Schemas, Logics, and Neural Assemblies

Ron Sun
Department of Computer Science
The University of Alabama
Tuscaloosa, AL 35487

rsun@athos.cs.ua.edu

To appear in: Applied Intelligence. Vol.5, No.2. 1995.

Abstract

To implement schemas and logics in connectionist models, some form of basic-level organization is needed. This paper proposes such an organization, which is termed a discrete neural assembly. Each discrete neural assembly is in turn made up of discrete neurons (nodes), that is, a node that process inputs based on a discrete mapping instead of a continuous function. A group of discrete neurons (nodes) closely interconnected form an assembly and carry out a basic functionality. Some substructures and superstructures of such assemblies are developed, to enable complex symbolic schemas to be represented and processed in connectionist networks. The paper shows that logical inference can be performed precisely, when necessary, in these networks and with certain generalization, more flexible inference (fuzzy inference) can also be performed. The development of various connectionist constructs demonstrates the possibility of implementing symbolic schemas, in their full complexity, in connectionist networks.

Acknowledgment: I wish to thank the two anonymous reviewers for their detailed critiques. This work is supported in part by the RGC grant #1640 from the Research Grant Committee, the University of Alabama.

1 Overview

This paper aims to deal with issues in implementing complex symbolic schema structures in connectionist networks. The advantages of connectionist models (especially the simple PDP style models; McClelland & Rumelhart [11]) compared with traditional purely symbolic models (e.g., LISP programs) have been demonstrated in many areas and they are therefore extremely appealing as an alternative paradigm for cognitive modeling. On the other hand, however, the *necessity* of having complex symbolic structures in cognitive processing have been argued for by many leading researchers (e.g., Fodor & Pylyshyn [4], Pinker & Prince [10], Arbib [1]; see also [19]). The basis of the arguments ranges from domain specific theories (e.g. [10]), to philosophical beliefs in the distinction of the conceptual level and the subconceptual level (e.g. [24]). The tension between the two approaches, one emphasizing parallel distributed network models and the other emphasizing centralized sequential symbolic manipulations, highlights the need to integrate the two approaches, in order to fully utilize their respective capabilities and potentials.

One possible way of integrating the two approaches is to "implement" symbolic structures (including schemas) in connectionist models; such an integration is not necessarily a strict implementation, but can be an incorporation of symbolic schemas and structures into parallel distributed network models for enhancing their processing capabilities. There are undoubtedly other ways for the integration of the symbolic and the subsymbolic, e.g., coupling separate symbolic and connectionist components, or using connectionist models as attached procedures in a symbolic system (see Sun & Bookman [20] for an overview). However, this way of "implementation" stands out as a promising approach in that it employs unified and principled solutions (i.e., based entirely on the connectionist paradigm) to the problem, which, being more minimalistic, fares better under Occam's razor.

Let us consider schemas. In the vast literature of philosophy, psychology, and AI over a long time in history, the word "schema" has been used to refer to many different things. From Kant's idea of schemas as innate frameworks through which one views the world, to Piaget's use of the word that includes various kinds of conceptual structures that can be constructed developmentally (Inhelder & Piaget [5]), to AI knowledge representation systems, which includes frames, scripts, semantic networks, and even production systems, schemas have played important roles in understanding mental processes and intelligent behaviors.

Let us look into some different forms of schemas. A *frame* (see, e.g., Minsky [9]) represents a concept in terms of its various attributes, or *slots*, each of which has a name (label) and can have a value (a filler). By organizing knowledge in frames, the access is made easier, in that all the relevant pieces of information regarding a concept are grouped together and the access can be made in a more structured way. With frames, it is also easy to add or remove relevant information about a particular situation, object, or event. For example, a kitchen frame may contain all the relevant information as in Figure 1.

A *semantic network* (see, e.g., Quillian [25]), on the other hand, consists a set of nodes, each of which represents a particular primitive concept, and labeled links between nodes, each of which represents a particular relation between nodes. Semantic networks also allow efficient access of knowledge by utilizing links that connect related concepts. As a simplified example, to represent "John gave Mary a book", we can build a semantic network as shown in Figure 2.

So the word "schema" in actuality encompasses various radically different forms of representation, not mentioning the radically differing contents represented in these forms. Therefore, in

Frame: kitchen	
SLOT	VALUE
refrigerator	(3, 5)
dishwasher	(5, 5)
stove	(4, 6)
pantry	(4, 6)

Figure 1: An Example of a Frame

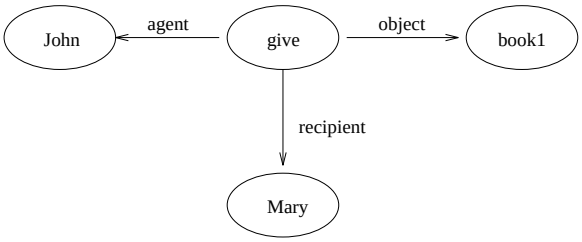


Figure 2: An Example of a Semantic Network

implementing schemas (in connectionist networks or any other media for representation), it is necessary to have a thorough investigation and to start with something simple, and incrementally add more and more complex structures into such implementations of schemas; this is especially important for implementation in a highly restrictive medium such as connectionist models.

There are many important considerations and ideas that need to be taken into consideration. For one thing, in implementations, we need not only strive for new techniques for representing various types of schema structures in a network fashion, but also for possibilities of utilizing useful characteristics of connectionist models, such as fault tolerance and partial matching, for the benefit of more powerful and more robust cognitive models. Other considerations that are relevant in this work include flexibility in representation, expressive adequacy, space and time complexity, and economy of representation.

The plan for the rest of this paper is as below. In Section 2, I will first discuss abstractly issues in implementing schemas and propose assemblies as basic connectionist constructs for implementing schemas. In Section 3, I will present formal details of assemblies, and show how they can be mapped into conventional neural network models. In Section 4, detailed exploration of implementational issues as outlined in Section 2 will follow, utilizing assembly structures. In Section 5, some remarks will complete the paper.

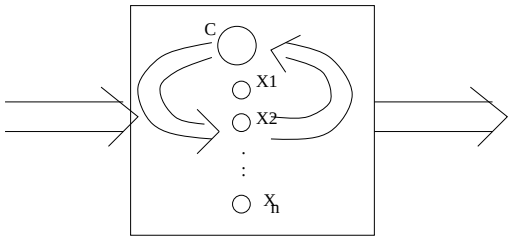


Figure 3: The General Idea of An Assembly

There are inputs and outputs to and from the assembly, There are also communications between C and $X's$.

2 Connectionist Implementation of Schemas

2.1 Basic Connectionist Structures: Assemblies

To map symbolic schema structures, as a basic-level organization in cognitive theories, into connectionist networks, we need to establish some corresponding organization in connectionist networks. The development of such structures has to be accomplished without violating the basic precepts and the fundamental characteristics of the connectionist approach. In other words, symbolic structures must be *smoothly* integrated into connectionist networks. The following requirements need to be satisfied: (1) uniformity and regularity of network layouts can be ensured (to a certain degree at least) while structures are built into (or emerged from) connectionist models, and (2) simplicity of individual nodes and links can be maintained while complex aggregate information is processed. These requirements constitute the design constraints.

In this work, as a basic-level organization, a discrete neural *assembly* [15] is adopted, which is a collection of nodes closely interconnected in various specific ways for encoding and processing a simple structured piece of aggregate information. Figure 3 shows a highly schematic diagram of an assembly: it consists of a set of complex nodes (i.e., DN nodes), each of which is for handling a certain aspect of an information aggregate (e.g. a schema or a symbolic structure of some other kind). A usual way of encoding information is for one of the nodes (denoted as C) to represent the overall value or confidence of the particular information aggregate which the assembly represents; each of the remaining nodes (denoted as X's) represents a concrete aspect of the aggregate, for example, a slot of a frame (in the sense of Minsky [5]), or an argument to a predicate (in case of first-order predicate calculus).

The internal structure of an assembly must ensure the consistency in processing the information aggregate (i.e., the schemas structure). For example, if a slot of a frame receives a value that violates certain built-in constraints, then the frame should not be activated, or at least, the confidence for that frame should be very low. As another example, if an argument variable (in first-order predicate calculus) is not instantiated, then the predicate containing the argument should not be allowed to activate (provided that variable is required to be instantiated as in most cases). Likewise, if a predicate is not invoked (or should not be invoked), even if some of its arguments are sent right values, they should not be allowed to accept, process, or propagate these values. The internal structure within an assembly need to deal with these and many more issues, to guarantee the proper handling of schemas and other symbolic structures.

