A Connectionist Representation of First-Order Formulae with Dynamic Variable Binding

Nam Seog Park

Ph.D.
University of Edinburgh
1997

Abstract

The relationship between symbolicism and connectionism has been one of the major issues in recent Artificial Intelligence research. An increasing number of researchers from each side have tried to adopt desirable characteristics of the other. These efforts have produced a number of different strategies for interfacing connectionist and symbolic AI. One of them is *connectionist symbol processing* which attempts to replicate symbol processing functionalities using connectionist components.

In this direction, this thesis develops a connectionist inference architecture which performs standard symbolic inference on a subclass of first-order predicate calculus. Our primary interest is in understanding how formulas which are described in a limited form of first-order predicate calculus may be implemented using a connectionist architecture. Our chosen knowledge representation scheme is a subset of *first-order Horn clause expressions* which is a set of universally quantified expressions in first-order predicate calculus. As a focus of attention we are developing techniques for *compiling* first-order Horn clause expressions into a connectionist network. This offers practical benefits but also forces limitations on the scope of the compiled system, since we are, in fact, merging an *interpreter* into the connectionist networks. The compilation process has to take into account not only first-order Horn clause expressions themselves but also the strategy which we intend to use for drawing inferences from them. Thus, this thesis explores the extent to which this type of a translation can build a connectionist inference model to accommodate desired symbolic inference.

This work first involves constructing efficient connectionist mechanisms to represent basic symbol components, dynamic bindings, basic symbolic inference procedures, and devising a set of algorithms which automatically translates input descriptions to neural networks using the above connectionist mechanisms. These connectionist mechanisms are built by taking an existing temporal synchrony mechanism and extending it further to obtain desirable features to represent and manipulate basic symbol structures. The existing synchrony mechanism represents dynamic bindings very efficiently using temporal synchronous activity between neuron elements but it has fundamental limitations in supporting standard symbolic inference. The extension addresses these limitations.

The ability of the connectionist inference model was tested using various types of first order Horn clause expressions. The results showed that the proposed connectionist inference model was able to encode significant sets of first order Horn clause expressions and replicated basic symbolic styles of inference in a connectionist manner. The system successfully demonstrated not only forward chaining but also backward chaining over the networks encoding the input expressions. The results, however, also showed that implementing a connectionist mechanism for full unification among groups of unifying arguments in rules, are encoding some types of rules, is difficult to achieve in a connectionist manner needs additional mechanisms. In addition, some difficult issues such as encoding rules having recursive definitions remained untouched.

Acknowledgements

Very many thanks, firstly, to my supervisors Mr. Dave Robertson and Prof. Keith Stenning for their countless comments and suggestions through the entire period of the PhD study. This thesis would not have been possible without their continuous advice and encouragement.

Many thanks to my examiners, Dr. John Hallam from the department and Dr. Leslie Smith from Centre for Cognitive and Computational Neuroscience, University of Stirling, who have made valuable comments on the thesis.

I would also like to thank many of the friends and colleagues who have provided helps, advice and companionship during the PhD study.

Special thanks to my parents who have been always the source of emotional and financial support, and to my wife, Jeong In Min, and my children, Yeon Chan and Herin, for being patient and understanding during all stages of the study, which have made the whole PhD pursuit more worthwhile.

I am very grateful to the Ministry of Science & Technology, Korea, and the British Council for first two years of scholarship under grant SCOT/KOR/2923/37/A, also to Doorae Institute for their sponsorship during the later periods the PhD study. Thanks to the Artificial Intelligence Department at Edinburgh University who have made it possible for me to attend conferences and workshops overseas.

Finally, thanks to God who has made all these happen in my life.

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research.

——————————————

Nam Seog Park
Edinburgh
3 January 1997

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

The relationship between symbolicism and connectionism has been one of the major issues in recent Artificial Intelligence research. An increasing number of researchers from each side have tried to adopt desirable characteristics of the other. These efforts have produced a number of different strategies for interfacing connectionist and symbolic AI.

This chapter first illustrates two important AI research paradigms, symbolicism and connectionism, and various neurosymbolic strategies that has been used to interface them. The scope of this thesis is a particular neurosymbolic strategy which was adopted in this research. The following presentation involves motivation, approach, and summarised results of this research. Finally the organisation of the thesis is described.

1.2 Symbolicism and Connectionism

1.2.1 Symbolicism

Symbolicism has long been a major paradigm of AI research. This paradigm attempts to understand intelligence by building a working symbolic model of a *mind* [Dreyfus & Dreyfus (1988)]. The guiding principle of symbolicism is the *physical symbol system hypothesis* [Newell & Simon (1976)]. The hypothesis states that:

> A physical symbol system has the *necessary* and *sufficient* means for *general*

> *intelligent action.*
>
> By "necessary" we mean that any system that exhibits general intelligence will prove upon analysis to be a physical symbol system. By "sufficient" we mean that any physical symbol system of sufficient size can be organised further to exhibit general intelligence. By "general intelligent action" we mean that the same scope of intelligence as we see in human action: that in any real situation behaviour appropriate to the ends of the system and adaptive to the demands of the environment can occur, within some limits of speed and complexity [Newell & Simon (1976)].

A physical symbol system is a symbol system which is composed of symbols, expressions and processes that operate on expressions. The use of the term "physical" is intended to imply that such a system would obey the laws of physics and be attainable in some practical sense. Symbols in a physical symbol system are symbolic mediums that represent concepts in our physical system and expressions are symbolic structures obtained by placing symbols in a physical relation. In their view, a computer is a physical symbol system that produces an evolving collection of symbolic structures through time [Bechtel & Abrahamsen (1991)]. One important emphasis made by Newell and Simon is that there is a semantics (designations and interpretation) within the system itself and intelligence cannot be achieved without this internal semantics. For example, expressions in stored programs designate locations in computer memory and these expressions can be interpreted by accessing those locations.

Although a similar realisation of a physical system and its semantics is possible in connectionism using neural networks, the major emphasis made by the physical symbol system hypothesis is the use explicitly of symbols. Hence, the physical symbolic system hypothesis suggests that a physical realisation of a symbol system – which has a suitable means for performing the designation and interpretation of its symbol structures into the world – is necessary and sufficient for intelligent action. This hypothesis is also an empirical hypothesis which is intended to specify a class of such symbol systems which demonstrate those capable of intelligent action. Two significant methodological commitments proposed by this hypothesis are the use of symbols and systems of symbols to describe the world and the empirical view of computer programs as experiments [Luger & Stubblefield (1989)]. Although Newell and Simon presented this hypothesis in the middle 1970s, all the symbolic processing-based AI research that has been done since AI started in the late 1950s has consisted of various attempts to investigate this hypothesis or experiments inspired by it.

This hypothesis was further supported by Smith's *knowledge representation hypothesis* [Smith (1982)] which states technical meaning of representation. Consequently, a number of important knowledge representation and inference techniques were developed out of AI research, which became basic building blocks to build knowledge-based systems (or Expert systems).

1.2.2 Connectionism

Connectionism, also called the subsymbolic paradigm, attempts to understand intelligence by building a working neural network model of the *brain* [Dreyfus & Dreyfus (1988)]. An artificial neural network consists of a number of units which are connected to each other with links. These units correspond to idealised neurons and links correspond to idealised synaptic connections. Computational features of nodes are very simple and limited: each of them receives inputs from other units and computes an output which is propagated to other units. They can only hold limited state information and their outputs do not have sufficient resolution to encode symbolic names or pointers [Rumelhart & McClelland (1986)]. Instead of using explicit symbols, expressions, and processes, the knowledge of a neural network emerges as patterns of activation over some of units of the network based on neural connections and threshold values.

The foundation of neural network research was laid by McCulloch and Pitts in early 1940s [McCulloch & Pitts (1943)]. They proposed a network model with very simple neuron-like units which have both inhibitory and excitory inputs. Since these units do not have threshold, they fire only if the total number of active excitory inputs exceeds the number of inhibitory inputs. On becoming active, the neuron elements project an output value of 1. They showed how configurations of these units carry out the logical operations of AND, OR and NOT. Further research demonstrated that such networks have the same computational power as a Universal Turing Machine.

Later von Neumann (1956) showed that introducing more redundant units made McCulloch and Pitts' networks reliably produce a desirable output even when some units malfunctioned. This idea was further developed by Winograd & Cowan (1963) to provide distributed redundant representations, where a unit contributed to the activation decision of some other units are also being affected by some other units. This procedure is considered as an early version of a *distributed representation* in neural networks.

A neural network architecture that uses adjustable thresholds and weights was proposed by Rosenblatt (1962). These networks were called *perceptrons*. By making weights of connection between units continuous rather than binary, as is the case in McCulloch and Pitts' networks, perceptrons were able to be trained to change their responses to solve classification problems. If an output of a unit is 0 when it should be 1, the weights on connections feeding into the unit are incremented. Whereas if an output of a unit is 1 when 0 is expected, the weights are decremented. Rosenblatt demonstrated that this training procedure converges in his Perceptron Convergence Theorem. Although perceptrons were successful and widely studied, their limitation in dealing with some elementary computations led to decline of interest of this model. This simplest example is the *exclusive or (XOR) problem* [Minsky & Papert (1969)]. Even though networks with more layers of units were studied, no learning algorithm which could determine the weights necessary to implement a given calculation was known.

It is 1980s that neural network research made dramatic progress on the problems of perceptrons and other models under the name of *connectionism*. The most influential development in this decade was multi-layer perceptrons and back propagation learning algorithm [Rumelhart & McClelland (1986)]. The previous limitation on perceptrons was lifted by introducing multiple layers of hidden units and the back propagation learning algorithm which allows error corrections to be passed through multiple layers of hidden units. Much recent research on connectionism is based on this model and its extension. Other network models included Hopfield nets [Hopfield (1982)] and Boltzmann machines [Ackley et al. (1985), Hinton & Sejnowski (1986)]. Hopfield nets gave some helpful physical insight by introducing an energy function and Boltzmann machines constructed formulations using a stochastic process for adjusting weights.

Further development of new architectures, new training algorithms, and advances in the mathematical basis of neural systems have made connectionism another important paradigm in AI research. Neural network systems as compared to symbol systems are robust and noise tolerant. When a system adopts distributed representations [Hinton (1984)], the malfunction of individual units has little impact on the system's reliability and performance (graceful degradation). Moreover, they support parallelism and are easily implemented on parallel computers. Another important attraction of these systems is the ability to learn new concepts.

Neural network systems has been successful to demonstrate low-level perceptual functionality in a number of areas such as signal processing, pattern recognition, and classifications from noise data. However, the simplicity of a unit's processing ability compared to the needs of symbolic computation, and the restriction on the complexity of messages exchanged by neurons, impose strong constraints on the nature of neural representations and processes [Feldman & Ballard (1982), Feldman (1989), Shastri (1991)]. This made it difficult to model higher-level cognitive tasks such as manipulating data structures, handling dynamic bindings, controlling inferencing and attention.

1.3 Neurosymbolic Integration

Since both AI research paradigms have their own strong advantages, an increasing number of researchers from each side have tried to adopt desirable characteristics of the other. These efforts have produced a number of subtle philosophies for interfacing connectionist and symbolic AI. These neurosymbolic integration strategies are categorised into two classes: *unified approaches* and *hybrid approaches*. Figure 1.1 gives more detailed view of these two classes of approaches[1].

The unified approaches build a system based on a single paradigm and try to enhance it to obtain both the connectionist and symbolic capabilities. Depending on the type of paradigm chosen, they are called a *symbol-based unified approach* or a *neural network-based unified approach*. The symbol-based unified approach uses symbolic components as basic building blocks and adopts a network architecture consisting of nodes and

[1] Although some attempts to classify these strategies was made by Handler (1989) and Lallement & Alexandre (1995), the terminologies used here are adopted mainly from Hilario (1995).

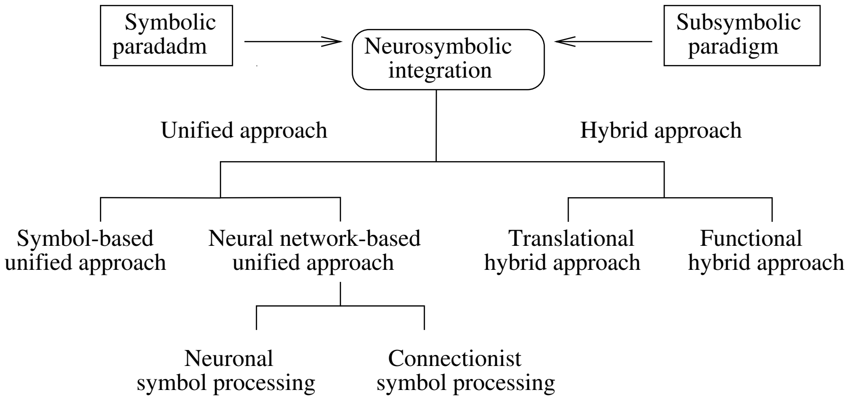


Figure 1.1: Categorisation of neurosymbolic strategies (based on the classification in Hilario (1995))

links and propagation of activation over the network. One example of such a system is the Net-Clause model proposed by Markov (1990). Its ability has been demonstrated in concept learning [Markov (1992)] and logical inference [Markov (1993)]. The other unified approach involves two trends: *neuronal symbol processing* and *connectionist symbol processing*. The aim of neuronal symbol processing is to model the brain's high-level functions. In these models, symbolic functions emerge from neural structures and processes which are grounded in biological reality (e.g. Alexandre et al. (1991), Edelman (1992), Nenov & Dyer (1993, 1994)). Connectionist symbolic processing, however, does not claim neurobiological plausibility. Thus, formal neurons are used as basic building blocks and model construction begins by taking for granted the fact that some high-level symbolic functions are performed and proceeded with the design of the appropriate connectionist infrastructure [Hilalio (1995)]. According to the underlying representation scheme employed, the architectures of these models can be either *localist* [Lange & Dyer (1989b), Shastri & Ajjanagadde (1993)], where each neuron represents a concept, or *distributed* [Gallant (1988), Touretzky & Hinton (1988), Browne & Pilkington (1994b)], where a concept emerges from the interaction of several neurons or combination of both [Sun (1992)].

Unlike the unified approaches, the hybrid approaches aim at synergistic combination in systems consisting of both neural and symbolic components. Neurosymbolic models using these approaches are classified into two groups: *translational hybrid models*[2] and *functional hybrid models*. The translational hybrid models do not have an independent symbolic component for data processing. They use neural network components as processors and adopt symbolic structures as input and output descriptions. They compile input symbolic structures into neural networks and extract symbolic structures from neural networks after refinement of encoded knowledge. The most widely used symbolic structures are rules [Giles & Omlin (1996), Towell & Shavlik (1993, 1994)]. The functional hybrid models, however, have independent symbolic and connectionist components which cooperate with each other. Depending on their *integration strength*, functional hybrid models can be either *loosely coupled* or *tightly coupled* systems. In loosely coupled systems, each component is clearly localised in time and space, and data and control is transfered from one component to the other through external structures accessible to both components. In tightly coupled systems, on the other hand, data and control transfer is carried out through common internal structures, such as memory, shared by both components. Thus, changes in one component directly affect the behaviour of the other component via these common structures. Another way of categorising the functional hybrid system is by four *integration schemes*: *coprocessing*, *subprocessing*, *chainprocessing*, and *metaprocessing*. In coprocessing, both types of components participate in the data processing as equal partners. Whereas, in subprocessing, one component is embedded in and controlled by the other. In chainprocessing, one component serves as a main processor and the other as a pre or post processor. Finally, in metaprocessing, one component is in charge of basic tasks and the other carries out meta-level tasks such as controlling and monitoring. Hilario (1995) describes some example systems which belong to each subcategory of functional hybrid models.

1.4 Connectionist Symbol Processing

Connectionist symbol processing attempts to replicate symbol processing functionalities using connectionist components. One important ability of a symbol system is easy representation of a separated symbolic structure and structure-sensitive operations

[2] These are also called *transformational models* [Medsker (1994)] or *semi-hybrid models* [Orsier & Labbi (1995)].

such as matching and substitution. For instance, the following expression,

$$own(father\_of(tom), holy\_bible),$$

represents a separated symbolic structure very easily: the predicate symbol *own* and the structured term *father_of(tom)* are clearly separated from each other in the physical medium of representation symbols. Similarly, within the structured term *father_of(tom)*, the symbol *tom* and the name of the structured term *father_of* are also separated from each other.

Implementing the ability to encode a separated constituent structure and structure-sensitive operations to manipulate the constituent structure in specific situations is essential to achieve symbol processing functionalities. However, the representational ability of a connectionist system to represent and manipulate constituent structure has been doubted. As McCarthy (1988) noted, most connectionist systems suffer from a propositional fixation and their representational power is restricted to unary predicates applied to a fixed object. Foder & McLaughlin (1990) and Foder & Pylyshin (1988) also claim that representation in a connectionist system does not have constituent structures (compositionality) and this prevents the system from supporting systematic, structure-sensitive operations of any interesting symbol processing. These critiques have acted as an incitant which has stimulated the development of many connectionist symbol processing models. These models have demonstrated the abilities to represent and manipulate basic symbolic structures such as lists [Tourentzky (1990)], trees [Chalmers (1990), Christman (1991), Kücheler & Goller (1996), Pollack (1988, 1990), Sperduti (1994, 1996), Sperduti et al. (1995)], stacks [Das et al. (1993), Sun et al. (1993)], and higher symbolic structures such as hierarchies [Bartfai (1996), Mani & Shastri (1991)], semantic networks [Lange & Dyer (1989a, 1989b), Pereira Castro (1995, 1996), Shastri & Feldman (1986)], and rules [Shastri & Ajjanagadde (1993), Touretzky & Hinton (1988), Sun (1992, 1994)]. Their areas of application involve solving constraint satisfaction problems [Pereira Castro (1995,1996)], term unification [Browne & Pilkington (1994a), Holldobler & Kurfeß(1991), Pollack (1988, 1990)], classification of structures [Sperduti (1996)], grammar processing [Das et al. (1993), Sun et al. (1993)], language transformations [Chalmers (1990), Christman (1991)], reactive sequential decision making [Sun & Peterson (1995)], and general inference [Ajjanagadde & Shastri (1993), Lange & Dyer (1989a), Touretzky & Hinton (1988), Sun (1992, 1994)]

Neural network models which use a distributed representation [Bartfai (1996), Browne & Pilkington (1994a), Chalmers(1990), Chrisman(1991), Elman (1989), Harris & Elman (1989), Pollack (1988, 1990), Pereira Castro (1995, 1996), Touretzky (1990), Touretzky & Hinton (1988), Sperduti (1994, 1996), Sun et al. (1993)] can be subjected to structure-sensitive operations without the need of unpacking the representations into their constituents or converting them into separated versions [Chalmers (1990)]. But even in these cases, there is a type of separated representation at a higher level of description if a connectionist activation pattern is divided into segments corresponding to the constituent that the representation intend to have [Barnden (1992)]. These activation patterns are the main form of short-term constituent structure in these distributed connectionist system. In other systems which employ a localist representation [Ajjanagadde & Shastri (1991), Holldobler & Kurfeß(1991), Lange & Dyer (1989a), Shastri & Ajjanagadde (1993)], a group of nodes is dedicated to represent always the same part of or whole symbolic structures. Activation patterns on different nodes or groups of nodes are regarded as different short-term constituent structures. Sun (1992,1994), however, employs both distributed representation and localist representations at the same time. In this system, an activation pattern within each component or that across both components are used to represent short-term constituent structures.

In contrast to representing short-term constituent structures by activation state, long-term encoding of constituent structures in connectionist symbolic processing systems is represented by the weight or specific structures of networks.

1.5 Scope of This Thesis

The aim of this thesis is to develop a connectionist inference architecture which can perform standard symbolic inference that can be observed in first-order predicate calculus [Chang & Lee (1973), Bundy (1983), Manna & Waldinger (1985), Luger & Stubblefield (1989)]. This work is, therefore, situated in developing a neural network-based unified model which employs connectionist symbolic processing with a localist network representation. Our primary interest is in understanding how formulas which are described in a limited fragment of first-order predicate calculus may be implemented using a connectionist architecture. Our chosen knowledge representation scheme is a subset of *first-order Horn clause expressions* (the detailed definition will be in Section 4.2.1) which is itself a subset of universally quantified expressions in first-order predicate calculus.

As a focus of attention we develop techniques for *compiling* first-order Horn clause expressions into a connectionist network which shows a standard symbolic inference behaviour. Unfortunately, this type of compilation is not straightforward since we, in fact, merge an *interpreter* into the connectionist networks. The compilation process therefore has to take into account not only the first-order Horn clauses themselves but also the strategy which we intend to use for drawing inferences from them. Although some connectionist inference systems have been proposed (as surveyed in Section 2.4.1), all of these impose strong representational limitations and/or rely on neural elements of high complexity. This thesis shows how some of these fundamental limitations can be removed, resulting in a significantly improved connectionist inference model.

