networks, but it is hoped that they will be of some value in the future when complex neural topologies are studied and simulated.

APPENDIX A

BIF File Format

BIF File Format

A BIF file consists of two parts: the group block and the CN block. The group block defines types of CN's used in the network. The CN block decribes the network in terms of CN's and connectivity. Delimiters are used to describe the bounds on the group block, the CN block, and each CN definition within the CN block. Reserved fields in bit vectors are used to delimit the sites and links within the each CN definition. The sitevec for the sites and the lnkvec for the links. Refer to [Bah88] for complete details on this.

The BIF file format is best described by a grammar. The grammar consists of three types of elements, the *non-terminal*, the field-terminal, and the **reserved word**. The final BIF file consists of the field-terminals and reserved words.

BIF GRAMMAR

bif-file:
    gblock cblock
gblock:
    sgbk *grouplist* **egbk**
grouplist:
    grouplist group
group:
    groupfields **egrp**
groupfields:
    index name initpot initstate
cblock:
    scbk *cnlist* **ecbk**
cnlist:
    cnlist cn
cn:
    cnfields sites **endc**
cnfields:
    group index procid delay bitvec *cn-options*
cn-options:
    history restpot pot state output error sd
sites:
    sites site
site:
    sitefields links
sitefields:
    value sitevec
links:
    links link
link:
    lnkvec history cn site link weight

APPENDIX B

BIF Environment

BIF Environment

The BIF environment consists of two function calls and six global variables. The function call

create_bif_attributes()

creates the BIF attribute base. It consists of four groups.

The *cngroup* describes the types of CN and corresponds to the group block in the BIF file. There are three types of CNs specified for the group block. Each is declared as a list under the *cngroup* group. They are the input, hidden, and output lists. These are the three types of CNs supported currently by BIF.

The cn group describes in detail each type of CN. As in the *cngroup* group, it is divided into three lists. Again, they are the input, hidden, and output lists.

The site group describes the two types of sites available using the BIF environment. The input site and the output site are each described in their corresponding list under the *site* group.

The *link* group describes the one type of link available in the BIF environment.

Once the attribute base is constructed, the six global variables are assigned to the attribute base. There are three CN attribute access lists, two site attribute access lists, and one link attribute access list.