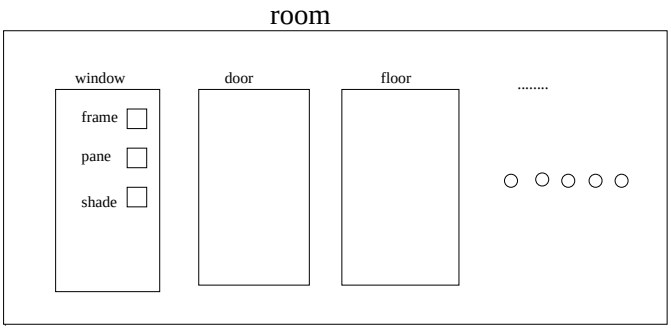


Figure 4: Nested Schemas

2.2 Superstructures and Substructures

Schemas can be complex, and examples of nested or otherwise complex schemas abound in the AI and psychology literature. To allow for nested schemas in connectionist networks, nested assemblies are necessary; that is, there is a need to have *subassemblies* nested within assemblies, and sub-subassemblies nested within subassemblies, and so on. For example, suppose we need to represent a two-level nested frame: $room = (window, door, floor, ceiling, ...)$, in which $window = (sill, frame, pane, shade, ...)$. See Figure 4. Embedded frames can be dealt with by nested subassemblies: we need to have an assembly for *room*, in which subassemblies are constructed each of which is for one embedded structure, such as *window*.

Besides such structures within assemblies, we may also need structures across multiple assemblies, (1) as a means of activating a schema which an assembly represents based on the propagation of activation from another schema (represented by another assembly), or (2) as a means of coordinating two schemas represented by two separate assemblies in ways of (a) *co-occurrence* (being activated simultaneously), (b) *mutual exclusion* (never being activated at the same time), and (c) *sequential activation* (one being activated after another in a fixed temporal order). Many situations require such coordination: co-occurring events need to be represented by the co-occurrence of schemas, as in case of the following two instances of schemas: *adjacent-to(area1, area2)* and *adjacent-to(area2, area1)*. Non-compatible events need to be handled by the mutual exclusion of relevant schemas, as in case of the two schemas: *shape-square(obj)* vs. *shape-circle(obj)*. Representing a sequence of events, such as those in Schank's scripts (Schank [12]) or in HOPE (Arbib [1]), requires sequential activation. Moreover, the coordination and activation may need to be accomplished in ways other than embedding schemas to be coordinated within one another, for the sake of economy of representation since a schemas may have to be connected to and coordinated with multiple other schemas and we only want to have one representation.

Yet another simple and common kind of superstructures that cross assemblies is as follows: suppose there is a rule: "if an area grows something, it is an agricultural area." It can be expressed as:

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

We may need to represent the schema concerning growing things, $grow(x, y)$, as an assembly and represent $agr(x)$ as another separate assembly. Thus, links connecting the two assemblies, which form a superstructure, are needed in order to pass activation from one assembly to another for carrying out an application of the rule in question. See Figure 5. There are some complicated

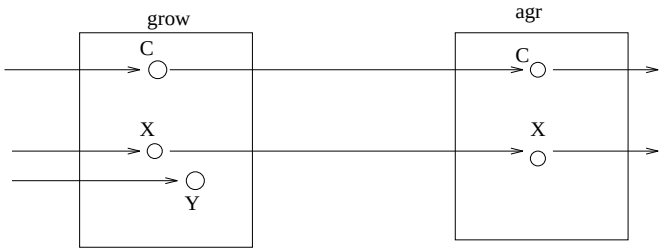


Figure 5: The Network for Representing [if *grow(x,y)* then *agr(x)*]
where corresponding nodes are linked across assemblies, and activation values are propagated along each link.

issues involved in linking together assemblies (i.e. imposing superstructures on top of assemblies). One is how to pass on the overall value (or confidence) of an assembly to other assemblies. Another issue is how to pass on bindings of variables, such as x in the $grow(x, y)$ assembly, to the other assemblies to which this assembly is connected.

2.3 Logics and Variables

Such connectionist structures should, as a minimum requirement, be able to represent some simple logic calculus and be able to handle variables and their bindings during a reasoning process (i.e. first-order logic), all of which are necessary in modeling human cognition. Inhelder & Piaget [5], for example, have argued for the importance of logical capabilities in a schema-based understanding of human cognition. It is also evident from existing models in this area that variables and their dynamic bindings are necessary and are crucial in achieving adequate expressive and computational power. Moreover, as has been argued elsewhere, it is widely believed that various schema structures are describable with logics and implementable in a logical framework (cf. [19]). Therefore, the importance of logics with regard to schemas is clear.

Issues involved in implementing logics in relation to schemas include how to represent arguments of a predicate, how to represent a particular binding (e.g., a constant), how to represent dynamic (and changing) bindings, how to propagate dynamic bindings (as mentioned before), and how to enforce the consistency within a predicate (regarding its arguments) to establish a coherent basic-level unit in processing.

Moreover, there is also the question of how to perform fuzzy inference in such models. Fuzzy inference can be viewed as a generalization of binary logic, in that it reasons with continuous analog values instead of simple binary true/false values. Thus it can handle graded reasoning situations (involving graded concepts such as "tall", "rather tall", "very tall", etc.) that cannot be handled by binary logic. Fuzzy inference constitutes an alternative, more general kind of processing for schema-based theories and systems, including connectionist implementations of schemas, and thus needs to be considered.

3 Details of Assemblies

3.1 An Automata-Theoretical Description

In order to implement various structures in assemblies as discussed above, we need some formal treatments that can qualify the idea of assemblies more clearly. Thus I will introduce here a very general formalism for connectionist models, the *Discrete Neuronal Model*, or the DN model. [1] It is a generalization of simple conventional connectionist models (such as weighted-sum neural networks). It can deal with variable binding, structural 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 formalism forms a unifying framework for developing complex reasoning systems with schemas.

An automaton-theoretic description of a DN node is:

$$\text{N}=< S, A, B, I, O, U, V >$$

where

S= the set of all the possible states of a node,
O= the output-line 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).

One can think of a DN node as having a set of input lines and a set of output lines (see Figure 6):

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

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

where the value of each element of I, I_i, is in B, and the value of each element of O, O_i, is in A. Therefore, the value of I is in $B \times B .... \times B = \Pi_n B$, and the value of O is in $\Pi_m A$. The state transition and action functions are [2]

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

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

DN assemblies can be easily defined based on DN nodes:

$$as = (C, X_1, X_2, ..., X_k)$$

where k is an arbitrary integer; C and X's are DN nodes. Therefore, 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 for

[1] This formalism is first proposed in Sun [14] and later further developed in [15, 17] and in Sun & Waltz [21].

[2] 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.

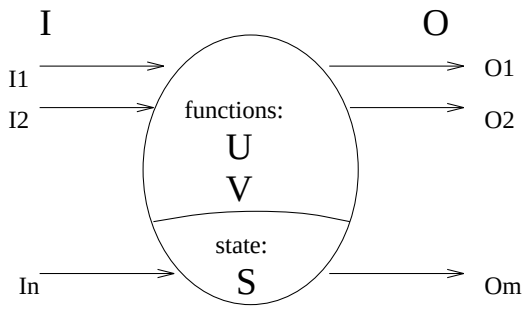


Figure 6: A DN node

the schema represented by the assembly and is connected to other nodes (mainly C nodes) in other assemblies representing different schemas that are related to it somehow. The X nodes take care of detailed aspects of a schema (such as various slots and arguments). There may also be connections from each X to C and from C to each X. Each node in an assembly, C or X's, has a specific DN state transition function and an action function.

On top of assembly structures explained above, superstructures can be established that link various assemblies in accordance with their relations to each other:

$$NET = (AS, L)$$

where

$$AS = \{as\}$$

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

and

$$L \subset \{(as_1, as_2, i, j) \mid as_1, as_2 \in AS, \quad i, j \in [0, l] \text{ are labels}$$

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

In other words, a network consists of a set of assemblies, and the connections among them are specified in L by labels that indicate a particular DN node in one assembly is connected to another particular DN node in another assembly.

3.2 A Neural Network Implementation

The question is: How can we justify such complex and general types of nodes and assemblies as in the DN formalism, since they seem to violate the connectionist principle of allowing only simple processing elements and simple messages? The answer is simple: DN nodes can be implemented with the conventional weighted-sum threshold networks or the sigmoidal networks, so that the DN formalism serves as a descriptive tool for the sake of clarity in developing complex network structures.

