14

A LOCALIST NETWORK ARCHITECTURE FOR LOGICAL INFERENCE

Nam Seog Park

Dave Robertson

Department of Artificial Intelligence
University of Edinburgh

1 INTRODUCTION

The primary interest of this chapter is in understanding how systems which are described in a symbolic specification style may be implemented using a connectionist architecture. As a focus of attention, techniques have been developed for *compiling* Horn clauses into a connectionist network. This offers significant practical benefits but also forces limitations on the scope of the compiled system. Executable symbolic specifications are (in the right hands) effective for describing systems and are comparatively easy for designers to understand. However, they normally require extra *machinery*, in the form of an interpreter or theorem proving system in order to be executed. For many applications (particularly when the system is to be implemented in hardware) such extra mechanisms are both inefficient and structurally complex. Connectionist systems, on the other hand, use structurally simple components, may provide very fast inference and have no need of a separate interpreter, but are difficult to use directly for specification because of the mass of connections between elements. By providing automatic translation from symbolic to connectionist representations, one should be able to cancel out the deficiencies of each style whilst retaining the advantages of both. Unfortunately, this type of compilation is not straightforward; since an *interpreter* has to be merged into the connectionist networks, the compilation process has to take into account not only the Horn clauses themselves but also the strategy, which is intended to be used for drawing inferences from them. It also appears that some fundamental aspects of symbolic inference are difficult to translate directly into a connectionist framework. In particular, it has proved to be difficult to provide a full translation of term unification (in the style of common Horn clause languages like Prolog) and this has, in turn, placed awkward limitations on the forms of

inferences which could be supported. The purpose of this chapter is to propose a localist network architecture which removes some of these fundamental limitations, resulting in a connectionist system which extends expressive power in achieving symbolic inference.

2 A TARGET SYMBOLIC MODEL

2.1 Symbolic Knowledge

Since a localist connectionist architecture is to be designed as a connectionist model which translates symbolic knowledge (described in the form of rules and facts) into a network, defining a symbolic expression to describe them is necessary. The symbolic expression that has been chosen to describe the target symbolic knowledge is a subset of first-order Horn clause expressions, which is defined as follows:

Definition: A subset of *first-order Horn clause expressions* is a set of universally quantified expressions in first-order predicate calculus of the form:

$$p_1(\cdots) \wedge p_2(\cdots) \wedge \cdots \wedge p_n(\cdots) \rightarrow q(\cdots),$$

where $p()_i$'s and $q()$ are all positive atomic expressions. The conjunction of $p()_i$ is the antecedent and the $q()$ the consequent. An expression with no antecedent is called a *fact* and an expression which has both antecedent and consequent is called a *rule*.

2.2 Symbolic Inference Procedures

To achieve symbolic style of inference, a localist network architecture also has to merge the strategies which are used for drawing inferences into connectionist mechanisms. Two basic symbolic inference procedures must be embedded into the network structure: *matching* and *substitution* procedures. Connectionist mechanisms which correspond to these two procedures will be designed in such a way that they perform counterparts to the symbolic tasks in a connectionist manner.

Let us consider the following rule:

$$p(X, Y) \rightarrow q(X, Y).$$

When the predicate, $p(a, U)$, is presented to the antecedent of the given rule, a forward chaining symbolic inference system will normally use a matching procedure to see if the antecedent of the rule matches with the presented predicate. This yields a set of substitutions between variables and constant in the matching terms.

$$Match[p(a, U),\ p(X, Y) \rightarrow q(X, Y)] = \{a/X,\ U/Y\}.$$

As the result of match procedure, two types of bindings are obtained: the *constant binding* between a and X and the *variable binding* between U and Y. These bindings are usually differentiated using different symbol names in a symbolic inference system (a constant symbol for a constant binding and a variable symbol for a variable binding).

Once a set of bindings is obtained from the match procedure, these bindings are then used for the symbolic substitution procedure. The substitution procedure takes the matched formula and consistently substitutes all occurrences of variables in the formula using the binding set. This procedure may be represented as follows:

$$Substitute[\{a/X, U/Y\},\ p(X, Y) \rightarrow q(X, Y)] = [\ p(a, U) \rightarrow q(a, U)].$$

From the result of this substitution, a symbolic inference system can apply the *modus ponens* inference rule to infer the new logical assertion, $q(a, U)$.