The function

```
print_bif(int ens)
```

prints out the group block followed by the definition of the network. Each component of the network will contain the attributes as defined by the BIF attribute base.

```
// declare the global attribute access lists

ATTRREF *ICN;
ATTRREF *HCN;
ATTRREF *OCN;
ATTRREF *ISITE;
ATTRREF *OSITE;
ATTRREF *ULINK;


// print the NDL version of BIF

void print_bif(int ens)
{
  print_cn_groups();
  print_ensemble(ens);
}


// print the cn group block

void print_cn_groups()
{
  printf("sgbk\n");
  print_list("cngroup","input");
  printf("egrp\n");
  print_list("cngroup","hidden");
  printf("egrp\n");
  print_list("cngroup","output");
  printf("egrp\n");
  printf("egbk\n");
}


// create the BIF Attribute Base

void create_bif_attributes()
{
        // create the cn groups

  create_attr_group("cngroup");
   create_attr_list("cngroup","input");
    create_attr_record("cngroup","input","index",0);
```

```
 create_attr_record("cngroup","input","name","input");
 create_attr_record("cngroup","input","initpot",0);
 create_attr_record("cngroup","input","initstate",0);
create_attr_list("cngroup","hidden");
 create_attr_record("cngroup","hidden","index",1);
 create_attr_record("cngroup","hidden","name","hidden");
 create_attr_record("cngroup","hidden","initpot",0);
 create_attr_record("cngroup","hidden","initstate",0);
create_attr_list("cngroup","output");
 create_attr_record("cngroup","output","index",2);
 create_attr_record("cngroup","output","name","output");
 create_attr_record("cngroup","output","initpot",0);
 create_attr_record("cngroup","output","initstate",0);

     // create the CN attribute lists

create_attr_group("cn");
 create_attr_list("cn","input");
  create_attr_record("cn","input","group",0);
  create_attr_record("cn","input","procid",-1);
  create_attr_record("cn","input","delay",0);
  create_attr_record("cn","input","bitvec",7);
  create_attr_record("cn","input","history",0);
  create_attr_record("cn","input","restpot",0);
  create_attr_record("cn","input","pot",0);
  create_attr_record("cn","input","state",0);
  create_attr_record("cn","input","output",0);
  create_attr_record("cn","input","error",0);
  create_attr_record("cn","input","sd",0);
 create_attr_list("cn","hidden");
  create_attr_record("cn","hidden","group",1);
  create_attr_record("cn","hidden","procid",-1);
  create_attr_record("cn","hidden","delay",0);
  create_attr_record("cn","hidden","bitvec",7);
  create_attr_record("cn","hidden","history",0);
  create_attr_record("cn","hidden","restpot",0);
  create_attr_record("cn","hidden","pot",0);
  create_attr_record("cn","hidden","state",0);
  create_attr_record("cn","hidden","output",0);
  create_attr_record("cn","hidden","error",0);
  create_attr_record("cn","hidden","sd",0);
 create_attr_list("cn","output");
  create_attr_record("cn","output","group",2);
```

```
    create_attr_record("cn","output","procid",-1);
    create_attr_record("cn","output","delay",0);
    create_attr_record("cn","output","bitvec",7);
    create_attr_record("cn","output","history",0);
    create_attr_record("cn","output","restpot",0);
    create_attr_record("cn","output","pot",0);
    create_attr_record("cn","output","state",0);
    create_attr_record("cn","output","output",0);
    create_attr_record("cn","output","error",0);
    create_attr_record("cn","output","sd",0);

        // create the site attribute lists

  create_attr_group("site");
   create_attr_list("site","input");
    create_attr_record("site","input","value",0);
    create_attr_record("site","input","sitevec",128);
   create_attr_list("site","output");
    create_attr_record("site","output","value",0);
    create_attr_record("site","output","sitevec",0);

        // create the link attribute lists

  create_attr_group("link");
   create_attr_list("link","all");
    create_attr_record("link","all","lnkvec",128);
    create_attr_record("link","all","history",0);

        // create the CN attribute access lists

  ICN = create_attr_ref("cn","input");
  HCN = create_attr_ref("cn","hidden");
  OCN = create_attr_ref("cn","output");

        // create the site attribute access lists

  ISITE = create_attr_ref("site","input");
  OSITE = create_attr_ref("site","output");

        // create the link attibute acces list

  ULINK = create_attr_ref("link");
}
```