First of all, to implement a DN node in a network of conventional connectionist nodes, (1) we need to take into consideration of representing input/output values of a DN node in conventional models, (2) we need to address the question of how to carry out input/output mapping (i.e., the action function of a DN node), and (3) we need to address the questions of how to represent states in the DN formalism and then how to carry out the state transition function of a DN node. In

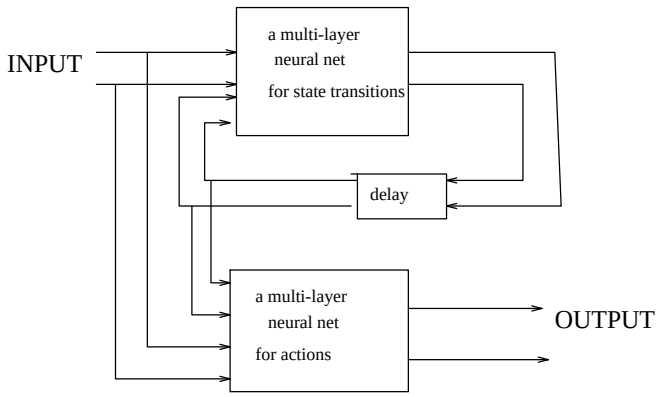


Figure 7: Implementing a DN node in Conventional Connectionist Networks

terms of the first question, we can find a mapping between an input/output value of a DN node and an activation vector of a set of simple conventional nodes, and we can use a set of simple conventional nodes (a layer of them) for each input/output line of a DN node. Each input/output value is now represented by an activation vector and the activation vector propagates through a group of links associated with these nodes. See Figure 7. By the same token, we can also represent a state with a set of simple conventional nodes (which form a hidden layer) and each state value corresponds to an activation vector of these nodes. To carry out the action function and state transition function, it has been shown that a multi-layer recurrent connectionist network (of conventional nodes) can do the job: it maps the current state and current inputs to a new state (represented by another layer of nodes) and outputs (represented by yet another layer of nodes). The new state is fedback, through some delay elements, as the new current state to the network in the next cycle, which helps to decide, along with new inputs, which state to enter next and what to output next. Backpropagation training can be used to set up such a network.

A network in the DN formalism consists of a set of interconnected assemblies, each of which is made up of DN nodes. Now, if we turn each of these DN nodes into a small network of simple conventional connectionist nodes (using only weighted-sum or sigmoid functions) and if we represent each link with a corresponding vector of links, we will wind up having a modularly distributed conventional connectionist model that are equivalent to the original network. For example, we have the following set of relations:

$$give(x,\ y,\ z) \longrightarrow own(y,\ z)$$
$$buy(x,\ y) \longrightarrow own(x,\ y)$$
$$own(x,\ y) \longrightarrow can\text{-}sell(x,\ y)$$

We can use an assembly for each schema (such as *give* and *buy*), and within each assembly each DN node can be implemented with a network of simple conventional nodes. Figure 8 shows a sketch of the resulting network.

Since this implementation of the DN nodes and networks involves distributed patterns, we can term it *the modularly distributed method*. There are obvious advantages in using this method, instead of globally distributed representation: (1) because distributed representation is used within each individual module, fault tolerance and partial matching capabilities of input, output, or state

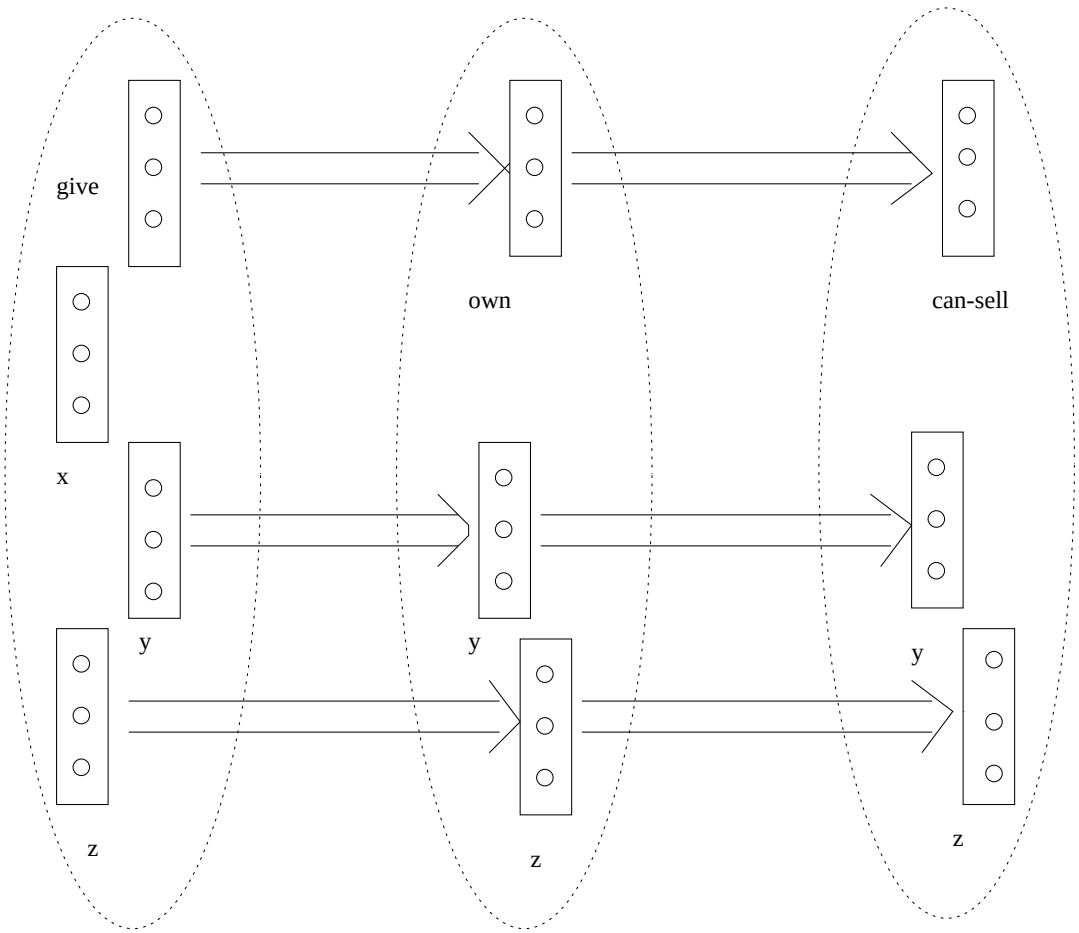


Figure 8: A Modularly Distributed Conventional Neural Network
where each box represents a small network implementing a DN node and each oval represents an assembly.

values are enabled; but (2) because the distributed representation is modularized, representations of different aspects of a DN node are isolated within their individual representational spaces (a particular set of nodes) without interference with each other. At a higher level, each DN node in an assembly is also represented distributively but separately, so that different aspects of a schema (represented by different DN nodes in an assembly) are also isolated within their individual representational spaces without interference with each other. At an even higher level, each assembly (for representing a complete schema) is represented distributively but separately, so that different schemas will not be mixed up and will not interfere with each other.

In sum, although the DN formalism is general and complex, it can be easily mapped into conventional neural network models. and therefore, this formalism should be viewed as a descriptive tool and/or a notational convention for presenting and explaining complex (conventional) network structures, which otherwise will be difficult to describe. I will only use the DN formalism thereafter, but it should be kept in mind that it is equivalent to conventional networks. It should also be mentioned that this formalism is aimed for generality beyond this work [14, 19], and the

actual mechanism as developed in this paper is more tractable.

4 Dealing with the Issues

4.1 Dealing with Super/Substructures

First of all, we need to deal with substructures in schemas. To do so, we can use *subassemblies* in connectionist implementations. Subassemblies can be defined by revising slightly the definition of assemblies as presented earlier: instead of viewing $X_i's$ as nodes, we can view them as subassemblies; that is,

$$X_i =_{\text{def}} (D, Y_1, Y_2, ..., Y_{t_i})$$

where $i \in [1, k]$. Notice that each component of an assembly can now be an assembly itself (i.e., a subassembly). As a degenerate case, a subassembly can contain a single node, in which case this structure collapses back into a simple assembly. The connectivity pattern in an assembly basically remains the same: All the Y nodes (corresponding to the X nodes in the original definition) will communicate with the D node the same way as the X nodes communicate with the C node as before in simple assemblies. The D nodes will communicate with the C node. k is the number of subassemblies. t_i is the number of Y nodes in a subassembly. There also need to be other connections between a subassembly and the overall structure of the assembly (especially the C node), for various communication needs as will be explored later.

Beside this simple one-level embedding, multiple levels of embedding are also permissible. Multiple levels of subassemblies can be constructed the same way as shown above for one level of embedding, by replacing an individual DN node in the above definition with a (sub)assembly structure and linking it up as appropriate.

Secondly, in addition to structures within assemblies, we may also need structures across multiple assemblies, namely superstructures, as discussed in Section 2.2. Let us look into a few cases. Coordinating two schemas represented by two separate assemblies can be accomplished by some simple superstructures as follows. The co-occurrence of schemas can be handled by two one-directional links, or one bi-directional link, connecting the two assemblies, mostly in their C nodes. Each such link can be assigned appropriate weight, e.g. the positive full weight 1, so that as soon as one of the assemblies is activated, the other will be activated immediately due to the activation flowing in from one of the two links, and vice versa. The negative weights need to be strong enough to overwrite other possible inputs to each of the two assemblies, if necessary. See Figure 9 for a diagram.

The mutual exclusion of two schemas can be handled by two negative links with appropriate weights, e.g. the negative full weight -1, connecting the two assemblies in both directions. The weights, in this case, should also be strong enough to overwrite other inputs. See Figure 10 for a diagram. Due to this mutual inhibitory connection, there can be at most one assembly being activated at one time.

The temporal coordination as discussed in Section 2.2 can also be accomplished, by one-directional chaining links from the first schema to the second, from the second to the third, etc. in a sequential order, with delay elements on each link (which can be easily accomplished with a DN node [14]). Such links can control the temporal order and sequentiality. See Figure 11.