In a symbolic system, these symbolic procedures are clearly separated and are usually implemented as independent modules. In a connectionist system, however, knowledge representation and inference procedures cannot be easily separated because inference procedures have to be represented either in the form of weights on the network or in the embedded structure of networks. In addition, there exists a wide range of algorithms to implement symbolic inference procedures in a symbolic inference system. We are, however, much more constrained in a connectionist system because of the uniformity of network

components and the simplicity of their computing abilities. Nevertheless, any connectionist system which wishes to resemble a symbolic style of inference must have connectionist mechanisms which represent symbolic knowledge and perform some form of matching with substitution.

3 A NETWORK ARCHITECTURE

The localist network architecture proposed in this chapter is a connectionist model which encodes symbolic rules and facts in first-order Horn clause expressions into a set of corresponding structured networks called *structured predicate networks* (SPNs) and performs symbolic inferences over these networks.

When a set of rules and facts are given, this input is compiled into SPNs by the knowledge compiler using algorithm devised. The general structure of a SPN is demonstrated in Figure 1.

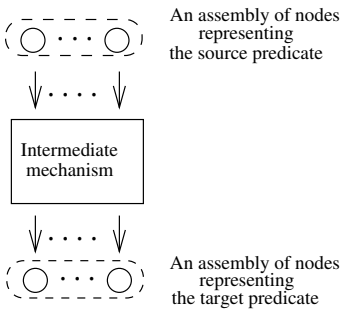

Figure 1 The structure of SPNs.

As can be seen, each SPN consists of three parts: an assembly of nodes for a source predicate, an assembly of node for a target predicate, and an intermediate mechanism between them. The corresponding source and target predicates are the antecedent and consequent predicates of a rule when the SPN is built to support forward chaining and vice versa to support backward chaining. In the case of a fact SPN, the source predicate is a base form of the fact given and the target predicate is a special assembly of nodes introduced to represent result bindings generated as a result of inference. This intermediate mechanism performs similar tasks to the two previously mentioned symbolic inference procedures during inference.

In many cases, a set of rules and facts share the same base predicate (by having the same predicate name and arity) and this allows their SPNs to be connected to each other through this common predicate when they are encoded. When performing inference, this series of connected SPNs is used to draw a chain of inference, which is carried out in a standard way over the network by activating a group of nodes and propagating the activation through these SPNs.

4 COMPONENTS OF SPNS

4.1 Basic Neuron-Like Elements

Before we define neuron elements that will be used to build SPNs, it may be helpful to explain two time scales which will be used throughout this chapter: a *phase* is a minimum time interval in which a neural element performs the basic computations (sampling its inputs and thresholding); an *oscillation cycle* is a window of time in which neuron elements show their oscillatory temporal behaviors. Each oscillation cycle consists of several phases and the number of phases in one oscillation cycle limits the number of distinct entities (constants and variables) which participate for bindings during inference.

To build SPNs, three basic neuron elements are required. Unlike ordinary nodes found in conventional neural network models, these elements have special temporal behaviors hypothesised. An ordinary node samples and sums all its inputs in a given cycle of the time and thresholding the sum to decide its output. Whereas the neuron-like elements sample their inputs over several phases of an oscillation cycle and determine their output patterns, depending on the input patterns sampled during this time period. The detailed behavior of each type of element is differentiated based on a type of input pattern they sample and a type of output pattern they produce in each oscillation cycle. On becoming active, these elements continually produce the output (oscillating) for the duration of inference.

Definition: The **neuron-like elements**[1] that will be used to build SPNs are defined as follows:

- a **π-btu element** becomes active on receiving one or more spikes in different phases of any oscillation cycle. On becoming active, a π-btu element produces an oscillatory spike that is in phase with the driving inputs;
- a **τ-or element** becomes active on receiving one or more spikes within a period of oscillation. On becoming active, a τ-or element produces an oscillatory pulse train whose pulse width is comparable to the period of an oscillation cycle;
- a **multiphase τ-or element** becomes active on receiving more than one input in different phases within a period of oscillation. On becoming active, a multiple τ-or element produces an oscillatory pulse train whose pulse width is comparable to the period of an oscillation cycle.