APPENDIX C

Pyriform Environment

Pyriform Environment

```
// ************************************************************
//
// PYRIFORM - a base class for a pyriform network
//

class PYRIFORM {
 int lotsize;          // LOT size
 int die;              // LOT diminish/replenish rate
 int patchlo;          // minimum number of layer II / patch
 int patchhi;          // maximum number of layer II / patch
 int lotconprob;       // probability of apical connection
 int basalprob;        // probability of basal connection
 int ens;              // ensemble index for LOT input neurons
 LOTCON lot;               // connection links for LOT
 PATCH patches;  // linked list of patches

public:
 PYRIFORM() {              // constructor
   lotsize = 0;
   die = 0;
   patchlo = 0;
   patchhi = 0;
   lotconlo = 0;
   lotconhi = 0;
   prob = 0;
   ens = -1;
   lot = NULL;
   patches = NULL;
 }
};
```

```
// ************************************************************
//
// PATCH - a class for containing the specific information concerning
//         a patch

class PATCH {
 int ens;             // ensemble index
 PATCH *next;

public:
 PATCH() {
   ens = 0;
   next = NULL;
 }
};



// ************************************************************
//
// LOTCON - a class for containing the LOT link ends

class LOTCON {
 int ens[MAXLOTSIZE];
 int cn[MAXLOTSIZE];
 int site[MAXLOTSIZE];

public:
 LOTCON() {
   ens = 0;
   cn = 0;
   site = 0;
 }
};
```

```
PYRIFORM pyriform;

// ***********************************************************

pyriform_environment()
{
 pyriform = new PYRIFORM();
}


// ***********************************************************
set_lot(int size,int die)
{
        // set the lot parameters

  pyriform->lotsize = size;
  pyriform->die = die;

        // create the LOT input neurons
        //  dedicated ensemble
        //    site 0 - input from olfactory bulb
        //    site 1 - output to LOT

  pyriform->ens = create_ensemble();
  create_cn(pyriform->ens,size);
  create_site(pyriform->ens,NULL,2);

        // create the LOT connection sites
        //  originally they will be the LOT neuron output sites,
        //  but later will become recurrent colateral output
        //  sites from the Layer II neurons

  lot = new LOTCON(size);
  for(int i = 0; i<size; i++)
    set_lot_con(i,pyriform->ens,i,1);
}
```

```
// ***********************************************************

set_patch(int plo, int phi, int lprob, int bprob)
{
 pyriform->patchlo = plo;
 pyriform->patchhi = phi;
 pyriform->lotconprob = lprob;
 pyriform->basalprob = bprob;
}

// ***********************************************************

create_patch(int n)
{
 for(i=n; i=0; i++)
   create_patch();
}
```

```
// ****************************************************************

create_patch()
{
      // create the patch class

  PATCH patch = new PATCH();
  patch->ens = create_ensemble();

      // create the Layer II Pyriform Neurons
      //     site 0 - apical input sites
      //     site 1 - inhibitory input site
      //     site 2 - basal input site
      //     site 3 - recurrent colateral output site

  int n = get_random(pyriform->patchlo,pyriform->patchhi);
  create_cn(patch->ens,n);
  create_site(patch->ens,NULL,4);
  AGGR *layerII = create_range_aggr(0,n-1);

      // create the LOT connections

  int ens,cn,site;
  // cycle thru patch CNs
  for(int i=0; i<n; i++) {
      // cycle thru LOT CNs
    for(int j=0; j<pyriform->lotsize; j++) {
      if(true_or_false(pyriform->lotconprob)) {
        get_lot_con(j,&ens,&cn,&site);
         create_link(ens,cn,site,patch->ens,i,0);
      }
    }
  }
      // add recurrent colaterals to LOT

  cn = 0;
  for(j=0; j<pyriform->lotsize; j++) {
    if(true_or_false(pyriform->lotconprob)) {
      set_con_lot(j,patch->ens,cn++,3);
    }
    if(cn >= n) break;
  }
```

```
        // create the Basal connections

    for(int i=0; i<n; i++) {
      for(int j=i+1; j<n; j++) {
        if(true_or_false(pyriform->basalprob))
          create_link(patch->ens,i,3,patch->ens,j,2);
      }
    }

        // create the Inhibitory interneuron

    int icn = create_cn(pyriform->ens);

        // create the inhibitory connections

    create_site(patch->ens,icn,2);
    create_link(patch->ens,icn,0,patch->ens,layerII,3);
    cerate_link(patch->ens,icn,1,patch->ens,layerII,1);

        // add patch to end of linked list

    if(pyriform->patches = NULL) {
      pyriform->patches = patch;
    }
    else {
      PATCH *p = pyriform->patches;
      while(p->next) p = p->next;
      p->next = patch;
    }
  }
```

```
// ************************************************************

pyriform_output()
{
 int i = 0;

      // print the lot input neuron connections

 printf("LOT INPUT NEURONS\n");
 print_ensemble(pyriform->ens);

      // print each patch neuron connections

 PATCH *p = pyriform->patches;
 while(p) {
  printf("PATCH #%d NEURONS\n",i++);
  print_ensemble(p->ens)
  p = p->next;
 }

      // print the final output lot sites

 printf("FINAL LOT OUTPUT\n");
 int ens,cn,site;
 for(i=0; i<pyriform->lotsize; i++) {
  get_lot_con(&ens,&cn,&site);
  printf("%5d  %5d  %5d0,ens,cn,site);
 }
 printf("END\n");
}
```

REFERENCES

[Bah88] Bahr, C., "ANNE : Another Neural Network Emulator," *Master Thesis*, Beaverton, Oregon, 1988.

