On Variable Binding in Connectionist Networks

Ron Sun

Department of Computer Science

The University of Alabama

Tuscaloosa, AL 35487

RUNNING HEAD: variable binding

KEY WORDS: connectionism, variable binding, logic, rule, network.

To appear in: *Connection Science*, 1992

ABSTRACT

This paper deals with the problem of variable binding in connectionist networks. Specifically, a more thorough solution to the variable binding problem based on the *Discrete Neuron* formalism is proposed and a number of issues arising in the solution are examined in relation to logic: consistency checking, binding generation, unification, and functions. We analyze what is needed in order to resolve these issues, and based on this analysis, a procedure is developed for systematically setting up connectionist networks for variable binding based on logic rules. This solution compares favorably to similar solutions in simplicity and completeness.

ACKNOWLEDGEMENTS. I wish to thank Dave Waltz, James Pustejovsky, and Tim Hickey for many discussions that helped me to elucidate ideas contained in this paper. I am also grateful to the three anonymous reviewers for their insightful criticisms and useful suggestions.

1 Introduction

When discussing connectionist models in relation to reasoning, one often omits issues related to variables in order to simplify the discussion, just as one often omits predicates and arguments, looking only at propositional forms, when discussing logics. However, variables are important, or even essential, for connectionist systems, as argued by many leading researchers (cf. Feldman & Ballard 1982, Pinker & Prince 1988, and Fodor & Pylyshyn 1988). As in Feldman & Ballard (1982), the following example demonstrates the necessity of having variables: supposing we have a red circle and a black square, we have to have a way of associating red with circle and black with square (such as *COLOR(obj1, red)*, *SHAPE(obj1, circle)*, etc.); this binding has to occur and cease dynamically when an agent or a system is reasoning with (visual) input data. Variable binding is essential in creating dynamic structures in connectionist networks.

In the same vein, based on linguistic data, Pinker & Prince (1988) point out that rules, modular structures and variables are essential for the human language faculty, and are thus important to be taken into account in connectionist models of linguistic processes and the like. The lack of variable binding was considered by them to be a major shortcoming of early connectionist models.

Variables and variable binding mechanisms are also needed to achieve greater computational power in connectionist systems. Early research in connectionist symbolic processing and high level connectionist models avoids the variable binding problem by reasoning with one object at a time or by using multiple instantiations, such as in Kosko (1988), Blelloch (1986), Gallant (1987), and Shastri (1988). This approach limits the processing power of the respective systems, and increases their computational complexities; that is, to deal with multiple objects, many instances of a rule have to be deployed: instead of

$$Inflation(x) \longrightarrow HighInterestRate(x)$$

we have to have

$$Inflation_{US} \longrightarrow HighInterestRate_{US}$$

$$Inflation_{Japan} \longrightarrow HighInterestRate_{Japan}$$

and so on (cf. also Hendler 1988).

The difficulty of variable binding stems from the simplicity of connectionist models, in which each node performs very simple processing (usually just weighted-sums or similar numerical calculations). And structurally these models are uniform (usually using simple processing elements of the same type with regular connection patterns). Therefore, it is extremely difficult to impose any particular dynamic symbolic structures onto them because of the uniformity (in comparison, to store or modify a symbolic structure in conventinal computers, one has to go through a sequence of operations with the help of central processing units and other components, which are not readily available to connectionist models). The problem of introducing variables into connectionist models amounts to the question of how to incorporate particular structures to networks without losing too much of that characteristic uniformity and how to enhance the processing power of each processing element to a degree sufficient for symbolic processing while maintaining its simplicity. Different existing solutions (Touretzky & Hinton 1985, Barnden 1988, Sun 1989, Lange & Dyer 1989 and Ajjanagadde & Shastri 1989) can be viewed as different ways of achieving the balance between these conflicting demands (see Section 4 for some analyses of existing work).

Our aim here is, first of all, to devise a network structure as uniform as possible and to keep node functions well-defined and relatively simple but at the same time to be able to create dynamic symbolic structures. Secondly, we are concerned with developing a generic, domain-independent framework, not tied to (and/or limited to) particular applications, so generality and descriptive precision are our main considerations (for application-oriented treatments, see e.g. Lange & Dyer 1989 and Ajjanagadde & Shastri 1989). Finally, since besides the basic capability of variable binding, there are in fact many more difficult problems to be solved in order to enable a connectionist reasoning system to incorporate logic rules and to possess reasoning power comparable to a sequential symbolic system, and most of these issues are not adequately addressed by most solutions to the variable binding problem, we will provide some treatment of these more intricate issues in this paper.

In the rest of this paper, we first present in complete detail of a more thorough and more rigorous solution to the variable binding problem (which was first presented in Sun 1989), within a specific architecture CONSYDERR, based on a formal model of node activations (Sun et al 1989). We then develop mechanisms for dealing with some difficult and important issues in variable binding. We introduce a set of procedures for compiling a rule set into connectionist networks, and we examine its correctness. To understand better the relationship between logic and connectionist models, we look into a logic formalism that directly maps onto the weighted-sum connectionist models and thus links connectionist models to logic.

2 Variable Binding with CONSYDERR

2.1 CONSYDERR in a Nutshell

CONSYDERR is a connectionist architecture integrating rule-based and similarity-based reasoning (see Sun 1991). It is a two-level, dual-representation architecture for commonsense reasoning, with one level, called CL, using localist representation and the other level, called CD, using distributed (feature-based) representation. The localist level (CL) is mainly for representing rules and concepts. Rules are represented via links between nodes. The distributed level (CD) is mainly for carrying out similarity-based reasoning, to supplement and enhance rule-based reasoning mechanisms. Each node in CL is connected to all the relevant feature nodes in CD; once a CL node is activated, the related CD nodes will be activated subsequently from inter-level connection, and vice versa. Links in CL are replicated diffusely in CD by multiple links between two sets of nodes representing a condition and a conclusion of a rule respectively. Similarity matching is actually accomplished through the interaction between the two levels, by a top-down/settling/bottom-up cycle. See Figure 1, where the top portion is CL and the bottom portion is CD. This model has been used to model elementary geographic reasoning (Sun 1990, 1991 and 1991a)

Here we will only discuss the CL level, because it is the level in which we will implement rules and variable binding. In its propositional form (without variables and variable binding), CL is a feed-forward weighted-sum connectionist model, in which a node represents a certain concept and links represent rules. If there is a rule $a\ b\ c \longrightarrow d$, then there is a link from each of the conditions, a b and c, to d, and the three links together represent the rule. Each node computes the weighted-sum of its inputs from incoming links, to add up its evidential support (confidence values). Once activated, a node will pass on its own activation (i.e. confidence value) to nodes downstream, that is, nodes which this node is connected to as a condition in a certain rule (and consequently these nodes are conclusions of the rules involved; see Sun 1991). Justifications for this interpretation of weighted-sum connectionist

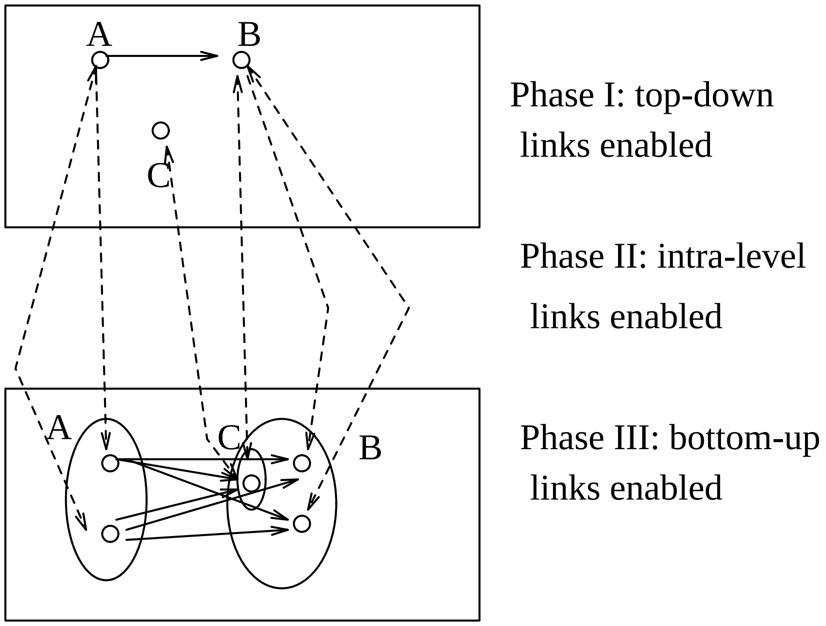

Figure 1: The CONSYDERR Architecture

A two-level architecture: CL is the top level and CD is the bottom level

models will be given in 3.4. When variables and variable binding are added, CL becomes much more complex, as will be explained in the following sections.

Note that although this work is the outgrowth of studies of connectionist models for commonsense reasoning (Sun 1991, 1992), this paper per se will not discuss issues in commonsense reasoning, but only variable binding as an enabling mechanism.

2.2 Representing Variables in CONSYDERR

First of all is the question of how to represent individual variables. As suggested in Sun (1989) (also cf. Lange & Dyer 1989), each variable in a predicate can be handled by one separate node. A variable can get bound by an application of a relevant rule that has in its RHS (right-hand side) the predicate that contains the variable. Basically, a binding is a numerical value, which represents a particular constant (a domain object), being passed along a link (which represents the rule in question) to the node representing the variable, from a node representing a variable having the same name in the LHS (left-hand side) of the rule applied. For example, suppose we have a rule: "if an area grows something, it is an agricultural area." It can be expressed as follows:

$$grow(x, y) \longrightarrow agr(x)$$

Suppose the input is $grow(a, b)$ (meaning "*area a* grows *b*"), where a and b are constants, say, $a = Florida$ and $b = orange$. The binding a is passed to the node representing x, and b to the one representing y in $grow(x, y)$. Then, by applying the rule, a is further passed from the node representing x in $grow(x, y)$ to the node representing x in $agr(x)$. (In order to do this, however, we need to extend the idea of weighted-sum node activation functions to allow the passing and selection of the numerical values or symbols representing bindings, as will be detailed in 2.3.)

So with this representation, for each predicate we need an *assembly*, or a collection of nodes interconnected in some way, one of which is for computing and storing (confidence) values of a predicate (we will call it the C node), and the rest are for variables (we will call them X nodes). Supposing we have k variables, an assembly then contains k+1 nodes, each of which (except one C node) is for storing and passing along bindings of the variable which it represents. Among these nodes, the C node of an assembly basically receives inputs from C nodes in other assemblies that are conditions (LHS) of a rule that use this assembly for its RHS, and computes the value that represents the confidence we have on the predicate which the assembly represents based on the values of the conditions. Each variable (X) node receives inputs from variable (X) nodes in the same other assemblies as its corresponding C node, picks up its own binding, and passes it on. See Figure 2.