All of these above superstructures can be described entirely with the DN formalism. The following subsections will provide further details regarding variables and other superstructures in

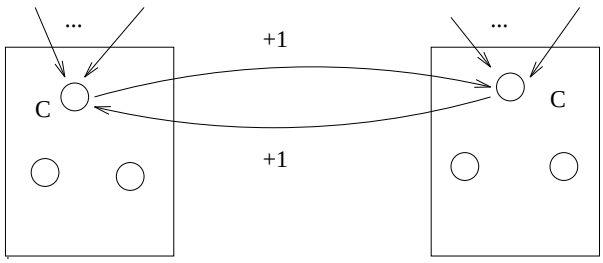


Figure 9: Superstructures for Co-occurring Schemas Implemented in Assemblies
where one assembly represents *adjacent(area1, area2)* and the other *adjacent(area2, area1)*

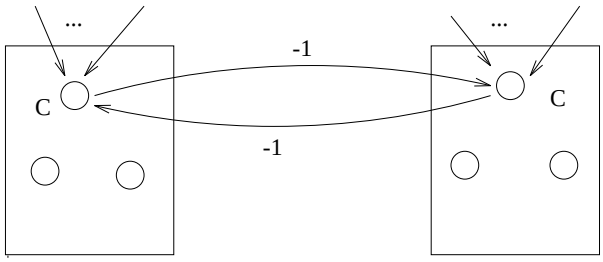


Figure 10: Superstructures for Mutually Exclusive Schemas
where one assembly represents *shape-square(obj1)* and the other *shape-circle(obj1)*

the form of rules.

4.2 Dealing with Logical Inference

First of all, simple forms of rule-based reasoning as often seen in AI systems are necessary for accurately capturing some of the cognitive processes or intelligent behaviors. Therefore, they need to be implemented in connectionist schema-based systems. One such simple form of rule-based reasoning is Horn clause logic, which consists of simple propositions such as p and implications such as $p_1 p_2 ... p_n \longrightarrow q$. Let us look into the propositional form first. We can show, through formal proofs, that simple weighted-sum threshold network computation. As shown in Figure 12, we can specify a special case of the weighted-sum threshold connectionist model, in which values associated with facts are binary, each weight is positive, total weights of each rule sum to 1, and all thresholds are set to 1. In such a case, the model can be shown to be sound and complete with respect to Horn clause logic. Here *soundness* means that whatever can be inferred in the

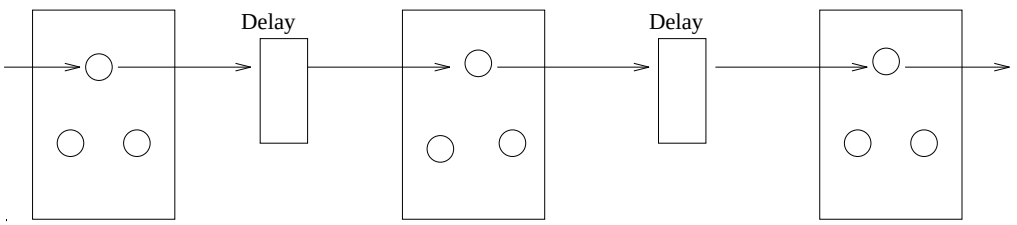


Figure 11: Superstructures for Sequential Schemas

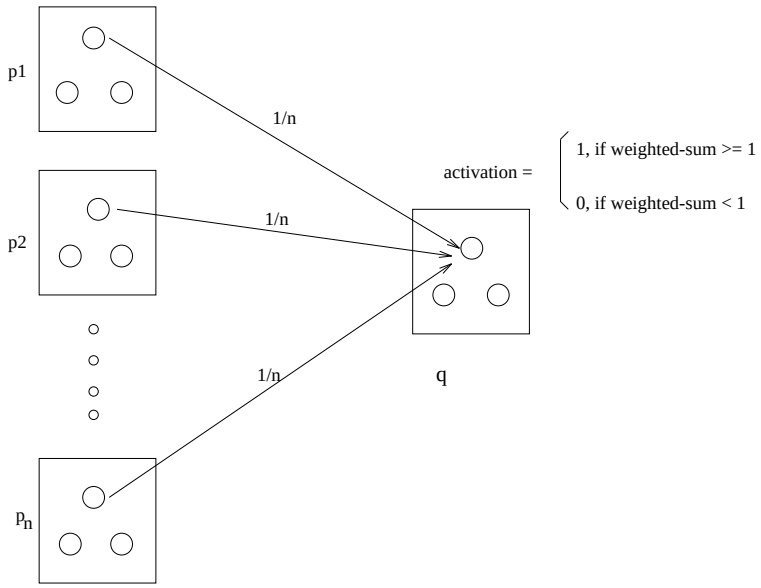


Figure 12: Implementing a Horn Clause Rule

For example, when $n = 3$, p_1 = own-book, p_2 = no-need-for-book, p_3 = book-has-value, and q = sell-book. Then the rule is: if one owns a book, has no need for the book, and the book has some value, then one can sell the book. All the weights are $\frac{1}{3}$.

connectionist implementation can be inferred in Horn clause logic; *completeness* means whatever can be inferred in Horn clause logic can be inferred in the connectionist implementation. The basic idea is simple: since we make the threshold of every node to be 1, and assign the same weight to each fact in the condition of a rule that equals 1 divided by the number of facts in the condition (so the total weights into the node representing the conclusion sum to 1), the fact in the conclusion of a rule is activated if and only if all of the facts in the condition of the rule is activated fully, because of the weighted-sum computation. The activation of the conclusion is either one or zero. So whatever is activated in this implementation is true in corresponding Horn clause logic, and vice versa. (For details of this implementation and the proof of soundness and completeness, see [18].) Since the weighted-sum model can be viewed as a special case of the DN model, it is clear that DN networks can carry out Horn clause logic.

Some popular paradigms of AI, which can be viewed as different instances of schemas such as production systems and semantic networks, can be expressed in Horn clause logic and are thus representable within connectionist models, especially with connectionist implementations of schemas.

Secondly, as an extension to this implementation, fuzzy inference that takes into account *cumulative evidentiality* can also be implemented with this simple weighted-sum computation (as a simple instance of the above DN formalism). Reasoning, especially in commonsense domains, is inexact (graded): fuzzy, uncertain, and imprecise. The inexactness of various sorts can all be represented by a numerical value, which measures the confidence one has about a particular fact as an instance of a concept (Zadeh [22]). In other words, it measures the "distance" or the "goodness of fit" of a fact to a concept (Sun [19]). There are also indications that reasoning is mostly evidential and evidential combination processes are often cumulative; or in other words,

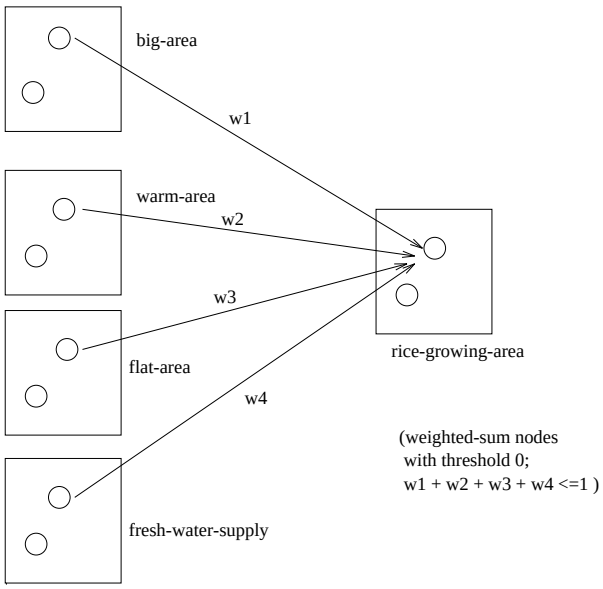


Figure 13: Implementing a Fuzzy Inference Rule

they tend to "add up" various pieces of evidence to reach a conclusion, with a confidence that is determined from the "sum" of the confidence values of the different pieces of evidence. Knowing two conditions in a rule results in a larger confidence than knowing only one. For example, suppose there is a rule (see Sun [16]):

$$\textit{big-area warm-area flat-area fresh-water-supply} \longrightarrow \textit{rice-growing-area}$$

If we know all of the four conditions, warm, flat, big and fresh water supply, we can draw the conclusion with full confidence; when we know only three out of the four conditions, we reach the same conclusion with less confidence. A cumulative evidential combination procedure is therefore necessary in reasoning with schemas (especially when implemented in connectionist models).