1.6 Motivation, Approach, and Results

Expressions in symbolic formulas 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, we should be able to cancel out the deficiencies of each style whilst retaining the advantages of both. This offers significant practical benefits but also forces limitations on the scope of the compiled system. In particular, it appears that some fundamental aspects of symbolic inference are difficult to translate directly into a connectionist framework. It also has proved 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 [Sun (1992), Shastri & Ajjanagadde (1993), Park (1992), Park et al. (1995)]. Therefore, some efforts are needed to explore the extent to which this type of a translation mechanism can build a connectionist inference architecture to accommodate standard symbolic style of inference. This thesis explores this through the following procedures:

- constructing efficient connectionist mechanisms to represent basic symbolic components of first order Horn clause expressions;
- building an efficient connectionist mechanism to represent dynamic bindings[3];
- finding connectionist equivalents to standard symbolic inference procedures;
- providing additional connectionist mechanisms which address key knowledge representation issues in a target network representation, such as consistency checking;
- devising a set of algorithms which automatically translate input descriptions in first order Horn clause expressions to neural networks using the above connectionist mechanisms.

Implementing the basic symbolic components of first order Horn clause expressions and an efficient dynamic binding mechanism are carried out by taking an existing mechanism, and extending it to obtain desirable features to represent and to manipulate symbolic structure. The connectionist mechanism that was employed for this work is a temporal synchrony mechanism proposed by Ajjanagadde & Shastri (1991) and Shastri & Ajjanagadde (1993). The temporal synchrony mechanism represents dynamic bindings very efficiently using temporal synchronous activity between neuron elements but it has fundamental limitations for supporting standard symbolic inference [Park (1992), Park et al. (1993), Park et al (1995)]. Park et al. (1995) extended the temporal synchrony mechanism to addresses these limitations. The surveys in Section 2.4.1 and

[3] Dynamic bindings refer to a set of bindings which can be represented dynamically. A connectionist representation makes a clear distinction between dynamic bindings and static bindings. See Section 2.3.1 for more detailed description.

Chapter 3 illustrate the advantages and disadvantages of this mechanism with respect to others.

A connectionist inference architecture proposed in this thesis is based on this extended mechanism and employs additional connectionist mechanisms to implement standard symbolic inference procedures[4] and to deal with some knowledge representation issues, such as consistency checking and unification between groups of unifying arguments. For a given set of first Horn clause expressions, the corresponding connectionist inference architecture is built by a number of algorithms that are devised for this purpose. During compilation, these algorithms automatically decide and build necessary connectionist components and integrate them to construct a localist neural network encoding the given expressions. These networks are the basic components of the proposed connectionist inference architecture over which symbolic inference is replicated in a connectionist manner.

The ability of the connectionist inference model was tested using various types of first order Horn clause expressions (as demonstrated in Chapter 7 and Appendix B). The results showed that the proposed connectionist inference model was able to encode significant sets of first order Horn clause expressions and replicated basic symbolic styles of inference in a connectionist manner. The system successfully demonstrated not only a forward chaining style of inference but also a backward chaining style of inference over the networks encoding the input expressions. The results, however, showed that a mechanism to support full unification among groups of unifying arguments in rules, and encoding of some types of rules, is difficult to achieve in a connectionist manner. In addition, some difficult issues such as encoding rules having recursive definitions are still outstanding.

1.7 The Organisation of the Thesis

The rest of the thesis is structured as follows:

Chapter 2 presents an overview of related work. First of all, the nature of standard symbolic inference is demonstrated. This is followed by presentation of some issues

[4] This is demonstrated in Section 2.2.

related to connectionist inference architecture. Then, it surveys the existing connectionist inference systems which were built based on various connectionist mechanisms for dynamic bindings. Each system will be presented in terms of the connectionist architecture adopted, a mechanism for dynamic bindings, and its expressive power, as well as its advantages and disadvantages.

Chapter 3 highlights in more detail a temporal synchrony solution to the dynamic biding problem proposed by Shastri & Ajjanagadde (1993). The basic mechanism of temporal synchrony solution, and the corresponding connectionist system, SHRUTI, will also be described. The rest of the chapter will focus on SHRUTI's rule and fact encoding mechanisms, how to encode rules and facts involving n-ary predicates and how to perform inference over networks. Finally, the limitations of their system will be explored and discussed.

Chapter 4 describes a connectionist architecture for symbolic inference (CASI) which was inspired by the temporal synchrony solution. Firstly, we describe the target symbolic inferences based on first-order formula which CASI is intended to achieve. Then the overall architecture of CASI will be presented. CASI defines a new neural element, called a π-btu node, and an entity node consisting of a pair of π-btu elements. This entity node is used to represent basic symbolic components: constant and variable entities and predicates. We also show how CASI's connectionist mechanisms correspond to the basic symbolic inference procedures.

Chapter 5 explores more deeply how symbolic rules and facts are encoded into networks. Although CASI adopts Shastri & Ajjanagadde's temporal synchrony approach to represent dynamic bindings, its rule and fact encoding mechanisms are designed to deal with various knowledge representation issues. This rule and fact encoding mechanisms consider unification and consistency checking not only within a group of unifying arguments but also across many groups of such arguments. The functional abilities of these mechanisms to achieve target symbolic inference procedures will be demonstrated by applying the connectionist inference procedure over example networks.

Chapter 6 discuses some possible extensions of CASI in order to extend its expressive power. The extensions include special mechanisms introduced to encode rules in a special condition and representing structured terms in rules and facts.

In Chapter 7, we first test the behaviour of CASI using variety of sets of rules and facts. Each of the test sets are carefully chosen to test target symbolic behaviour of various types of symbolic rules and facts. Each result of the test will be summarised and analysed. Next, CASI's time and space complexities will be analysed. Then, CASI's architecture and ability to resemble symbolic inference will be compared with SHRUTI and other similar connectionist inference systems.

Finally, chapter 8 concludes the thesis and discusses some future directions.

1.8 Summary

This thesis develops a connectionist inference architecture, based on connectionist symbol processing, which performs standard symbolic inference on a subclass of first-order predicate calculus. Implementing such a connectionist architecture requires connectionist mechanisms which represent symbolic data structures and perform basic symbol processing functionalities. Moreover, such mechanisms needs to handle some specific issues that any type of connectionist inference models have to cope with. The next chapter portrays these issues in greater detail and surveys how existing connectionist inference models address these problems.

Chapter 2

Connectionist Inference Systems

2.1 Overview

This chapter firstly illustrates the nature of standard symbolic inference procedures with a simple set of first order predicates. Secondly, some issues related to implementing connectionist inference architectures to achieve symbolic inference will be presented. The reset of this chapter surveys existing symbolic inference architectures to demonstrate how they cope with these issues in their solutions. The surveyed architectures are compared at the end of this chapter in terms of representation schemes adopted, structure of networks, and expressive power.

2.2 Nature of Standard Symbolic Inference

Predicate calculus is a formally defined representation scheme which allows us to express the knowledge needed to solve a problem. Unlike propositional calculus, it allows expressions to contain variables and these variables let us create general assertions about classes of entities. Although the predicate calculus is only one of the representational languages[1] used to represent symbolic knowledge, it is the most widely used representation scheme [Turner 1984].

To demonstrate standard symbolic inference that we would like to achieve with the proposed connectionist architecture, let us think of a step of simple inference in the

[1] Other representation schemes involve semantic networks, objects, scripts, conceptual dependencies, and frames [Luger & Stubblefield (1989)].

predicate calculus. If we are given a description of general knowledge, "Every man is mortal. Adam is a man.", this statement can be translated into first-order Horn clause expressions as follows. The first statement is represented by the expression

$$man(X) \rightarrow mortal(X).$$

based on the notion that "X is a man" by the predicate *man(X)*, and "X is mortal" by the predicate *mortal(X)*. The two symbols *man* and *mortal* are called predicate names and X's in the bracket are called *arguments* of the predicates. The symbol $\rightarrow$ is normally read as "implies" and called the *implication*. The second sentence, "Adam is a man", is then represented by the expression.

$$man(adam).$$

In order to draw inference based on these expressions, we need inference rules. An inference rule is a mechanical means of producing a new expression from other expressions. Two frequently used inference rules are *universal instantiation* and *modus ponens* [Luger and Stubblefield (1989)]:

- *universal instantiation* states that if any universally quantified variable in a true expression is replaced by any appropriate term from the domain, the result is a true expression. Thus, if a is from the domain of X, the $\forall X$ *p(X)* lets us infer *p(a)*. The meaning of the symbol $\forall$ is "for all" and is called the *universal quantifier*.
- if the fact p and the rule $p \rightarrow q$ are known to be true, then *modus ponens* lets us infer q.

Because the X in the rule is universally quantified, we may substitute any value (also called a *filler*) in the domain for X and it still has true statement under the inference rule of universal instantiation. Based on the known fact, *man(adam)*, we can substitute *adam* for X in the rule. As a result, we obtain the expression, $man(adam) \rightarrow mortal(adam)$. From this new expression and the known fact, *man(adam)*, we can now apply modus ponens and infer the conclusion *mortal(adam)*.

In order to apply the inference rule, modus ponens, an inference system must be able to determine when two expressions are the same or match. This requires a decision process for determining the *variable substitutions* under which two or more expressions can be made identical. In the example rule and fact, for instance, the antecedent of the rule *man(X)* and the fact *man(adam)* can be identical when we substitute *adam* for the variable X. This substitution is called a *binding* and represented by the notation $\{adam/X\}$. This binding is used later to substitute the value *adam* for all the occurrences of the variable X in the rule to obtain a new expression on which we can apply the modus ponens inference rule. A number of issues must be taken into account during the substitution procedure: (i) A problem-solving process must maintain consistency of variable substitutions. It is important that any unifying substitution is made consistently across all occurrences of the variable in both expressions being matched. (ii) Once a variable has been bound, future unifications and inferences must take the value of this binding into account. If a variable is bound to a constant, it may not be given a new binding in a future unification. (iii) Neither can two different constants be substituted for one variable [Luger & Stubblefield (1989)]. In summary, a step of forward chaining inference based on modus ponens is carried out by two sub-tasks, *matching* and *substitution*. For a given set of expressions already known, when an expression is presented, the matching sub-task tries to match between the known expressions and the expression presented. If there is any match, it produces a set of bindings. The substitution sub-task then uses these bindings to substitute all occurrence of variables in the expression matched. The resulting expression is used to infer the conclusion of inference.

2.3 Some Related Issues

The ability to produce new constituent structure from existing constituent structures is an important feature of any type of inference mechanism. When constituent structures are represented in a connectionist manner, structure-sensitive operations are needed to attain symbolic styles of inference. The previous section emphasised two such important operations, matching and substitution. Since these operations communicate each other through a common symbolic data structure, called bindings, it is required for any connectionist inference system to implement an efficient binding mechanism in a connectionist manner to support symbolic styles of inference.

2.3.1 Representing Dynamic Bindings

The weakness of standard neural networks in representing dynamic bindings was stressed by Barnden (1984), Feldman (1982), Fodor & Pylyshyn (1988), Malsburg (1986). For instance, in the following expression,

$$give(john, mary, book1),$$

the entire constituent structure cannot be represented by simply activating the nodes representing the arguments, *give_arg1*, *give_arg2*, and *give_arg3*, and the nodes representing the fillers *john*, *mary*, and *book1*. The problem is that the given expression does not merely express an association between the constituents, it also expresses a specific relation in which each constituent plays a distinct role. What is required is a connectionist representation of correct bindings between predicate arguments and fillers: {*john/give_arg1, mary/give_arg2, book1/give_arg3*} [Shastri & Ajjanagadde (1993)].

The dynamic representation of expressions should also be able to represent multiple sets of bindings with the same predicate such as *give(john, mary, book1)* and *give(andrew, sarah, car2)* with different sets of fillers without creating an undesirable ghost expression[2] such as *give(andrew, mary, book1)* [Shastri & Ajjanagadde (1993)]. In addition, attaining a chain of inference on a connectionist architecture also requires the dynamic generation of inferred expression through bindings. The importance of this is observed when a set of bindings produced as a result of one step of inference is used again for the next step of inference. In the following two expressions, for instance:

$$buy(X, Y) \rightarrow own(X, Y)$$
$$own(X, Y) \rightarrow can\_sell(X, Y)$$

starting with dynamic representation of *buy(mary, car77)* based on the bindings, {*mary/buy_X, car77/buy_Y*}, should be capable of creating subsequent inferred expres-

[2] The process which produces unwanted ghost expressions is called *cross talk*.

sions *own(mary, car77)* and *can_sell(mary, car77)* with two sets of dynamic bindings generated, $\{$*mary/own_X, car77/own_Y*$\}$ and $\{$*mary/can_sell_X, car77/can_sell_Y*$\}$, without cross talk over the network encoding these rules.

This shows that dynamic representation of bindings differs from the static representation found in Shastri & Feldman (1986) and Shastri (1988) where each binder node binds the appropriate filler to the corresponding argument and the *focal node* provides the requisite grouping between the set of bindings that make up the instantiated predicate. Although such a static binding may be suitable for representing long-term constituent structures, it is implausible for representing dynamic changes of bindings with one set of fillers and another.

Thus, the dynamic variable binding problem in a connectionist system arises when the system needs a representation of a constituent structure whose constituents (fillers) change from one step of inference to others. Any connectionist inference systems which attempt to replicate symbolic style of inference should be capable of representing a basic form of constituent structures as well as creating and destroying the relationships between constituents dynamically without decomposing the pre-established structure of the network.

2.3.2 Knowledge Representation Issues

Even if we use a particular connectionist approach to represent constituent structure and dynamic bindings, building a connectionist inference system requires additional solutions to further knowledge representation issues [Sun (1992)] which arise in replicating standard symbolic inference. A difficulty is that representing symbolic knowledge does not simply encode expressions into corresponding networks. It also needs how to implement some symbolic constraints that the syntax of expression imposes in a connectionist manner.

The first representation issue is representing and distinguishing constant and variable bindings. The match procedure of a symbolic inference system generates two types of bindings: a constant binding and a variable binding. A constant binding refers to the binding between a constant and variable and a variable binding to a binding between variables. For instance, if the expression, $p(a, U, V)$, is presented to the following expression,

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

the bindings, $\{a/X,\ U/Y,\ V/Z\}$, will be obtained as the result of the matching procedure. In symbolic systems, the constant binding $\{a/X\}$ is easily differentiated from the variable bindings $\{U/Y, V/Z\}$, and also one variable binding from the other. The unique symbolic names make each binding differ from others. When this rule is encoded into a network, however, both types of bindings are equally represented as activation patterns over neural elements. Nevertheless, a connectionist inference system must provide a connectionist mechanism to represent these bindings and to distinguish one binding from others.

Secondly, consistency checking within a group of unifying arguments or across groups of unifying arguments during inference is a problem in a connectionist system. When expressions which have constant or repeated variable arguments are encoded, the following consistency checking is required

- any constant argument should get bound to the constant filler which must have the same name as the constant argument;
- any repeated variable arguments should get bound to the same constant fillers or free variables.

This is due to the fact that a variable is not allowed to get bound to two different constants at the same time. In particular, if there are more than one groups of unifying arguments, more complicated consistency checking is necessary to draw appropriate inference. For instance, if we have the following expression

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

presenting the expression, $p(a, U, U, V)$, should force consistency not only within each repeated argument, X's and Y's, but also between them, to produce the desirable expression, $q(a,a)$, as the result of inference.

Another issue is how to represent a structured term as an argument of various expressions because terms of first-order predicate calculus allow any depth of nested structured terms in formulas.

2.4 Connectionist Inference Architecture Survey

Until now, many connectionist systems have been proposed to provide solutions to the dynamic binding problem and to build a connectionist inference architecture on the basis of those solutions. This section summarises eight connectionist models from the following point of view:

- what is the basic architecture?
- what is a solution to the dynamic binding problem?
- how is constituent structure represented?
- what are the advantages and disadvantages?

2.4.1 Distributed Connectionist Production Systems

Distributed connectionist production system (DCPS) is a connectionist interpreter described by Touretzky & Hinton (1988) that uses *coarse coded* representation [Hinton et al. (1986)] to represent variable bindings. Their system is composed of five groups of cells called *spaces*: two spaces for working memory and production rules; one for the bindings; and the remaining two for clauses. The basic unit of constituent structure used for each space is a triple of terms which corresponds to frame-slot-filler combinations.

Rules used in DCPS have the form

$$T_1\, T_2 \rightarrow -T_3 - T_2 + T_4$$

where each T_i is a triple of terms. The antecedent of each rule only consists of two triples of terms which are either ground or of the form (x A A) and (x B B) where the capital letters are constants and x is a variable. A variable only appears as the first argument of each triple and only one variable is allowed in each rule. Each rule is represented by a collection of 40 rule units and each rule unit positively connected to the clause spaces. The 40 units comprising the rules form a clique. Since units inhibit others outside their clique, the rule space is organised as a *winner-take-all* network [Feldman & Ballard (1982)]. When the network is settled, the units representing one rule become active and the other units stay inactive. When the antecedent of a rule is matched, the triples on the consequent of the rule are either added (+) to or deleted (-) from working memory.

The working memory consists of 2000 binary state units. Each unit has a receptive field table and its receptive field is defined to be the cross-product of the six symbols in each of the three columns. This gives 216 tuples (6^3) per field. Since the triples are built out of 25 symbols, there are $25^3 = 15{,}625$ distinct triples. When these triples are distributed among the units in the working memory, each triple is recognised by approximately 28 receptive fields ($6^3/25^3 * 2000$). This means that a triple is stored in the working memory by activating all units whose receptive fields contain this triple. Multiple triple bindings are represented by superimposing their receptive fields. Due to the coarse coding used, the units represent other triples as well. This exhibits local blurring of closely related triples if too many triples are stored at the same time. An activated triple can be deleted from the memory by inhibiting all units whose receptive fields contain this triple. If many triples are deleted, this may cause other stored triples to be deactivated since the units present more than one triple.

The two clause spaces C1 and C2 are exact copies of the working memory. They also consists of 2000 units and each unit in the clause space is connected to the corresponding cell in the working memory. The units in clause space inhibit one another to limit the total number of units that can be activated at the same time to 28. This is because each clause space is designed to hold only one triple at a time. These two clause spaces are used to represent two triples of the antecedent of a rule.

To execute a rule firing cycle, DCPS first performs energy minimisation to find a rule whose left-hand side matches two of the triples in the working memory. When a rule has a variable, x, the rule is enabled if the system contains the first triple of the antecedent in C1 and the second triple in C2. If the first symbol of the both triples matched in C1 and C2 are the same, then variable x will get bound to the symbol. This binding is realised in the bind space. Since bind space contains a total of 333 units, each symbol takes the receptive field of 40 bind units (3/25 * 333). And the binding x to a unique symbol (the first symbol) is represented in a situation where a set of 40 units in the bind space which vote for a particular symbol as the value of the bound variable become active. Once a rule is settled, gated connections from the rule space add and remove receptive fields from the working memory to execute the consequent of the rule. The cycle is then repeated for another match-execute cycle.

Since DCPS can only deal with single variable rules and allows only two triples of terms on the antecedent, it has limited expressive power. Each step of reasoning process involves one rule like a traditional production system, i.e. the system performs sequential rule-based reasoning and parallel firing of rules is not allowed. The system has very complex structures and is costly in terms of computational resources, but it uses only very simple node functions. The rules encoded can be used many times in different inference cycles.

2.4.2 Tensor Product Production System

Dolan & Smolensky (1989)'s Tensor Product Production System (TPPS) provides a systematic and principled approach to representing and processing structured data and variable binding in connectionist networks based on tensor products [Smolensky (1987, 1990)].

The tensor product can be considered as a generalisation of the outer product of two vectors. When used for variable bindings, arguments and constants are viewed as a n and m dimensional vectors and argument-constant binding is viewed as the $n * m$ dimensional vector obtained by taking the tensor product of the appropriate argument and constant vector. Basically, each combination of variables and values is represented in the system by a simple node, or a group of them, which detects dynamically if the right binding is present. If the tensor product of three vectors is used, the data structure such as a frame s with slots r_i and fillers f_i, (s, r_i, f_i), can be represented by $s * r_i * f_i$. Using a network of $n * m$ nodes, a tensor product based variable mechanism can only encode $n * m$ bindings without cross-talk.

TPPS was built to show that the tensor product technique can be effectively applied to the inference task achieved by DCPS. Unlike DCPS, TPPS uses linear working memory units where superimposed representations are numerically added together. Also TPPS's clean-up units perform a function similar to the DCPS bind space but its size is smaller than DCPS's bind space. Since TPPS does not use clause spaces, it is almost a feed-forward network. Feed-forward networks are easy to design using tensor representation because the connection patterns are given by simple tensor algebra expressions. This makes both design and analysis of TPPS much easier than with custom coarse codings, as is the case in DCPS. Although the tensor product utilises simple node functions, its network can be of high computational complexity.

2.4.3 CHCL

CHCL [Hölldobler & Kurfeß(1991)] is a connectionist inference system which can handle first-order Horn clauses. CHCL consists of 6 layers: the spanning set layer, connection layer, reduction layer, term layer, unification layer, and occur check layer. Computing elements in each layer are simple neuron-like units which perform thresholding operations. They are interconnected through weighted links. The terms of the formula are represented in the term layer and the connections of the formula are represented in the connection layer. By clamping on certain input units, the spanning set layer first gets activated and then this activation will spread through the connection layer and the unification layer until certain output units are activated indicating whether a proof of the formula has been found or not.