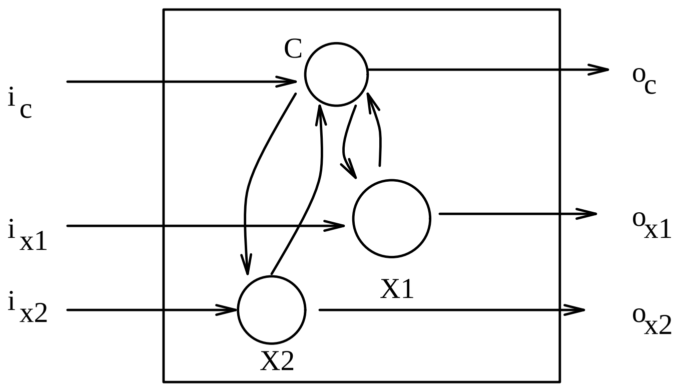

Figure 2: An Assembly

[1] For the above example: $grow(x, y) \longrightarrow agr(x)$, we will have two assemblies: one for grow(x,y) and the other for agr(x). The first assembly has three nodes: C, X, and Y. The second assembly has two nodes: C and X. The C node in the first assembly is linked to the C node in the second assembly. The X node in the first assembly is linked also to the X node in the second assembly. See Figure 3. Variable binding is then taken care of by passing values along links (Sun 1989 and Lange & Dyer 1989). Details will follow.

2.3 A Formal Model of Node Activations

One question that needs to be answered is exactly what kind of processing elements we need and what functions we must have in a processing element in order to deal with variables. We thus need some formal treatments that can qualify the problem more clearly. We will use a very general formalism for connectionist models, the

[1]Each of the variable nodes in an assembly should also communicate with the C node to send its bindings, so that the C node can check consistency and other matters, and should receive "instructions" from the C node, which are to tell a variable node whether or not to pass on a binding and which binding to take. These issues will be addressed later in the paper.

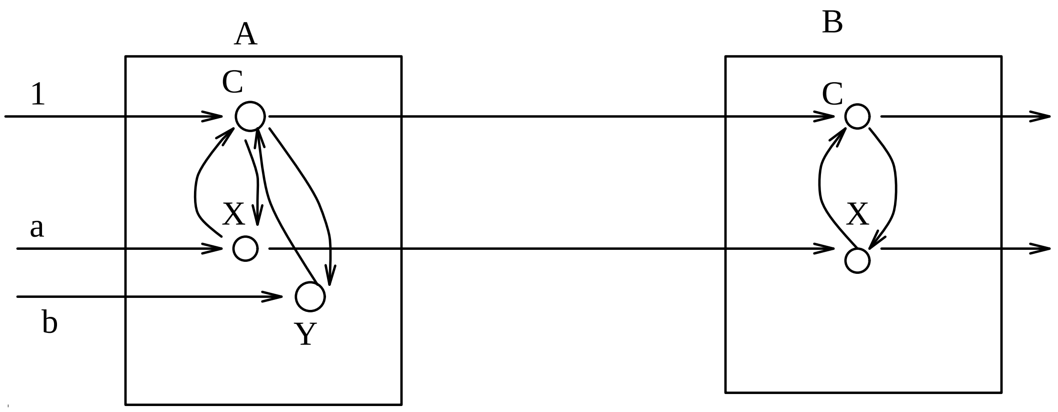

Figure 3: The Network for Representing the Rule: if grow(X,Y) then agr(X)

Discrete Neuronal Model, or the DN model. [2] It can deal with variable binding, rule matching, and confidence value propagation all at the same time, and it can be mapped back into conventional models, such as weighted-sum models. Thus the model forms a unifying framework for developing a complex reasoning system incorporating variables.

An automaton-theoretic description of a node in a DN model is:

N=$< S, A, B, I, O, U, V >$

S= the set of all the possible states of a node,

O= the output vector,

A= the set of all the output symbols,

I= the input-line vector,

B= the set of all input symbols,

U= state transition function,

V= action function (output function).

[2]This formalism is first proposed in (Sun et al 1989, Sun 1989) and later further developed in (Sun 1990, 1991c).

We think of a DN node as having a set of input lines and a set of output lines:

$$I = (I_1, I_2, ..., I_n)$$

$$O = (O_1, O_2, ..., O_m)$$

where each element of I, I_i, is in B, and each element of O, O_i, is in A. Therefore, the value of I is in $\Pi_n B$, and the value of O is in $\Pi_m A$. And the state transition and action functions are [3]

$$U : S \times \Pi_n B \longrightarrow S$$

$$V : S \times \Pi_n B \longrightarrow \Pi_m A$$

A network of DN nodes is a graph with (multiple) directed edges between pairs of nodes. An added constraint is if there is a directed link from the node i to the node j, then $A_i \subset B_j$.

Although the formalism is general and complex (it is aimed for generality beyond this work; cf. Sun 1989, 1990 and 1991c), the *actual* mechanism for variable binding is simple, and it can be easily mapped into conventional models, as will be seen below. [4] Because of the generality, the DN model can emulate conventional connectionist models (see Appendix A for details). Conversely, DN nodes can also be implemented with conventional models (see also Appendix A). These two facts together further ensure the compatibility of this formalism with more conventional models. This formalism should thus be viewed as a descriptive tool

[3]There is a timing issue here: for some DN nodes, the functions can be carried out earlier, while for others later. In other words, each cycle of state transitions and actions is divided into different phases, each of which is for different functions and/or for different nodes. The above equations should be refined to take into consideration of this issue; this has not been done here, for the sake of readability. This issue will be explicated later in relation to node assemblies.

[4]By no means are we claiming that this model is more biologically plausible, but only that it can be a useful formalism.

and/or a notional convention for presenting and explaning complex (conventional) connectionist network structures as will be developed in this paper.

We will describe now how the DN formalism can be used to construct the CL level of CONSYDERR, given some first order logic rules with variables. We can map from the first-order rules to a CL network (with variable binding) as follows: a predicate is represented by an assembly of DN nodes, and rules are encoded in a network by the connections which an assembly has to the other assemblies. For example, Figure 4 shows how assemblies representing predicates are wired into a network. So CL is made up of assemblies which in turn are made up of DN nodes representing predicates and arguments, that is,

$$CL =_{\text{def}} (AS, L)$$

where

$$AS = \{(C, X_1, X_2, ..., X_k) | \text{ k is an arbitrary integer; C and X's are}$$

$$\text{a set of DN nodes as specified below}\}$$

$$L \subset \{(AS_1, AS_2, i, j) \mid AS_1, AS_2 \in AS, \;\; i, j \in [0, l] \;\; \text{are labels}$$

$$\text{denoting nodes within an assembly}\}$$

The connections among nodes are specified in L by labels which indicate a particular node in one assembly is connected to another particular node in another assembly. In each assembly, there are k+1 nodes of the DN type: C, $X_1, X_2, ..., X_k$. The C node contains the (confidence) value of the predicate (representing the confidence one has on the concept) and is connected to other nodes (mainly C nodes) in other assemblies representing different predicates that are related to it by rules. The X nodes take care of variable bindings (for arguments to the predicate) for a total of k variables. There may also be connections from each X to C and from C

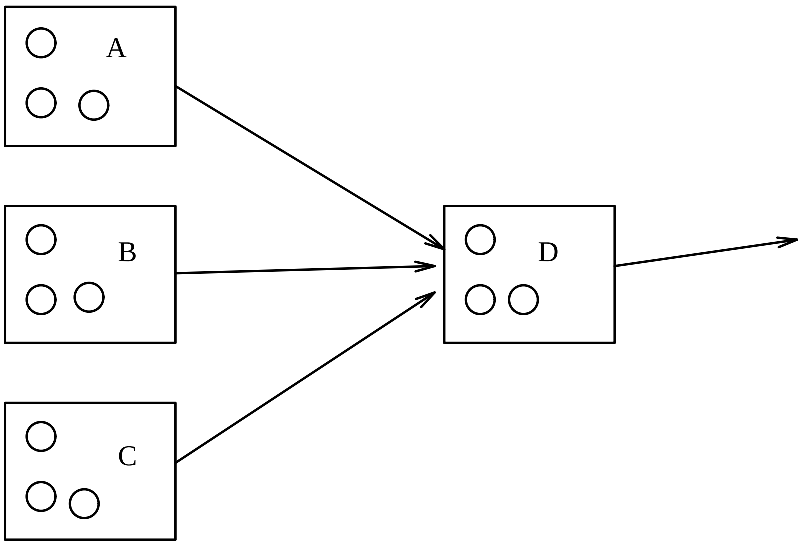

Figure 4: The Wiring of a Network for Representing the Rule: if A(X) B(Y) C(Z), then D(W)
(where X, Y, Z and W are vectors of arguments)

to each X. The set of input/output symbols (A and B) in the variable nodes are values (numerical or symbolic) representing all possible bindings used in a task, the range of which is fixed a priori but the meaning can be dynamically assigned (cf. Ajjanagadde & Shastri 1990). l denotes the maximum number of nodes in an assembly. Each node in an assembly, C or X's, has a specific action function (note that a state transition function is not needed for this application), as specified below:

For C:

$$O_C^1 = V_C^1(I_1, I_2, ...I_l, O_{X_1}^2, ...O_{X_k}^2) \in A_C$$

$$O_C^2 = V_C^2(I_1, I_2, ...I_l, O_{X_1}^2, ...O_{X_k}^2) \in A_C$$

NOTE. O_C^1 and O_C^2 denote the values of the corresponding output lines. Action function V_C^1 here is a weighted-sum function (weights will be discussed later). V_C^2 produces a signal to the X nodes (see examples later). $I_i's$ are outputs of other nodes (mainly C nodes but possibly X nodes) in other assemblies impinging on this node (here I_i denotes the value of the input line, not the line per se; the same below). The C node also receives inputs from X nodes in the same assembly, useful in certain situations which will be explicated later. l is the total number of inter-assembly inputs, so the total number of inputs (intra-assembly and inter-assembly) is $l + k$. A_C is the set of output symbols in the C node. Superscript 1 denotes the inter-assembly output and 2 the intra-assembly output.

And for X_i: i=1,2,....k,

$$O_{X_i}^1 = V_{X_i}^1(I_1, I_2, ...I_n, O_C^2) \in A_{X_i}$$

$$O_{X_i}^2 = V_{X_i}^2(I_1, I_2, ...I_n) \in A_{X_i}$$

NOTE. $O_{X_i}^1$ and $O_{X_i}^2$ denote the values of the corresponding output lines. Action function $V_{X_i}^1$ here is a mapping specifying the output of X_i based on I_i's, inputs from X nodes in other assemblies. O_C^2 is the signal from the C node in the same assembly. $V_{X_i}^2$ produces a signal to the C node (see examples later). A_{X_i} is the set of output symbols in the node X_i.

The above formalism reveals some subtle timing considerations that need to be addressed (recall the earlier mentioning of this issue). Outputs of a node depend on not only inputs from other assemblies, but also inputs from other nodes in the same assembly, and thus it is important to make sure that there is no mutual waiting (deadlock) situation. In this case, intra-assembly communication from

X to C does not depend on input from other nodes in the same assembly, only on input from other assemblies. So this intra-assembly communication can finish first, and next the intra-assembly communication from C to X is facilitated. Then outputs to other assemblies can be computed, without any further impediments. We refine I and O into two parts respectively as follows:

$$I = I^1 \circ I^2$$

$$O = O^1 \circ O^2$$

where superscript 1 denotes inter-assembly input/output, superscript 2 denotes intra-assembly input/output, and $\circ$ denotes vector concatenation. For state transition and action functions:

$$V = V^1 \cup V^2$$

where for X nodes

$$V^1 : B^I \longrightarrow A^{O^1}$$

$$V^2 : B^{I^1} \longrightarrow A^{O^2}$$

and for C nodes

$$V^1 : B^I \longrightarrow A^{O^1}$$

$$V^2 : B^I \longrightarrow A^{O^2}$$

Note that O^2 of X is I^2 of C and O^2 of C is I^2 of X. V^2 is done during the first phase of a cycle, communicating within an assembly. V^1 is done during the second phase of a cycle, sending outputs to other assemblies downstream.