A threshold, n, associated with each neuron-like element indicates that the element will fire only if it receives n or more spike inputs in the same phase.

Figure 2 shows graphical notation of each neuron-like element and its temporal characteristics.

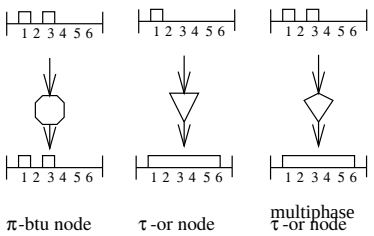

Figure 2 Temporal behaviors of the neuron-like elements.

4.2 Entity Nodes

Symbolic inference systems use symbols to represent components of knowledge. Each symbol is represented using a unique symbolic name and this unique

[1] The τ-or and multiphase τ-or elements are the same as defined in Shastri & Ajjanagadde [1] and a π-btu element is newly defined.

name is used to distinguish not only each symbol from the others but also each symbolic binding from the others. A variable symbol can unify with a constant, a variable, or even with a function in some inference systems but cannot unify with two different constants at the same time. During unification, the system must always be able to identify which variable has been unified.

For proper representation of symbolic entities, the proposed network architecture introduces an entity node.

Definition: An **entity node** is a pair of π-btu elements. The left element is used to represent a variable role of the entity and the right element a constant role. A symbolic entity name may be used as a label of an entity node. Symbolically an entity node is represented using the notation, *entity_name*$[\phi,\phi]$, where the left position of the bracket represents the state of the left π-btu element and the right position that of the right π-btu element. The symbol "ϕ" denotes the state in which the element is inactive. The subscripts, $_{left}$ or $_{right}$, are used as a suffix of a symbolic entity name to differentiate the left element from the right one. The notations, *entity_name*$_{left}[\phi]$ or *entity_name*$_{right}[\phi]$ are used to indicate each state of the left or right π-btu element.

When an entity node is used to represent a constant, only its right element becomes active during a chain of inference. Whereas when the entity node is used to represent a variable, only its left node becomes active. However, when it is used to represent an argument of the predicate, either or both its elements may become active during a chain of inference.

4.3 Predicate Assemblies

A predicate is a basic structured unit of symbolic rules and facts. To represent predicates, an assembly of nodes, called a *predicate assembly*, is used. Like a symbolic predicate, a predicate assembly also has associated arity since each predicate assembly is composed of n entity nodes that correspond to the n arguments.

When a predicate is encoded using a predicate assembly, it is always represented in the form of base predicate which is defined as follows:

Definition: A **base predicate** of an n-ary predicate, $p(A_1,A_2,\ldots,A_n)$, is a predicate which has the same predicate name p followed by the n argument

names, $arg_1, arg_2, \ldots, arg_n$. All predicates which share the same predicate name and arity have the same base predicate.

A more detailed definition of a predicate assembly is as follows:

Definition: An n-ary **predicate assembly** is composed of n entity nodes, $arg_1, arg_2, \ldots, arg_n$. Each of these entity nodes is called an *argument node*. We use the notation, *pred_name:*{ $arg_1[\phi,\phi]$, $arg_2[\phi,\phi]$, ..., $arg_n[\phi,\phi]$} to represent an n-ary predicate assembly symbolically. When any individual argument node needs to be named, a prefix "*pred_name:*" is used before a symbolic name of the argument node.

Figure 3 depicts the structure of a predicate assembly graphically.

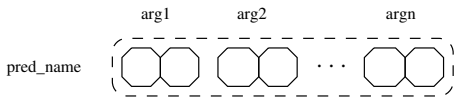

Figure 3 The structure of a predicate assembly.