The major technique for variable binding within CHCL relies on a unification algorithm which computes the most general unifier of two first-order terms [Hölldobler (1990)]. The overall strategy pursued by CHCL is to identify so called spanning matings [Bibel (1987)] in the formula under investigation. A spanning mating defines a proof iff all connected literals [Stickel (1987)] are simultaneously unifiable. To compose a solution, the terms in these candidate matings have to be unifiable, i.e. finding a spanning mating corresponds to deciding the satisfiability of the propositional structure of the given formula. The search space can be reduced with the help of techniques such as removal of non-unifiable connections, useless connections, and solved connections. CHCL reduces a formula, generates the spanning sets one-by-one and simultaneously unifies all connected literals in such a set in parallel. Although the time required for the core unification algorithm is linear to the size of the terms the number of nodes required is quadratic to the size of terms.

Browne & Pilkington (1994b) implemented Holldobler (1990)'s style of unification using a distributed representation. They used Hölldobler's scheme to produce a symbolic representation for terms to be unified, and have used an auto-associative network to produce a distributed representation. They constructed the network to accommodate two-argument compound terms such as $f(X, Y)$ or $f(a(X), Y)$ which consist of two constants a and f and two variables X and Y. A training and test set were chosen from a sets of 9216 two-argument terms which was generated by taking all possible permutations of the numbers of given alphabets, with the restriction that only f appears at the functor position of a compound term. Their model has shown that a complex structure-sensitive process such as unification can be performed on a distributed representation. However, their model currently performs only a single resolution step and needs more connectionist mechanisms to support a complete inference system.

2.4.4 ROBIN

Lange & Dyer (1989a, 1989b) describe a connectionist system, ROBIN, capable of performing high level inferences over structured connections of nodes that encode world knowledge in semantic networks. It uses structured connections of nodes to encode a semantic knowledge base of related frames. Each frame has one or more role, with each role having expectations and logical constraints on its fillers. Every frame can be related to one or more other frames, with path ways between corresponding role for inferencing. When constructing a structured semantic network, a special node called a *signature node* is attached to each role of a frame-like concept. During inference ROBIN permanently allocates a unique signature to each concept. Distinct concepts have different signatures permanently allocated. Signatures are activation values or patterns emitted by the signature nodes. Dynamic role-filler associations are achieved by activating a role's biding node and filler's signature nodes with the same activation value. Therefore it can maintain a large number of dynamic bindings and deals with rules having multiple variables. Their model can deal with issues such as concurrent multiple bindings, cyclic connections, and global inhibition. It also supports knowledge level parallelism. A problem with the use of signatures is that the signatures must be high precision quantities if each entity is to have a unique signature. Thus propagating bindings will require that nodes propagate and compare high precision analog values.

2.4.5 COMPOSIT

Barnden(1989) presents COMPOSIT, a connectionist rule-based system that performs syllogistic reasoning based on some core aspects of Johnson-Laird's mental model theory [Johnson-Laird & Bara (1984)]. COMPOSIT uses a scratch pad of registers called Configuration Matrices which hold the short-term data structures on which production rules act. Such a matrix consists of 32×32 registers, each register associated with a symbol and a vector of binary flags. The symbols denote classes, individuals of classes, or situations like overlapping classes. The flags denote relationships between neighbouring registers, for example, the member relationship between a register representing an individual and a register representing a class. The patterns are associated by means of the relative position and similarity of these registers which contain those patterns [Barnden & Srinivas (1991)]. The pattern similarity association refers to a technique where the use of several occurrences of the same symbol in different registers achieves a linking power similar to linking by associative addressing in computers. The relative position encoding refers to a technique to put things into association with each other rapidly and flexibly by putting them next to each other.

COMPOSIT uses production rules to manipulate a configuration matrix. A production rule consists of a condition part that tests for the presence or absence of specific sorts of state configuration in the configuration matrix. The action part of a production rule changes the symbols and/or highlights states of registers for dynamic variable binding and propagation. The registers to be changed are chosen on the basis of their current highlighting or symbol states and the state of their immediate neighbours. By applying production rules, syllogistic inferences are made.

In a neural-net implementation of COMPOSIT, each Configuration Matrix (CM) register is implemented as a neural subnetwork. The rest of the circuitry in the register is concerned with the registers reaction to command signals. Each CM register network is connected to the subnetworks for immediately neighbouring registers and used to represent the sensitive to neighbouring register state. Each register subnetwork is also connected to the so called parallel distributor, which receives command signals from nodes corresponding to rule action parts. This parallel distributor uses a temporal winner-takes-all contention resolution mechanism [Barnden & Srinivas (1990)] to perform the arbitrary selection among registers satisfying the symbol or highlighting conditions specified in a command signal.

To summarise, COMPOSIT can manipulate complex symbolic data structures and constrained variable bindings and this allows COMPOSIT to deal with the rule: if A loves B, B loves C, and C is not A, then A is jealous of C. Although the structure of COMPOSIT is relatively simple, but the symbolic manipulation necessary to match rules against data is a complicated process. As a result of the matching process, parallelism in reasoning is not fully captured. The use of an interpreter and control signals that drive the operation of the system makes COMPOSIT basically a sequential rule-based system.

2.4.6 CONSYDERR

Sun (1992) has proposed the network model, called CONSYDERR, for commonsense reasoning. It consists of two sub-systems one of which uses localist representation, the other distributed representation. The localist sub-system is used mainly for representing rules and concepts and the distributed sub-system mainly for similarity-based reasoning, to supplement and enhance rule-based reasoning mechanisms. Each node in the localist sub-system is connected to all the relevant feature nodes in the distributed sub-system. Similarity matching is achieved by the interaction between these two sub-systems. During inference, activation of a node in the localist sub-system will be firstly propagated to related nodes in the distributed sub-system. After a settling down phase, some activation of nodes in the distributed sub-system goes back to the other nodes in the localist sub-system.

Rules and variable bindings are dealt by the localist sub-system. Predicates of a rule are represented using an assembly of nodes interconnected, one of which is a predicate node used for computing and storing values of the predicate and the rest are used for arguments. Thus a predicate with k arguments is represented with one predicate node and k argument nodes. To encode a rule, the antecedent and consequent predicates have to be built using an assembly of nodes and then the predicate node of the antecedent predicates is connected to that of the consequent predicate. Also each argument node in the antecedent predicate is linked to the corresponding argument node in the consequent predicate. A binding between a constant and an argument of the antecedent predicate is established by assigning the same numerical value to them. Propagation of bindings from the antecedent to the consequent is carried out by passing these values from an argument node to the node representing the argument having the same name in the consequent.

Unlike other connectionist inference systems, CONSYDERR's localist sub-system can deal with a number of important issues in dynamic bindings such as representing a variable binding[3] – a situation where an argument gets bound to a variable – consistency checking, unification, and function terms. However, the need for high computational ability of the assemblies and the use of abstraction which hides network details is beyond simple summation with thresholding and so demands much more sophisticated neuron elements than we usually require.

2.4.7 SHRUTI And EXCON

Shastri & Ajjanagadde (1993) presented a connectionist model, called SHRUTI, which performs backward and forward chaining inferences based on variable bindings handled by temporal codings. They described in detail a technique for using simple *neuron-like* elements to encode rules and facts involving n-ary predicates with variables. A key part of their mechanism to solve the dynamic binding problem lies in matching rhythmic patterns of activity, temporal synchrony, between two nodes. A node presenting an argument produces a temporal activation pattern (a spike train) with a specific phase that can be synchronised with a node representing a particular constant. This mechanism efficiently solves the dynamic binding problem. Network circuits for

[3] In CONSYDERR, this is called a pseudo binding.

rules and facts are devised to enable such a synchronisation dynamically and this allows very fast inference. SHRUTI's network has relatively simple structures but needs nodes with temporal properties, capable of detecting the activation phase. And it has some limitations in dealing with knowledge representation issues which are important in enabling their model to achieve better symbolic processing capabilities. Chapter 3 gives a more detailed description of their system.

Grant (1991) and Rohwer et al. (1992) implemented a core part of SHRUTI using a distributed representation rather than a localist representation as is the case in SHRUTI. Their system, called EXCON, was built on a backpropagation through time model [Rohwer (1991)] which allows processing to take place through time split into a specified number of distinct phases. To encode knowledge, EXCON transforms rules and facts written in a constrained variety of first order logic into a neural network weight matrix which encodes target knowledge. Further training of networks is possible using large data sets representing input queries and the desired responses from these input. This allows the rules encoded in a network to be modified in such a way as to encapsulate new rules reflected in the training data that were not previously catered for in the original encoded knowledge. Once the network has been built, EXCON can be queried in the traditional expert system fashion. Since EXCON is a limited implementation of SHRUTI, it has limited expressive power in the form of rules allowed. It also needs to be extended to be able to deal with knowledge representation issues.

2.5 Comparisons

2.5.1 Regarding Representation Adopted

Distributed connectionist models represent knowledge as patterns of activation across nodes, rather than the single unit representation of individual concepts as is the case in localist connectionist models. Each of the distributed connectionist models, categorised in Table 2.1, uses the energy minimisation metaphor to represent individual variable bindings or rule firings.

Each of these models has successfully demonstrated that distributed connectionist models have the ability to represent and use explicit rules. Furthermore, their use of dis-

	Representation Adopted	Parallelism in Rules	Inference Speed
DCPS	distributed	no	very slow
TPPS	distributed	no	slow
CHCL	localist	no	slow
ROBIN	localist	yes	fast
COMPOSIT	localist	no	slow
CONSYDERR	both	yes	fast
SHRUTI	localist	yes	very fast
EXCON	distributed	yes	fast

Table 2.1: Comparison regarding implementation mechanisms

tributed representations allows their models to have robustness to damage and noise-resistant associative retrieval. The primary problem with each of these distributed connectionist models is that although they select their rules through massively-parallel constraint satisfaction, they actually behave serially at the knowledge level because they select and fire only one rule at a time. Although one exception is found in EXCON (based on SHRUTI's original localist model) most current rule-based distributed models are serial at the knowledge-level. They still exhibit many of the same problems that traditional symbolic rule-based systems have. This becomes a major problem when the tasks are complex and involve high level inferencing such as language understanding and planning tasks which generally require exploration of many possible inference paths in parallel. Other problem of distributed connectionist models involves lack of the ability in representing constituent structure needed to handle complex conceptual relationships [Feldman 1989].

Localist connectionist systems, on the other hand, represent structural relationships between concepts. They represent knowledge by simple nodes and their weighted links, with each node standing for a distinct concept. Activation of a conceptual node represents the amount of evidence available for that concept in the current context. Since multiple inference paths can be pursued simultaneously through these networks, localist connectionist systems are able to fire many rules at once, i.e. they are parallel at the knowledge level. This dramatically decreases the time required to search through the rule space to find inference paths connecting the inputs. Two localist systems, CHCL and COMPOSIT do not allow parallelism in rules. This is because CHCL is built based on a connectionist unification algorithm which computes the most general unifier of only two first-order terms at a time, while COMPOSIT is a neural net implementation of a serial rule-based system which manipulates a short term, complex symbolic data structure. The main problems with localist connectionist models are difficulties in learning new rules and weakness to damage.

2.5.2 Regarding Structure of Networks

Addition comparisons in relation to structures of networks are summarised in Table 2.2.

	Node Functional Requirement	Network Scalability	Inference Type
DCPS	simple	bad	forward
TPPS	simple	bad	forward
CHCL	simple	bad	backward
ROBIN	simple	good	forward
COMPOSIT	simple	bad	forward
CONSYDERR	complex	good	forward
SHRUTI	simple+temporal	good	both
EXCON	simple	bad	backward

Table 2.2: Comparison regarding network structure and complexities

Apart from CONSYDERR and SHRUTI, all connectionist systems use very simple neuron elements that perform simple computations on their inputs: summation, summation with thresholding and decay. CONSYDERR uses not only these simple elements but also another two types of elements which perform selection of one binding out of many or constraint checking and/or unification and also binding generation when necessary. SHRUTI uses simple elements with temporal oscillation properties.

As can be seen, scaling up the networks to accommodate more knowledge is difficult in distributed connectionist systems. To encode large amounts of knowledge such as knowledge related to commonsense, these models should cope with the scale-up problem that most of traditional distributed connectionist models suffer from [Güsgen & Hölldobler (1991)]. Localist connectionist systems are relatively more flexible. Since each node stands for a distinct concept and the connectivity of structured networks implicitly represents structural relationships between concepts, more concepts can be added to the network using additional nodes representing new concepts and additional links.

As for the types of inference supported, only SHRUTI can perform both backward and forward inferences. CHCL and EXCON only perform backward chaining, while the other systems perform only forward chaining.

2.5.3 Regarding Expressive Power

The abilities of connectionist systems to deal with some knowledge representation are summarised in Table 2.3.

	Representation of Variable Bindings	Consistency Checking	unification	Function Terms
DCPS	no	no	yes	no
TPPS	no	no	yes	no
CHCL	no	yes	yes	yes
ROBIN	no	no	no	no
COMPOSIT	no	no	no	no
CONSYDERR	yes	yes	yes	yes
SHRUTI	no	limited	limited	limited
EXCON	no	no	limited	no

Table 2.3: Comparison regarding ability in dealing with some issues

CONSYDERR addresses to all these issues and proposed solutions for them. It can represent variable bindings and deals with consistency checking and unification as well as function terms in rules. However, CONSYDERR handles these issues not by a uniform connectionist mechanism but by different types of complex network elements which are hypothesised to have such a function (see Section 3.2 of Sun(1992)).

CHCL can also deals with consistency checking and unification implicitly by a connectionist unification algorithm [Holldobler (1990)] adopted. Its representation scheme of terms using a set of position-label pairs can also easily represent function terms in the initial stages of inference and variable bindings if any are generated during inference.

SHRUTI performs limited consistency checking and unification but can not represent full variable bindings. A preliminary solution to incorporate function terms in SHRUTI is suggested in Ajjanagadde (1990). EXCON can perform limited unification but can not handle consistency checking, function terms, and variable bindings. DCPS and TPPS only have built-in unification mechanisms in restricted forms of rules because of the distributed representation they use. It is not clear how other connectionist models deal with these issues.

2.6 Summary

Eight existing connectionist inference systems are surveyed in this chapter. They employed various mechanisms and strategies to address these issues to achieve symbolic inference. Among them, the three localist connectionist systems, ROBIN, CONSYDERR, and SHRUTI, provide the most efficient connectionist mechanisms. The biggest advantage of these systems is that the time taken for a chain of inference is only proportional to the size of the network, rather than to the number of rules encoded. CONSYDERR and SHRUTI can perform logical reasoning, whereas ROBIN carries out reasoning based on frame-like knowledge. Although CONSYDERR appears to have more expressive power in dealing with knowledge representation issues, the need of high computational ability of the assemblies and the use of abstraction which hides network details is beyond simple summation with thresholding. SHRUTI, on the other hand, uses simpler neuron elements and its network can do both types of reasoning, forward and backward chaining, with a very efficient dynamic binding mechanism. But it also shows weakness in dealing with knowledge representation issues. In Chapter 3 we explore the advantages and disadvantages of SHRUTI further to get more detailed design criteria for a connectionist inference system for logical inference.

Chapter 3

Temporal Synchrony Solution and SHRUTI

3.1 Overview

This chapter describes in more detail SHRUTI's temporal synchrony solution to the dynamic binding problem. Firstly, the principle of temporal synchrony will be summarised from the following points of view:

- the types of neural element used;
- how to represent constituent structures involving symbolic concepts and n-ary predicates;
- how to represent dynamic bindings over constituent structures.

Secondly, rule and fact encoding mechanisms of SHRUTI will be outlined by showing the encoding procedures using some example rules and facts. Two types of inference are also exemplified. Finally, this chapter discusses some advantages and limitations of SHRUTI.

3.2 Temporal Synchrony Solution

Shastri & Ajjanagadde (1993) observed that the existence of rhythmic, temporal by synchronous activity in the animal brain suggested one solution to the dynamic binding problem in connectionist systems. The fundamental features of the temporal synchrony solution to the dynamic variable binding problem lies in a separation of an inference cycle (an oscillation cycle) into several phases and the use of phase-sensitive neuron-like elements.

3.2.1 Phase-Sensitive Neuron Elements

Four different types of neuron-like elements are used to represent symbolic knowledge in this approach. The behaviours of these elements are differentiated based on the window of time over which they sample their inputs and the width of their output pulse.

A *ρ-btu element* becomes active on receiving a periodic spike train and produces a periodic spike train that is in-phase with the driving input. In particular, if node A is connected to node B then a periodic firing of A leads to a periodic firing of B that is in-phase with the firing of A. They assume that ρ-btu nodes can respond in this manner as long as the period of oscillation, π, lies in the interval $[\pi_{min}, \pi_{max}]$. This interval can be interpreted as defining the frequency range over which ρ-btu nodes can sustain oscillations. A threshold, n, associated with a ρ-btu node indicates that the element will fire only if it receives n or more synchronous inputs.

A *τ-and element* becomes active on receiving an oscillatory input consisting of a train of pulses of width comparable to the period of oscillation. A τ-and element behaves like a temporal "*and*" element that becomes active if it receives uninterrupted input over a period of oscillation. On becoming active, a τ-and element produces an oscillatory pulse train whose period of oscillation and pulse width matches that of the input. A threshold, n, associated with a τ-and element indicates that the element will fire only if it receives n or more pulse trains of width comparable to the period of oscillation.

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 oscillation, π. A threshold, n, associated with a τ-or element indicates that the element will fire only if it receives n or more inputs in the same phase.

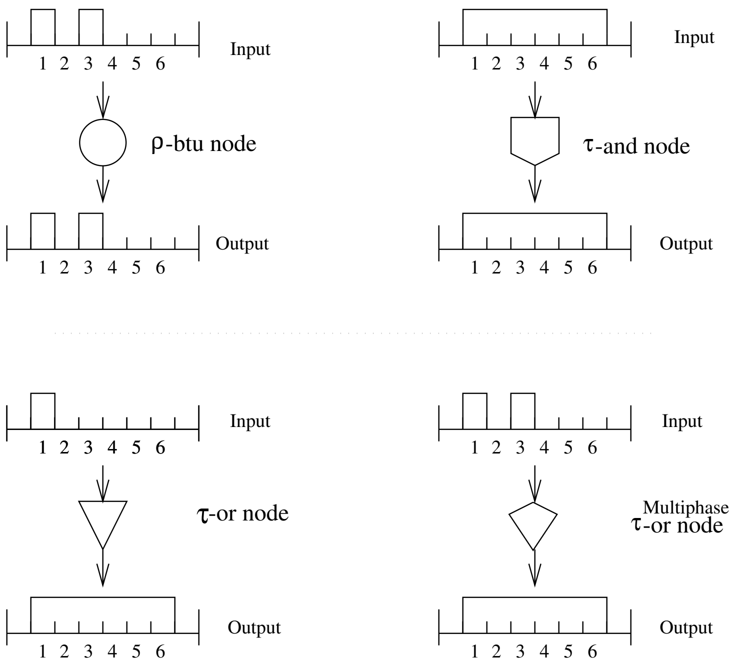


Figure 3.1: The functional behaviours of neuron-like elements

A *multiphase τ-or element* becomes active on receiving more than one inputs 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 oscillation, π. A threshold, n, associated with a multiphase τ-or element indicates that the element will fire only if it receives n or more inputs in the same phase.

Figure 3.1 depicts the temporal behaviour of these elements. All link weights and all thresholds of elements are 1 unless otherwise specified. The number of phases in one oscillation cycle is assumed to be 6 for the purpose of explanation. When a ρ-btu element, depicted with a circle, receives one spike signal in an oscillation cycle, it produces a periodic spike train that is in-phase with the driving input. A τ-and element is drawn with a pentagon. It behaves like a temporal *and* element that becomes active only if it receives uninterrupted input over a period of oscillation. A τ-or element drawn with a triangle shape becomes active when it receives one or more spike signals within a period of oscillation. A *multiphase* τ-or element drawn with a diamond shape, on the other hand, becomes active only when it receives more than two spike signals in different phases within a period of oscillation. Because of this temporal behaviour, the multiphase τ-or node can be used to check if an input signal contains more than two spikes in each oscillation cycle. On becoming active, both of them produce an oscillatory pulse train whose pulse width is comparable to the period of oscillation.

On becoming active, each element may continue to oscillate as long as it is under the current focus of attention for a chain of inference. Otherwise, its oscillation decays in a short time unless it is used for another chain of inference.