2.4 An Example

As an illustration, we will derive in the DN formalism a representation of a reasoning process:

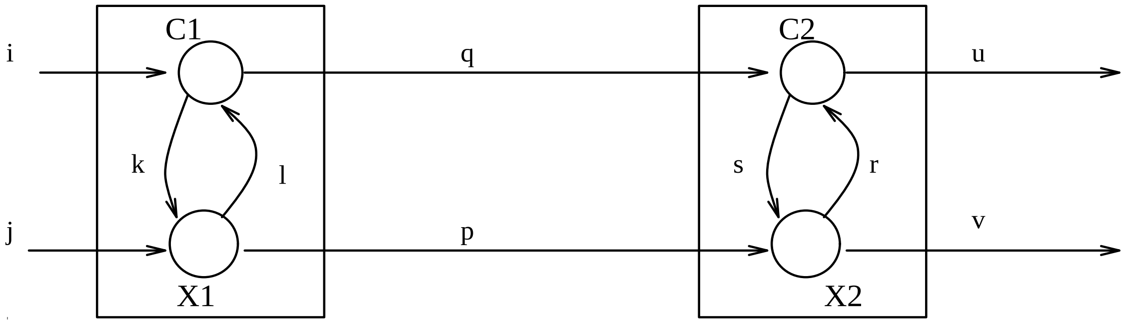

Figure 5: An Example Network — Socrates

if one is human, one is mortal.

Socrates is human,

so Socrates is mortal.

The rule to be represented is

$$human(x) \longrightarrow mortal(x)$$

First we will set up a network for representing the rule. Applying the DN formalism, we build up two assemblies (see Figure 5). The first assembly represents "X is human" and the second assembly represents "X is mortal". The input j is the string "Socrates" or its numerical representation, and the input i is the value of the proposition "Socrates is human", which is one in this case. The output v in this case should be the string "Socrates" or its numerical representation, and the output u should be the value of the proposition "Socrates is mortal", which is also one in this case. (States and state transitions in the DN formalism are not needed here, so we will not be concerned with states in the following discussion.)

Let R stand for real numbers, and Σ stand for string (or their numeric representation). For C1,

$$I = (i, l)$$

$$O = (k, q)$$

where B_i = R, $B_l = \Sigma$, A_k = R, and A_q= R.

The signals from variable nodes to the C node are to inform it whether there is a proper binding present at each variable node: when there is no needed binding, the C node should not be activated. The signals from the C node to variable nodes are to tell them if they should pass on bindings: when the C node is not activated, the variable nodes need not work on the binding. This provides double insurance for practical purposes, although it is not necessary for logical purposes.

Now the action function for C1 is:

$$V : R \times \Sigma \longrightarrow R^2$$

Expressing it in functional forms, we have

$$q = V^1_{C_1}(i, l) = i$$

$$k = V^2_{C_1}(i, l) = 1 \text{ if } i \neq 0, \quad \perp otherwise$$

where $\perp$ denotes no output, and q, k, i and j denote the values of the corresponding input or output lines (below, without further ado, the same applies for other input and output lines).

For X1,

$$I = (j, k)$$

$$O = (l, p)$$

where $B_j = \Sigma$, B_k = R, $A_l = \Sigma$, and $A_p = \Sigma$. The action function is

$$V : \Sigma \times R \longrightarrow \Sigma^2$$

Expressing it in functional forms, we have

$$p = V^1_{X_1}(j, k) = j \text{ if } k = 1, \quad \perp otherwise$$

$$l = V^2_{X_1}(j,k) = j$$

It is the same for C2 and X2. In case of C2:

$$u = V^1_{C_2}(q,r) = q * w$$

$$s = V^2_{C_2}(q,r) = 1 \text{ if } q \neq 0, \quad \perp otherwise$$

where w is the weight on the link from C1 to C2 (in this case, we have $w = 1$). And for X2:

$$v = V^1_{X_2}(p,s) = p \text{ if } s = 1, \quad \perp otherwise$$

$$r = V^2_{X_2}(p,s) = p$$

Simplifying the above notations, we come up with simple assignment statements (so that they can be easily programmable) as follows (assume prime signs represent the next time period). For C1,

$$q' = i$$

$$k' = 1 \ \text{ if } i \neq 0, \quad \perp otherwise$$

For X1,

$$l' = j$$

$$p' = j \text{ if } k' = 1, \quad \perp otherwise$$

For C2,

$$u' = q * w$$

$$s' = 1 \text{ if } q \neq 0, \quad \perp otherwise$$

For X2,

$$r' = p$$

$$v' = p \text{ if } s' = 1, \quad \perp otherwise$$

After two time periods, the result can be described as follows:

$$u'' = q' * w$$

and

$$v'' = p' \text{ if } s'' = 1, \quad \perp \ otherwise$$

With the above equations regarding p', q' and s'', by simple substitution we have,

$$v'' = j \text{ if } i \neq 0, \quad \perp \ otherwise$$

$$u'' = i * w$$

where u'' is the output from the C node and v'' is the output from the variable node, all in the "mortal" assembly. In this case, the input i is 1, the input j is "Socrates", and the weight w is 1. So the output u'' is 1, and the output v'' is "Socrates", which together mean that "Socrates is mortal".

2.5 More on the Model

Let us discuss some possible objections to the model presented above, and explicate some implicit assumptions we have.

One objection to the above variable binding mechanism is that given a limited resolution in a node, it is impossible to represent all of the domain objects. However, I hold that, for a cognitive agent, only a relatively small set of objects are involved in any particular reasoning task at a given time, and therefore a network with a reasonable resolution should be able to handle such a set. A simple calculation suffices: suppose the value range is $[-1, 1]$, and the resolution is 0.001, then total 2000 objects can be handled at one time. A mechanism for dynamically assigning values to objects is necessary though (cf. Ajjanagadde & Shastri 1990), to fully utilize an available value range. We can dynamically map all relevant visual or mental objects onto a set of values within a range; this can be accomplished by a connectionist network in the form of a simple counter-like circuit, which assigns to each incoming object the (distinct) current value to be used in the reasoning system, and increments its own value by a small amount for the next object. This mechanism, which is mainly concerned with inputs for a reasoning system, need not be part of the reasoning system per se. Another technique for expanding ranges is to use a group of nodes (and a group of links attached to them) for representing the binding of one variable (which in this case has to be a vector), and thus to achieve much wider ranges than one single node. More nodes can be added to the group to reach any given resolution (see Appendix A for the detail of the mapping). To simplify the discussion, the above two mechanisms are not assumed in the following sections.

A related objection is that the above mechanism violates a standard connectionist precept that only simple messages are allowed to pass along links (because activation values represent firing frequencies of neurons, etc.). The answer is three-fold: (1) as shown in Appendix A, DN nodes can be mapped to a network of the standard type, so that no complex computation and messages are necessary; the DN formalism provides the necessary convenience in describing complex structures; (2) as indicated by many leading experts, the neural model with "activation values representing firing frequencies" is clearly too simplistic and probably wrongheaded (see Sun 1989 and Sun & Waltz 1990, and the references cited therein for details); although the generic DN node is too powerful, its various instantiations used in this work are more plausible; (3) similar network models have been advocated and used by some leading researchers, and they are proven advantageous in certain respects (see e.g. Aleksander 1989).

Another objection questions the use of abstract formalisms. The reason for using the DN formalism here is simply that we need (1) a rigorous means of descriptions in order to address some complex issues (see 3.1), (2) an abstraction that hides some network details unnecessary for discussing the problem in hand, (3) a formal and meticulous model that can reveal problems that might otherwise be overlooked, and (4) a conceptual tool and a guide for hardware implementation. Basic elements of this formalism can be readily implemented in conventional connectionist models.

3 Dealing with Difficult Issues

3.1 Consistency Checking, Unification, and Functions

There are a number of important issues in variable binding beyond what was discussed above. These issues were not adequately addressed in existing work but are crucial in achieving adequate symbolic processing capabilities. A detailed treatment of these issues is provided below.

First of all, a clarification needs to be made: X nodes do not necessarily contain a particular numerical value or symbol that represents a particular domain object (a constant); in other words, they can represent free variables. For example, if we have a rule:

$$a(x) \longrightarrow b(x)$$

given a(z), where z is a variable, we can derive b(z), by representing z with a special value in the X nodes of the assemblies for a(x) and b(x). In order to distinguish a constant (a real binding) from a variable (*a pseudo-binding*), variables are represented by values greater than a certain threshold θ, so a simple test (such as "if $x > \theta$") suffices to distinguish them.

In connectionist models, instead of simply passing bindings on, an X node representing an argument of a predicate may need to generate bindings. The situation can be illustrated by the following example:

$$a(x_1) \longrightarrow b(x_1, x_2, c)$$

Here the value for x_1 is simply passed from a to b, but two other values need to be generated in the b assembly to cover the last two arguments — x_2 and c (note that c is a constant but x_2 is a variable). The issue of *binding generation* is taken care of implicitly in the inference engine in symbolic systems, but it has to be taken care of explicitly in connectionist models. There are two cases both of which can be illustrated by this example: (1) the value generated for c is a genuine binding, and it should have been preset in the network, but (2) the value generated for x_2 simply denotes a variable (i.e. a possible binding, or a *pseudo-binding*). The same as usually done in Horn clause logic, a universal quantifier is assumed over x_2. The same as in performing resolution theorem proving for Horn clause logic (cf. Chang & Lee 1973), we can assume that different rules have different variable names before being compiled into a network (so that there will be no conflict); then a unique value representing x_2 can also be preset in the network. Both cases can be taken care of in the assembly representing the RHS of the rule (the b assembly in this case):

> In the X node of the first type, if the input from the C node is not zero, then output c, otherwise output nothing ($\perp$).
>
> In the X node of the second type, if the input from the C node is not zero, then output $p > \theta$, otherwise output nothing ($\perp$).

So the functions used for the C node in the b assembly are as follows:

$$O_C^1 = V_C^1(O_{C_a}^1, O_{X_1}^2, O_{X_2}^2, O_{X_3}^2)$$

$$= W * O^1_{C_a} \text{ if } O^2_{X_1} \neq \perp \; and \; O^2_{X_2} \neq \perp \; and \; O^2_{X_3} \neq \perp \; and \; O^1_{C_a} \neq \perp, \;\; \perp \; otherwise$$

$$O^2_C = V^2_C(O^1_{C_a}, O^2_{X_1}, O^2_{X_2}, O^2_{X_3})$$

$$= 1 \text{ if } O^2_{X_1} \neq \perp \; and \; O^2_{X_2} \neq \perp \; and \; O^2_{X_3} \neq \perp \; and \; O^1_{C_a} \neq \perp, \;\; \perp \; otherwise$$

where $O^1_{C_a}$ is the input from the C node in a, and W is the weight between the C node in a and its counterpart in b. The functions for the three X nodes carry out the above two kinds of generation:

$$O^1_{X_1} = V^1_{X_1}(O^1_{X1_a}, O^2_C) = O^1_{X1_a} \text{ if } O^2_C = 1, \;\; \perp \; otherwise$$

$$O^2_{X_1} = V^2_{X_1}(O^1_{X1_a}) = O^1_{X1_a}$$

$$O^1_{X_2} = V^1_{X_2}(O^2_C) = p > \theta \text{ if } O^2_C = 1, \;\; \perp \; otherwise$$

$$O^2_{X_2} = 1$$

$$O^1_{X_3} = V^1_{X_3}(O^2_C) = c \text{ if } O^2_C = 1, \;\; \perp \; otherwise$$

$$O^2_{X_3} = 1$$

where $O^1_{X1_a}$ is the input from the X node in a; the three nodes perform the binding as described before. As will be shown later, these functions can be systematically set up when compiling a rule set into a network. Note that pseudo-bindings will not cause any problem downstream, because it will be treated as a variable with a (implicit) universal quantifier, and can be unified in just the same way as any other variables (as will become clearer later).