Fortunately, the generic weighted-sum computation of connectionist models provides exactly what we need in an evidential combination procedure: numerical gradedness and cumulative evidentiality [18]. In the previous example, we assign (or use learning algorithms to acquire) numerical, graded weights each of which reflects the relative importance of the corresponding condition in determining the conclusion from a rule (along with numerous other relevant or irrelevant conditions and rules). The sum of all the weights in a rule is less than or equal to 1, depending on the confidence in the rule. We set thresholds to be 0 and use graded activations. Thus, with the weights given, if we know all the four conditions, we can draw the conclusion with full confidence; that is, the weighted-sum is maximized. When we know only three out of the four conditions (or less), we reach the same conclusion with less confidence; that is, the weighted-sum is smaller, because some conditions (involved in the weighted-sum) have 0 or very small values (confidence values, or in terms of connectionist models, activation values). See Figure 13.

4.3 Dealing with Variable Binding

To add variables, we have to address the question of how to represent variables in connectionist networks, not just for logical inference but also for generic schema structures. There are several existing ways of representing variables in connectionist networks: (1) using a separate node for each variable, which can take on different values for representing different bindings (e.g., Lange & Dyer [6], Sun [14]), or (2) using the temporal activation patterns (or simply phase synchronization) for representing bindings for different variables (e.g., Shastri & Ajjanagadde [13]). We will adopt the first approach here for its simplicity and its intuitive appeal; for a detailed comparison of various approaches, see [17]. In this approach, each variable (such as an argument in a predicate or a slot in a frame) is handled by one node. A variable can become bound from the working of a superstructure that connects the assemblies containing the variable; for example, an application of a relevant rule that has a variable in its conclusion will result in the variable get bound by a value. A binding is represented by a numerical activation value standing for a particular constant; the value can be passed along a link connecting two assemblies, from a node representing a variable in the source assembly to a node in the target assembly representing a variable having the same name.

For example, suppose a rule is:

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

The input can be $grow(a, b)$ (meaning *area* a grows b), where a and b are constants (e.g., a=*Florida* and b=*orange*). The binding a should be passed to the node representing x in grow(x,y), and b to the node representing y in $grow(x, y)$. Then by applying the rule, a should be further passed from the node representing x in $grow(x, y)$ to the node representing x in $agr(x)$.

Let us see how this can be done with DN assemblies. Suppose there are k variables; an assembly then contains k+1 nodes, each of which except one (the C node) is for storing and passing along bindings of the variable which it represents (called the X nodes). Among these nodes, the C node of an assembly receives inputs from C nodes in other assemblies that are conditions of a rule that use this assembly for its conclusion. Each variable (X) node receives inputs from variable (X) nodes in the same set of assemblies as its corresponding C node, picks up its own binding, and passes it on (see Figure 3). As an example, for representing $grow(x, y) \longrightarrow agr(x)$, there need to be 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 5). First the C node in the first assembly is activated (in response to the input $grow(a, b)$), and the binding a is passed to the X node in the assembly $grow(x, y)$, and b to the Y node in the same assembly. Then by applying the rule (i.e., propagating activation along the link from one assembly to the other), the C node in the second assembly is activated, and a is passed to the X node in $agr(x)$. Thus a conclusion is reached: $agr(a)$.

In addition, each of the X nodes in an assembly should also communicate with the C node to send its bindings, so that the C node can check consistency and others matters, and should receive "instructions" from the C node; the "instructions" are to tell an X node whether to accept bindings or not and which binding to accept in case there are multiple inputs for an X node. These issues will be discussed next.

4.4 Further Issues in Variable Binding

There are many more issues involved in variable binding in logical inference; below I will discuss in some detail a number of these issues.

4.4.1 Free Variables

Let me first clarify the notion of a binding. In the DN formalism, an X node in an assembly can receive and pass on a value which represents a constant (i.e., an object in the domain). Equivalently, in terms of the distributed implementation, an X "node", which is a network by itself, can also receive and pass on a binding, which is an activation vector in this case. Such an X node (either as a DN node or as a subnetwork of the conventional type) does not necessarily contain a particular value (or vector) that represents a constant or a particular domain object: in other words, it can represent a free variable. For example, if we have a rule:

$$p(x) \longrightarrow q(x)$$

given $p(z)$, where z is a variable, we can derive $q(z)$; we can do this by representing z with a special value in the X nodes of the assemblies for representing $p(x)$ and $q(x)$. In order to distinguish a constant (a real binding) from a free variable (*a pseudo-binding*), free variables are represented by values (or vectors) different from a real binding. In the DN formalism, they are greater than a certain threshold θ, so a simple test, such as "if $v > \theta$", suffices to distinguish them. In the corresponding modularly distributed implementation, they are vectors that have some particular values in some elements at the beginning of a vector, such as (1 1 1 v_1 v_2 .... v_n) for pseudo-bindings and (0 0 0 v_1 v_2 .... v_n) for real bindings, so that a similarly simple test can distinguish them. It is easy to see that the two methods are equivalent to each other: if you convert a vector to a value by concatenating the elements of the vector, then the test used for the DN model (with a threshold value) can be used for the converted vector, and vice versa.

4.4.2 Binding Generation

Instead of simply passing bindings on, an X node (e.g., representing an argument of a predicate) may need to generate bindings. This situation can be illustrated by the following example:

$$p(x_1) \longrightarrow q(x_1, x_2, c)$$

Here the value for x_1 is simply passed from p to q, but two other values need to be generated in the q assembly to cover the last two arguments: x_2 and c. Note that c is a constant, but x_2 is a free variable and needs to be taken care of differently. In either cases, we need to produce a binding that was not there before the assembly is activated. Let us look into the two cases: (1) the value generated for c is a genuine binding, but (2) the value generated for x_2 simply denotes a possible binding or a pseudo-binding. In the case of (1), the value for c should have been preset in the corresponding argument node in the assembly for $q(x_1, x_2, c)$. In the case of (2), we need to make a few assumptions in order to preset a value as in (1). In the same way as usually done in Horn clause logic, a universal quantifier can be assumed over x_2. And in the same way as in performing resolution theorem proving for Horn clause logic [23], we can assume that different rules use different variable names, before being used to construct a network, so that there will

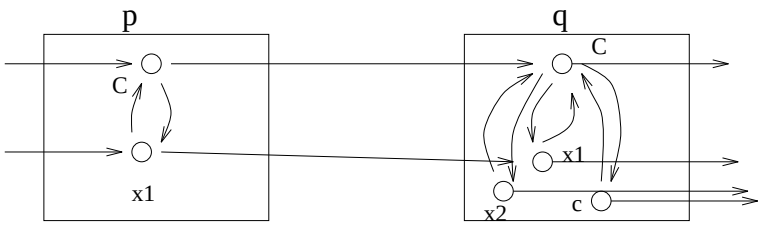


Figure 14: Binding Generation
where C denotes a predicate node and X denotes a argument node.

be no naming conflict. Based on these assumptions, in constructing the network, a unique value representing the pseudo-binding of x_2 (which must be above a certain threshold) can be preset in the second argument node of the assembly representing $q(x_1, x_2, c)$.

So, the two cases can be taken care of as follows (see Figure 14):

> In the X node of the first type, if the input from the C node is not zero, then output a particular value 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 a particular value v (where $v > \theta$), otherwise output nothing ($\perp$).

The functions used for the C node in the q assembly are as follows (where C denotes the predicate):

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

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

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

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

where $O^1_{C_p}$ is the input from the C node in p, and W is the weight between the C node in p and its counterpart in q. O^1_C denotes the inter-assembly output of C and O^2_C denotes the intra-assembly output of C, or in other words, a signal to $X's$ to enable their processing. (From now on, I will always use superscript 1 to indicate inter-assembly communication and 2 to indicate intra-assembly communication.) The functions for the three X nodes carry out the above two kinds of generation:

$$O^1_{X_1} = V^1_{X_1}(O^1_{X1_p}, O^2_C) = O^1_{X1_p} \text{ if } O^2_C = 1, \ \perp \textit{ otherwise}$$

$$O^2_{X_1} = V^2_{X_1}(O^1_{X1_p}) = O^1_{X1_p}$$

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

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

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

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

where $O^1_{X1_p}$ is the input from the X node in p. The three nodes perform the respective binding properly as described before. $O^1_{X_i}$ denotes the inter-assembly output of X_i and $O^2_{X_i}$ denotes the intra-assembly output of X_i, or in other words, a message to C for its checking. The overall functionality is accomplished through the communication and cooperation between C and $X's$.

4.4.3 Consistency Checking

In dealing with multiple bindings at the same time, we may need to check the consistency of bindings, since oftentimes certain constraints are imposed regarding bindings of different variables. This is the issue of *consistency checking*. For example, suppose we have the following rule:

$$p(x_1, x_2)\, q(x_2, x_3) \longrightarrow r(x_1)$$

where the constraint is that the x_2 in p is to be bound to the same value as the x_2 in q. Another example is

$$p(x_1, x_2)\ \ x_1 \neq x_2 \longrightarrow r(x_1, x_2)$$

where we have to make sure that the inequality holds.