3.2.2 Representing Entities and N-ary Predicates

One of the basic components of symbolic knowledge is an entity which represents a concept. Connectionist systems use nodes and links to encode domain knowledge. In the temporal synchrony approach, each constant entity is represented using a ρ-btu node. The entities, "*book1*", "*john*", and "*mary*", for example, are represented using the dedicated ρ-btu elements as shown in Figure 3.2. An n-ary predicate, on the other hand, is represented using an assembly of nodes: two τ-and elements and n ρ-btu elements. These two special τ-and elements are referred to as the *collector* (c) and the *enabler* (e) of the predicate. The role of these two elements will be explained in the later section of this chapter. The n arguments of the predicate are then represented using n ρ-btu elements, called argument nodes. For example, the ternary predicate *give* is represented by the three ρ-btu elements, labelled *arg1*, *arg2*, and *arg3* as shown in figure 3.2. As can be seen, the binary predicate *own* is also described using another assembly of nodes in the figure.

3.2.3 Representation of Dynamic Bindings

When each entity and predicate assembly is represented using neuron elements, dynamic bindings are represented on these nodes by synchronous firing of related nodes. With reference to the nodes in Figure 3.2, the dynamic representation of the bindings {*john/give_arg1, mary/give_arg2, book1/give_arg3*} is established by activating *john* and *give_arg1* in the first phase, *mary* and *give_arg2* in the second, and *book1* and *give_arg3* in the third phase. This activation will lead rhythmic pattern of activity shown in Figure 3.3a and this is considered as the dynamic fact *give(john,mary,book1)*.

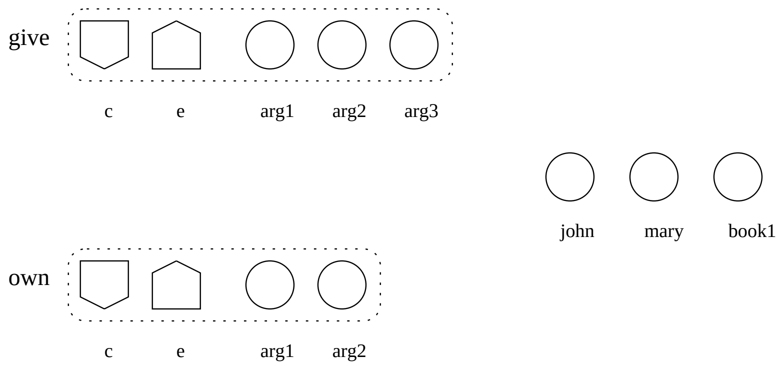


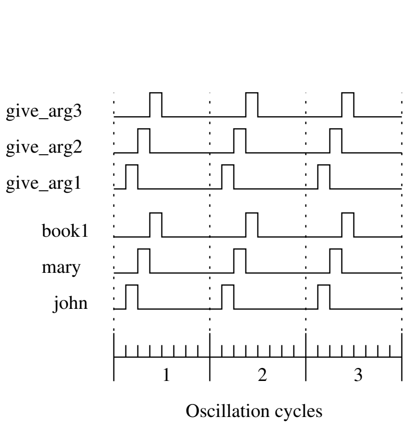
Figure 3.2: The representation of entities and n-ary predicates

The additional dynamic fact *own(mary,book1)* is obtained by simply activating *own_arg1* node in the second and *own_arg2* node in the third phase. Then all these relevant synchronous activities will represent multiple dynamic bindings: {*john/give_arg1, mary/give_arg2, book1/give_arg3, mary/own_arg1, book1/own_arg2*} as shown in Figure 3.3b.

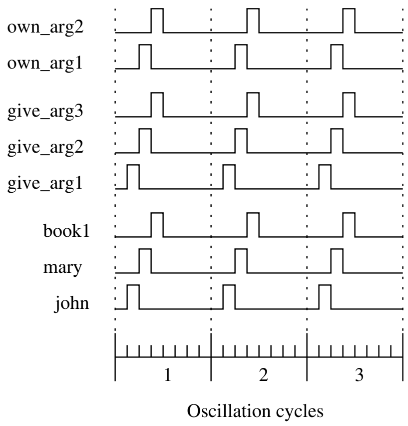
One important aspect of this approach is that the number of distinct phases within an oscillation cycle only equals to the number of distinct entities participating in the dynamic bindings. This does not depend on the total number of dynamic bindings being represented by the activation pattern. The number of distinct entities that participate in the bindings at the same time is limited by the ratio of the period of the rhythmic activity, π, and the width of individual spikes, ω. The width of a phase, ω, and the period of inference cycle, π, are limited to those indicated by physiological measurements (see Section 3.5.1).

3.3 SHRUTI

Based on the temporal synchronous approach to the dynamic binding problem, Shastri & Ajjanagadde (1993) developed a connectionist inference system, called SHRUTI, which encodes rules and facts involving n-ary predicates into networks and performs interesting class of inference. They suggested algorithms to encode rules of the form



(a) Single dynamic binding



(b) Multiple dynamic bindings

Figure 3.3: The phasic representation of dynamic bindings

$$\forall X_1, \ldots, X_m \; p_1(\cdots) \wedge p_2(\cdots) \wedge \cdots \wedge p_n(\cdots) \rightarrow \exists Z_1, \ldots, Z_l \; q(\cdots),$$

where $p_i(\ldots)$'s are the antecedent of a rule whose arguments are elements of $\{X_1, X_2, \ldots, X_m\}$ and $q(\ldots)$ is the consequent of a rule whose arguments are either elements of $\{X_1, \ldots, X_m\}$, or elements of $\{Z_1, \ldots, Z_l\}$, or constants. They also provided an algorithm to encode a fact of the form $p(t_1, t_2, \ldots, t_k)$ where t_i's are either constants or distinct existentially quantified variables.

3.3.1 Encoding Rules

Each symbolic rule is divided into two parts, the antecedent and consequent. The distinction between them is made by the symbol $\rightarrow$. Encoding each symbolic rule consists of two stages. Firstly, the antecedent and consequent predicates of a rule are represented using corresponding predicate assemblies. These predicate assemblies are used to represent the bindings between constant fillers and their argument nodes using synchronous activation. Secondly, an intermediate mechanism is constructed between the antecedent predicate assembly and the consequent predicate assembly according to the types of conditions that the rule has to enforce to achieve desirable inference.

An intermediate mechanism is composed of links and neuronal elements and provides a means of propagating bindings from one predicate assembly to an other. Links are directional and constructed in such a way that they propagate bindings either from the antecedent predicate assembly to the consequent predicate asembly or vice versa. This directional property decides the types of inference that the network can support: forward chaining or backward chaining. In order to support both types of inference, two independent intermediate mechanisms have to be built.

Figure 3.4 shows one of the simplest intermediate mechanisms constructed to encode the following rule to support forward chaining:

$$\forall X, Y, Z \; give(X, Y, Z) \rightarrow own(Y, Z).$$

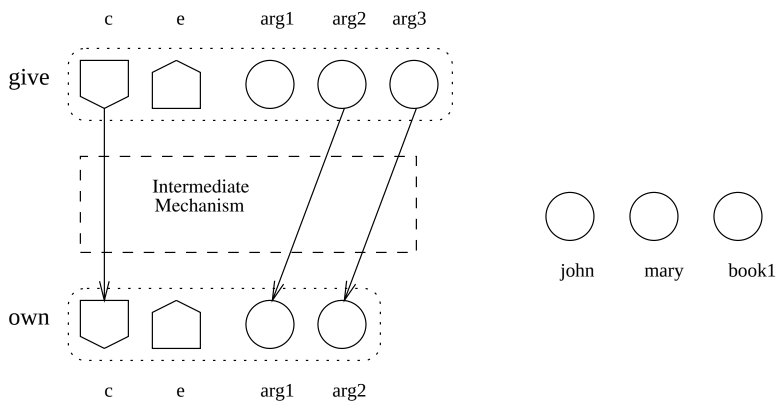


Figure 3.4: Encoding of the sample rule, $\forall X, Y, Z$ *give(X, Y,Z)* $\rightarrow$ *own(Y,Z)*

Since the example rule only requires simple binding propagation from the antecedent to the consequent, additional sub-mechanisms are not necessary. Only simple links are needed as the intermediate mechanism. The enabler of a predicate assembly is used to indicate the activation of the predicate assembly. It becomes active whenever the system is queried about the predicate assembly. On the other hand, the collector of a predicate assembly is used to indicate that the dynamic bindings of the arguments of the predicate assembly are recognised as an intermediate result or a result obtained during a chain of inference.

For example, when the fact *give(john,mary,book1)* is presented to the system, it establishes the initial bindings {*john/give_arg1, mary/give_arg2, book1/give_arg3*} on the *give* predicate assembly by activating the constant nodes (*john*, *mary*, and *book1*) and their corresponding arguments in distinct phases. The collector of the *give* predicate assembly (*give_c*) will also be activated to indicate the situation that the predicate assembly are active with proper binding set up.

Apart from the simple rule shown above, SHRUTI can also encode other types of rules which require special treatments to force special conditions that may be specified by the syntax of the rules. These types of rules need more complicated intermediate mechanisms which involve not only links but also some additional neuron elements. The types of rules which require special treatments are as follows (for more details see Section 4.5 and 4.6 of Shastri & Ajjanagadde(1993)):

- rules with constants and existentially quantified variables in the consequent;
- rules with repeated variables in the consequent;
- rules with multiple antecedent predicates.

Their rule encoding mechanism conceptually creates a directed inferential dependency graph: each predicate argument is represented as a node and each rule is represented by links between nodes denoting the arguments of the antecedent and consequent predicate assemblies. Propagation of bindings from one predicate assembly to others to achieve a chain of inference corresponds to a parallel breadth-first traversal of the directed inferential dependency graph.

3.3.2 Encoding Facts

Facts are a permanent record of a set of bindings describing a particular situation. The connectionist representation of a fact should encode the bindings pertaining to the fact in a way that allows the connectionist inference system to rapidly recognise dynamic bindings that match the encoded fact. Figure 3.5 illustrates the fact encoding mechanism introduced by Shastri & Ajjanagadde (1993) with the fact *give(john,mary,book1)*.

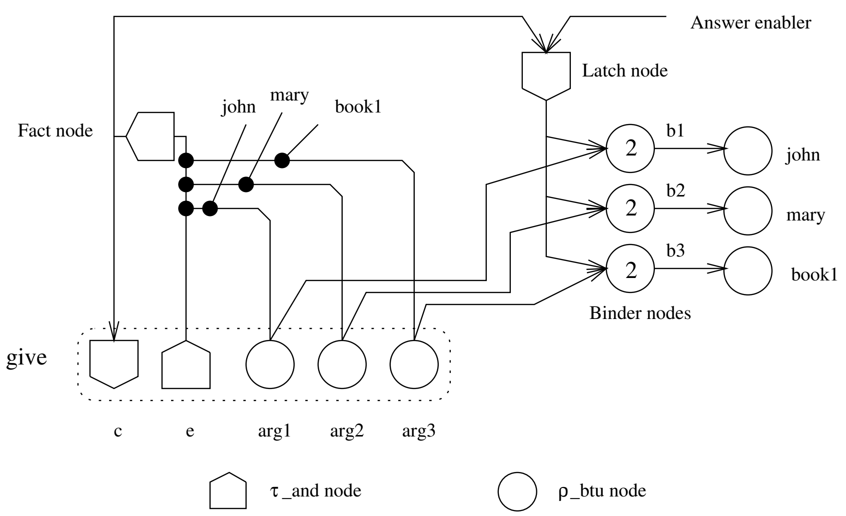


Figure 3.5: Encoding of the sample fact, *give(john,mary,book1)*

A fact is encoded using a τ-and element, called a fact node, which receives an input from the enabler of the associated predicate assembly. This input is modified by inhibitory links from argument nodes of the predicate assembly. If an argument is bound to an entity, the modifier inputs from the argument nodes are in turn modified by inhibitory links from the appropriate entity nodes. The output of the fact node is connected to the collector of the associated predicate assembly. All the facts which have the name, *give*, with three arguments will be associated with the the same predicate assembly, *give*. The additional mechanisms, the latch node and the binder nodes, are introduced for a special type of query processing which will be explained in a later subsection.

This fact encoding mechanism enables a fact node to be active only when the static bindings it represents match with the dynamic bindings represented in the network's state of activation. The *give_e* in the network shown in Figure 3.5 becomes active whenever any query involving the predicate *give* is presented to the system. On becoming active, *give_e* projects an output to the fact node, which will be in turn modified by an active argument of *give* unless each filler of this argument specified by the fact is firing in synchrony with the corresponding argument. The fact node will be activated only when there is no inhibitory signal coming from the argument nodes.

3.3.3 Performing Inference

Once a set of symbolic rules and facts are encoded into a network, SHRUTI can perform either forward chaining or backward chaining styles of inference over the network.

Forward chaining is started by presenting a predicate to the system. The presented predicate has the form $p(t_1, t_2, \ldots, t_m)$, where t_i's are either constants or existentially quantified variables. Presenting a predicate to the system involves specifying the predicate and the argument bindings of the predicate to the antecedent predicate assembly of the encoded rule. The effect of specifying the query predicate to the system is achieved by activating the collector of the corresponding predicate assembly in the network. The effect of setting up the initial bindings is obtained by assigning the phase delay δ to fillers of the presented predicate and their corresponding argument nodes in the network. The δ is defined $\lfloor \pi/n \rfloor$, where n is the number of distinct constant fillers appearing in the presented predicate. If t_0 is assumed to be the starting point of π, the phase $t_0 + \delta_i$ will be assigned to the ith distinct constant filler and its corresponding argument node. The value of δ decreases as the presented predicate has a larger number of distinct constant fillers. However, δ can not be smaller than ω which is the minimum width of the phase in which a single entity is represented in the temporal synchrony approach. After the initial bindings are set up on the antecedent predicate assembly, a result of forward chaining is obtained by propagating these initial bindings to the consequent predicate assembly. Facts which are associated with predicate assemblies are not used during forward chaining.

In the case of the example rule encoded in Figure 3.4, presenting the predicate, *give(john, mary, book1)*, establishes the initial bindings {*john/give_arg1, mary/give_arg2, book1/give_arg3*} on *give* predicate assembly by activating the constant nodes and the argument node as follows:

john=[1], mary=[2], book1=[3],
give_e=[*],give_arg1=[1], give_arg2=[2], give_arg3=[3].

The symbol "*" stands for activation throughout the entire oscillation cycle and the number indicates the specific phase. In the next oscillation cycle, these initial bindings propagate to the *own* predicate assembly through the intermediate mechanism (links in this case), which eventually activate *own_arg1* in the same phase as *give_arg2* and *own_arg2* in the same phase as *give_arg3*. The collector, *own_c*, also becomes active by *give_c*. Consequently, the bindings {*mary/own_arg1, book1/own_arg2*} are obtained from the *own* predicate assembly as the result of inference. Time taken to process this sort of inference is proportional to πd, where d is the maximum diameter of the network.

Backward chaining is started by posing a query to the system. There are two types of queries: *yes-no* and *wh-type*. For example, the yes-no query, *give(john,mary,book1)*? or *give(john,mary,U)*?, can be posed to see whether or not any fact which matches with the given query is encoded into the network. The query containing variables, such as *give(john,mary,U)*?, may be considered as a wh-type query to obtain the constant which will get bound to U of the query if there is any encoded fact matched. Posing either type of query involves specifying the query predicate to the predicate assembly having the same argument name and arity. This is achieved by activating the enabler of the predicate assembly and setting up initial bindings between fillers of the query predicate and their corresponding arguments nodes. Performing backward chaining then corresponds to propagating these initial bindings to the antecedent predicate assembly and any fact node associated. Yes-no query processing succeeds only if there is a fact encoded into the network which matches with the presented query. Any matched fact will activate the collector of a predicate assembly with which it is associated and this activation comes back to the original predicate assembly where initial bindings were set up. The activation of the collector of the predicate assembly indicates the affirmative answer, otherwise the negative answer is assumed.

For detailed understanding of SHRUTI's fact encoding mechanism, let us think of the wh-type query, *give(john,U,V)*, to the network in Figure 3.5 which encodes the fact *give(john,mary,book1)*. SHRUTI's wh-type query processing requires two independent stages: the yes-no query processing stage and the answer extraction stage.

Stage 1 : firstly the system performs yes-no query processing to see if there exists a fact which matches with the given query in a network:

- posing the query *give(john, U, V)* to the system activates *give_e*. At the same time phase allocation occurs such that *john=[1], U=[0], V=[0]* where the symbol "0" represents the *null* phase[1] (no phase allocation). The corresponding argument nodes of the *give* predicate will also get activated in the same phases, *give_arg1=[1], give_arg2=[0], give_arg3=[0]*. This establishes the initial dynamic bindings between them;
- in the next oscillation cycle, the active output from *give_e* activates the fact node since there is no inhibitory signal coming from the argument nodes. The output of *give_arg1* is blocked by the same phase signal coming from the constant node *john*. The *give_arg2, give_arg3* do not project output signals because they are not activated during the query instantiation procedure;
- after one more oscillation cycle, the fact node activates the *give_c* and this indicates an affirmative answer to the given *yes-no* query. The output from the fact node also activates the *latch* node which receives input from the *answer enable*. The answer enable signal is activated when a *wh-type* query is posed. Once a latch node becomes active it holds activation until the second stage of wh-type query processing finishes.

Stage 2 : if the result of *yes-no* query processing is affirmative, the system must wait until all facts which match the query and their corresponding latch nodes in the network become active. Then the system performs the answer extraction procedure as follows. Otherwise inference stops and wh-type query processing fails.

- at the beginning of the second stage, the system now allocates distinct phases to the argument nodes which were bound to variable fillers of the input query and were left without phase allocation during the first stage. Consequently, new phase allocation is done for two argument nodes: *give_arg2=[2], give_arg3=[3]*;
- this activation then propagates to the *binder nodes* which also receive active input from the latch node. Note that the threshold of binder nodes is 2. The nodes *b2* and *b3* become active in the second and third phase as a result;

[1] To be precise, an inactive status of node throughout the entire oscillation cycle.

- the activation of binder nodes then propagates to the constant nodes, resulting that *mary=[2]* and *book1=[3]*. The desired answer of the given wh-type query is, therefore, obtained by observing in-phase activity between constant nodes and argument nodes, i.e. $\{U/mary, V/book1\}$.

Time taken to process each wh-type query is proportional to $4\pi(d+1)$: $2\pi(l+1)$ for the yes-no query processing stage and $2\pi(2d\text{-}l+1)$ for the answer extraction stage, where d is the maximum diameter of the network (see Park (1992) for more details).

3.3.4 Combining ISA Hierarchy with Rule-Based Reasoner

The rule-based reasoner described in the previous sub-sections can be integrated with an ISA hierarchy representing entities, types, the instance-of relations between entities and types, and the super/sub-class concept relations between types. Shastri & Ajjanagadde refer to the instance-of, super-concept, and sub-concept relations collectively using the ISA relation. This ISA hierarchy allows:

- the occurrence of types as well as entities in rules, facts, and queries;
- knowledge in the ISA hierarchy to interact with the encoding of the systematic aspects of a rule in order to enforce type restrictions and type preferences on argument-fillers.

They observed that one of the most important problems which must be solved in order to integrate the ISA hierarchy and the rule-based component is a representation of instantiations of a concept and a predicate [Mani & Shastri (1993)]. To address this problem, Shastri & Ajjanagadde introduced a bank of concept nodes and a switching mechanism. Each bank of concept node is composed of k ρ-btu elements, where k is the multiple instantiation constant. Thus, instead of using a single ρ-btu element, each concept in a ISA hierarchy is represented using a bank of elements. Two banks of concept nodes corresponding two concepts in ISA relationship are linked through the switching mechanism which mediates communication between them. When a concept bank needs to represent multiple instantiations during inference this switching mechanism automatically allocates available concept node in the bank. In the same way, rules and facts are encoded using banks of predicates and the switching mechanism. Each predicate in a predicate bank is composed of its own collector, enabler, and n argument nodes. In case of a rule, the antecedent and consequent predicates are encoded using two banks of predicates with the switching mechanism in the middle. Again the switching mechanism coordinates multiple instantiation of a predicate during inference when necessary.

Since these mechanisms impose significant space and time costs and structures, in terms of complexity and specificity of connections [Palm (1993), Eckhorn (1993)], they estimated the multiple instantiation constant to be a small number, 3. They claimed that the maximum number of multiple instantiation involved an episode of reflexive reasoning is no more than 3 (see Section 8.2 of Shastri & Ajjanagadde (1993)).

3.4 Advantages of SHRUTI

First of all, the temporal synchrony solution provides an extra time dimension to a connectionist representation. Consequently, the synchronous activities between neurons make it possible to represent dynamic bindings in a connectionist manner. It can efficiently represent a large number of dynamic bindings at the same time and there is no limit in a number of entities involved in the same bindings. As long as entity nodes oscillate in the same phase, all of them represent the same variable bindings. The number of distinct entities involved in these bindings will be only limited by physiological measurement of π and ω.

Secondly, SHRUTI can encode rules and facts involving n-ary predicates into a structured (localist) network. The structure of the network allows parallelism in the rules and facts encoded. During inference, it allows large numbers of rule to fire at the same time and a lot of facts encoded in a network can be active simultaneously as the predicate assemblies with which they are associated are activated. This parallelism and the efficient dynamic binding mechanism make SHRUTI supports very rapid inference – a chain of inference in several hundred msecs [Shastri (1992)]. The time taken for a chain of inference is independent of the total number of rules and equals no more than $l\pi$ where l is the length of a chain of inference and π is the width of an oscillation cycle.