There is also the issue of *consistency checking*. We may need to check the consistency of bindings, in case certain constraints are imposed regarding values of possible bindings for different arguments. For example, suppose we have the following rule:

$$a(x_1, x_2) \; b(x_2, x_3) \longrightarrow c(x_1)$$

where the constraint is that the x_2 in a is to be bound to the same value as the x_2 in b; another example is

$$a(x_1, x_2) \;\; x_1 \neq x_2 \longrightarrow c(x_1, x_2)$$

where we have to make sure that the inequality holds. In order to check input consistency, all X nodes have to pass on their inputs to a central location, namely the C node, for comparisons. The C node in the c assembly then performs the equality or inequality checking (cf. also Shastri & Ajjanagadde 1990). Given the DN formalism, these checkings can be easily formalized. With the first example, the functions used for X in the c assembly are:

$$O_X^1 = V_X^1(O_{X1_a}^1, O_C^2) = O_{X1_a}^1 \; if \; O_C^2 = 1, \;\; \perp \; otherwise$$

$$O_X^2 = V_X^2(O_{X1_a}^1) = O_{X1_a}^1$$

where $O_{X1_a}^1$ is the input from X_1 in the a assembly; and the function forms for C are:

$$O_C^1 = V_C^1(O_{C_a}^1, O_{C_b}^1, O_X^2, O_{X2_a}^1, O_{X2_b}^1)$$

$$O_C^2 = V_C^2(O_{C_a}^1, O_{C_b}^1, O_X^2, O_{X2_a}^1, O_{X2_b}^1)$$

where $O_{X2_a}^1, O_{X2_b}^1$ are inputs from X_2 nodes in assemblies a and b, and $O_{C_a}^1$ and $O_{C_b}^1$ are inputs from C nodes in a and b respectively; the functions for the C node in the c assembly perform the following mappings (where $I_1 = O_{C_a}^1$ and $I_2 = O_{C_b}^1$):

	O_C^1	O_C^2
$O_{X2_a}^1 = O_{X2_b}^1 \quad I_1 \neq \perp \quad I_2 \neq \perp$	$\sum W_i I_i$	1
otherwise	$\perp$	$\perp$

These mappings take care of consistency checking for this example. The other example above can be dealt with in a similar fashion.

An extension of the issue of consistency checking is that of *unification*, a much restricted version of the unification problem in first order logic (we perform unifications of variable/variable and variable/constant). Suppose we have a rule:

$$a(x, n) \longrightarrow b(x)$$

when given an input, say, $a(z, y)$, we have to make sure that y is unifiable with n; that is:

if $y = n$ or y is a variable, then they are unifiable.

This can be done in the X_2 node in the a assembly: [5]

$$O^1_{X_2} = V^1_{X_2}(y, O^2_C) = n \text{ if } O^2_C = 1, \quad \perp \ otherwise$$

$$O^2_{X_2} = V^2_{X_2}(y) = n \text{ if } y = n \ or \ y > \theta, \quad \perp \ otherwise$$

A mapping table for $O^2_{X_2}$ is as follows ($O^1_{X_2}$ is too simple to tabulate):

y	$O^2_{X_2}$
$y = n$	n
$y > \theta$	n
otherwise	$\perp$

Other functions (for C and X_1) remain the same.

What is especially difficult is the situation in which unification has to be done across predicates, as illustrated by the following rule:

$$a(x_1, x_2)\ b(x_2, x_3) \longrightarrow c(x_1)$$

Suppose we are given the following conditions:

$$a(m, n)$$

[5] Note that for each predicate appearing in a rule with a different set of arguments (ignoring the difference in variable names), there is a unique assembly specifically for it. See 3.2 for details.

and

$$b(x_4, p)$$

where m, n, and p are constants. Now we are in the following state:

$$a(m, n)\; b(x_4, p) \longrightarrow c(x_1)$$

We should unify x_4 with n (and we have to make sure that x_4 is indeed unifiable with n), and bind x_1 to m. The unification can be done within the C node in the c assembly as follows:

> if x_2 (of a) $= x_2$ (of b), or if x_2 (of a) $> \theta$, or if x_2 (of b) $> \theta$, then they are unifiable.

When the unification is done, a message is sent to the X node in the c assembly to enable it to accept input from the X_1 node in the a assembly; otherwise, inputs are deemed inconsistent with the rule and further activations are blocked. With the same equations as in the case of consistency checking, the C node performs the following mappings for unification (which carries out the above test):

$I_1 = O^1_{C_a}$	$I_2 = O^1_{C_b}$	O^2_X	$O^1_{X2_a}$	$O^1_{X2_b}$	O^1_C	O^2_C
not $\perp$	not $\perp$	not $\perp$	$< \theta, = O^1_{X2_b}$	$< \theta, = O^1_{X2_a}$	$\sum W_i I_i$	1
not $\perp$	not $\perp$	not $\perp$	$\geq \theta$	not $\perp$	$\sum W_i I_i$	1
not $\perp$	not $\perp$	not $\perp$	not $\perp$	$\geq \theta$	$\sum W_i I_i$	1
otherwise					$\perp$	$\perp$

Note that there is no sequential search involved in performing unification here. In cases where there are multiple occurences of variables and constants within and/or across predicates, the same approach applies, but the mappings will be more complicated. For the sake of space, such cases will not be discussed here.

A question here is how we should deal with the situation in which *multiple rules* having the same conclusion (but different conditions) reach their conclusions with different values; that is, how we combine conclusions from different rules. For example:

$$a(m,x)\ b(n,x) \longrightarrow c(m,n)$$

$$a(m,x)\ d(x,n) \longrightarrow c(m,n)$$

If both rules are activated, we have two conclusions, e.g., $c(m,n)$ with a confidence value 0.9, and $c(m,n)$ with a confidence value 0.8. The question is how we should combine them (or choose one of them). We want to choose the one with the maximum activation value (i.e. with the highest confidence value). This seems to be a reasonable choice (cf. Sun 1989 and Lange & Dyer 1989); it is actually the *best* choice when considering the rule interaction issue (which is beyond the scope of this paper; see Sun 1991). Nevertheless, we lose completeness by doing that (more on this later).

Given the answer to the above question, the issue of *selection* arises; that is, how do we choose and pass on the right binding, in case there are multiple sets of bindings available? With the DN model, we do the following: (1) the C node (in the assembly representing the conclusion) receives inputs from different rules and MAX them, and will output k if its kth input is the maximum; (2) then the corresponding X nodes will pass on the kth inter-assembly input, if the input from the C node (the intra-assembly input) is k. This simple two-step procedure basically takes care of the issue; the mappings for the C node in the c assembly can be easily set up as follows:

$$\begin{aligned} O_C^1 &= V_C^1(O_{C_a}^1, O_{C_b}^1, O_{C_d}^1, O_{X_1}^2, O_{X_2}^2) \\ &= max(W_1^1 O_{C_a}^1 + W_2^1 O_{C_b}^1,\ W_1^2 O_{C_a}^1 + W_2^2 O_{C_d}^1) \text{ if } O_{X_1}^2 \neq \perp\ and\ O_{X_2}^2 \neq \perp, \\ &\qquad \text{and the maximum is not } \perp \\ &\quad \perp\ otherwise \end{aligned}$$

$$
\begin{aligned}
O_C^2 &= V_C^2(O_{C_a}^1, O_{C_b}^1, O_{C_d}^1, O_{X_1}^2, O_{X_2}^2) \\
&= 1 \text{ if } W_1^1 O_{C_a}^1 + W_1^1 O_{C_b}^1 \geq W_1^1 O_{C_a}^1 + W_1^1 O_{C_d}^1 \\
& 2 \text{ if } W_1^1 O_{C_a}^1 + W_1^1 O_{C_b}^1 < W_1^1 O_{C_a}^1 + W_1^1 O_{C_d}^1 \\
& \perp \ otherwise
\end{aligned}
$$

where $O_{C_a}^1, O_{C_b}^1, O_{C_d}^1$ are inputs from C nodes in the a, b, and d assemblies, respectively. If one of the inputs is $\perp$, then the weighted-sum is $\perp$. If both weighted-sums are $\perp$, then the result is $\perp$. (These two functions can be easily put into table forms as before.) For X_1 and X_2,

$$O_{X_1}^1 = V_{X_1}^1(O_{X1_a}^1, O_C^1)$$

$$= O_{X1_a}^1 \text{ if } O_C^1 = 1 \ or \ O_C^1 = 2, \ \perp \ otherwise$$

$$O_{X_1}^2 = V_{X_1}^2(O_{X1_a}^1)$$

$$= 1 \text{ if } O_{X1_a}^1 \neq \perp, \ \perp \ otherwise$$

$$O_{X_2}^1 = V_{X_2}^1(O_{X1_b}^1, O_{X2_d}^1, O_C^1)$$

$$= O_{X1_b}^1 \text{ if } O_C^1 = 1, \ \ O_{X2_d}^1 \text{ if } O_C^1 = 2, \ \perp \ otherwise$$

$$O_{X_2}^2 = V_{X_2}^2(O_{X1_b}^1, O_{X2_d}^1)$$

$$= 1 \text{ if } O_{X1_b}^1 \neq \perp \ or \ O_{X2_d}^1 \neq \perp, \ \perp \ otherwise$$

where $O_{X1_a}^1, O_{X1_b}^1, O_{X2_b}^1, O_{X2_d}^1$ are inputs from X_1 or X_2 nodes in assemblies a, b, or d, respectively.

Yet another issue is how we extend variable binding to include *functions* (such as *living-in(father(john), boston, mass)*), or even *second order predicates.* The basic idea for solving this problem is: instead of one node for one argument, we can use a set of nodes, or a *subassembly*, for an argument. By doing this, a function, with all its parameters, can be readily assigned to a subassembly, in ways similar to assigning a predicate and its arguments to an assembly: in a subassembly, one node represents the function symbol, and the other nodes represent parameters of the function. Since a parameter of a function can be a function itself, the structure can be arbitrarily long. However, in any implementation, a subassembly can only contain a limited number of nodes, and this creates a dilemma. We would like to have as many nodes as possible, but want to limit the computational resources used. In practice, we can always choose a reasonable number of nodes for a subassembly, taking into consideration the characteristics of the domain.