We need to find efficient ways of dealing with this issue in a connectionist framework. In order to check input consistency, all X nodes can pass on their inputs to a central location, namely the C node, for comparisons. In the r assembly of the example above, after all the X nodes pass their inputs to the C node, it then performs the equality or inequality checking. See figure 15. With the DN formalism, these checks can be easily formalized (and can then be easily converted to the modularly distributed implementation). In the first example, the action functions used for X in the r assembly are:

$$O^1_X = V^1_X(O^1_{X1_p}, O^2_C) = O^1_{X1_p}\ if\ O^2_C = 1,\ \ \perp\ otherwise$$

$$O^2_X = V^2_X(O^1_{X1_p}) = O^1_{X1_p}$$

where $O^1_{X1_p}$ is the input from X_1 in the p assembly. And the action function forms for the C node are:

$$O^1_C = V^1_C(O^1_{C_p}, O^1_{C_q}, O^2_X, O^1_{X2_p}, O^1_{X2_q})$$

$$O^2_C = V^2_C(O^1_{C_p}, O^1_{C_q}, O^2_X, O^1_{X2_p}, O^1_{X2_q})$$

where $O^1_{X2_p}, O^1_{X2_q}$ are inputs from X_2 nodes in assemblies p and q, and $O^1_{C_p}$ and $O^1_{C_q}$ are inputs from the C nodes in p and q respectively; the functions for the C node in the r assembly perform the following mappings (where $I_1 = O^1_{C_p}$ and $I_2 = O^1_{C_q}$):

	O^1_C	O^2_C
$O^1_{X2_p} = O^1_{X2_q} \quad I_1 \neq \perp \quad I_2 \neq \perp$	$\sum W_i I_i$	1
otherwise	$\perp$	$\perp$

These functions together perform consistency checking for this example. The other example above can be dealt with in a similar fashion.

4.4.4 Unification

As in any first-order logic, unification is needed to match terms, that is, the known or derived facts and the conditions of rules. This issue is a natural extension of the issue of consistency checking. I will perform only unifications of variable/variable and variable/constant for now.

Let us examine an example. Suppose there is a rule:

$$p(x, n) \longrightarrow q(x)$$

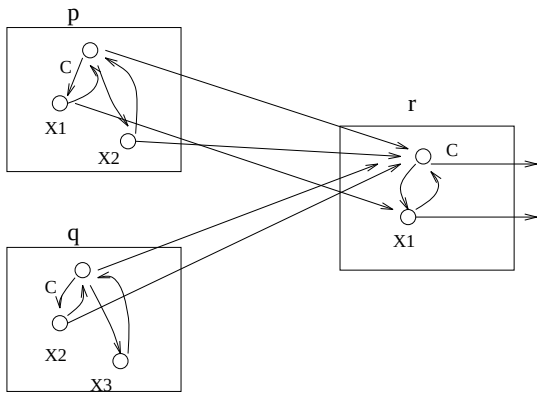


Figure 15: Consistency Checking

where C denotes a predicate node and X denotes a argument node.

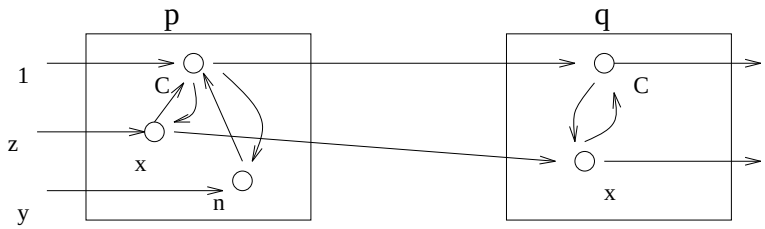


Figure 16: A Simple Unification

where x is a variable and n is a constant. When given an input, say $p(z, y)$, where z and y are both variables, we have to make sure that z is unifiable with x, which is trivial, and y is unifiable with n, which is a real unification. See Figure 16. In order to unify y with n, we can perform the following test: If $y = n$ or y is a variable, then they are unifiable. In general, the rule is:

> To unify x and y, compare x and y. If $x = y$, then they are unifiable. Or, if either x or y is a variable, then they are unifiable.

This rule can be carried out in X nodes involved in unification. In the example above, it can be done in the X_2 node in the p assembly:

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

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

Note that variables are represented by values greater than a certain threshold θ; hence we use the test $y > \theta$.

A mapping table for $O^2_{X_2}$ is as follows:

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

The mapping for $O^1_{X_2}$ is simple and thus will not be tabulated. In this case, other necessary functions (for the C node and X_1) remain the same.

What is especially difficult is the situation in which unification has to be done across predicates. This situations can be illustrated by the following example. Suppose we have a rule:

$$p(x_1, x_2)\, q(x_2, x_3) \longrightarrow r(x_1)$$

Suppose the following conditions are given:

$$p(m, n)$$

and

$$q(x_4, l)$$

where m, n, and l are constants and x_4 is a variable. The network will unify x_1 with m, x_2 (in p) with n, x_2 (in q) with x_4, and x_3 with l. This is done by allowing the input to pass through the nodes representing x_1 and x_2 (in p), and x_2 and x_3 (in q), when a proper signal is received from the C nodes in p and in q, respectively. Now we are in the following state:

$$p(m, n)\, q(x_4, l) \longrightarrow r(x_1)$$

Because the second argument in p and the first argument in q must be the same in the rule above, we should also unify x_4 with n and we have to make sure that x_4 is indeed unifiable with n. The unification can be done within the C node in the r assembly according to the following principle:

If x_2 (of p) = x_2 (of q), or if x_2 (of p) $> \theta$, or if x_2 (of q) $> \theta$, then they are unifiable.

See Figure 17. To carry out the process outlined above, the C node performs the following mappings for unification, with the same equations as in the case of consistency checking:

$I_1 = O^1_{C_p}$	$I_2 = O^1_{C_q}$	O^2_X	$O^1_{X2_p}$	$O^1_{X2_q}$	O^1_C	O^2_C
not $\perp$	not $\perp$	not $\perp$	$< \theta$, $= O^1_{X2_q}$	$< \theta$, $= O^1_{X2_p}$	$\sum W_i I_i$	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					$\perp$	$\perp$

When the unification is done, a message is sent to the X node in the r assembly to enable it to accept input from the X_1 node in the p assembly; otherwise, inputs are deemed inconsistent with the rule and further activations are blocked.

Note that there is no sequential search involved in performing unification here, in contrast to symbolic reasoning systems. Note also that in cases there are multiple occurrences of variables and constants within and/or across predicates, the same approach applies, but the mappings will be more complicated.

4.4.5 Multiple Rules

Another issue that needs to be addressed is how we should deal with the situation in which *multiple rules* that have the same conclusion but different conditions reach their conclusions with different values (confidence levels) and different bindings (for variables in the conclusion). We

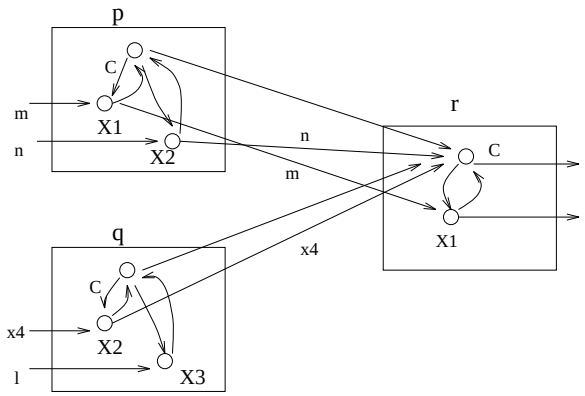


Figure 17: Unification Across Predicates
where C denotes a predicate node and X denotes an argument node.

will look into the first half of the question, that is, how we combine different confidence values of the same conclusion reached from different rules.

Let us look into an example. Suppose we have the following two rules:

$$p(m,x)\, q(n,x) \longrightarrow r(m,n)$$

$$p(m,x)\, s(x,n) \longrightarrow r(m,n)$$

If both rules are activated, two conclusions will be reached; for example, we reach from the first rule $c(m,n)$ with a confidence value 0.9, and from the second rule $c(m,n)$ with a confidence value 0.8. The question is how we can combine them. One simple way is to choose the conclusion with the maximum activation value (or in other words, with the highest confidence level). This method can be more fully justified; see [17] for the detail.

4.4.6 Selection

After dealing with combining values of multiple rules, the issue of *selection* immediately arises; that is, how do we choose a set of bindings and pass them on, when there are multiple sets of bindings available (each of which is derived from a different rule)?

We can choose the set of bindings associated with the winner rule. This can be accomplished with the following steps: (1) the node C (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.