Finally, this model is built based on neurally plausible data. All nodes of the network are designed and implemented based on the core features of connectionism presented in Feldman & Ballard (1982) and Rumelhart & McClelland (1986). And the selection of the width of phases and that of oscillation cycles is based on physiological measurements. This makes the model more neurally plausible than other connectionist inference systems surveyed in Section 2.4.1.

3.5 Some Limitations

3.5.1 Fundamental Constraints

A Number of Distinct Entities Allowed

The most clear constraint on SHRUTI's binding mechanism is the limitation of the number of different entities allowed to participate simultaneously in dynamic variable bindings. This limitation is a natural outcome of employing temporal synchrony to solve the dynamic variable binding problem since each entity of a predicate should occupy one phase of each oscillation cycle in order to avoid cross-talk. The maximum number of different entities allowed at a time, therefore, cannot exceed the number of phases in an oscillation cycle.

If the number of phases is large, the system can accommodate a larger number of dynamic bindings at a time but the time taken for each oscillation cycle will be increased too. On the other hand, if the number of phases is small, the system cannot support an enough number of dynamic bindings simultaneously and this will result in frequent cross-talk during a chain of inference. How can the suitable number of phases can be decided? Shastri & Ajjanagadde (1993) estimated this on the basis of biological and psychological evidence. Biologically, the limitation on the number of distinct entities depends on the smallest feasible values of ω, the width of a phase, and π, the width of an oscillation cycle, i.e. the maximum number of distinct entities allowed in their mechanism is bounded by $\lfloor \pi_{max}/\omega \rfloor$. They estimated neurally plausible value of π_{max} is about 28 msec and ω is about 6 msec. This revels that the number of entities referenced by the dynamic predicates is five or less. Psychologically, they justify this value in relation with 7 ± 2, a commonly accepted measure of short-term memory capacity [Miller (1956)]. Consequently, no rule may involve more than 7 to 10 distinct entities at a time.

Types of Rules Allowed

SHRUTI has a difficulty in dealing with a certain class of rules. When rules are encoded for backward reasoning,

> "any variable occurring in multiple argument positions in the antecedent of a rule should also appear in the consequent of the rule and get bound during the query answering process [Shastri & Ajjanagadde (1993)]. "

The problem with this constraint is that the system cannot sometimes force the condition that repeated variables in the antecedent should be bound to the same constant. In the rule $\forall X, Y \; p(X, X, Y) \rightarrow q(Y)$, for example, the variable X occurs in multiple positions in the antecedent but does not appear in the consequent. Therefore, the system cannot enforce the condition that its first and second arguments should bound to the same constant when the query $q(a)$ is given to the system. This is because the given query produces the dynamic query $p(?, ?, a)$ where the first and second arguments are unspecified. If the variable X occurs in the consequent such as: $\forall X, Y \; p(X, X, Y) \rightarrow q(X)$, the dynamic query of $q(a)$ will be $p(a, a, ?)$ and the condition of the rule can be forced by the system. The same constraint can occur when rules are encoded for forward reasoning: a variable occurring in multiple argument positions in the consequent of a rule should also appear in the antecedent of the rule and get bound during the query processing.

The important effect of this constraint is that the mechanism cannot be used to infer the transitivity of a relation in backward reasoning. For example, the following rule of transitivity has the variable Y occurring twice in the antecedent but does not appear in the consequent,

$$\forall X, Y, Z \;\; love(X, Y) \wedge love(Y, Z) \rightarrow jealous(X, Z),$$

thus cannot be used to answer correctly to the query *jealous(john,mary)?*.

3.5.2 Knowledge Representation Limitations

Disambiguation of Variable Bindings

As symbolic inference systems clearly distinguish different types of symbols (constant, variable, and function symbols for example), a connectionist inference system which replicates symbolic inference has to provide a mechanism to differentiate them in a connectionist manner.

During inference, SHRUTI only assigns a distinct phase to a constant filler appearing in a presented predicate and leaves a variable filler completely unspecified, without allocating any phases. This means that SHRUTI can only represent constant bindings (bindings between constant fillers and argument nodes) and fails to represent some forms of variable bindings (bindings between variable fillers and argument nodes). Since all variable fillers are represented using the same temporal tag – inactive states throughout the entire oscillation cycle – SHRUTI is unable to differentiate variable bindings. This prevents SHRUTI from performing unification across different groups of unifying arguments [Park(1992), Park & Robertson (1996a), Park et al. (1995)].

Unification Across Different Groups of Unifying Arguments

Suppose the situation where the predicate fact *p(a,U)* is presented to the following rule:

$$\forall X\ p(X, X) \rightarrow q(X),$$

unification between a and U is required to force consistency within the repeated arguments of the rule. Firing of the rule should allow us to deduce *q(a)* with the unification result, $\{U/a\}$.

If the rule has an additional variable argument in a different name, we need to keep track of the unification result during inference, i.e. unification is required across the unifying groups of arguments. Suppose we add the additional argument Y to the above rule:

$$\forall X, Y \; p(X, X, Y) \rightarrow q(X, Y),$$

then we will have two unifying argument groups for X and Y. Presenting the predicate *p(a,U,U)* now requires not only unification within repeated arguments (X's) but also unification between two groups of unifying arguments (X's and Y). What is expected as the desired answer, in this case, is *q(a,a)*.

This type of unification is well supported by most symbolic rule-based systems which use a symbolic matching procedure. Connectionist systems, however, are much more constrained in dealing with this type of unification because they require the network nodes to exhibit only simple, localised behaviours.

Figure 3.6 depicts how the given rule is encoded using SHRUTI's rule encoding mechanism for forward chaining.

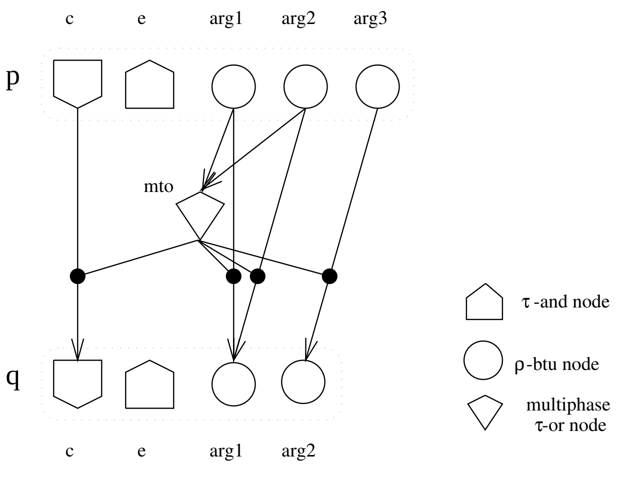


Figure 3.6: Encoding the rule which requires unification across different different groups of unifying arguments

The predicates, p and q, are represented using two predicate assemblies. The intermediate mechanism of the rule is then inserted by connecting appropriate links between the corresponding argument nodes. Since enablers are used only for backward chaining, there is no link between the two enablers. As described in the previous section, presenting the predicate to the system allocates a unique phase to constant fillers and activates them and their corresponding argument nodes in the same phase. Posing the query *p(a,U,U)* results in:

phase allocation and constant activation:*a=[1], U=[0]*;
predicate activation: *p_c=[*], p_e=[0], p_arg1=[1], p_arg2=[0], p_arg3=[0].*

This initial in-phase oscillation between a and *p:arg1* represents the initial variable binding, {*a/p_arg1*}. Note that their system does not allocate any phases to the variable fillers (*U's*) and their corresponding argument nodes (*p_arg2* and *p_arg3*).

Once initial variable bindings are set up, these bindings propagate to the consequent predicate assembly through the directed links. At this point, the network needs a way of performing unification and forcing the consistency condition for the first two arguments of the p predicate assembly because they are represented using the same variable name in the rule. The *mto* node, which is a multiphase τ-or node, is inserted for this purpose. As defined earlier, the *mto* node gets activated only when two different constant fillers are assigned to *p_arg1* and *p_arg2*. In the case of the presented predicate, *p(a,U,U)*, the *mto* node remains inactive because only the input coming from *p:arg1* is active during inference. As a result, the q predicate assembly will be activated in the following phases:

q_c=[*], q_arg1=[1], q_arg2=[0].

This incorrectly indicates the answer *q(a,U)* over the q predicate assembly and this is not the desirable result, *q(a,a)*. This has been caused simple because the unification restrictions in the antecedent of the rule are not carried forward to the consequent. Even if the predicate *p(a,U,b)* is presented to the network, the unification result {*U/a*} is only observed in the procedure of inference and is not explicitly represented on the the arguments of the q predicate. In addition, if the predicate *p(U,V,b)* which has variable fillers in the repeated argument positions is presented, there is no way to represent the binding {*U/V*} in SHRUTI's dynamic binding mechanism.

Consistency Checking Over Encoded Facts

When the fact $p(a, a, b)$ is encoded into a connectionist inference system, the system should be able to answer both, *yes-no* and *wh-type* queries. For example, the yes-no query *p(a,a,b)*? can be posed to see whether or not any fact which matches with the given query is encoded into the network. Also the wh-type query *p(a,U,V)?* may be posed to obtain the constants which will get bound to U and V of the fact if there is any encoded fact matched.

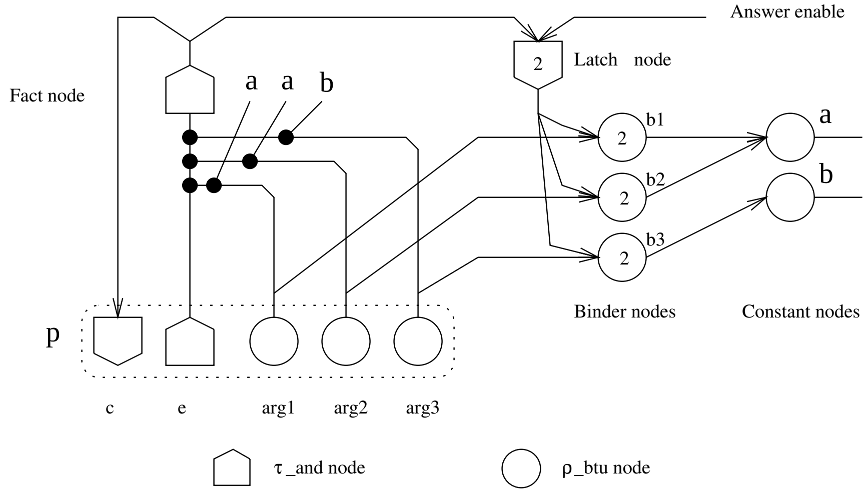


Figure 3.7: The network which encodes the fact *p(a,a,b)* using Shastri & Ajjanagadde's mechanism

Figure 3.7 shows the network with the fact node and the answer extraction mechanism, which encode the fact, *p(a,a,b)*. If we explore this network by posing a series of wh-type queries, we discover that SHRUTI has limitations in performing proper unification between the posed queries and encoded facts. As shown in Table 3.1, SHRUTI's fact encoding mechanism fails in forcing consistency on repeated variables in the input queries. Their mechanism can not produce proper answers to the the last three queries which require any repeated variable fillers in the input query should get bound to the same constant during inference.

Query Posed	Desired Answer	S&A's Answer	Result
p(U,V,W)	Yes - {U/a,V/a,W/b}	Yes - {U/a,V/a,W/b}	right
p(U,U,V)	Yes - {U/a,U/a,V/b}	Yes - {U/a,U/a,V/b}	right
p(U,U,U)	No	Yes - {U/a,U/a,U/b}	wrong
p(U,V,U)	No	Yes - {U/a,V/a,U/b}	wrong
p(a,U,U)	No	Yes - {a/a,U/a,U/b}	wrong

Table 3.1: Response to the various queries presented to the encoded fact

Clarifying Bindings Among Multiple Encoded Facts

A further limitation of SHRUTI's fact encoding mechanism is that it does not disambiguate bindings obtained by wh-type query processing over multiple facts encoded (facts which share the same predicate name and arity). Suppose the following facts are encoded into a network:

$$f1 : p(a, b, c),\quad f2 : p(a, a, b),\quad f3 : p(a, a, a).$$

Since all facts have the same predicate name and arity, all of them are associated with the same predicate assembly *p*. Each fact requires its own answer extraction mechanism for wh-type query processing, which consists of a latch node and a set of binder nodes as described earlier.

If we pose the wh-type query, *p(U, V, W)*, the system first performs yes-no query processing. During this stage, all three fact nodes and their corresponding latch nodes become active. At the beginning of the answer extraction stage, the system activates the argument nodes which correspond to the variable fillers as follows:

p_arg1=[1], p_arg2=[2], p_arg3=[3].

As inference cycles continue, the constant nodes, *a, b*, and *c*, will be activated by each set of binder nodes of the facts as follows:

by the binder nodes of f1: *a=[1], b=[2], c=[3]*;
by the binder nodes of f2: *a=[1], a=[2], b=[3]*;
by the binder nodes of f3: *a=[1], a=[2], a=[3]*.

The final representation of these results will be:

$$a=[1,2,3],\ b=[2,3],\ c=[3].$$

where $a=[1,2,3]$ represents that the node a becomes active in the first, second, and third phases of each oscillation cycle. If a single ρ-btu node is used for each constant, these results cannot be represented properly because this type of node handles only a single phase. Using the special mechanisms, a bank of concept nodes and a switching mechanism, introduced to address multiple instantiations, the above result, $a=[1,2,3]$, will be represented using the constant bank a by making the first constant node of the bank active in the first phase, the second constant node of the bank in the second phase, and the last constant node in the third phase. The rest of results, $b=[2,3]$ and $c=[3]$, have to be represented using constant banks b and c.

From this representation, what we can observe is that the constant a occupies the first, second, and third argument positions of the predicate p; the constant b the second and third argument positions; and the constant c the third argument position. These in-phase oscillations between constant banks and argument nodes activated in the same phase as those of variable fillers do not necessarily disambiguate the desired bindings, $\{U/a,\ V/b,\ W/c\}$, $\{U/a,\ V/a,\ W/b\}$, $\{U/a,\ V/a,\ W/a\}$. The problem is that although we can tell which constants occupy which argument positions as the result of inference, we can not tell externally what the valid combinations are. Additional mechanism may be necessary to enable their fact encoding mechanism to provide enough information to disambiguate the corresponding bindings for each encoded fact.

3.6 Summary

The temporal synchrony solution is efficient mechanism which provides dynamic bindings in a connectionist manner. Also the way that SHRUTI encodes rules and fact using assembly of nodes allows parallelism in the network constructed. However, this solution is fundamentally limited in representing variable bindings during inference. This limitation in turn makes SHRUTI unable to deal with important knowledge representation issues, especially disambiguation of bindings and unification across different groups of unifying arguments. Furthermore SHRUTI's rule and fact encoding mechanisms do not consider all the consistency conditions that the syntax of rules and facts enforces when encoding rules and facts.

Therefore, it would be interesting to build a connectionist system which utilises the advantages of the temporal synchrony solution, yet overcomes the limitations highlighted in this chapter. Exploring such a system will help us to understand the extent to which this type of a connectionist architecture can accommodate basic concepts of symbolic inference. Chapter 4 extends this idea further and proposes a design and implementation of such a connectionist inference system.

Chapter 4

A Connectionist Architecture

4.1 Overview

Having recognised the limitations of existing connectionist models, this chapter proposes another connectionist model, called a Connectionist Architecture for Symbolic Inference (CASI). CASI adopts the temporal synchrony approach and extends it to obtain desirable features.

This chapter firstly defines a target symbolic model whose knowledge representation and inferential ability will be replicated by the proposed architecture. Then, we present CASI's overall architecture and details about the extension made to provide better dynamic binding mechanism will be described. This extension can be summarised as follows:

- generalisation of a ρ-btu element
- representing entities and predicates
- representing dynamic bindings

Based on this extension, this chapter describes possible connectionist interpretations of standard symbolic inference procedures which become an important infrastructure of CASI.

4.2 A Target Symbolic Model

4.2.1 A Symbolic Knowledge Representation Scheme

The language that has been chosen to describe the target symbolic knowledge is a subset of first-order Horn clause expressions defined as follows:

Definition: *First-order Horn clause expressions* are 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*. □

To describe a target symbolic model, we use the subset of first-order Horn clause expressions which do not contain negations or recursive rules which have multiple predicate definitions. Some example facts and rules in the right form are:

man(adam),
love(mary,tom),
p(p,b),
$man(X) \rightarrow mortal(X),$
$p(X,Y) \rightarrow q(X,Y).$

The form of fact and rule which are not supported by the proposed connectionist architecture is:

$\neg son(abraham, jacob),$
$path(X,Y) \wedge path(Y,Z) \rightarrow path(X,Z).$

Symbols used in first-order Horn clause expressions are the same as those defined in first-order predicate calculus [Chang & Lee (1973), Bundy (1983), Manna & Waldinger

(1985), Luger & Stubblefield (1989)]. The basic symbolic constituent structure of rules and facts is a predicate which consists of a predicate symbol followed by n constant or variable arguments.

4.2.2 Target Symbolic Inference Procedures

Two important procedures of symbolic inferences have been recruited to be implemented in a connectionist manner. Section 2.2 described the importance of these two inference procedures, match and substitution, in standard symbolic inference based on first-order predicate calculus.

To summarise their roles again in drawing symbolic inference, let us consider the following rule written in a first-order Horn clause expression:

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

When the predicate, $p(a, U, V)$, is presented to the antecedent of the given rule to instantiate forward chaining, an inference system will normally use the match procedure to see if the antecedent of the rule matches with the presented predicate. If it is, this procedure yields a set of bindings between variables and constants in the matching terms as represented in the following notation:

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

From the match procedure, two types of bindings are produced: the constant binding between a and X and the variable binding between U and X and between V and Y. The term a *constant binding* refers to the situation where a variable gets bound to a constant value and a *variable binding* the situation where a variable gets bound to a variable value. 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 expression and consistently substitutes all occurrences of variables in the expression using the binding set. This procedure may be represented as follows:

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

Given the result of this substitution, a symbolic inference system can apply the *modus ponens* inference rule to infer the new logical assertion, $q(a,V)$. Note that the first two bindings, $\{a/X, U/X\}$, were obtained from the repeated arguments (X's), in other words, from the same group of unifying arguments. The intermediate binding, $\{a/U\}$, obtained from these bindings is necessary for the substitution procedure to produce the desirable result, $q(a,V)$.

In a symbolic system, these standard inference 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 design of the appropriate connectionist infrastructure. In addition, there exists a wide range of algorithms to implement symbolic inference procedures in a symbolic inference system. However, we are 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 follow a symbolic style of inference must have connectionist mechanisms which represent symbolic knowledge and perform some form of matching with substitution.

4.2.3 Dynamic Bindings

As can be seen in the above inference procedures, one of the most important short-term constituent structure in symbolic inferences is the dynamic bindings representing the relationship between the presented fillers and arguments of the predicate matched. The inference procedures firstly shows that there are clear distinction between constant bindings and variable bindings as well as between variable bindings.

Secondly, the substitution procedure requires bindings obtained from the same group of unifying arguments to be represented explicitly during inference so that intermediate bindings can be deduced. In the previous inference, for example, bindings pertaining to the same unifying argument group were represented using the same argument name such as $\{a/X, U/X\}$, from which the intermediate binding, $\{a/U\}$, is obtained. This is necessary for the substitution procedure to substitute all occurrences of the argument X in the consequent of the rule with the constant a rather than the variable U.

More complicated set of bindings which belong to the same unifying argument group can be seen in the situation where the predicate $p(a, U, V)$ is presented to the following rule:

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

In this case, a set of bindings produced by the match procedure is $\{a/X, U/X, V/X\}$ and this generates the intermediate bindings, $\{a/U, a/V\}$. These intermediate bindings are needed for the substitution procedure to produce the desirable result $q(a)$.

This suggests that the ability to represent both constant and variable bindings dynamically is an essential feature for a connectionist system which replicate a standard symbolic style of inference. In addition, a way of representing a set of bindings pertaining to a group of unifying arguments is necessary to deduce intermediate bindings when necessary. A connectionist architecture that will be introduced in the following section will be designed to provide such connectionist mechanisms to represent not only required dynamic bindings efficiently but also the strategies used to draw inference in a connectionist manner.

4.3 A Connectionist Architecture

4.3.1 Overall Architecture

A Connectionist Architecture for Symbolic Inference (CASI) is a connectionist model which encodes symbolic rules and facts in first-order Horn clause expressions into a set of corresponding networks called structured predicate networks (SPNs). Once input rules and facts are encoded, it can perform symbolic inferences over these networks. Having SPNs in common, CASI provides two independent phases to achieve symbolic inference: The knowledge compilation phase and the inference phase. Figure 4.1 illustrates the overall structure of CASI.