In the following discussion, to simplify the matter, we will only deal with one level of functions, that is, only functions of constants and variables. To wire up rules that can contain one level of functional terms, we can set up assemblies for all predicates involved just as before, but in place of each X node, we have an X subassembly, which consists of an F node (for a function symbol) and a number of Y nodes (for function parameters). Revising the original definition in 2.3 regarding AS:

$$AS =_{\text{def}} \{(C, X_1, X_2, ..., X_k)| \text{ k is an arbitrary integer; C is a node,}$$

$$\text{and X's are subassemblies}\}$$

where

$$X_i =_{\text{def}} (F, Y_1, Y_2, ..., Y_t)$$

The connectivity pattern in an assembly basically remains the same, and all Y nodes (corresponding to X nodes in the original definition) will send messages (their inputs) to the C node as before. Each F node sends its input to the C node too. Then the C node can perform consistency checking, unification, and other tasks. t is the maximum number of parameters for a function. A special case is when the binding of a whole subassembly is just one constant or one variable, only one of the nodes (F) is used. In order to distinguish these cases, the C node has to distinguish a functional term, a constant, and a variable. This can be accomplished by the following test (extended from the test used before):

> suppose we have two thresholds θ_1 and θ_2; if $I_i < \theta_1$, then I_i is a constant; if $I_i > \theta_2$, then I_i is a variable; if $I_i < \theta_2$ and $I_i > \theta_1$, then it is a function symbol.

Based on this, we can specify the mappings used in the C, F and Y nodes as an extension of the previously specified mappings. As a simplified example, suppose

$$AS =_{\text{def}} \{(C, X)\}$$

where

$$X =_{\text{def}} (F, Y_1, Y_2)$$

that is, we only allow functions of up to two parameters in this example (other cases are similar). Suppose we want to set up a network connection for a rule:

$$a(x) \longrightarrow b(x)$$

Let us look into the assembly for b. In the assembly, we first determine the function forms for C:

$$O_C^1 = V_C^1(O_{C_a}^1, O_F^2, O_{Y_1}^2, O_{Y_2}^2)$$

$$O_C^2 = V_C^2(O_{C_a}^1, O_F^2, O_{Y_1}^2, O_{Y_2}^2)$$

where $O_{C_a}^1$ is the output from the C node in the a assembly. Next the functions for F are determined:

$$O_F^1 = V_F^1(O_{F_a}^1, O_C^2) = O_{F_a}^1 \text{ if } O_C^2 = 1, \ \perp \ otherwise$$

$$O_F^2 = V_F^2(O_{F_a}^1) = O_{F_a}^1$$

where $O_{F_a}^1$ is the input from the F node in the a assembly; the functions for Y_1 are

$$O_{Y_1}^1 = V_{Y_1}^1(O_{Y1_a}^1, O_C^2) = O_{Y1_a}^1 \text{ if } O_C^2 = 1, \ \ \perp \textit{otherwise}$$

$$O_{Y_1}^2 = V_{Y_1}^2(O_{Y1_a}^1) = O_{Y1_a}^1$$

where $O_{Y1_a}^1$ is the input from the Y_1 node in the a assembly; the functions for Y_2 are similar and thus omitted. Now the detailed mapping for C can be determined, which basically distinguishes three different cases (a constant, a variable, or a function):

$O_{C_a}^1$	O_F^2	$O_{Y_1}^2$	$O_{Y_2}^2$	O_C^1	O_C^1
not $\perp$	$\geq \theta_1, < \theta_2$	any	any	$W * O_{C_a}^1$	1
not $\perp$	$\geq \theta_2$	$\perp$	$\perp$	$W * O_{C_a}^1$	1
not $\perp$	$< \theta_1$	$\perp$	$\perp$	$W * O_{C_a}^1$	1
otherwise				$\perp$	$\perp$

Given a functional term, say $a(f(m))$ or $a(g(m,n))$, as input to the a assembly, the above functions can handle properly the propagation of bindings.

To further investigate the issue, look at a variation of the above example:

$$a(x) \longrightarrow b(f(x,m))$$

In this case, the functions for F in the b assembly have to generate the binding (recall our earlier discussion of binding generation):

$$O_F^1 = f \text{ if } O_C^2 = 1, \ \ \perp \textit{otherwise}$$

$$O_F^2 = f$$

The functions for Y_1 in the b assembly are

$$O_{Y_1}^1 = O_{F_a}^1 \text{ if } O_C^2 = 1, \ \ \perp \textit{otherwise}$$

$$O^2_{Y_1} = O^1_{F_a}$$

where $O^1_{F_a}$ is the input from the F node in the a assembly. The functions for Y_2 in the b assembly also have to generate a binding:

$$O^1_{Y_2} = m \text{ if } O^2_C = 1, \quad \perp \; otherwise$$

$$O^2_{Y_1} = m$$

The functions for C remain the same. The three nodes (F, Y_1, and Y_2) give their respective outputs, when given permission by the C node (through O^2_C). Note that in more complex cases, the mappings will also have to take into account the other issues discussed above (see Appendix B).

The above approach is not limited to one level only, as the above example seems to suggest. However, when multi-level embedding of functional terms is allowed, the complexity will be greatly increased, since many more nodes have to be allocated for each assembly. In the end, we have to impose a limit on how complex a formula can get. This special problem in implementing functional terms in connectionist networks stems from the fact that function symbols introduce structures the size of which is indeterminate at the time of constructing the network (at the "compile-time") and can only be decided when the actual binding is formed (at the "run-time"), whereas the other elements of rules and logics have pre-determinable sizes and structures. [6] Additional techniques, such as dynamic recruitment of nodes into assemblies, will be explored in the future to circumvent the difficulty, for practical purposes.

[6] It may not be unreasonable to impose an artificial limit. As a comparison, in human reasoning, no structural embedding of infinite depth can occur. For example, despite the theoretical possibility to do so with transformational grammars, no human can actually produce and comprehend a very long spoken sentence beyond a certain length and complexity limit.

3.2 Compilation

The above examples are small and isolated. For a large rule set, it is not easy to hand-wire a network. We need a systematic procedure for building up a network, when given a set of rules with weights. CFELDNN (*Compilation of Fuzzy Evidential Logic into DN Networks* [7]) is such a procedure.

We use three different types of assemblies in this procedure (see Figure 6):

- Ordinary assemblies – computing weighted sums of inputs and passing along bindings for the variables.
- OR assemblies – computing the maximum of inputs and selecting one set of bindings out of many based on who is the winner (the maximum).
- Complex assemblies – the same as ordinary assemblies, except performing constraint checking and/or unification, and also binding generation when necessary. (It does not perform OR functions, because we do not want to make complex assemblies too complex and unmanagable).

Assume each node has a set of intra-assembly inputs and a set of inter-assembly inputs; likewise, each node has a set of intra-assembly outputs and a set of inter-assembly outputs; assume there is no recursion in the rule set (the rule set is hierarchical); assume rules come with weights attached. The main procedure is as follows:

SET-UP-RULE-NET:

———————————————————————————

```
Initialize the rule set to include all the rules to be implemented.
Repeat until the set is empty:
```

[7] *Fuzzy Evidential Logic* will be explained later.

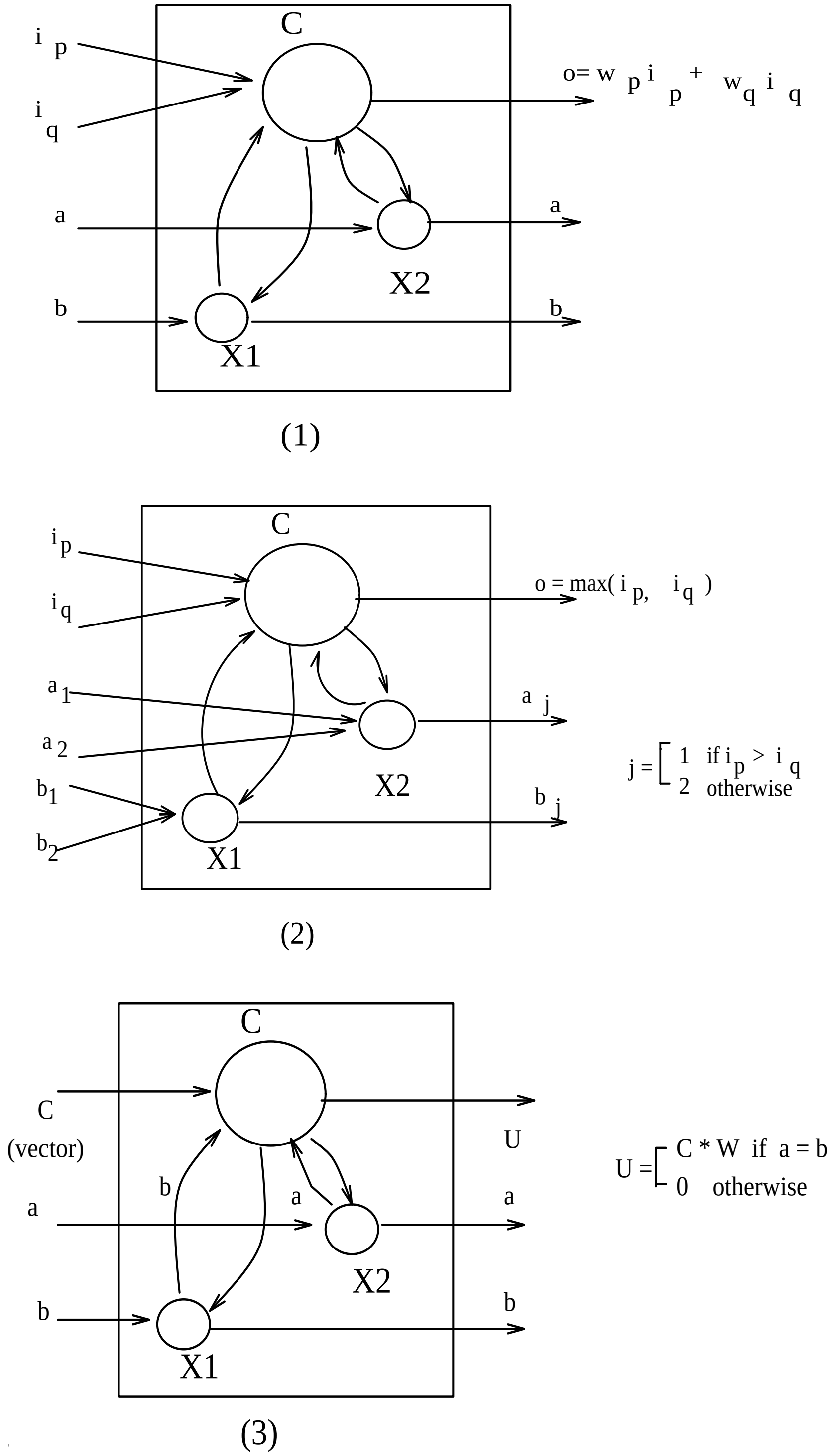

Figure 6: The Three Types of Assemblies

```
Choose a rule ( P1(X1) P2(X2) ...... Pn(Xn) --> Q(X) | w ) to work on,
     if there is no rule in the set that has Pi as conclusion.
Identify all existing assemblies for Pi, i=1,2,....,n,
   if there is none (with the same binding requirement), make one.
Identify all existing assemblies for Q,
   if there is none (with the same binding requirement), make one.
If there is only one assembly for each Pi and there is no in-link to Q,
   link each Pi to Q, assign weights to each Pi.
If there is in-link to Q,
   link Pi's to an intermediate assembly, then or-link this assembly
   with existing Q to a new Q assembly which assumes all out-links of Q,
   and assign weights as appropriate.
If there are multiple assemblies for Pi,
   or-link all assemblies for that Pi to an intermediate assembly,
   and then link that intermediate assembly to Q,
   and assign weights as appropriate.
Delete from the set the rule implemented.
```

Note that if we deal only with acyclic rule sets, the relations between LHSs and RHSs of various rules impose a partial ordering among the rule set; what the algorithm does is to choose a full ordering out of the partial ordering. Here we do not use any nodes with multiple sites in this procedure (Feldman & Ballard 1982); this is because we simply eliminate the need by introducing multiple nodes, each of which replaces one site, for the sake of clarity. Since we always put a bunch of related nodes together in an assembly, this means that instead of one assembly with nodes having multiple sites, now we simply have multiple assemblies consisting of nodes with only a single site. In the algorithm, "with the same binding requirement" refers to consistency checking, binding generation and other internal mechanisms that have to be individualized for particular rules. When such requirements exist for a predicate (in the LHS or RHS of a rule), we have to make a separate assembly for that predicate in that particular rule. All the basic ideas concerning this procedure have been discussed before, therefore we will leave all the details of subprocedures for setting up assemblies, including C and X nodes, for linking together assemblies in various ways, and for dealing with issues identified earlier, such as consistency checking, selection, and unification, to the Appendix B.