5 REPRESENTING DYNAMIC BINDINGS

Representing dynamic bindings in a connectionist system has been recognized as one of the main technical obstacles to achieving a symbolic style of inference (see a survey in Park et al [2]). The network architecture proposed here adopts the basic idea of the *temporal synchrony approach* which is credited to Ajjanagadde & Shastri [3]. However their approach can only represent constant bindings and fails to represent variable bindings (limiting the inferential power of their system). The binding mechanism used in this chapter is based on the extended temporal synchrony mechanism described in Park et al [2].

To help the understanding of this mechanism, let us consider the constant a, the variable U, and the two arguments arg_i and arg_j. Before setting up the bindings, they first need to be represented using entity node as follows:

$$a[\phi,\phi],\ U[\phi,\phi],\ arg_i[\phi,\phi],\ arg_j[\phi,\phi].$$

A constant binding and a variable binding between these entity nodes can be represented in the following ways:

- the constant binding between a and arg_i, $\{a/arg_i\}$, is represented by activating the right element of $a[\phi,\phi]$ and $arg_i[\phi,\phi]$ ($a_{right}[\phi]$ and $arg_{i_{right}}[\phi]$) in the same phase, say the first phase, which results in $a[\phi,1]$ and $arg_i[\phi,1]$, where the number 1 stands for the first phase;
- the variable binding between U and arg_j is represented by activating their left elements ($U_{left}[\phi]$ and $arg_{j_{left}}[\phi]$) in a different phase, the second phase for example, which results in $U[2,\phi]$ and $arg_j[2,\phi]$, where the number 2 indicates the second phase.

Basically the in-phase activation between two or more entity nodes represents temporal bindings between them. By changing synchronous activity among entity nodes, different temporal bindings can be obtained easily. A constant binding and a variable binding are distinguished by which element of an entity node become active.

6 BUILDING SPNS FOR RULES

A symbolic rule is encoded by building the antecedent and consequent predicate assemblies and the intermediate connectionist mechanism between them. Since predicates of the rule are easily encoded using two base predicate assemblies, the main concern is how to decide the structure of the intermediate mechanism. The complexity of its structure depends on the following factors:

- *type of an argument*: if a constant argument or repeated variable argument appears in the antecedent predicate, a consistency checking sub-mechanism is inserted into the intermediate mechanism. This sub-mechanism forces the condition that any constant filler which will get bound to a constant argument should be the same as the constant argument and any constant fillers which get bound to repeated arguments should be the same constant;
- *argument matching between the antecedent and consequent*: if the same argument appears both in the antecedent and consequent, a binding propagation sub-mechanism is required between those argument nodes to propagate dynamic bindings from the antecedent to the consequent predicate assembly. Those arguments which only appear in the antecedent do not need

this sub-mechanism; however those arguments which only appear in the consequent require a special sub-mechanism called the binding generation sub-mechanism;

- *binding interaction between different groups of unifying arguments*: if there are more than one different groups of unifying arguments in the antecedent of the rule, the appropriate bindings in one unifying argument group may need to be propagated to the other unifying argument groups when both of them are assigned the same variable filler.

When encoding each symbolic rule, the knowledge compiler must examine all the above factors and has to decide the structure of the intermediate mechanism required. The complexity of the intermediate mechanism of a rule therefore depends on the complexity of the conditions that the syntax of the symbolic rule imposes.

To exemplify this encoding procedure, let us consider the following example rule:

$$p(X, X, Y) \rightarrow q(X, Y).$$

Encoding the rule starts with constructing the base predicate assemblies of the predicates appearing in the rule, *p(X, Y, Z)* and *q(X, Y)*. Then the intermediate mechanism between them is built, which involves the following procedure to determine the required sub-mechanisms.

First of all, a number of groups of unifying arguments is examined. The antecedent of the give rule has two groups of unifying arguments (the X's and the Y's). The required sub-mechanism in this case is a consistency checking sub-mechanism for the repeated X's in the antecedent so that any fillers which get bound to these repeated arguments should be the same constant or unifiable ones.