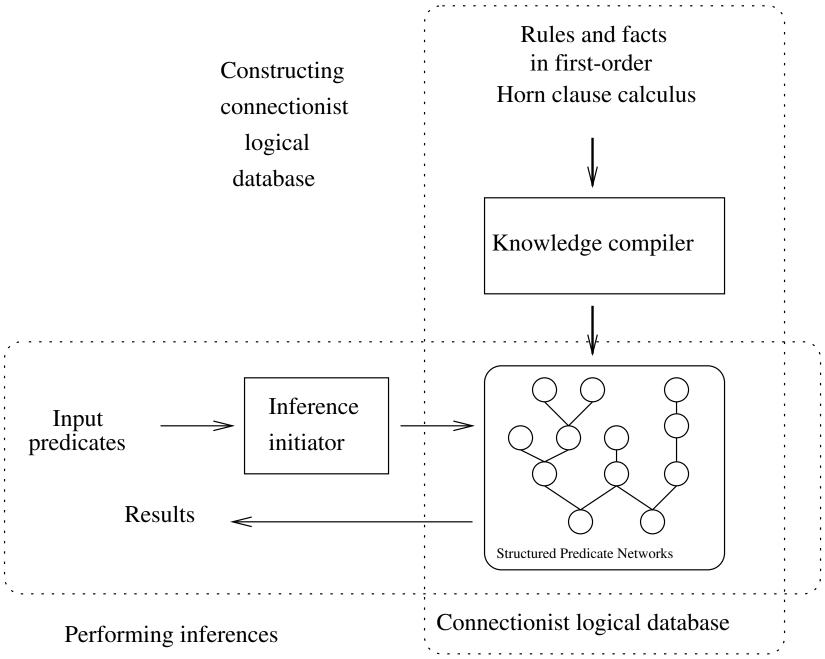


Figure 4.1: The structure of Connectionist Architecture for Symbolic Inference

During the knowledge compilation phase, CASI uses a knowledge compiler to build SPNs from input rules and facts. On receiving input rules and facts, the knowledge compiler translates them into the corresponding SPNs using special algorithms devised. A collection of SPNs is considered as a connectionist logical database (CLDB) which may correspond to a symbolic logical database in a symbolic inference system. Once SPNs are constructed, CASI can perform inference in a standard way over these networks by activating a group of nodes corresponding to a presented query predicate and propagating their activation to other groups of nodes through links. The initial activation of a group of nodes to start inference involves how to establish dynamic bindings between the presented predicate and a group of nodes corresponding to it. CASI's architecture must provide an efficient way of representing dynamic bindings between them. This series of initial binding set up procedure is carried out by the inference initiator. Since we consider Shastri & Ajjanagadde's temporal synchrony approach is an efficient way of representing constant bindings in a connectionist manner, CASI's dynamic binding mechanism adopts this approach and extends it to explore the extent to which this type of connectionist architecture can accommodate basic concepts of symbolic style of dynamic bindings.

4.3.2 Structured Predicate Networks

SPNs are the most important component of CASI, which are defined as follows:

Definition: A *structured predicate network* (SPN) is a localist network which encodes a rule or a fact. Each SPN is composed of three parts, $SPN_i = (P_{s_i}, M_i, P_{t_i})$, where P_{s_i} is a group of nodes representing the *source predicates*, P_{t_i} is a group of nodes representing the *target predicates*, and M_i is the *intermediate mechanism* which connects them. When encoding a rule for forward chaining, the corresponding source and target predicates are the antecedent and consequent of the given rule and vice versa when the rule is encoded for backward chaining. When encoding a fact, the source predicate is the base form of the given fact and the target predicate is an additional predicate to be inserted. □

Figure 4.2 depicts the structure of a SPN.

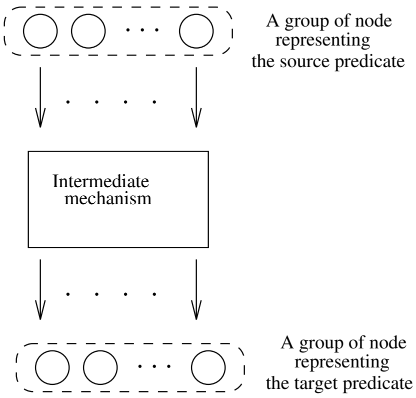


Figure 4.2: The structure of SPNs

Whenever a rule or a fact is encoded, the knowledge compiler translates it into the corresponding SPN and puts it into the CLDB. Since syntax of some rules or facts impose additional conditions such as consistency of an argument for a chain of inference, one part of the SPN has to take care of those conditions in a connectionist manner. The SPN's intermediate mechanism, M_i, is introduced for this purpose. Each M_i for a rule or a fact will be implemented in such a way that it includes several sub-mechanisms which carry out all the necessary sub-tasks to achieve the target symbolic inference.

In many cases, a set of rules and facts share the same base form of predicate – by having the same predicate name and arity. When encoded, this allows their SPNs to be connected to each other through this common predicate. This series of connected SPNs is used to draw a chain of inference. For instance, when two rules, $p(X) \rightarrow q(X)$ and $q(X) \rightarrow r(X)$, are encoded, only three groups of nodes representing the predicates, p, q, and r, are required because the consequent of the first rule is the same as the antecedent of the second rule. The SPN encoding the rule, $p(X) \rightarrow q(X)$, is connected to the SPN representing the rule, $q(X) \rightarrow r(X)$, through the common groups of nodes representing the predicate $q(X)$. In a similar way, SPNs encoding facts which have the same predicate name and arity share a group of nodes representing a base form of the predicate. This allows parallelism when they are used in a chain of inference.

4.4 Building The Components of SPNs

To build major structural components of an SPN, we need neuron elements and links. This section describes the necessary neuron elements to be employed in CASI to extend the temporal synchrony approach.

4.4.1 Basic Neuron Elements

The first important extension made to extend the temporal synchrony approach is generalisation of a ρ-btu element and employing it as a basic element to build groups of nodes for an SPN.

Before we define a generalised ρ-btu element, it may be helpful to remind two time scales used in the temporal synchrony approach: 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 behaviours. Like a ρ-btu element, a generalised ρ-btu element, which will be called a π-btu element, also samples their inputs over several phases of an oscillation cycle and determines their output patterns depend on the input patterns sampled during this time period.

The temporal behaviour of a π-btu element is defined as follows:

Definition: A π-*btu element* is a generalised ρ-btu element. It becomes active on receiving one or more spikes in any oscillation cycle. On becoming active, a π-btu element projects an oscillatory spikes that are in-phase with the driving input. On becoming active, the π-btu element continually produces the output (oscillating) even after its input has ceased – while they are under the current focus of attention for a chain of inference. Otherwise, their output decays in a short time unless they are used for another chain of inference. A threshold, n, associated with this element indicates that the element will fire only if it receives n or more spike inputs in the same phase. $\square$

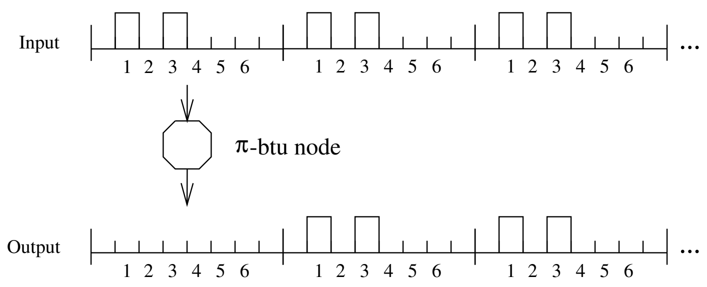


Figure 4.3: The temporal behaviours of a π-btu element

Figure 4.3 depicts the temporal behaviour of a π-btu element. The number of phases in one oscillation cycle is assumed to be 6 for the purpose of explanation. When the π-btu element, depicted with an octagon, receives two spikes in an oscillation cycle as shown in the figure, it produces the same spike train that is in-phase with the driving input. The behavioural difference between our π-btu element with respect to that of Shastri & Ajjanagadde's ρ-btu element is that: the temporal property of a ρ-btu element is restricted to carrying only a single phase signal per oscillation cycle, whereas CASI's π-btu element allows multiple phase signals per oscillation cycle and propagates these in synchrony with the driving inputs. This requires higher signal transfer rates than Shastri & Ajjanagadde's ρ-btu element. In relation to this, Eckhorn (1993) observed that the signal transfer rates on biological neurons are much higher than those of ρ-btu elements. Such high rates in a single neuron are capable of signalling complex messages, including routing dynamic representations. This allows the possibility that there exists a type of neuron whose signal transfer ability is even higher than the π-btu element.

In CASI, the defined π-btu element will be used as a basic element to build groups of nodes of different types. These groups of nodes are used to represent entities and predicates. To build an intermediate mechanism of a SPN, different types of phase-sensitive nodes are needed. CASI adopts other three types of phase-sensitive elements from SHRUTI [Shastri & Ajjanagadde (1993)]: the τ-and, the τ-or and the multiphase τ-or elements (refer to Section 3.2.1 for detailed definitions of them).

4.4.2 Entity Nodes

Symbolic inference systems use symbols to represent components of knowledge. Each symbol is represented using a unique symbolic name to be differentiated from others. Two types of symbols are used to represent entities in first-order Horn clause expressions: a constant symbol and a variable symbol. A constant symbol is used only to represent a known entity or concept but a variable symbol plays a different role. A variable symbol can unify with a constant, a variable, or even with a structured term in some inference systems but cannot unify with two different constants at the same time. Therefore identifying one variable symbol from others is an important property in first-order Horn clause expressions because sometimes it may be necessary to represent many groups of unifying variables during inference. For example, we may want to represent that "Variables X, Y, Z unify and Variables U, V, W unify." Any connectionist system which resembles symbolic inference must provide a connectionist mechanism that differentiates not only one constant entity from others but also one variable entity from others.

To cater for this, CASI introduces a special cluster of neuron elements, called an entity node. The detailed structure of the entity node is defined as follows:

Definition: An **entity node** is a pair of π-btu elements. The left element is used to represent a variable role of the entity node and the right element a constant role. A symbolic name may be used as a label of an entity node to differentiate one node from others. For convenience in discussion, we represent an entity node symbolically using the notation, *entity_name([0],[0])*, where the left square bracket represents the state of the left π-btu element and the right square bracket that of the right π-btu element. The symbol "0" denotes the state in which an element is inactive. The notations, $entity\_name_{left}[0]$ or $entity\_name_{right}[0]$ are used to indicate each inactive state of the left or right π-btu element. When an entity node is used simply to represent a constant or a variable, the notations, *var_name[0]* or *const_name[0]* are used to shorten the representation. □

According to the usage of either or both elements, an entity node is used to represent a constant node, a variable node and a binding node. When an entity node is used to represent a constant entity, only its right element is used during inference. When the entity node is used to represent a variable entity, only its left node is used. When it is used to represent a binding entity, either or both elements may be used at the same time. In CASI, constant nodes and variable nodes are used to represent constant fillers and variable fillers and binding nodes to represent arguments. When both elements of a binding node become active, the left element represents variable binding(s) and the right element constant binding(s) involving an argument represented by it. Figure 4.4 gives a graphical and symbolic representation of each role of an entity node.

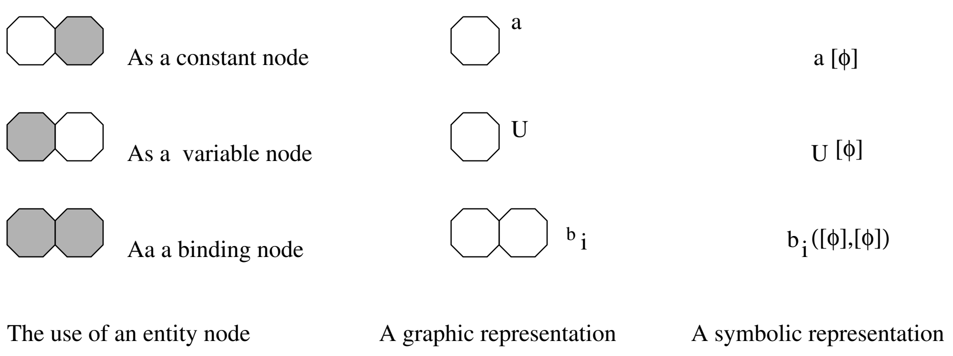


Figure 4.4: The representation of roles of an entity node

In the graphical representation, entity nodes used for constant and variable nodes are represented using only one π-btu element to simplify the representation because only their right or left elements are used respectively. To differentiate these two graphical representations, symbolic tags are used. A single π-btu element with a symbolic name beginning with an upper-case letter is considered as a variable node and one with a symbolic name beginning with a lower-case character as a constant node. When the entity node is used as a binding node, a specific name or b_i will be used as a symbolic label.

4.4.3 Predicate Assemblies

A predicate is a basic structured unit of symbolic rules and facts. CASI uses an assembly of nodes, called a predicate assembly, to represent symbolic predicates. Like a symbolic predicate, a predicate assembly has an arity. When a predicate is encoded using a predicate assembly, it is always represented in the form of a 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. For each A_i of the given predicate, we say that arg_i of the base predicate is the *base argument* of A_i, where A_i is either a constant or a variable. $\Box$

Some example base predicates are shown as follows:

Predicates	The base predicates
$p(X,Y,Z)$	$p(arg_1,arg_2,arg_3)$
$p(a,U,b)$	$p(arg_1,arg_2,arg_3)$
$father(john,tom)$	$father(arg_1,arg_2)$
$triangle(X)$	$triangle(arg_1)$

The first two predicates have the same base predicate because they have the same argument name and arity. In the first predicate, the base arguments of X and Y are arg_1 and arg_2.

Based on the above definition, an n-ary predicate assembly is defined as follows:

Definition: An n-ary *predicate assembly* is composed of one τ-and node and n binding nodes. A τ-and node is called a predicate activator (labelled *pa*) and n binding nodes, which will be also called argument nodes (labelled arg_1, arg_2, ..., arg_n). The predicate activator is used to indicate whether the predicate assembly is active or not. Dynamic bindings represented on its argument nodes will be valid only when the predicate activator becomes active. The notation, *pred_name:*$\{pa[0], arg_1([0],[0]),\ arg_2([0],[0]),\ \ldots,\ arg_n([0],[0])\}$ will be used to represent an n-ary predicate assembly symbolically. When any individual argument node needs to be named, a prefix "*pred_name:*" is used before the symbolic name of the argument node. □

The predicate, p(X,Y), for example, is represented using one predicate activator and two argument nodes. The symbolic representation of the predicate assembly will be

$$p{:}\{pa[0], arg_1([0],[0]),\ arg_2([0],[0])\}$$

where *pa* is the symbolic name of the predicate activator and arg_1 and arg_2 are those of argument nodes (each corresponds to X and Y arguments). When the node arg_1 needs to be named individually, the prefix "*p:*" is used before the name such as $p{:}arg_1$.

Figure 4.5 illustrates the graphical representation of this predicate.

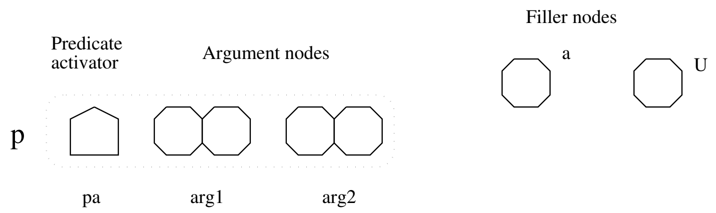


Figure 4.5: The representation of a predicate assembly

A predicate activator and two argument nodes are found in the predicate assembly called *p*. One argument node is differentiated from the other by their symbolic labels. Two fillers, the constant *a* and the variable *U*, are also shown in the figure. The distinction between a constant filler and a variable filler is made by whether their symbolic labels start with a lower-case letter or a upper-case letter. Note that all such labels are a notational convenience for the reader. In the system itself, differentiation of roles of nodes is determined by their connections in the network.

4.4.4 An Example of an SPN

To give an example how a rule is encoded into its corresponding SPN, let us reconsider the following example rule:

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

If this rule is encoded to support forward chaining, the predicate *p(X,X,Y)* is considered as the source predicate of the rule and the predicate *q(X,Y)* as the target predicate. The SPN is built by creating two predicate assemblies corresponding to the source and target predicates. An intermediate mechanism is then needed between them. The structure of the SPN will therefore looks like the network shown in Figure 4.6.

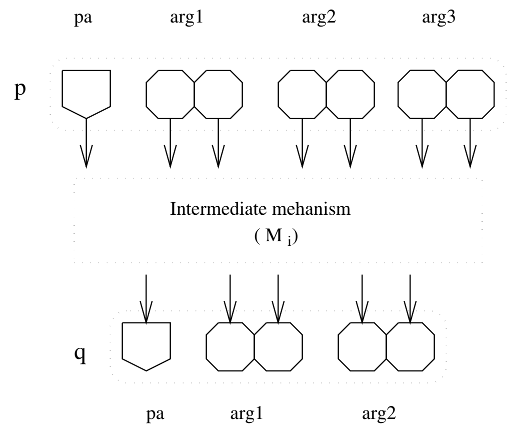


Figure 4.6: The example SPN encoding the rule, $p(X, X, Y) \rightarrow q(X, Y)$

Assuming the name of the example rule to be "ex1", we use the following symbolic notation to describe the SPN:

$$SPN_{ex1} = \{P_{s_{ex1}}, M_{ex1}, P_{t_{ex1}}\}, \text{ where}$$

$$P_{s_{ex1}} = p{:}\{pa[0],\ arg_1([0],[0]),\ arg_2([0],[0]),\ arg_3([0],[0])\},$$
$$P_{t_{ex1}} = q{:}\{pa[0],\ arg_1([0],[0]),\ arg_2([0],[0])\},$$
$$M_{ex1} = \text{an intermediate mechanism between them.}$$

The whole structure of the SPN will be completed after the intermediate mechanism is built. Each intermediate mechanism of a rule is constructed with its own set of connectionist sub-mechanisms which perform the required sub-tasks, such as consistency checking, that syntax of the rule may enforce. The detailed design and implementation of intermediate mechanisms according to types of rules will be described in Chapter 5. The rest of this chapter gives more details about CASI's connectionist interpretations of dynamic bindings and symbolic inference procedures. CASI provides connectionist mechanisms to perform the tasks required for symbolic style of inference based on these interpretations.

4.5 Representing Dynamic Bindings

In the description of the target symbolic system, we emphasised the importance of representing constant and variable bindings as well as a set of bindings which belong to the same group of unifying arguments. To meet these requirements CASI's dynamic binding mechanism uses temporal synchronous activities between constant or variable node and argument nodes to represent dynamic bindings.

To illustrate the extension made, let us consider the two filler nodes, *U[0]* and *a[0]*, representing the variable *U* and the constant *a*, and the two argument nodes, $arg_i([0],[0])$ and $arg_j([0],[0])$. CASI's dynamic binding mechanism establishes a constant binding and a variable binding between the filler nodes and the argument nodes as follows:

- the variable binding between *U* and arg_i, $\{\text{U}/arg_i\}$, is represented by activating the variable node representing *U* and the left element of the argument node representing arg_i in the first phase, so that *U[0]* becomes *U[1]* and $arg_{i_{left}}[0]$ becomes $arg_{i_{left}}[1]$, where the number 1 indicates the first phase;
- the constant binding between *a* and arg_j, $\{a/arg_i\}$, is represented by activating the constant node representing *a* and the right element of the argument node

representing arg_j in a different phase, so that *a[0]* becomes *a[2]* and $arg_{j_{right}}$*[0]* becomes $arg_{j_{right}}$*[2]*, where the number 2 stands for the second phase.

Figure 4.7 illustrates these two types of dynamic bindings graphically. A shaded element indicates that the element is in active state.

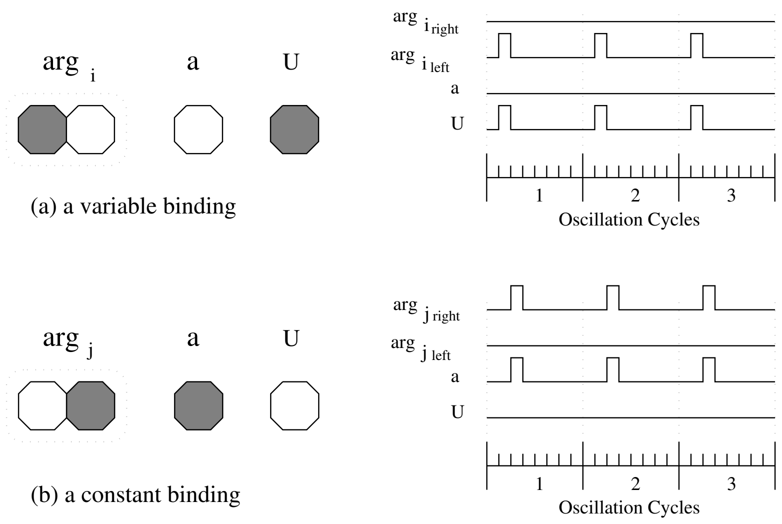


Figure 4.7: CASI's representation of two types of dynamic bindings

As shown in Figure 4.7a and b, a constant binding or a variable binding is distinguished by the position of the activated element of the argument involved. By activating a different set of filler nodes in-phase with the argument nodes, a set of different temporal dynamic bindings can be obtained.