3.3 Implementation with Conventional Models

Although we mentioned briefly before that DN nodes can be implemented in conventional connectionist models, we now want to discuss in more details that the basic node types used in the network constructed by CFELDNN can be readily implemented in conventional connectionist networks (and thus a whole network of DN nodes can be implemented in conventional connectionist networks):

- For ordinary assemblies, the C node receives messages from X nodes, computes weighted sums of inputs, and sends a simple message to X nodes; its functionality can be fully determined by the mappings specified before, which can be used to train a three-layer backpropagation network. The X nodes pass along bindings when receiving proper messages from the C node, and their implementation in conventional models are straightforward by doing a backpropagation training of a three-layer network (mapping strings to numerical values and $\perp$ to 0, etc.) (see Appendix A).

- For OR assemblies, the C node computes the maximum of inputs, and the computation can be done by a three-layer backpropagation network. Each X node chooses one binding out of many based on which set is the winner (having the maximum activation for the C node); this can be easily carried out by a three-layer backpropagation network too. Intra-assembly communications can be taken care of along with the above basic functionalities.

- Complex assemblies perform constraint checking and/or unification (and other operations, except OR), and from the discussion in 3.1, it is clear that they can be implemented in conventional networks based on the mappings specified in table forms (which can be used to set up and train backpropagation networks).

Clearly distributed representation is used in this implementation in conventional connectionist networks. But contrary to Touretzky & Hinton (1985), which uses globally distributed representation (i.e. the same representational space for all symbols), we use modularly distributed representation, in which each variable has its own representational space. We can avoid a lot of interferences and crosstalks this way (cf. Shastri & Ajjanagadde (1990)'s analysis of crosstalks in Touretzky & Hinton 1985)

3.4 Logic and Variable Binding

To better see the relationship between logics and connectionist models, let us look into a logic FEL (which stands for *Fuzzy Evidential Logic*; Sun & Waltz 1991), and its relation to weighted-sum connectionist models and Horn clause logic. A **fact** in FEL is an atom or its negation, represented by a letter (with or without a negation symbol) and having a value between l and u (e.g. $[0, 1]$ or $[-1, 1]$), representing its confidence. The value of an atom is related to the value of its negation by a specific method, so that knowing the value of an atom results in immediately knowing the value of its negation, and vice versa. A **rule** in FEL is in turn a structure composed of two parts: a left-hand side (LHS), which consists of one or more facts, and a right-hand side (RHS), which consists of one fact. When facts in the LHS get assigned values, the fact in the RHS can be assigned a value according to a *weighting scheme*[8]. For example, $a\ b\ c \longrightarrow d$ is a rule, and a weighting scheme associated with it will tell us the value of d once we know the values of a, b and c.

The above description is common to almost all logics or production systems, but numerical calculations associated with this logic are different from either probabilistic reasoning (cf. Pearl 1988) or common forms of fuzzy logic (cf. Zadeh 1988), in that a simpler and more intuitive scheme is employed (Sun 1991). [9] We use a **weighting scheme**, which is a way of assigning a weight to each fact (reflecting its relative importance in determining the conclusion) in the LHS of a rule, with the total weight (the sum of the absolute values of weights) less than or equal to 1, and of determining the value (the confidence) of the fact in the RHS of a rule by thresholded (if thresholds are used) weighted-sum of the values of the facts in the LHS (i.e. the inner-product of the weight vector and the vector of the values of the LHS facts). When a threshold is used and the range of values is continuous, then the weighted-sum is passed on if its absolute value is greater than the threshold, or the result is 0. When a threshold is used and the range of values is binary (or bipolar), then the result will be one or the other depending on whether the weighted-sum (or the absolute value of it) is greater than the threshold or not

[8]When the value of a fact in the LHS is unknown, assign a particular value, say zero, as its value.

[9]However, because of the simplicity, we may lose some finer details of the probability theory, including interactions among different premises and rules.

(usually the result will be 1 if the weighted-sum is greater than the threshold, 0 or -1 if otherwise). For example, a weighting scheme for the above rule is

$$w_1 = 0.3, \quad w_2 = 0.3, \quad w_3 = 0.4.$$

where each weight reflects the relative importance of the corresponding fact in determining the conclusion, and the value of d is calculated by the weighted-sum (or inner-product) of the values of the facts in the LHS (i.e. the confidence values of the conditions):

$$d = w_1 a + w_2 b + w_3 c$$

A **conclusion** in FEL is a value associated with a fact, calculated from rules and facts by doing the following: for each rule having that fact in its RHS, obtain conclusions of all the facts in its LHS (if any fact is unobtainable, assume it to be zero, or any value used for representing *unknown*), and then calculate the value of the conclusion in question using the weighting scheme.

Note that *conclusion* is defined here recursively. Thus we may want to restrict the structure of theories allowed: a rule set is said to be **hierarchical**, if the graph depicting the rule set is acyclic; the graph is constructed by drawing a unidirectional link from each fact in LHS of a rule to the fact in RHS of a rule. Making a rule set hierarchical avoids circular reasoning.

Now FEL can be defined as follows:

> A **Fuzzy Evidential Logic** (FEL) is a 5-tuple: $< A, R, W, T, I >$, where A is a set of facts, the values of which are initially assumed to be zero (or any value used for representing *unknown*), R is a set of rules, W is a weighting scheme for R, T is a set of thresholds each of which is for one rule, and I is the initial condition: a set of elements of the form (f, v), where f is a fact, and v is a value associated with f.

In reasoning, if we do not know anything about a fact, we often assume its value to be zero (especially when the value range is $[-1, 1]$). This is mentioned in the above definition of FEL. However, to accomplish this in CONSYDERR, we have to explicitly set all input lines, except those that represent known information, to zero. Otherwise, the inputs and the outputs of the assemblies representing those unknown facts will be $\perp$. An alternative is to equate $\perp$ with zero in implementation, so that this assumption will be taken care of automatically. (The $\perp$ symbol in the DN formulation of CONSYDERR is provided for more flexibility, so that other possible mappings are also allowed, and also for a clearer semantics, so that implicit assumptions have to be brought into the open.)

A binary version of FEL can simulate Horn clause logic. A **binary FEL** is a reduced version of FEL, in which values associated with facts are in $[-1, 1]$, each weight is positive, the total weight (the weighted sum of the absolute values of all the weights) of each rule is 1, and all thresholds are set to 1.

Theorem 1 *The binary FEL is sound and complete with respect to (propositional) Horn clause logic.* [10]

Proof: The inference rule for FEL can be defined as a variant of forward chaining (note the definition of *conclusion*). Let K be a set of FEL facts and their values, in the form of pairs (f,v), where f is a fact and v is its value, a real number between -1 and 1 representing the confidence that f is true. Assume that all facts are uniquely represented in K (though their confidence values may be zero). The inference rule will simply add FEL facts to K until no new ones can be added:

(1) Given the FEL rule: $a_1.....a_r \longrightarrow b \quad (w_1, ..., w_r)$

[10]Here *soundness* means that whatever can be inferred in binary FEL can be inferred in Horn clause logic; *completeness* means whatever can be inferred in Horn clause logic can be inferred in binary FEL.

If (a_1, v_1),....(a_r, v_r) are in K, then let $v' = w_1 * v_1 + ... + w_r * v_r$ and assume (b,v) be in K. If $|v'| \geq \theta$ and $|v'| > |v|$, then replace (b, v) by (b, v').

(2) Given the FEL fact: (b, 1) (the known fact), we simply replace (b,v) in K by (b,1).

In case of binary FEL, we have $\theta = 1$. Also, if we know that the value of b is 1, then we know that the value of $\neg b$ is -1, vice versa.

Given a Horn clause theory H, one can produce a corresponding binary FEL theory F as follows: each Horn clause $a_1, ...a_r \longrightarrow$ b is transformed into a FEL rule by associating a weight w_i with each fact a_i, in such a way that $w_i > 0$ and their sum is 1. Conversely, given a binary FEL theory F, one can also produce a corresponding Horn clause theory H as follows: each FEL rule $a_1, ...a_r \longrightarrow b \quad (w_1, w_2, ..., w_r)$ is transformed into a Horn rule directly, by ignoring weights w_i's, since according to the definition of binary FEL, we always have $w_i > 0$ and their sum is 1.

Assume that all facts in K initially have values 0. Then (b,1) is a FEL conclusion of F iff b is a logical consequence of H. To see this, observe that a FEL fact (b,1) is added to K if and only if all of the facts in the LHS of the rule have values 1. Thus, (b,1) is only introduced to K if there exists facts (a_1,1),....,(a_r,1) in K. It follows that the FEL inference rule behaves exactly like the forward chaining operator for Horn clause logic when we restrict our attention to facts of value 1. b is a conclusion of H, iff b can be inferred by forward chaining in H, iff (b,1) can be inferred by the FEL inference rule.

Therefore, because of the fact that the Horn clause forward-chaining inference rule is sound and complete, binary FEL is sound and complete with respect to Horn clause Logic.

Q.E.D

Having dealt with Horn clause logic, we want to mention also that FEL can handle as well an extension of Horn clause logic – Shoham's logic (Shoham 1987; a modal logic formalism purporting to explain causality), further demonstrating the power of FEL (Sun 1991).

The advantage of FEL lies in the fact that, in spite of its generality, it can be implemented in connectionist networks directly, and thus it bridges the gap between logics and connectionist models. We link FEL to the CL level of CONSYDERR: let a node represent one and only one fact (including its negation) and links represent rules that go from the nodes representing conditions of a rule to the node representing the conclusion of the rule; then FEL is represented by a network of nodes connected via links, which is exactly a CL. Through FEL, the logical capability of the weighted-sum connectionist models is established.

To add variables to this logic, the above definitions and descriptions can be extended in a straightforward way. This extension is omitted for avoiding redundancy (see Sun 1991).

3.5 Correctness of the Solution

Let us look into the logical correctness of the CONSYDERR architecture.