Second, the matching between arguments in the antecedent and those of in the consequent must be checked. The variable arguments X and Y appear both in the antecedent and consequent of the rule. Therefore binding propagation sub-mechanisms are needed between the first two arguments of p and the first argument of q and between the third argument of p and the second argument of q.

Finally, binding interaction between different groups of unifying arguments will be accommodated. When the same variable filler is assigned to both groups of unifying arguments, X's and Y, the binding obtained from the argument group X's must be propagated to the argument group Y. Therefore, we need binding interaction sub-mechanisms between them to support this.

After this analysis, the knowledge compiler automatically figures out the necessary sub-mechanisms for the intermediate mechanism and builds the corresponding network to the given rule (refer to Park et al [4] for a more detailed algorithm).

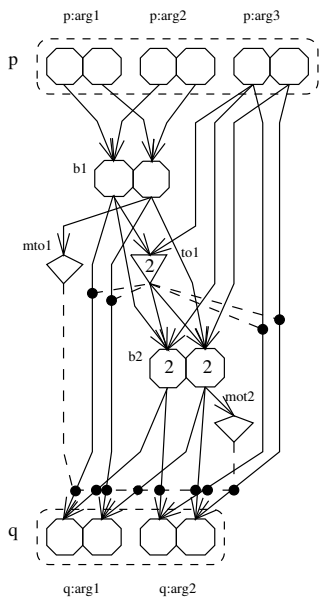

Figure 4 The network which encodes the rule $p(X, X, Y) \rightarrow q(X, Y)$.

Figure 4 shows the corresponding SPN which encodes the given rule. Two base predicate assemblies are shown at the top and bottom of the network and between them is the intermediate mechanism. The $b1_{right}$ and *mto1* connections provide the consistency checking sub-mechanism for the repeated arguments, *p:arg1* and *p:arg2* (corresponding X's in the given rule). The links from *b1* to *q:arg1* and those from *p:arg3* to *q:arg2* are the binding propagation sub-mechanisms for the arguments corresponding to X and Y in the given rule. The

nodes *b1*, *to1*, *b2* are used as the binding interaction sub-mechanism between two different groups of unifying arguments X's and Y. And the node *mot2* is used to check the consistency of the bindings generated after binding interaction. The links drawn in a solid line are excitory and those in a dashed line are inhibitory. The eight black dots which are connected with a dashed line near to the bottom of the network are used to reduce the complexity of the network in the drawing. On becoming active, any inhibitory sources connected to one of these black dots send the same inhibitory signal to the the rest of dots.

7 BUILDING SPNS FOR FACTS

Symbolically, an n-ary fact is represented by the fact name followed by n constant arguments. And this has the same structure as that of ordinary predicate except that it has pre-established bindings. When a fact is encoded, one operation that can be performed against the encoded fact is posing a query predicate to the network to see if the proposed query matches with the encoded fact. Since fillers of the posed query can be a mixture of constants and variables, two types of consistency need to be forced during this match procedure. If the fact, *p(a,a,b)*, is encoded, for example, posing the query *p(a,X,X)* requires the intermediate mechanism to check both constant and variable consistency between fillers and arguments of the encoded fact.

This sort of query processing is relatively easy in symbolic inference systems because matching between two predicates is done by a symbolic pattern matching. The bindings produced are easily shared by other variables using pointers. However, it is difficult to achieve in a connectionist manner because of the following reasons:

- it requires built-in mechanisms for consistency checking (for both constant fillers and variable fillers);
- unlike symbolic system, the result bindings should be explicitly represented over certain nodes and this should be achieved by automatic propagation of bindings during inference.

The proposed network architecture tackles these problems by using an additional predicate assembly, called a *fact assembly*, and the intermediate mechanism. To encode each fact, the fact encoding mechanism requires two predicate assemblies, a base predicate assembly and a fact assembly. The fact assembly

has the same structure as an ordinary predicate assembly and is used to represent the result of inference involving the fact. The base predicate assembly provides the basic structure of facts to be encoded. All facts which share the same predicate name and the same arity are associated with the same base predicate assembly. This allows parallelism in inference over encoded facts. The consistency checking mechanisms will be built into the intermediate mechanism, which will connect between the base predicate assembly and the fact assembly.