[BaC87] Bailey, D. A. and Cuny, J. E., "Graph Grammar Based Specification of Interconnection Structures for Massively Parallel Computation," COINS Technical Report 87-23, Amherst, Massachusetts, June 1987. University of Massachusetts, Computer and Information Science.

[Bai88] Bailey, J., "A VLSI Interconnect Structure for Neural Networks," *Ph.D. Dissertation*, Beaverton, Oregon, 1988.

[CaG88] Campbell, B. and Goodman, J. M., "HAM: A General Purpose Hypertext Abstract Machine," *Communications of the ACM*, July, 1988.

[Dij76] Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, 1976.

[Fan86] Fanty, M., A Connectionist Simulator for the BBN Butterfly Multiprocessor, vol. TR164, University of Rochester, January, 1986.

[Gar88] Garg, P. K., "Abstraction Mechanisms in Hypertext," *Communications of the ACM*, July, 1988.

[GhH88] Ghosh, J. and Hwang, K., "Critical Issues in Mapping Neural Networks on Message-Passing Multicomputers," *IEEE - Transactions on Computers*, Febuary, 1988.

[GrL] Granger, R. and Lynch, G., "Derivation of encoding characteristics of layer II cerebral cortex," Technical Report, Irvine, CA. Center for the Neurobiology of Learning and Memory.

[Isn82] Isner, J. F., "A Fortran Programming Methodology Based on Data Abstraction," *Communications of the ACM*, vol. 25(Oct, 1982), .

[Jag89] Jagla, K., "HAS : Hierarchical Architecural Simulator," *Master Thesis*, Beaverton, Oregon, June 1989.

[Kra] Kraft, T., ANSpec Language Definition, Science Applications International Corporation.

[May88] May, N., "Fault Simulation of a Wafer-Scale Neural Network," *Master Thesis*, Beaverton, Oregon, February 1988.

[Pla87] Plate, T., "A design for the simulation of connectionist models on coarse grained parallel computers," MCCS-87-106, Las Cruces, NM, November 22, 1987. New Mexico State University.

[RuM86] Rumelhart, D. E. and McClelland, J. L., *Parallel Distributed Processing*, vol. Volume 1 - Foundations, Massachusetts Institute

Of Technology, 1986.

[Wat88] Watrous, R. L., "GRADSIM: A Connectionist Simulator using Gradient Optimization Techniques," MS-CIS- -8-16, Philadelphia, PA, March, 1988. University of Pennsylvania, Department of Computer Science.

[Win79] Winograd, T., "Beyond Programming Languages," *Communications of the ACM*, vol. 22(July, 1979), .

Biographical Note

The author was born 31 August 1955, in Eau Claire, Wisconsin. In 1956 he moved to Portland, Oregon where he attended public grammar schools and graduated from Reynolds High School in 1973.

In September 1973 the author attended Southern Oregon State College for one year. In September 1974 the author attended Oregon State University studying Nuclear Engineering for one year. In August 1975 the author entered the Navy studying Nuclear Engineering and electronics. In March 1977 the author again attended Oregon State University and in May 1980 received a B.S. in Electrical and Computer Engineering.

The author designed industrial control systems, specializing in Machine Vision and Factory Automation for the next six years. In the fall of 1986, the author entered the Masters Degree Program at the Oregon Graduate Institute. In the early summer of 1990, the author graduated with a M.S. in Computer Science and Engineering.

The author has been married two years to Deanna Rockwell and they have one child, Christopher, age 18 months, and another on the way in late July 1990.

The author is now working for the neural network start-up company Adaptive Solutions in Beaverton, Oregon.