CASI's dynamic binding mechanism also provides an easy way of representing a set of dynamic bindings generated by the same group of unifying arguments. Two bindings, $\{a/arg_i,\ U/arg_i\}$, which have been produced by the same argument name arg_i, for instance, can be represented by activating two filler nodes and the argument node as follows:

$$a[1],\ U[2],\ arg_i([2],[1]).$$

Assuming that the constant node and the variable node is activated in the first and the second phases already, *a[1]* and *U[2]*, activating the left and right elements of the

arg_i([0][0]) node in-phase with these filler nodes gives the required representation of the dynamic bindings. The left element of the argument node ($arg_{i_{left}}$*[2]*) and *U[2]* indicates the variable binding, $\{U/arg_i\}$, and the right element ($arg_{i_{right}}$*[1]*) and *a[1]* the constant binding $\{a/arg_i\}$, from which $\{a/U\}$ is obtained.

Figure 4.8 depicts this situation.

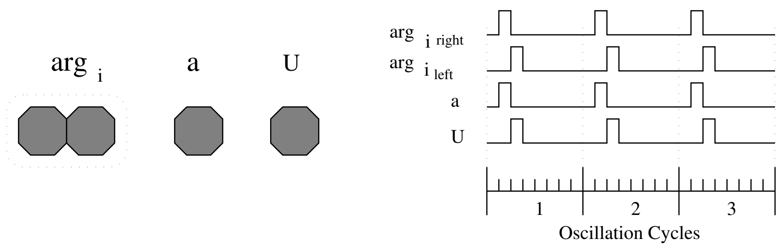


Figure 4.8: Representing two bindings $\{a/arg_i, U/arg_i\}$ belonging to the same unifying group

If more than one variable fillers are involved in dynamic bindings pertaining in the same unifying group such as $\{a/arg_i,\ U/arg_i,\ V/arg_i\}$, this situation is represented as follows:

$$a[1],\ U[2],\ V[3],\ arg_i([2,3],[1]).$$

If the filler nodes are active in *a[1]*, *V[2]* and *V[3]*, this can be achieved by activating the left element of the argument node in the second and third phase and the right element in the first phase. Because the left element of the arg_i is active both the second and third phases in each oscillation cycle this is interpreted as the two variable bindings, $\{U/arg_i, U/arg_i\}$. On the other hand, the activation of the right element in the first phase produces the constant binding $\{a/arg_i\}$. Based on these, the intermediate bindings $\{a/U,a/V\}$ are obtained. Figure 4.9 demonstrates this situation.

4.6 Connectionist Inference Procedures

To replicate standard forms of symbolic inference procedure, CASI has to provide connectionist equivalents to two important inference procedures, match and substitution.

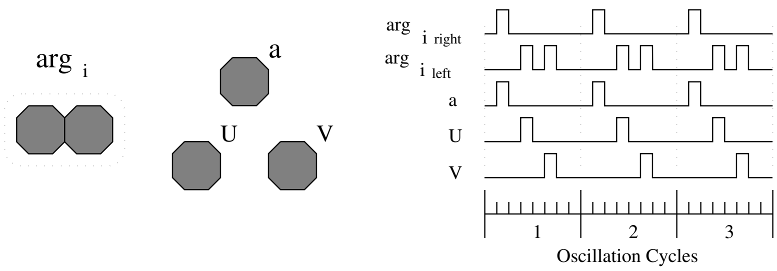


Figure 4.9: Representing the dynamic bindings $\{a/arg_i, U/arg_i, V/arg_i\}$ pertaining to the same unifying group

This section illustrates how these procedures can be implemented in a connectionist manner using the connectionist dynamic binding mechanism previously introduced.

4.6.1 A Connectionist Match procedure

If we reconsider the symbolic match procedure between the predicate $p(a, U, V)$ and the rule, $p(X, X, Y) \rightarrow q(X, Y)$:

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

Applying the match procedure produces the set of bindings $\{a/X, U/X, V/Y\}$. Although the detailed sub-tasks required for the match procedure are not specified, one obviously necessary sub-task is consistency checking on the bindings produced by the repeated arguments. The output of the match procedure should be generated only if the bindings do not violate the consistency condition (that the repeated arguments should be unifiable). Thus, representing the symbolic match procedure in a connectionist manner requires us to provide connectionist equivalents to sub-tasks involved in the symbolic match procedure, especially consistency checking and representation of the generated bindings.

In CASI, this symbolic match procedure is resembled by providing connectionist submechanisms which perform the necessary sub-tasks. Instead of implementing the match procedure as a whole, its sub-tasks are implemented by separate connectionist sub-mechanisms within the corresponding SPN. When the given example rule is encoded, it is translated into the following SPN:

$$
\begin{aligned}
SPN_i &= \{P_{s_i}, M_i, P_{t_i}\}, \text{ where} \\
P_{s_i} &= p{:}\{pa[0],\ arg_1([0],[0]),\ arg_2([0],[0]),\ arg_3([0],[0])\}, \\
P_{t_i} &= q{:}\{pa[0],\ arg_1([0],[0]),\ arg_2([0],[0])\}, \\
M_i &= \text{the intermediate mechanism between } \mathrm{P}_{s_i} \text{ and } \mathrm{P}_{t_i}.
\end{aligned}
$$

The effect of representing a set of dynamic bindings, $\{a/X, U/X, V/Y\}$, – which is the same as obtaining these bindings from the symbolic match procedure – on this SPN is achieved in the following way:

1. for each filler appears in the presented predicate, create its corresponding filler node using an entity node. In the case of the presented predicate, $p(a, U, V)$, the constant node $a[0]$ and the two variable filler nodes, $U[0]$ and $V[0]$, are created as the result;

2. set up initial dynamic bindings between the filler nodes and argument nodes of the source predicate assembly by activating them in the same phase. In the example SPN, for instance, this is done by activating the nodes involved in the following phase:

   $$
   \begin{gathered}
   a[1],\ U[2],\ V[3], \\
   (p{:}\{pa[*],\ arg_1([0],[1]),\ arg_2([2],[0]),\ arg_3([3],[0])\},\ M_i, \\
   q{:}\{pa[0],\ arg_1([0],[0]),\ arg_2([0],[0])\}).
   \end{gathered}
   $$

   These in-phase oscillations between the filler nodes and argument nodes of the p predicate assembly represent the set of bindings $\{a/p{:}arg_1, U/p{:}arg_2, V/p{:}arg_3\}$. Note this binding set is the same as the one obtained by the symbolic match procedure, $\{a/X, U/X, V/Y\}$. The phase activator $pa[*]$ also becomes active throughout the entire oscillation cycle to indicate that the bindings represented on the argument nodes can be recognised from the predicate assembly;

3. since the above sub-procedure does not involve any type of consistency checking over these initial bindings, insert a consistency checking sub-mechanism into the

M_i to carry out this task. Make the rest part of the SPN use these initial bindings only if they do not violate any consistency condition imposed by the sub-mechanism inserted.

The symbolic match procedure is usually carried out by the unification algorithm which involves necessary consistency checking. Thus, a set of binding is produced only when the two input predicates matched satisfy all the consistency conditions forced by the algorithm. Whereas CASI's connectionist match procedure sets up the initial bindings without any consistency checking involved. The necessary consistency checking is carried out by separate sub-mechanisms inserted into the intermediate mechanism. The initial bindings will be valid on the SPN only if they satisfy consistency conditions forced by these sub-mechanisms. A detailed algorithm that will be used to determine the structure of these sub-mechanisms will be described in Chapter 5.

In general, when a predicate, $p(F_1, F_2, \ldots, F_n)$, is presented, where F_i is each filler of the predicate, CASI uses the following algorithm to set up the initial bindings:

PROCEDURE *Instantiation of the initial bindings for Forward Chaining*

Step1: let $p(arg_1, arg_2, \ldots, arg_n)$ be the base predicate of $p(F_1, F_2, \ldots, F_n)$
Step2: **if** the base predicate assembly is encoded into CLDB
Step3: **then**
Step4: **do** $i = 1$ **until** $i = n$
Step5: **if** node F_i is not in CLDB
Step6: **then** create an entity node with the label F_i
Step7: **if** F_i is already active in phase P_j
Step8: **then if** F_i is a variable name
Step9: **then** activate $arg_{i_{left}}$ in the phase P_j
Step10: **else** activate $arg_{i_{right}}$ in the phase P_j
Step11: **else** get a new phase available P_k
Step12: **if** F_i is a variable name
Step13: **then** activate $F_{i_{left}}$ and $arg_{i_{left}}$ in the phase P_k
Step14: **else** activate $F_{i_{right}}$ and $arg_{i_{right}}$ in the phase P_k
Step15: increase i by 1

Step16: activate *pa[0]* in the phase P_k

Step17: **end do**

Step18: **else** return(fail)

At the beginning of the algorithm, the existence of the base predicate assembly corresponding to the presented predicate is checked first. If such a predicate assembly does not exist, inference has to stop. Otherwise the binding instantiation procedure continues to set up the initial bindings between the presented predicate and the antecedent predicate assembly using the dynamic binding mechanism. Each dynamic binding is set up by activating the filler node and the right or the left element of the argument node in the same phase according to the type of binding required. When this procedure is finished, the predicate activator of the base predicate assembly becomes active throughout the entire oscillation cycle (*p:pa[*]*). Note that this predicate activator becomes active in each phase during the whole initial binding instantiation procedure.

4.6.2 A Connectionist Substitute Procedure

After the match procedure, the generated bindings are needed for the substitution procedure:

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

to produce the new expression, $p(a, a, V) \rightarrow q(a, V)$, from which the conclusion, $q(a, V)$, is obtained by applying the modus ponens inference rule. Note that two initial bindings, $\{a/X,\ U/X\}$, generated between the fillers and repeated arguments of the antecedent are used to deduce the intermediate bindings, $\{a/U\}$, so that the substitution procedure substitutes all the occurrences of the argument X in the consequent with the constant a.

To complete inference, the same effect of the substitution procedure must be achieved in CASI by replicating it in a connectionist manner. Note that this substitution result represents the situation where all arguments (corresponding X's and Y) of the antecedent and consequent predicates are substituted with the constant a and the variable V. In CASI, this is achieved by activating all the argument nodes of the antecedent and consequent predicates in phase with the constant node *a[0]* and the variable node *U[0]* in the same phase. Because the argument nodes of the antecedent predicate p and the filler nodes already have been activated by the initial binding instantiation (in the first phase in case of the example), all we have to do is activate the predicate activator and the argument nodes of the consequent predicate q in the same phase as follows:

$$\begin{gathered}
a[1],\ U[2],\ V[3] \\
(p{:}\{pa[*],\ arg_1([0],[1]),\ arg_2([2],[0]),\ arg_2([3],[0])\},\ M_i, \\
q{:}\{pa[*],\ arg_1([2],[1]),\ arg_2([3],[0])\}\ )
\end{gathered}$$

The new in phase oscillatory behaviour between $q{:}arg_1([2],[1])$ and the filler nodes, *a[1]* and *U[2]*, represents the new binding $\{a/q{:}arg_1\}$ with the unification result $\{a/U\}$ and the other in phase oscillation between $q{:}arg_2([3],[0])$ and the variable node *V[3]* represents another new binding $\{V/q{:}arg_2\}$. This situation, as a whole, can be interpreted as the result of a symbolic substitution procedure, to yield $p(a, a, V) \rightarrow q(a, V)$. A connectionist way of achieving this effect is propagating the initial bindings represented on the antecedent predicate assembly to the consequent predicate assembly according to argument match between them. The easiest way to implement this is setting up binding propagation sub-mechanisms between the antecedent argument nodes and the constant argument nodes as well as between the predicate activator of the p predicate assembly and that of the q predicate assembly. The structure of these binding propagation sub-mechanisms can be as simple as ordinary links if a rule does not impose any consistency conditions. The example rule, however, requires the consistency checking sub-mechanism for the repeated arguments, X's in the antecedent, to complement the match procedure, the binding propagation sub-mechanism has to cooperate with a consistency checking sub-mechanism mentioned in the connectionist match procedure. The initial bindings on the antecedent predicate assembly should propagate to the consequent predicate assembly only if they do not violate consistency conditions imposed by this consistency checking sub-mechanism. Note that the first argument node of the q predicate assembly represent both the inference result and the unification result generated by the repeated argument of the antecedent. To achieve this activation state, we need yet an additional sub-mechanism to collect an intermediate bindings from the repeated argument nodes during inference.

This connectionist substitution mechanism suggests the intermediate mechanism, M_i, of the SPN to have not only the consistency checking sub-mechanisms but also the binding propagation sub-mechanism and an additional sub-mechanism to represent intermediate bindings. Chapter 5 will be devoted to describe how CASI decides all the necessary sub-mechanisms automatically to build a proper intermediate mechanism when encoding a rule and a fact.

4.7 Summary

This chapter proposes a connectionist architecture for symbolic inference (CASI). CASI can translates rules and facts in first-order Horn clause expressions into corresponding networks, called structured predicate networks (SPNs), and performs standard symbolic style of inference over them. To build SPNs from input rules and facts, CASI uses entity nodes and predicate assemblies consisting of π-btu elements.

To replicate symbolic inference, CASI extends the temporal synchrony approach to accommodate more general representation of dynamic bindings and provides connectionist equivalents to two basic symbolic inference procedures.

The initial binding instantiation procedure and the consistency checking sub-mechanism of the intermediate mechanism performs the task similar to that of the symbolic match procedure. In the similar way, the binding propagation sub-mechanism of the intermediate mechanism performs a similar task of symbolic substitution. When the presented predicate satisfies all the consistency condition forced by the consistency checking sub-mechanism, the initial activation will automatically propagate from the source predicate assembly to the target predicate assembly. The result of inference is then obtained by observing the predicate assemblies activated during inference.

Therefore, the role of a intermediate mechanism in CASI is important. It should include both consistency checking mechanisms and a binding propagation sub-mechanism. To achieve proper symbolic inference according to various types of rules, the knowledge compiler should be capable of constructing different types of intermediate mechanisms according to the types of rules and facts given. Chapter 5 will deal with all these issues in greater detail.

Chapter 5

Building Intermediate Mechanisms

5.1 Overview

One of the most important issues in encoding rules and facts into the corresponding SPNs is how to build an intermediate mechanism for each type of rule or fact. Since the representation of the source and target predicate assemblies and the dynamic binding mechanism alone cannot support the entire symbolic inference procedure required, the intermediate mechanism has to perform necessary sub-tasks to complete those procedures. To complete the match procedure, the intermediate mechanism has to provide consistency checking sub-mechanisms and to complete the substitute procedure the intermediate mechanism needs binding propagation sub-mechanisms.

This chapter describes the detailed procedure to decide the necessary sub-mechanisms of the intermediate mechanism when encoding each rule and fact. The contents are:

- basic definitions which are used throughout this chapter to describe the necessary conditions to decide the structure of an intermediate mechanism;
- the detailed procedure to determine the sub-mechanisms of the intermediate mechanism for each rule, with an example rule;
- the detailed procedure to decide the sub-mechanisms of the intermediate mechanism for each fact, with an example fact;

- the overall inference procedure over SPNs, encoding the example rule and fact.

5.2 Basic Definitions

The following are some basic definitions that will be used to decide the structure of intermediate mechanisms.

Definition For a given predicate and its base predicate, $p(A_1,\ A_2,\ \ldots,\ A_n)$ and $p(arg_1, arg_2, \ldots, arg_n)$, a *unifying argument group* (UAG) of the predicate p, G_{A_i}, is a set of base arguments whose corresponding arguments have an identical symbol name A_i. If arguments have a constant name, G_{A_i} is called a *constant UAG* and if argument have a variable name, G_{A_i} is called a *variable UAG*. Since each symbolic rule has source and target predicates, we can obtain two sets of UAGs from them. To differentiate a set of UAGs generated by the source predicate from that produced by the target predicate, we use two symbols, S_s and S_t. □

Definition For the two sets of UAGs corresponding to the source and target predicates of a rule:

$$S_s = \{G_{A_1}, G_{A_2}, \ldots, G_{A_n}\},$$
$$S_t = \{G_{A_1}, G_{A_2}, \ldots, G_{A_m}\},$$

we say that G_{A_i} in S_s is *related* to G_{A_j} in S_t and vice versa if A_i and A_j refer to the same symbolic argument name, i.e. $A_i = A_j$. In other words, two UAGs obtained from the source and target predicates are related if they are obtained from the same symbolic argument name. Any UAG which does not have the related UAG is called an *isolated* UAG. The set operators, "$\subset$", "$\supset$", and "$\equiv$" are used to represent relationships between any two sets of UAGs. □

For a more detailed explanation, let us consider the following rule:

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

When the given rule is seen in terms of forward chaining, we can obtain a set of UAGs from the source predicate, $p(X,X,Y,Y)$, and its base predicate, $p(arg_1,arg_2,arg_3,arg_4)$, as follows:

$$S_s = \{G_X, G_Y\}, \text{ where } G_X = \{p{:}arg_1, p{:}arg_2\},\ G_Y = \{p{:}arg_3, p{:}arg_4\}.$$

From the target predicate, $q(X, Y)$, and its base predicate, $q(arg_1, arg_2)$, we obtain:

$$S_t = \{G_X, G_Y\}, \text{ where } G_X = \{q{:}arg_1\},\ G_Y = \{q{:}arg_2\}$$

From S_s and S_t, we say that G_X in S_s is related to G_X in S_t because they are obtained from the same argument name X. Also G_Y in S_s is related to G_Y in S_t. Since every UAG in S_s is related to a UAG in S_t, we can say that $S_s \equiv S_t$.

Definition The *size of* a set of UAGs is the number of UAGs in the set and the size of a UAG is the number of base arguments in the UAG. If the size of a UAG is greater than 1, this means that the UAG was obtained from repeated arguments of a predicate. □

When considering the following rule in terms of forward chaining:

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

The S_s and S_t of the rule are

$$S_s = \{G_X, G_Y\}, \text{ where } G_X = \{p{:}arg_1\},\ G_Y = \{p{:}arg_2\},$$
$$S_t = \{G_X, G_Y, G_Z\}, \text{ where } G_X = \{q{:}arg_1\},\ G_Y = \{q{:}arg_2\},\ G_Z = \{q{:}arg_3\}.$$

The size of S_s and S_t are $size(S_s) = 2$ and $size(S_t) = 3$ and the size of all UAGs in S_s and S_t are 1.

Definition A *balanced rule* is a rule which does not have any isolated UAG in the target predicate. A rule which has isolated UAG(s) in the target predicate is called an *unbalanced rule*. □

The above example rule is an unbalanced rule because, G_Z in S_t is an isolated UAG.

5.3 Intermediate Mechanisms for Rules

5.3.1 Some Considerations

When a symbolic rule is given to be encoded into the corresponding SPN, its source and target predicates are represented by creating their corresponding base predicate assemblies as described in Section 4.4.4. The main issue is how to build a proper intermediate mechanism between them because it has to be designed not only to provide a path for binding propagation but also to prevent incorrect substitution during unification. This needs to take into account binding interaction within each UAG and across different UAGs in the source predicate to produce intermediate bindings when necessary; consistency checking on intermediate bindings generated by binding interaction to complete unification; and propagation of initial bindings from a UAG in the source predicate to its related UAG in the target predicate to achieve the effect of substitution procedure. To decide necessary sub-mechanisms which perform these tasks, the compiler has to consider the various syntactic conditions that each rule imposes.

To explain a procedure for deciding necessary sub-mechanisms, let us reconsider the following example rule:

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

The corresponding SPN of this rule is

$$\begin{aligned} SPN_i &= \{P_{s_i}, M_i, P_{t_i}\}, \text{ where} \\ P_{s_i} &= p{:}\{pa[0],\ arg_1([0],[0]),\ arg_2([0],[0]),\ arg_3([0],[0])\}, \\ P_{t_i} &= q{:}\{pa[0],\ arg_1([0],[0]),\ arg_2([0],[0])\}, \\ M_i &= \text{the intermediate mechanism between } \mathrm{P}_{s_i} \text{ and } \mathrm{P}_{t_i}. \end{aligned}$$

To refer a sub-mechanism needed between arguments in the predicate assemblies, we obtain S_s and S_t of the rule as follows:

$$\begin{aligned} S_s &= \{G_X, G_Y\}, \text{ where } G_X = \{p{:}arg_1, p{:}arg_2\},\ G_Y = \{p{:}arg_3\}, \\ S_t &= \{G_X, G_Y\}, \text{ where } G_X = \{q{:}arg_1\},\ G_Y = \{q{:}arg_2\}. \end{aligned}$$

Using a UAG of the source or target predicate of a rule is one way of grouping arguments appearing both predicates according to the symbolic names by which they are represented in the rule. When the compiler determines necessary sub-mechanisms for the intermediate mechanism, base arguments of a UAG in a predicate correspond to argument nodes of the predicate assembly encoding that predicate. Thus building sub-mechanisms between UAGs means constructing connectionist sub-mechanisms between argument nodes corresponding base arguments of UAGs.

The required sub-mechanisms can be categorised into two classes:

- sub-mechanisms required within each UAG in the source predicate;
- sub-mechanisms required between each pair of UAGs in the source predicate.