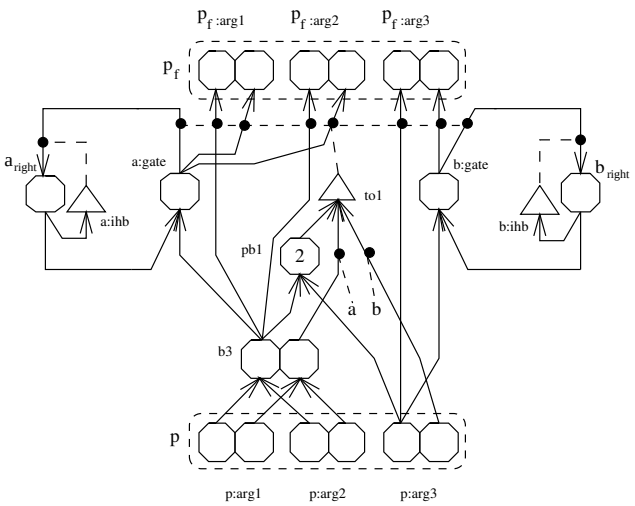

Figure 5 Encoding of the fact, $p(a, a, b)$.

Figure 5 shows the detailed structure of the intermediate mechanism required to encode the fact, *p(a,a,b)*. We can see the fact assembly p_f and its base predicate assembly *p*. Between them is the intermediate mechanism which involves the consistency checking sub-mechanisms and the binding propagation sub-mechanism. The node *b3* is used to collect initial bindings from the repeated constant arguments (*a*'s in the given fact). The fact inhibitory links between $b3_{right}$ and *to1*, between $p{:}arg3_{right}$ and *to1*, are used for constant consistency checking. The nodes, $b3_{left}$, $p{:}arg3_{left}$, *pb1*, and *to1* and related links between them are used for variable consistency checking. The links between *b3* and $p_f{:}arg1$, $p_f{:}arg2$ and those between *p:arg3* and $p_f{:}arg3$ are used as the binding propagation sub-mechanism. When a posed query violates any forced consistency conditions, *to1* will be activated and this prevents the fact

assembly, p_f, from becoming active. Otherwise, the initial bindings will propagate from the base predicate assembly to the fact assembly. Consequently, each argument node of the fact assembly will be activated by the binding propagation sub-mechanism and will represent the result of unification occurred during inference. The nodes *a:gate* and *b:gate* are used to activate the appropriate argument nodes of the fact assembly and their corresponding constant node at the same time. The nodes *a:ihb* and *b:ihb* are used to make each constant node become active only once during a chain of inference. Once constant nodes *a* and *b* become active, their activation makes associated *ihb* nodes active and, as a result, all inputs coming to these constant nodes will be inhibited.

8 PERFORMING INFERENCE

Once a set of symbolic rules and facts are encoded into a network, the inference system can perform either forward chaining or backward chaining over the network (to support both, separate intermediate mechanisms have to be built).

Forward chaining is started by presenting a predicate to the system. The presented predicate has the form $pred_name(t_1, t_2, \ldots, t_m)$, where t_i's are either constants or existentially quantified variables. Presenting a predicate to the system involves specifying the bindings between the presented predicate and its corresponding base predicate assembly in the network. Setting up the initial bindings is done as follows:

- assign each entity node (called a filler node) to each filler which appears in the input predicate
- activate each filler node and the corresponding argument of the base predicate in a unique phase in such a way that: if the filler is a constant, activate both the right π-btu element of the filler and the argument nodes; if the filler is a variable, activate both the left π-btu element of the filler and the argument nodes.

Once these initial bindings are set up using temporal synchrony, these bindings are then propagated to the consequent predicate assembly and, as a result, its arguments will represent the result of inference. In Figure 4, for instance, presenting the input fact, *p(a,U,U)*, to the network will set up initial bindings, {*a/p:arg1, U/p:arg2, U/p:arg3*} by activating the filler nodes and their corresponding argument nodes in the following phases:

$$a_{right}[1],\ U_{left}[2],$$
$$p{:}\{arg1[\phi,1],\ arg2[2,\phi],\ arg3[2,\phi]\},$$

where the numbers specify the particular phases. These activations propagate to the q predicate assembly through the intermediate mechanism and activate its argument nodes

$$q{:}\{arg1[2,1],\ arg2[2,1]\},$$

which represents the result of inference $q(a,a)$ and the result of unification generated during the inference $\{U/a\}$.