To implement this procedure with the DN formalism, we set up functions for all the DN nodes involved. The mappings for the C node in the r assembly can be easily set up as follows:

$$\begin{aligned} O^1_C &= V^1_C(O^1_{C_p}, O^1_{C_q}, O^1_{C_s}, O^2_{X_1}, O^2_{X_2}) \\ &= max(W^1_1 O^1_{C_p} + W^1_2 O^1_{C_q},\ W^2_1 O^1_{C_p} + W^2_2 O^1_{C_s}) \text{ if } O^2_{X_1} \neq \perp \textit{ and } O^2_{X_2} \neq \perp, \\ &\qquad \text{and the maximum is not } \perp \end{aligned}$$

$\perp$ *otherwise*

$$\begin{aligned} O_C^2 &= V_C^2(O_{C_p}^1, O_{C_q}^1, O_{C_s}^1, O_{X_1}^2, O_{X_2}^2) \\ &= 1 \text{ if } W_1^1 O_{C_p}^1 + W_1^1 O_{C_q}^1 \geq W_1^1 O_{C_p}^1 + W_1^1 O_{C_s}^1 \\ &\quad 2 \text{ if } W_1^1 O_{C_p}^1 + W_1^1 O_{C_q}^1 < W_1^1 O_{C_p}^1 + W_1^1 O_{C_s}^1 \\ &\quad \perp \textit{ otherwise} \end{aligned}$$

where $O_{C_p}^1, O_{C_q}^1, O_{C_s}^1$ are inputs from C nodes in the p, q, and s 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 the argument nodes X_1 and X_2,

$$O_{X_1}^1 = V_{X_1}^1(O_{X1_p}^1, O_C^1)$$

$$= O_{X1_p}^1 \text{ if } O_C^1 = 1 \textit{ or } O_C^1 = 2, \ \perp \textit{otherwise}$$

$$O_{X_1}^2 = V_{X_1}^2(O_{X1_p}^1)$$

$$= 1 \text{ if } O_{X1_p}^1 \neq \perp, \ \perp \textit{otherwise}$$

$$O_{X_2}^1 = V_{X_2}^1(O_{X1_q}^1, O_{X2_s}^1, O_C^1)$$

$$= O_{X1_q}^1 \text{ if } O_C^1 = 1, \ O_{X2_s}^1 \text{ if } O_C^1 = 2, \ \perp \textit{otherwise}$$

$$O_{X_2}^2 = V_{X_2}^2(O_{X1_q}^1, O_{X2_s}^1)$$

$$= 1 \text{ if } O_{X1_q}^1 \neq \perp \textit{ or } O_{X2_s}^1 \neq \perp, \ \perp \textit{otherwise}$$

where $O_{X1_p}^1$, $O_{X1_q}^1$, $O_{X2_q}^1$, and $O_{X2_s}^1$ are inputs from X_1 or X_2 nodes in assemblies p, q, or s, respectively.

4.4.7 Functions

Yet another issue is how to extend variable binding to include *functional terms* in predicate as in standard first-order predicate logic. For example, $living(brother(john), boston)$ is a predicate that contains a functional term — $brother(john)$, which is a way of identifying an (unknown) constant (i.e. John's brother) based on its relation to known constants (i.e. John).

The basic idea for solving this problem is as follows: instead of one node for one argument as before, we can use a set of nodes, which form 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. However, since a parameter of a function can be a function itself, the structure can be arbitrarily large. Nevertheless, in any network implementation, a subassembly can only contain a limited number of nodes. This creates a dilemma: we would like to have as many nodes as possible, but want to limit the

computational resources used. In practice, though, we can always choose a reasonable number of nodes for a subassembly, taking into consideration the particular domain involved.

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. We can adopt the entended definition of assemblies: in place of each X node, there is now a subassembly X, which consists of an F node (for a function symbol) and a number of Y nodes (for function parameters). All Y nodes will send signals (basically 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. There is one interesting special case: 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 such a case, the C node has to be able to distinguish a functional term, a constant, and a variable. We can accomplish this with the following test (extended from the test used before):

> There are two thresholds: θ_1 and θ_2. If $v < \theta_1$, then v is a constant; if $v > \theta_2$, then v is a variable; if $v < \theta_2$ and $v > \theta_1$, 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 can be handled similarly.) Now suppose we want to set up a network connection for a rule:

$$p(x) \longrightarrow q(x)$$

Let us look into the assembly for q. See Figure 18. In the assembly, we can first determine the forms of the output functions for the C node:

$$O^1_C = V^1_C(O^1_{C_a}, O^2_F, O^2_{Y_1}, O^2_{Y_2})$$

$$O^2_C = V^2_C(O^1_{C_p}, O^2_F, O^2_{Y_1}, O^2_{Y_2})$$

where $O^1_{C_p}$ is the output from the C node in the p assembly. Next the functions for F can be determined:

$$O^1_F = V^1_F(O^1_{F_p}, O^2_C) = O^1_{F_p} \text{ if } O^2_C = 1, \ \perp \textit{otherwise}$$

$$O^2_F = V^2_F(O^1_{F_p}) = O^1_{F_p}$$

where $O^1_{F_p}$ is the input from the F node in the p assembly. The functions for Y_1 are

$$O^1_{Y_1} = V^1_{Y_1}(O^1_{Y1_p}, O^2_C) = O^1_{Y1_p} \text{ if } O^2_C = 1, \ \perp \textit{otherwise}$$

$$O^2_{Y_1} = V^2_{Y_1}(O^1_{Y1_p}) = O^1_{Y1_p}$$

where $O^1_{Y1_p}$ is the input from the Y_1 node in the p assembly. The functions for Y_2 are similar and thus omitted.

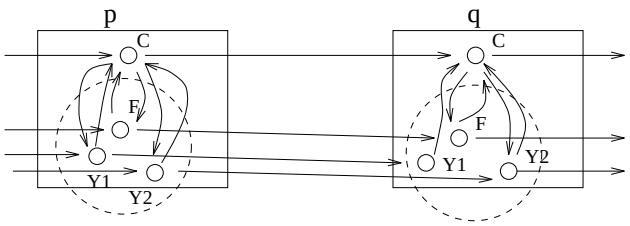


Figure 18: Handling Functional Terms

Now the detailed mapping for the C node can be determined, which basically distinguishes three different cases (a constant, a variable, or a function):

$O^1_{C_p}$	O^2_F	$O^2_{Y_1}$	$O^2_{Y_2}$	O^1_C	O^1_C
not $\perp$	$\geq \theta_1, < \theta_2$	any	any	$W * O^1_{C_p}$	1
not $\perp$	$\geq \theta_2$	$\perp$	$\perp$	$W * O^1_{C_p}$	1
not $\perp$	$< \theta_1$	$\perp$	$\perp$	$W * O^1_{C_p}$	1
	otherwise			$\perp$	$\perp$

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

4.5 Automatic Construction of Networks

With the exception of small and isolated systems, for any reasonably large schema-based system, it is not easy to hand-wire a network. We need an automatic procedure for building up a connectionist network, which implements schema structures and their connections. Let us consider a special case here in which schemas are described by predicates, with arguments/variables for handling individual aspects of schemas, and in which superstructures are handled by rules connecting predicates. We need to build up a network on the basis of a set of first-order predicates (describing schemas) and a set of first-order rules (with proper weights attached) connecting schemas.

CFELDNN (*Compilation of Fuzzy Evidential Logic into DN Networks* [19]) is a procedure for accomplishing this task. In this procedure, for simplifying implementational details, we use uniformly three different types of assemblies for various purposes (see Figure 19) as follows:

- Ordinary assemblies – computing weighted sums of input and passing along bindings for the variables.
- OR assemblies – computing the maximum of input 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, which is left to OR assemblies.)

Note that these types of assemblies can be easily formulated by the DN formalism. Furthermore, the formulated DN nodes can be easily implemented with conventional neural network models, since they are much limited than the general case of DN.

The main procedure for CFELDNN is as follows:

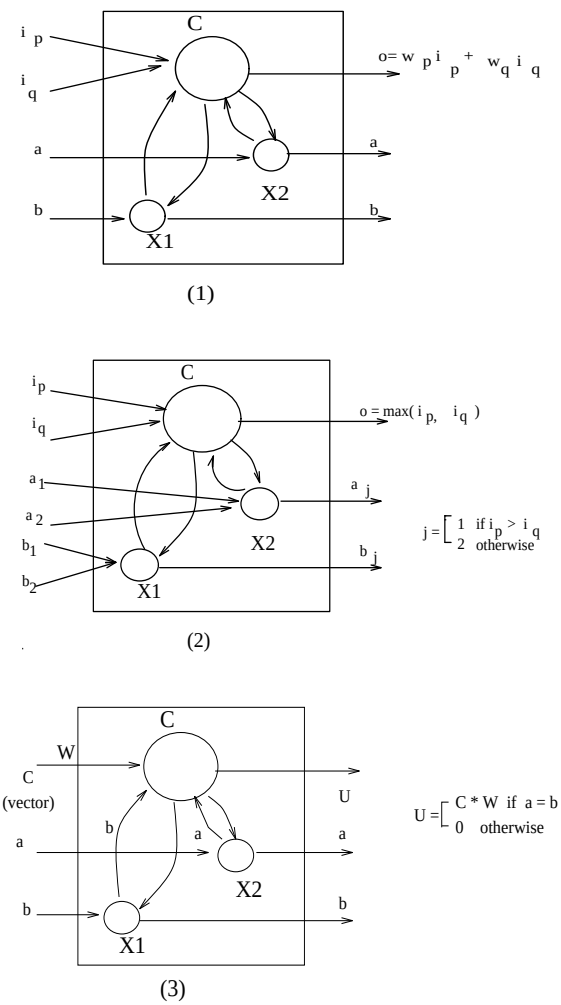


Figure 19: The Three Types of Assemblies
(1) Ordinary assemblies (2) OR assemblies (3) Complex assemblies

SET-UP-RULE-NET:

```
Initialize the rule set to include all the rules to be implemented.
Repeat until the set is empty:
  Choose a rule P1(X1) P2(X2) ..... Pn(Xn) --> Q(X) 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 given weights to the C links.
  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 weight 1 to each C link to the new Q assembly.
  If there are multiple assemblies for a Pi,
     or-link all assemblies for that Pi to an intermediate assembly,
     and then link that intermediate assembly to Q,
     and assign weight 1 to each C link to the intermediate assembly.
  Delete from the set the rule implemented.
```

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, we have to make a separate assembly for that predicate with that particular requirement. "C links" refer to those links that connect the C nodes of different assemblies. Since all the basic ideas have been discussed before, I will not repeat the details of subprocedures for setting up assemblies, including C and X nodes, for linking together assemblies in various ways (especially linking different X nodes), and for dealing with issues identified earlier, such as consistency checking, selection, and unification (see Sun [17] for a complete description).