Theorem 2 *Assume that there is no recursion in FEL (i.e. with a hierarchical rule set) and there is no functional term, the implementation of FEL in* CONSYDERR *with DN is complete and sound with respect to FEL.* [11]

A sketch of proof:

[11] Here *soundness* means that whatever can be inferred in the implementation can be inferred in FEL; *completeness* means whatever can be inferred in FEL can be inferred in the implementation.

Since the FEL rule set in question is hierarchical, we can order them into levels. Then an inductive proof on the number of levels will do. From all of the previous discussions regarding the issues, we can prove the base case, and carry out the induction.

Note that the completeness (that is, whatever can be inferred in FEL can be inferred in the implementation) comes under the assumption that we replicate and rename a predicate (and all rules involving that predicate) throughout the entire rule set if it appears more than once with different kinds of arguments in RHSs of different rules (note that this is only possible when the rule set is hierarchical). Otherwise, the different outcomes (with different bindings) from these different rules will be MAXed, and therefore the implemented system loses its completeness. For example,

$$a(m,x)\ b(n,x) \longrightarrow c(x,n)$$

$$a(m,x)\ d(x,l) \longrightarrow c(x,l)$$

If both rules are activated, we have two conclusions — but only one will be sustained because the MAX operation in combining the results (see 3.1). If we reach $c(m,n)$ with a confidence value 0.9, and $c(m,l)$ with a confidence value 0.8, then only the first conclusion will remain. What we can do is to replicate c:

$$a(m,x)\ b(n,x) \longrightarrow c(x,n)$$

$$a(m,x)\ d(x,l) \longrightarrow c'(x,l)$$

and also replicate all the rules that have c in their LHSs: suppose originally we have a rule

$$c(x,y) \longrightarrow e(x,y)$$

then we will have the following duplicated rules:

$$c(x,y) \longrightarrow e(x,y)$$

$$c'(x, y) \longrightarrow e'(x, y)$$

and this process continues recursively until we exhaust all such rules. [12]

With recursive rules, either direct or indirect, the system will lose its completeness, if implemented with cyclic connections. However we are mainly interested in hierarchical FEL (as a comparison, note that human reasoning is mainly forward-chaining and does not involve circular inferences; cf. Posner 1989), so this is not a problem. Besides, there are also a number of ways we can circumvent recursions. One way is to replicate the predicate a number of times in the network, up to the maximum expected depth of the recursion, provided that we have that information a priori. Another possible way is to use an external stack, which stores and pops out multiple bindings when necessary.

Corollary 1 *Assume that there is no recursion and there is no function symbol, the above implementation of FEL is sound and complete in the binary case with respect to the first order Horn clause logic.*

Observe the fact that the theorem relating binary FEL to Horn clause logic can be extended to the first order case, so that binary first order FEL is sound and complete with respect to first order Horn clause logic. The corollary follows directly, from the above theorem and this fact.

It should be noted that the above theorems do not imply that rule sets used in CONSYDERR *must* be hierarchical; rather, they only show that in such special cases where the rule sets are indeed hierarchical, the implementation in CONSYDERR can carry out some logics precisely. In fact human reasoning in general

[12]Note that this implementation depends on the rule set being hierarchical. Admittedly, however, many connectionist reasoning systems are not hierarchical.

can hardly be characterized as being very logical, nevertheless it can be precise in some occasions, and these above theorems abstractly demonstrate this possibility.

It is also interesting to note the following:

Theorem 3 *In case of propositional FEL, no matter whether or not it is hierarchical, the implementation of FEL as described before is always sound and complete with respect to propositional FEL.*

A sketch of proof:

An inductive proof on the number of steps (rules applied) will do. The base case is concerned with all the rules that are directly applicable given the initial input. Then with inductive hypothesis, induction can be carried out by examining all the rules that can be applied given the current state. Note that this can be done because there is no multiple binding problem at all for propositional logics.

4 Related Work

Touretzky & Hinton (1985) is the first work that confronts the problem of variable binding. Their system, imitating the structure of a symbolic production system, is made up of several separate modules, for working memory, production rules and facts, respectively. With distributed representation in these modules, the system is able to match variables against data in the working memory module by *winner-take-all.* The system has very complex structures and is costly in terms of computational resources, but it uses only very simple node functions (see also Sun 1991b). The system is only capable of performing sequential rule-based reasoning (at the global level). Another piece of early work is Smolensky (1987), which proposes using *tensor products* for variable binding, with either local or distributed representation. Basically, each combination of variables and values is represented in the system by a single node, or a group of them, which detects dynamically if the right binding is present. It utilizes simple node functions but a tensor product network can be of high computational complexity.

Barnden (1988) presents another system involving variable binding. In this system, data reside in grid form networks; data structures are represented with the help of adjacency relations and highlighting techniques; rules are hardwired with circuitry for detecting the presence of data that match a particular rule; once matched, an "Action Part" module associated with a rule is used to add a new data structure representing the conclusion from the matched rule. The structure of the system is relatively simple, but the symbolic manipulation necessary to match rules against data is a complicated process; as a result of the matching process, parallelism in reasoning is not fully captured. The system is thus basically a sequential rule-based system (at the global level).

Concurrent with CONSYDERR, there are two pieces of similar work (Ajjanagadde & Shastri 1989 and Lange & Dyer 1989; both appeared in the same volume as Sun 1989). Ajjanagadde & Shastri (1989, 1990) presents systems that perform backward chaining inferences with variable binding handled by temporal codings of two different types. The later version basically works this way: a node representing a variable produces a temporal activation pattern (a spike train) with a specific phase that can be synchronized with a node representing a particular binding. Circuits are devised to enable such a synchronization dynamically. This scheme has relatively simple network structures but needs nodes with temporal properties, capable of detecting activation phases.

Lange & Dyer (1989, 1989a) presents a solution very similar to the approach in this paper; that is, a distinct value is used to represent a particular binding, which they call a *signature*. Sigatures are used to represent different domain objects, and are passed along in rule applications. In their system, which is frame-based (similar to assemblies used in this paper to a certain extent), rules are used for selecting and activating frames. Activated frames help in deciding meanings of words in text, as the system is applied to the task of word sense disambiguation. Their system is task specific, and thus the variable binding mechanism is not, and does not need to be, very sophisticated. However, they deal with issues such as concurrent multiple bindings, cyclic connections, and global inhibition.

Now let us examine how these above solutions handle the important issues in variable binding which we discussed in this paper:

- The study of the *relationship between connectionist models and logics* helps us in delineating the scope of solutions to the variable binding problem, in gaining new insights into the matter and in using logics as yardsticks for measuring various solution strategies. Shastri & Ajjanagadde (1990) looks briefly into this issue in their phase-based system, but they only treated binary cases. Lange & Dyer (1989), Barnden (1988) and Dolan & Smolensky (1989) do not deal with the issue. Touretzky & Hinton (1985) only allows very limited forms of rules (with one variable).

- Touretzky & Hinton (1985) does not take into account *graded and fuzzy concepts* in reasoning. The other solutions cited above allow more general forms of rules and thus can potentially accommodate these but they do not provide any treatment.

- In Touretzky & Hinton (1985), the issue of *binding generation* is taken care of automatically because of the way they encode rules, predicates (i.e. tuples in their terminology), and arguments, in a distributed fashion. Shastri & Ajjanagadde (1990) does not deal with this issue explicitly, but it seems that they should be able to deal with the issue simply by not providing any temporal firing patterns synchronized with a variable. Others did not deal with the issue explicitly.

- When there are *multiple rules* applicable, Touretzky & Hinton (1985)'s system picks one arbitrarily, and thus the result is unpredictable. This is unsatisfactory, if we want to make sure that all our relevant knowledge is brought to bear on the problem. Lange & Dyer (1989) uses MAX for combining activations of rules, but do not handle conflicting bindings. It is not clear how others deal with the problem.

- Shastri & Ajjanagadde (1990) performs limited *consistency checking* and *unification* in their systems with means different from ours. Touretzky & Hinton (1985) has built-in *unification* because of the distributed representation, but one drawback is that they only allow very restricted forms of rules. Lange & Dyer (1989) does not deal with this issue.

- As for *functional terms*, Ajjanagadde (1990) explors it briefly in a phase-based system. To my best knowledge, the other solutions did not deal with this problem.

However, some of the existing solutions do have some apparent advantages. We will look into the following issues:

- Touretzky & Hinton (1985) is capable of reusing the same rules many times so that infinitely long chains of reasoning are possible. This can be handled in CONSYDERR by feeding the current output back into the network to start the next cycle.

- By using distributed representations, Touretzky & Hinton (1985) exhibits many interesting characteristics of the PDP approach, such as approximate pattern matching, generalization, and graceful degradation. In contrast, the approach adopted in this paper is mainly localist, which usually does not have the above characteristics. However, by implementing DN nodes in distributed networks (as explained in Appendix A), we can obtain similar characteristics in our system. The extent to which CONSYDERR exhibits these characteristics is yet to be determined.

- Shastri & Ajjanagadde (1990) is capable of dynamically assigning different phases to different variables and objects at different times, and therefore is able to fully utilize the limited range available. I shall note that this on-the-fly assignment of phases or frequencies is not an intrinsic part of the reasoning system, and is carried out by mechanisms external to the network performing inferences. Similar mechanisms can be added to CONSYDERR to fully utilize the available value range (which has only limited resolution), by dynamically mapping all relevant objects onto a set of values within the range (as mentioned in 2.5). The value range in CONSYDERR is at least as wide as the phase or frequency range, especially if one believes that activation values represent neuron firing phases or frequencies.

- Lange & Dyer (1989) incorporates mechanisms for inhibitions and gatings. CONSYDERR can certainly accommodate some mechanisms for inhibitions and gatings, in addition to activation propagations, because of the generality of the DN formalism. However, unlike the activation propagation (which can be characterized by FEL), the logical characteristics of such mechanisms are unclear.

5 Concluding Remarks

This paper discusses the problem of variable binding in connectionist networks, in the context of the CONSYDERR architecture. We present a thorough and rigorous solution to the variable binding problem, and formal definitions are introduced to clarify subtle details that were overlooked before. We analyze the solution to see how it can deal with the intricate issues involved in variable binding. To help with this analysis, a logic FEL is defined, which is a direct reformulation of connectionist networks, demonstrating logical capabilities of connectionist models. A procedure is developed to transform a set of FEL rules into a DN network systematically. This solution to the variable binding problem aims for structural simplicity while maintaining well-definedness and simplicity in individual processing elements.

A Compatibility of the DN model with other connectionist models

To see how a DN model emulates other connectionist models, let us look into various connectionist (neural network) models. In general, connectionist models can be divided into four classes:

- Continuous input/discrete activation models (e.g. the linear threshold model in McClelland & Rumelhart 1986),
- Discrete input/discrete activation models (e.g. Feldman & Ballard 1982),
- Continuous input/continuous activation models (e.g. the interactive activation and competition model; McClelland & Rumelhart 1986),
- Discrete input/continuous activation models (as another possibility).

All of them can be easily handled by our general formalism. To simulate a continuous input/discrete activation model (suppose a particular continuous activation function, such as the weighted-sum threshold function, is used), let a DN model be

$< S, A, B, I, O, U, V >$

where

B=$\{0, 1\}$,

A=$\{0, 1\}$,

S=$\{\}$

U is not needed,

V= the original weighted-sum thresholding activation function.