Backward chaining over encoded facts is also started by posing a query predicate to the network. The query predicate has the same form as that used in forward chaining. Posing a query predicate sets up the initial bindings between the posed query and the corresponding base predicate assembly (the consequent predicate assembly in case of a rule) in the same way as shown for forward chaining. The result of inference is then obtained by propagating these initial bindings to the antecedent predicate assembly and the associated fact assemblies if any. An associated fact assembly will only be activated if it matches with the query predicate posed. If the posed query contains variable fillers, the arguments of the activated fact assembly will represent the result of unification obtained from the inference. Therefore, after a query is posed to the network, the final result of backward chaining is obtained by observing the states of each fact assembly and its base predicate asembly in the network.

For example, posing the query, $q(a,U,V)$, to the network shown in Figure 5 will set up initial bindings, $\{a/p{:}arg1,\ U/p{:}arg2,\ V/p{:}arg3\}$, by activating the filler nodes and their corresponding argument nodes in the following phases:

$$a_{right}[1],\ U_{left}[2],\ V_{left}[3],$$
$$p{:}\{arg1[\phi,1],\ arg2[2,\phi],\ arg3[3,\phi]\}.$$

These initial bindings then propagate into the intermediate mechanism, including the consistency checking sub-mechanism. Since the posed query does not violate the consistency condition, these bindings propagate to the fact asembly through the binding propagation sub-mechanism. As a result, the arguments of the fact assembly and their corresponding constant nodes will be activated as follows:

$$b_{right}[3],$$
$$p_f:\{arg1[\phi,1],\ arg2[2,1],\ arg2[3,3]\}.$$

This represents that the encoded fact, $p_f(a,a,b)$, matches with the posed query and the result of unification obtained from the second and third argument nodes of the fact predicate is $\{U/a, V/b\}$.

9 SOME RELATED WORK

Applying temporal synchrony for a connectionist model to support various inference is found in the early work of Fahlman [5] in the form of an abstract computational mechanism in which temporal synchrony is represented in the notion of *marker passing*. Each node in the network stores a small number of discrete markers. These markers propagate between nodes under the supervision of the network controller to achieve dynamic bindings in a symbolic style of inference. Thus nodes in his system were required to have high computational ability such as storing, matching, and selectively propagating marker bits. Later, Clossman [6] used synchronous activity to represent argument-filler bindings for a connectionist model of categorization and learning without efficient encoding mechanisms for rules and facts. Ajjanagadde & Shastri [3] proposed a connectionist dynamic binding mechanism using temporal synchrony. They also suggested integrated rule and fact-encoding mechanisms which cooperate with their dynamic binding mechanism. Later, Shastri & Ajjanagadde [1] built a connectionist model for reflexive reasoning (called SHRUTI) on the basis of their mechanisms. It provides an efficient connectionist mechanism for dynamic bindings based on temporal synchrony together with encoding mechanisms for rules and facts involving n-ary predicates.

Since the basic connectionist binding mechanism using temporal synchrony is inspired by SHRUTI's dynamic binding mechanism, it is interesting to compare their system with ours. First, Shastri & Ajjanagadde's binding mechanism only represents constant bindings explicitly. It represents variable bindings by inactivity of argument nodes during inference. It does not therefore differentiate one variable binding from others, which is a basic, and fundamentally important feature of symbolic inference. Second, Shastri & Ajjanagadde's rule and fact-encoding mechanisms do not consider all the conditions that symbolic rules and facts may impose. In particular, a sub-mechanism for binding interaction between arguments is not considered in their system at all. However our model can represent groups of unifying arguments even when it is not known to which

individual the variable in those arguments is bound. Finally, the proposed fact-encoding mechanism does not need different network configurations for yes-no and and wh-type query processing (as is the case in Shastri & Ajjanagadde's system). These two query procedures are merged into one fact-encoding mechanism. This mechanism does not even require two separate stages of wh-type query processing because the result of unification is automatically represented on each argument node of the fact assembly during query processing.