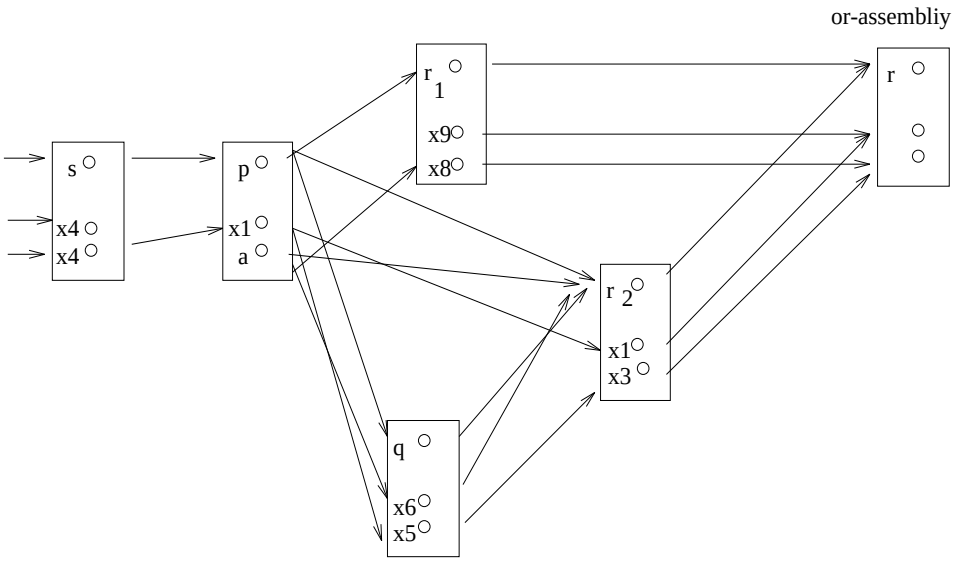
An example is in order here. Suppose we want to implement the following rule set:

```
I)    p(x1, a) q(a, x3) --> r(x1, x3)
II)   s(x4, x4) --> p(x4, a)
III)  p(x5, x6) --> q(x6, x5)
IV)   p(x7, x8) --> r(x9, x8)
```

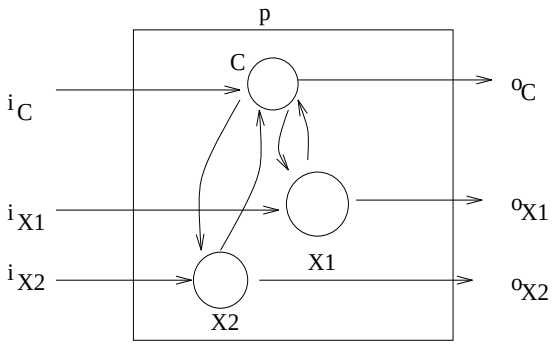
where a is a constant and $x's$ are variables. This set of rules involves many issues discussed above, such as multiple rules having the same conclusion (I and IV), consistency checking (I), and binding generation (IV). The solution obtained according to the procedure is shown in Figure 20.

5 Concluding Remarks

The present work is related to some existing works on schemas, including schema-based cooperative computation models, localist connectionist models of frames and semantic networks, and distributed connectionist models of scripts and natural language understanding. Arbib [1] has long been advocating schema-based cooperative computation systems for AI research; the systems that he and his associates developed utilize a large variety of schema structures. Their



(1) the network



(2) an assembly in the network

Figure 20: An Example Network for a Complex Rule Set
(1) The connections of the network, (2) The diagram of one assembly. The weight assignment and the detailed node functions are not shown there.

systems are mostly implemented in symbolic forms and thus are not unified with the connectionist framework. In order to explore the partial matching and fault tolerance characteristics of connectionist models, it is advantageous to incorporate schemas into connectionist models.

Lange & Dyer [6, 7] present a connectionist system for word sense disambiguation based on frames (a form of schemas, which is implemented in a connectionist fashion). Their frames have slots for variables and thus they handle variable binding with distinct values for different constants (the same as in the present work). However, they do not deal with substructures (including functional terms) and some other issues.

Miikkulainen & Dyer [8] present a connectionist system for reasoning with scripts (another form of schemas). Because of the distributed representation used, their system is capable of utilizing the partial matching capability for completing scripts based on partial information. The system does not handle superstructures and generic logical inference capabilities, though, which are nevertheless necessary in certain reasoning situations.

The advantage of implementing schemas in the way as in the present paper is as follows:

- First of all, compared with purely symbolic systems, this implementation allows reasoning to be carried out in a massively paralel fashion, not only at the implementation level, but also at the knowledge level.

- This implementation adopts a modularly distributed representation (though it can be viewed as localist too), thus it allows certain properties commonly associated with such representation: partial match, pattern completion, and fault tolerance.

- Compared with some other existing connectionist models, the present model allows complex schema structures (including multiple embedded structures and imposed superstructures) to be adequately represented.

- Moreover, it allows a variety of logical reasoning, including with binary rules and fuzzy rules, to be carried out efficiently.

In sum, this paper presents ways of implementing schema structures in connectionist networks and performing logical reasoning, by extending and aggregating basic network constructs. A basic-level organization, namely DN assemblies, is proposed, which consists of DN nodes, and super/substructures are developed on the basis of assemblies. Detailed solutions to various issues involved in this implementation and results concerning logical capabilities of the solutions are sketched out. It is not only possible to incorporate complex schema structures into connectionist networks but there are also some advantages in doing so.

REFERENCES

[1] M. Arbib, (1987). *From schema theory to language.* Oxford U. Press. New York, NY.

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

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

[4] Fodor, J. and Pylyshyn, Z. (1988). Connectionism and Cognitive Architecture: A Critical Analysis. *Connections and Symbols.* Cambridge, MA: MIT Press.

[5] B. Inhelder & J. Piaget, (1958). *The Growth of Logical Thinking from Childhood to Adolescence.* Routledge & Kegan Paul, London, England.

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

[7] Lange, T. and Dyer, M. (1989b). High Level Inferencing in a Connectionist Net work. *Connection Science*, 181-217.

[8] R. Miikkulainen & M. Dyer, (1991). Natural language processing with modular PDP networks and distributed lexicons. *Cognitive Science.* 15(3). pp.343-399

[9] M. Minsky, (1983). A Framework For Representing Knowledge, in: J. Hughland (ed.) *Mind Design.* MIT Press. Cambridge, MA.

[10] Pinker, S. and Prince, A. (1988). On Language and Connectionism. *Connections and Symbols.* Cambridge, MA: MIT Press.

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

[12] R. Schank, (1977). *Scripts, Plans and Goals.* LEA, Hillsdale, NJ.

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

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

[15] Sun, R. (1991) 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.

[16] Sun, R. (1992a) Connectionist Models of Commonsense Reasoning Incorporating Rules and Similarities. *Knowledge Acquisition*, Vol.4. 293-321.

[17] R. Sun, (1992b). On variable binding in connectionist networks. *Connection Science*, Vol.4, No.2, pp.93-124.

[18] R. Sun, (1993a). A Neural Network Model of Causality. *IEEE Transaction on Neural Networks*, in press

[19] R. Sun, (1993b). *Integrating Rules and Connectionism for Robust Reasoning* (A Monograph), John Wiley and Sons, New York, NY. forthcoming.

[20] R. Sun and L. Bookman, (1993). How do symbols and networks fit together? *Artificial Intelligence* magazine, in press.

[21] 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.

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

[23] C. Chang and R. C. Lee, (1973). *Symbolic Logic and Mechanical Theorem Proving*, Academic Press. Reading, MA.

[24] P. Smolensky. (1988). On the proper treatment of connectionism, *Behavioral and Brain Sciences*, 11, 1-43.

[25] M. R. Quillian, (1968). Semantic memory. in: M. Minsky, (ed.) *Semantic Information Processing*. MIT Press.