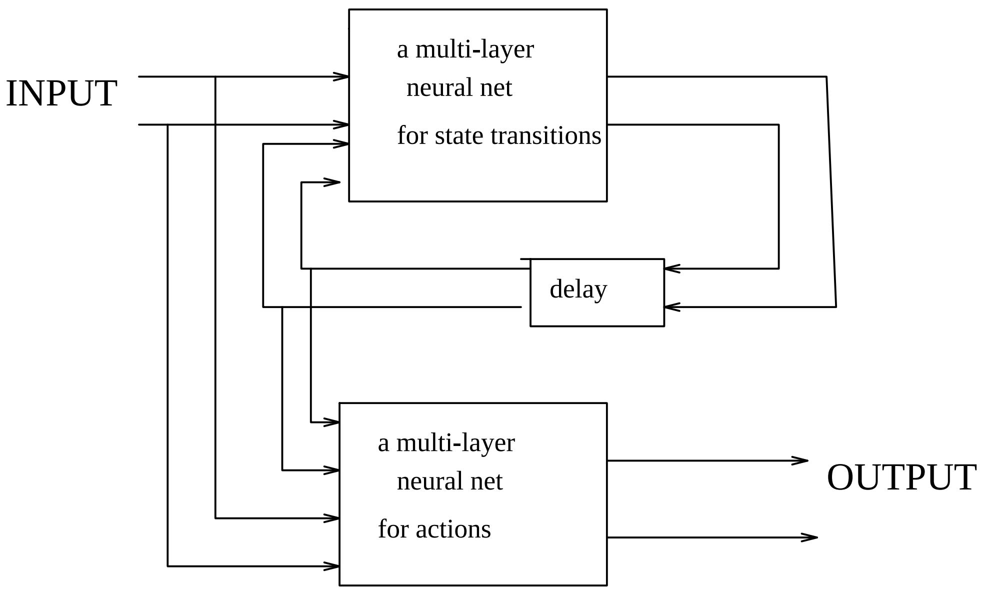

Figure 7: Implementing a Finite State Automaton in Weighted-sum Threshold Nets

This model can emulate the original model and produce the same output: 0 if the weighted-sum of inputs is less than the threshold and 1 if the weighted-sum of inputs is greater than the threshold. The model can carry out the computation exactly as its conventional counterparts.

Conversely, DN nodes can also be implemented with the conventional weighted-sum threshold model or the sigmoid activation model. The question here is how to implement the state variable and the state transition function of a DN node. It has been shown that a modular multi-layer network can do the job: a hidden layer represents the current state and each input/output value is represented by a group of nodes and is sent through a group of links associated with these nodes. The output from the hidden layer is fed back into the input layer to help decide, together with current inputs, which state to enter next and what to output (Fang & Wilson 1989). See Figure 7.

If, for certain applications, the state is not needed, then the top half of the diagram (concerning state transitions and delays) can be removed.

B Compilation Procedures

The procedure for setting up each individual assembly is

SET-UP-ASSEMBLY (for ordinary assemblies):

```
Set-Up-C

Set-Up-X
```

SET-UP-C (for ordinary assemblies):

```
  (I= (I1, I2, .....In) is the input from other assemblies)
  (I'= (I1', I2', ...)  are the inputs from Xs in the same assembly)
output to other Cs:  W * I, if I' not empty and I not empty
                     nothing, otherwise
output to Xs:  1 if I' not empty and I not empty
               nothing, otherwise
```

The output to Xs are to inform them if they should pass on bindings. I' is used to determine if there is right binding present at variable nodes. "not empty" means that none of the components of the vector involved is empty ($\perp$).

SET-UP-X (for ordinary assemblies):

```
             (I is the input from Xs in other assemblies )
             (I' is the input from C)
output to C: I
             nothing, otherwise
output to Xs in other assemblies: I if I'=1
                                  nothing, otherwise
```

OR-LINK-ASSEMBLY (for OR assemblies):

```
For C node:
        (I is the input from other assemblies. k denotes the group)
        (each group is a vector by itself)
  output to other Cs: W(k) * I(k) = max( W(j) * I(j)) if not empty
                      nothing, otherwise
  output to Xs: k if I(k) not empty
                nothing, otherwise
For X nodes:
           (I is the input from other assemblies)
           (Each element of I is a scalar)
           (I' is the input from C)
  output to C:  1 if I not empty
                nothing, otherwise
  output to Xs in other assemblies: Ik, if I'=k
                                    nothing, if I' empty
```

GENERATE-ARGUMENT (for complex assemblies):

```
For C node:
  (the same as for ordinary assembly)
For X:
  output to Xs: p if I'=1
                nothing, otherwise
  (the same as before otherwise)
```

CONSTRAIN-ARGUMENT (for complex assemblies):

```
For C node:
  if constraints are satisfied, then outputs are
    the same as above
```

```
  nothing, otherwise
```

UNIFY-ARGUMENT (for complex assemblies):

```
For C node:
       (I is the input from Cs in other assemblies)
       (I' is the input from Xs)
  output to Xs: 1 if I' can be unified and I  not empty
                nothing, otherwise
  (the same as above otherwise)
For X node:
       (I is the input from Xs in other assemblies )
       (I' is the input from C)
  output to C: I if I can be locally unified and I not empty
               nothing, otherwise
  output to Xs in other assemblies: the unified I, if I'=1
                                    nothing, otherwise
```

REFERENCES

Ajjanagadde, V. (1990) Reasoning with Function Symbols in a Connectionist System. *Proc. 12th Cognitive Science Conference*, pp.??.

Ajjanagadde, V. and Shastri, L. (1989) Efficient Inference with Multi-place Predicates and Variables in a Connectionist System. *Proc.11th Cognitive Science Society Conference*, pp.396-403, Hillsdale, NJ: Lawrence Erlbaum Associates.

Aleksander, I. (1989) The logic of connectionist system, in: I. Aleksander ed. *Neural Computing Architectures*, Cambridge, MA: MIT Press.

Barnden, J. (1988) The right of free association: relative-position encoding for connectionist data structures, *Proc.10th Conference of Cognitive Science Society*, pp.503-509, Hillsdale, NJ: Lawrence Erlbaum Associates.

Blelloch, G. (1986) AFL-1: A Programming Language for Massively Parallel Computers, Master's Thesis, MIT AI Lab.

Chang, C. and Lee, R. C. (1973) *Symbolic Logic and Mechanical Theorem Proving*, Academic Press.

Dolan, C. and Smolensky, P. (1989) Implementing a connectionist production system using tensor products. in D. Touretzky et al. eds. *Proc.1988 Connectionist Summer School*, pp.265-272. San Mateo, CA: Morgan Kaufman.

Fang, L. and Wilson, W. (1989) A Study of Sequence Processing on Neural Networks. *Proc.IJCNN*, Washington DC, pp.588-598, IEEE Press.

Feldman, J. and Ballard, D. (1982) Connectionist models and their properties. *Cognitive Science*, 205-254.

Fodor, J. and Pylyshyn, Z. (1988) Connectionism and Cognitive Architecture:

A Critical Analysis. in: Pinker and Mehler (eds.) *Connections and Symbols.* Cambridge, MA: MIT Press.

Gallant, S. (1988) Connectionist Expert Systems. *Communication of ACM*, V.31(2), 152-169.

Hendler, J. (1988) *Integrating Marker-passing and Problem Solving.* Hillsdale, NJ: Erlbaum.

Kosko, B. (1986) Fuzzy Cognitive Maps. *International Journal of Man-Machine Studies*, 24, 65-75.

Lange, T. and Dyer, M. (1989) Frame selection in a connectionist model, *Proc.11th Cognitive Science Conference*, 706-713. Hillsdale, NJ: Lawrence Erlbaum Associates.

Lange, T. and Dyer, M. (1989 a) High Level Inferencing in a Connectionist Network. *Connection Science*, 181-217.

Pearl, J. (1988) *Probabilistic Reasoning in Intelligent Systems.* San Mateo, CA: Morgan Kaufman.

Pinker, S. and Prince, A. (1988) On Language and Connectionism. in: S. Pinker and J. Mehler (eds.), *Connections and Symbols.* Cambridge, MA: MIT Press.

Posner, M. (1989) (ed.) *Foundations of Cognitive Science.* Cambridge, MA: MIT Press.

Rumelhart, D., McClelland, J. and the PDP Research Group (1986) *Parallel Distributed Processing: Explorations in the Microstructures of Cognition.* Cambridge, MA: MIT Press.

Shastri, L. (1988) A Connectionist Approach to Knowledge Representation and

Limited Inference. *Cognitive Science*, 12, 331-392.

Shastri, L. and Ajjanagadde, V. (1990) From simple association to systematic reasoning. Tech Report MS-CIS-90-05, University of Pennsylvania.

Shoham, Y. (1987) *Reasoning about Change: Time and Causation from the standpoint of Artificial Intelligence.* Ph.D Dissertation, Computer Science Department, Yale University.

Smolensky, P. (1987) On variable binding and representation of symbolic structure. Tech Report, University of Colorado, Boulder, CO.

Sun, R., Marder, E. and Waltz, D. (1989) Model local neural networks in the lobster stomatogastric ganglion. *Proc.IJCNN*, p.II-604, IEEE Press.

Sun, R. (1989) A discrete neural network model for conceptual representation and reasoning. *Proc.11th Cognitive Science Society Conference*, pp.916-923. Hillsdale, NJ: Lawrence Erlbaum Associates.

Sun, R. and Waltz, D. (1990) Neural networks and human intelligence. *Journal of Mathematical Psychology*, 34(4), 483-488.

Sun, R. (1990) Integrating Rules and Connectionism for Robust Reasoning, *Proc.AAAI Workshop on Integrating Neural and Symbolic Processes.* Boston, MA.

Sun, R. (1991) *Integrating Rules and Connectionism for Robust Reasoning.* Ph.D Thesis, Brandeis University, Waltham, MA.

Sun, R. (1991 a) Connectionist Models of Rule-Based Reasoning, *Proc.13th Cognitive Science Conference*, pp.437-442. Hillsdale, NJ: Lawrence Erlbaum Associates.

Sun, R. (1991 b) Chunking and Connectionism, *Neural Network Review*, Vol.4, No.2, 76-78.

Sun, R. (1991 c) The Discrete Neuronal Models and the Probabilistic Discrete Neuronal Models. in: B. Soucek ed. *Neural and Intelligent System Integration*, pp.161-178. New York, NY: John Wiley & Sons.

Sun, R. and Waltz, D. (1991) Neurally Inspired Massively Parallel Model of Rule-Based Reasoning. in: B. Soucek ed. *Neural and Intelligent System Integration*, pp.341-381. New York, NY: John Wiley & Sons.

Sun, R. (1992) Connectionist Models of Commonsense Reasoning Incorporating Rules and Similarities. *Knowledge Acquisition*, in press.

Touretzky, D. and Hinton, G. (1985) Symbols Among Neurons. *Proc.9th IJCAI*, pp.238-243. San Mateo, CA: Morgan Kaufman.

Zadeh, L. (1965) Fuzzy Sets. *Information and Control*, 8, 338-353.

Zadeh, L. (1988) Fuzzy Logic. *Computer*, Vol.21, No.4, 83-93.