Other connectionist systems which tried to achieve Horn clause calculus-based expressive power, without using the temporal synchrony approach, can be found in Hölldobler & Kurfeß' CHCL [7] and Sun's CONSYDERR [8]. CHCL can handle unification over arbitrary first-order terms. However, the total number of nodes required in their system is quadratic in the size of its knowledge base. CONSYDERR also provides a compilation mechanism of rules and facts in a subset of Horn clause and is able to handle various knowledge representation issues, but the need for high computational ability of the assemblies adopted and the use of abstraction which hides network details is beyond simple summation with thresholding.

10 SUMMARY

Any connectionist knowledge-based system which aspires to imitate symbolic inference should be able to represent variables and constants and unification between them not only within a group of unifying arguments but also across many groups of unifying arguments. The proposed localist network architecture is a step in this direction. In the proposed inference system, both constant and variable entities are represented using entity nodes. Also, arguments of a predicate are represented using the same entity nodes. This representation together with the connectionist dynamic binding mechanism enables the proposed architecture to deal with both constant and variable bindings without additional mechanisms. The use of a predicate assembly to represent rules and facts allows the inference system to represent easily the result of inference and unification on each argument node. The various intermediate mechanisms adopted in the rule and fact encoding mechanisms provide a way of representing not only a group of unifying arguments but also many groups of such arguments, with consistency checking between them.

In summary, the proposed inference system provides a mechanism which translates a significant subset of Horn-clause into a connectionist representation,

which may be executed very efficiently (not only in a forward chaining but also in a backward chaining style). However, in order to have the full expressive power of Horn clause FOPC, we need to add the ability to represent structured terms as arguments of a predicate and unbounded recursion in rules. Currently, no connectionist system has provided a convincing solution to all these problems (given the established problems of decidability for full FOPC) and we know that to do so, in general, we would have to sacrifice one of the most attractive features of this class of system: its ability to guarantee an answer in finite time.

ACKNOWLEDGMENTS

This work was partly supported by the Ministry of Science & Technology, Korea, and the British Council under grant SCOT/KOR/2923/37/A.

REFERENCES

[1] Shastri, L., and Ajjanagadde, V., "From simple associations to systematic reasoning: A connectionist representation of rules, variables, and dynamic bindings using temporal synchrony" *Behavioral and Brain Sciences*, Vol. 16, No. 3, 1993.

[2] Park, N. S., Robertson, D., and Stenning, K., "An extension of the temporal synchrony approach to dynamic variable binding in a connectionist inference system" *Knowledge-Based Systems*, Vol. 8, No. 6, 1995, pp. 345–357.

[3] Ajjanagadde, V., and Shastri, L., "Rules and variables in neural nets" *Neural Computation*, Vol. 3, 1991, pp. 121–134.

[4] Park, N. S., Robertson, D., and Stenning, K. Oscillation of neurons, dynamic bindings, and a symbolic knowledge representation with an embedded inference mechanism, In D. Levine, B. Brown, and T. Shirey (Eds.), *Oscillation in Neural Systems*, Lawrence Erlbaum Associates, Inc., in print.

[5] Fahlman, S. E., *NETL: A System for Representing Real-World Knowledge*, MIT Press, 1979.

[6] Clossman, G., A model of categorization and learning in a connectionist broadcast system, Ph.D. dissertation, Department of Computer Science, Indiana University, 1988.

[7] Hölldobler, S. H., and Kurfeß, F., "CHCL - A connectionist inference system" In B. Fronhöfer, and G. Wrightson (Eds.) *Parallelization in Inference Systems*, Lecture notes in computer science, Springer, 1991, pp 318–342.

[8] Sun, R., "On variable binding in connectionist networks" *Connection Science*, Vol. 4, No. 2, 1992, pp 93–124.