Firstly, there are three types of sub-mechanisms required within each UAG in the S_s: the binding collection sub-mechanism, the consistency checking sub-mechanism, and the binding propagation sub-mechanism. Binding collection refers to the limited unification procedure within a UAG if it has repeated arguments. When these repeated arguments get bound to a constant or free variable fillers, the binding collection procedure carries out a limited unification between the fillers. If the predicate $p(a, U, V)$ is presented to the example SPN, two fillers, a and U, are assigned to the UAG, G_X in S_s. Therefore, a binding collection sub-mechanism is necessary to collect bindings represented on the repeated arguments so that the intermediate binding, $\{a/U\}$, can be represented. Since this procedure simply collect bindings without checking any consistency for them, it is a very limited unification procedure and should not be used on its own. A UAG which has a single variable argument does not require this sub-mechanism.

Before the binding propagation is carried out, the consistency of the intermediate bindings and the initial bindings are checked first. This ensures the conditions that: any arguments of a constant UAG in the source predicate should not get bound to the constant filler other than the one specified as the constant argument name; and the arguments of the UAG whose size is greater than 1 (a repeated variable UAG) should not get bound to more than one constant filler at the same time. Therefore special mechanisms are needed to enforce these consistency conditions during inference. If the predicate, *p(a,b,V)*, is presented to the example SPN, this consistency checking sub-mechanism should be able to detect a consistency violation within the UAG, G_X in S_s, because two different constants are assigned to the repeated variable arguments.

After considering the binding collection and consistency checking, initial bindings established on argument nodes of each UAG in the source predicate need to be propagated to the those of the related UAG in the target predicate. This binding propagation will be achieved by a binding propagation sub-mechanism. When the predicate, *p(a,a,b)*, is presented to the example SPN, the initial constant bindings established on the G_X and G_Y in S_s must propagate to the G_X and G_Y in S_t to produce the desirable result, *q(a,b)*. Those UAGs which only appear in the source predicate (isolated UAGs) do not need this sub-mechanism but they may be used for further consistency checking or binding interaction across different UAGs in the source predicate.

Secondly, there are two sub-mechanisms required between each pair of UAGs. The first sub-mechanism performs binding interaction between two UAGs and the other sub-mechanism carries out consistency checking after binding interaction. Binding interaction between UAGs refers to the situation where two sets of bindings generated by two UAGs need to unified together. This situation occurs when the same variable filler is assigned to arguments belonging to two different UAGs, provided one of UAGs has a constant argument or repeated variable arguments. This procedure makes sure the further unification process between two UAGs – a required property to achieve symbolic inference. For instance, if the predicate *p(a,U,U)* is presented to the example SPN, the intermediate binding obtained from G_X in S_s needs further interaction with the variable binding obtained from G_Y in S_s because they share the same variable filler. In CASI's rule encoding mechanism, this interaction will be carried out by a separate sub-mechanism. The result, *p(a,a)*, cannot be obtained without further binding interaction by this sub-mechanism. After this binding interaction, another consistency checking procedure is needed to make sure of the consistency of the bindings produced as the result of the binding interaction. Whenever a consistency violation occurs, this procedure detects it and blocks the binding propagation from the source UAG to the target UAG.

The following procedure summarises an overall procedure of encoding a symbolic rule including how to determine the necessary sub-mechanisms for the intermediate mechanism:

PROCEDURE: *encoding a symbolic rule*

Step 1: compute S_s and S_t for a given rule;

Step 2: build base predicate assemblies corresponding to the source and target predicates;

Step 3: build the intermediate mechanism in such a way that

(1) within each UAG:

- for a UAG, G_{A_i}, in S_s whose size is greater than 1 (repeated arguments) build a sub-mechanism for binding collection;
- for a UAG, G_{A_i}, in S_s, if it is a constant UAG or a variable UAG whose size is greater than 1, build a sub-mechanisms for consistency checking;
- for a UAG, G_{A_i}, in S_s which has the related UAG, G_{A_j}, in S_t, where $A_i = A_j$, build a sub-mechanism for binding propagation from G_{A_i} in S_s to G_{A_j} in S_t;

(2) between a pair of UAGs:

- for each pair of UAGs, G_{A_i} and G_{A_j} in S_s, where $A_i \neq A_j$, if any of G_{A_i} and G_{A_j} is a constant UAG or a variable UAG whose size is greater than 1, build sub-mechanisms for further binding interaction and consistency checking.

The following subsections will describe in more detail procedures for deciding necessary sub-mechanisms within each UAG and between a pair of UAGs in the source predicate.

5.3.2 Sub-Mechanisms Required within Each UAG

The sub-mechanism required within each UAG are the binding collection sub-mechanism, the consistency checking sub-mechanism, and the binding propagation sub-mechanism.

The Binding Collection Sub-Mechanism

The purpose of the binding collection sub-mechanism (BCM) is to produce intermediate bindings and to represent them in a connectionist manner. Each UAG in the source predicate can has either a single argument or repeated arguments. A UAG which has a single argument will represent only one binding on the corresponding argument node during the initial binding instantiation procedure. A UAG which has repeated arguments, on the other hand, will represent a set of bindings over all its argument nodes. Since any repeated arguments should get bound to the same constant filler or free variable fillers, the BCM is required to collect all bindings generated from the repeated arguments for further consistency checking. A group of UAGs which requires this mechanism can be determined by checking their size as follows:

$$BCM = \{G_{A_i} |\ G_{A_i} \in S_s, size(G_{A_i}) > 1\}.$$

Figure 5.1 depicts the structure of a BIM for G_{A_i} in S_s. The argument nodes labelled arg_j, ..., arg_k represent the repeated arguments of the UAG. The binding node *b1* and links from the argument nodes serve as the BCM. When the initial bindings are assigned to the argument nodes, all their variable bindings are propagate to the left element of *b1* and all their constant bindings to the right element of *b1*. Consequently, the left element of *b1* represents all the variable bindings of the argument nodes and the right element all the constant bindings of the argument nodes. These bindings are called the *intermediate bindings* on *b1*.

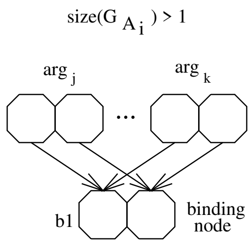


Figure 5.1: The structure of a binding collection sub-mechanism

The Consistency Checking Sub-Mechanism

Two types of sub-mechanisms are required for consistency checking during inference: consistency checking for a constant UAG (CCM_c); consistency checking for a repeated variable UAG (CCM_v). The CCM_c forces the condition that all arguments of a constant UAG in the source predicate must get bound to the constant specified in the constant argument or free variable fillers during inference. This condition should be forced whether the UAG is obtained from repeated arguments or not (regardless of its size). In the same way, the CCM_v forces the condition that all the repeated arguments in a variable UAG should get bound to the same constant filler or free variable fillers. The groups needing these treatments are determined by

$$CCM_c = \{G_{A_i}|\ G_{A_i} \in S_s, const(A_i)\},$$
$$CCM_v = \{G_{A_i}|\ G_{A_i} \in S_s, var(A_i), size(G_{A_i}) > 1\},$$

the predicates, *const(A_i)* and *var(A_i)*, check if the name of the UAG, G_{A_i}, is a constant or a variable. Constructing corresponding networks for these two types of consistency checking sub-mechanisms require us to consider whether the given UAG is isolated or not. Since non-isolated UAGs do not need a BPM, this affects the structure of the network to be built.

Figure 5.2a illustrates the CCM_c for an ordinary constant UAG, G_{A_i} in S_s, which has the related UAG, G_{A_i} in S_t. The name of the constant UAG is assumed to be c, i.e. $A_i = c$, for the purpose of explanation. The CCM_c consists of the *to1* node, which is a τ-or element, and the link from the right element of the source node which is modified by the constant node c. The source node is the associated binding node, inserted as the BCM, if a UAG has repeated argument, otherwise the source node is the argument node(s) of the UAG, G_{A_i} in S_s. The dashed lines are inhibitory links and the solid lines are excitory ones. The *to1* node will get activated whenever the right element of the source node gets bound to a constant other than c. The link from the right element of the source node which is modified by the inhibitory signal coming from the constant node ensures this condition. On becoming active, the *to1* node projects an inhibitory signal which will be used as an argument and rule inhibitory signal. As an argument inhibitory signal, it inhibits the binding propagation sub-mechanism that

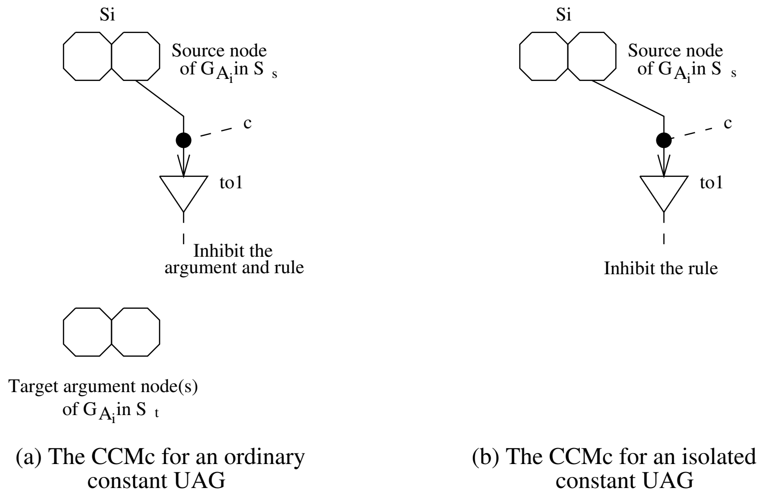


Figure 5.2: The structure of a consistency checking sub-mechanism for a constant UAG

will be inserted between the source node of G_{A_i} in S_s, and the target argument node(s) of G_{A_i} in S_t, which prevents the target argument nodes from becoming active. As a rule inhibitory signal, it is used to inhibit other binding propagation sub-mechanisms to be built for other UAGs in the source predicate.

If the given constant UAG is an isolated one, the CCM_c will be established as a stand-alone sub-mechanism. Figure 5.2b depicts the corresponding network for this. If consistency violation occurs during inference, *to1* becomes active and its inhibitory output will be used as a rule inhibitory signal.

The other sub-mechanism, CCM_v, is relatively easy to implement. When a repeated variable UAG in S_s has the related UAG in the target predicate, the CCM_v for the UAG is implemented using an *mto1* node, which is a multiphase τ-or element, as shown in Figure 5.3a. Whenever the right element of the source node receives two different constant bindings, the *mto1* node will detect this and projects an inhibitory signal which will be used as an argument and rule inhibitory signal.

When the variable UAG is an isolated UAG, the output of *mto1* is used only as a global inhibitory signal as can be seen in Figure 5.3b.

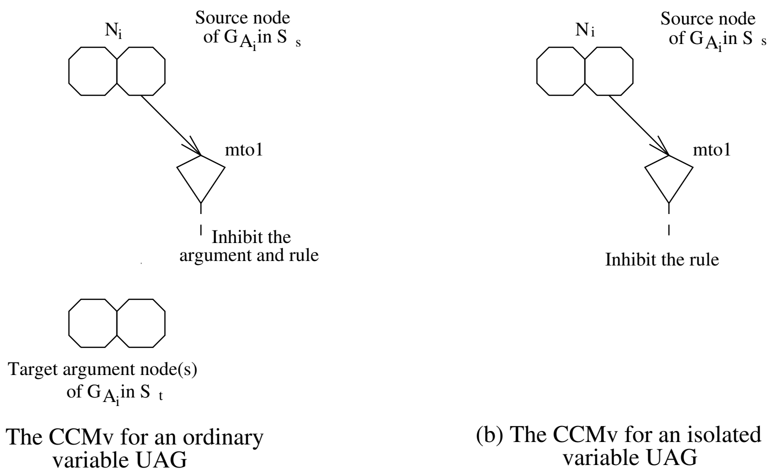


Figure 5.3: The structure of a consistency checking sub-mechanism for a variable UAG

The Binding Propagation Sub-Mechanism

Whenever a UAG, G_{A_i} in S_s, has the related UAG, G_{A_j} in S_t, a binding propagation mechanism (BPM) is needed to provide a path for binding propagation between them. This sub-mechanism directly connects the argument nodes representing arguments of the source UAG and their counterpart representing arguments of the target UAG. If the source UAG has an associated BCM, the links for the BPM start from the binding node inserted as the BCM.

The BPM constructing procedure involves providing one of two different circuitries for each type of UAG: a BPM for a UAG which has constant argument (BIM_c); a BPM for a UAG which has variable arguments (BIM_v). Firstly, the set of UAGs which require a BPM_c are determined by the following condition:

$$BPM_c = \{G_{A_i} |\, G_{A_i} \in S_s, G_{A_i} \in S_t, const(A_i)\},$$

Figure 5.4 demonstrates a sub-mechanism corresponding to the BPM_c. Again the constant argument, A_i, is assumed to be c for the purpose of explanation. When the target UAG has a single argument, the BPM_c connects the source node and the argument node representing the target argument as shown in Figure 5.4a. Variable binding propagation is achieved by a simple link between $N_{i_{left}}$ and the left element of the target argument node and constant binding propagation by two separate links between c and the right element of the target argument node and between *c:gate* and the right element of the target argument node to deal with two different situations.

The first situation occurs if the source node gets bound to a variable filler after initial binding instantiation. The binding set up on the left element of the source node first propagates to the *c:gate* node and then to the right element of the target argument node, as well as to the constant node c. This makes both the right element of the target node and the constant node c active in the same phase. While, the binding set up on the left element of the source node also propagates to the left element of the target argument node. The activation states of the target argument nodes represent the result of the unification between the variable filler and the constant argument. When several rules share the same constant, each rule requires a separate *gate* node for the constant to represent its own binding state.

The second situation occurs if the source node gets bound to a constant filler c, for instance, the right element of the target argument node becomes active by the direct input coming from the constant node. The constant node becomes active during the initial binding instantiation when the filler is presented to the argument of the source predicate. Therefore, receiving activation from the c node makes the right element of the target argument node active in-phase with the constant node. This achieves the required constant binding propagation to the target argument node. Activation of the constant node also makes the *c:ihb* node active, whose output blocks the other binding propagation path from *c:gate* to the right element of the target argument node so that the target argument nodes receives input only from the constant node.

If the related UAG, G_{A_i} in S_t, has repeated arguments, a separate set of binding propagation mechanisms are necessary from the source node to each target argument node. Figure 5.4b shows n different sets of connections between the source node and the n target argument nodes when the related UAG has n repeated arguments.

Note that, in Figure 5.4a, it takes two oscillation cycles for activation to be propagated from $N_{i_{left}}$ to the left element of the target argument node, whereas it takes one

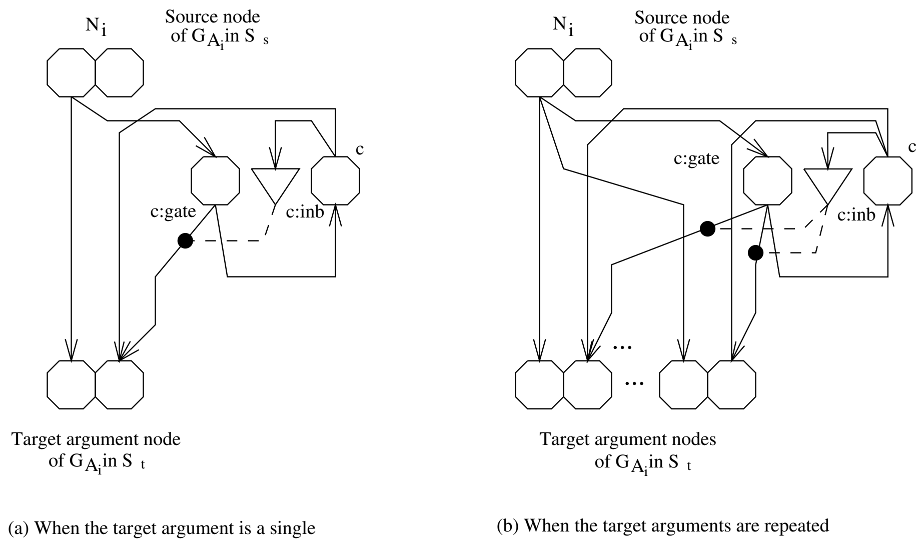


Figure 5.4: The structure of a binding propagation sub-mechanism for a constant UAG

oscillation cycle from $N_{i_{left}}$ to the *c:gate* node and another one oscillation cycle from the *c:gate* node to the right element of the target argument node. This time delay between $N_{i_{left}}$ and the left element of the argument node is to ensure the activation projected from $N_{i_{left}}$ reaches the left and right element of the target argument node at the same time to represent binding correctly. This sort of time delay, whose length depends on a type of intermediate mechanism, is computed by the rule compiler in advance and applied when the corresponding network is built. Also note that neuron elements continue to oscillate in phase with an input signal even after the element's input has ceased – as long as they are under the current focus of attention for a chain of inference.

In a similar way, a set of UAGs which require BPM_v is determined by the following condition:

$$BPM_v = \{G_{A_i} | \; G_{A_i} \in S_s, G_{A_i} \in S_t, var(A_i)\},$$

If the related UAG has a single argument, the structure of the BPM_v simply connects the source node and the target argument node of the related UAG as shown in Figure 5.5a. Unlike the $BPMc$, the BPM_v consists of simple links and does not have any associated nodes. Whenever the source node gets activated, this activation directly propagates to the target argument node through these links. If the related UAG has repeated arguments, additional sets of links are required between the source node and each target argument node as shown in Figure 5.5b.

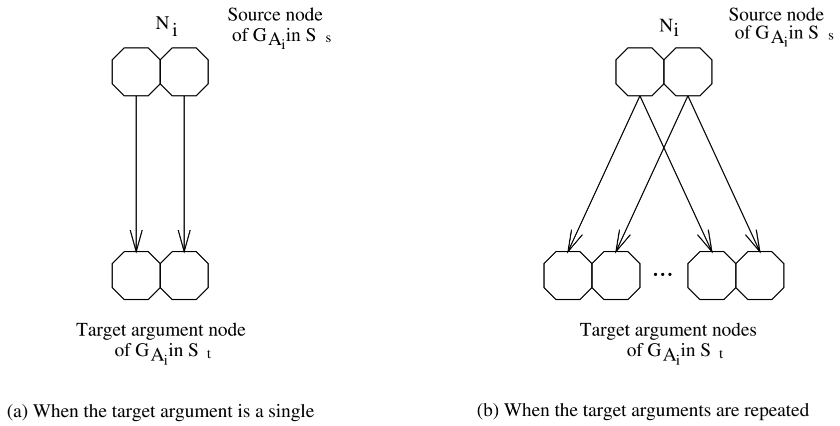


Figure 5.5: The structure of a binding propagation sub-mechanism for a variable UAG

If the source UAG, G_{A_i}, does not have a related UAG in the target predicate (i.e. isolated), the BPM is not required to provide paths for binding propagation between them. However, the source node N_i may be used for a consistency checking sub-mechanism within the UAG or for a binding interaction sub-mechanism across different UAGs.

5.3.3 Sub-Mechanisms Required Between Each Pair of UAGs

For any pair of UAGs in the source predicate, if either or both of them are constant UAGs or variable UAGs which have repeated arguments, binding interaction across those two UAGs and consistency checking after binding interaction is required to ensure proper propagation of bindings from the source predicate assembly to the target predicate assembly. There are three combinations in pairing two different types of UAGs:

- a pair of a constant UAG and a variable UAG,

- a pair of two variable UAGs,
- a pair of two constant UAGs.

The following specifies detailed structures of the required sub-mechanisms between each pair of UAGs.

Sub-Mechanisms Between Constant and Variable UAGs

Between a constant UAG and a variable UAG, we require a binding interaction sub-mechanism (BIM_{cv}) and a consistency checking sub-mechanism (CCM_{cv}). The binding interaction between UAGs refers to the situation where the same variable filler is assigned to argument nodes belonging to two different UAGs. This situation requires the binding obtained between the variable filler and the constant UAG to migrate to the variable UAG. Since this binding interaction procedure does not involve consistency checking, a separate consistency checking sub-mechanism is needed. The pairs of UAGs which require these two sub-mechanisms are determined by the following condition:

$$BIM\&CCM_{cv} = \{(G_{A_i}, G_{A_j})|\ G_{A_i} \in S_s, G_{A_j} \in S_s, const(A_i), var(A_j)\}.$$

Before these sub-mechanisms are built, the sub-mechanisms required within each UAG have to be constructed first: a BCM if any of UAGs has repeated arguments, CCMs for constant UAGs and for variable UAGs which has repeated arguments, and BPMs. The $BIM\&CCM_{cv}$ is then inserted into the intermediate mechanism. The structure of these mechanisms is illustrated in Figure 5.6a.

The N_i is the source node of the constant UAG (G_{A_i}) and the N_j that of the variable UAG (G_{A_j}). The name of the constant UAG is assumed to be c.

Firstly, the BIM_{cv} consists of *to2*, *b2*, and links coming from the source nodes and the constant node c. The node *to2* is used to check if both left elements of source nodes get bound to the same variable filler. This is done by checking if those elements are active in the same phase. Since the *to2* has threshold 2, it becomes active only when it receives two inputs in the same phase. On becoming active, *to2* sends an inhibitory

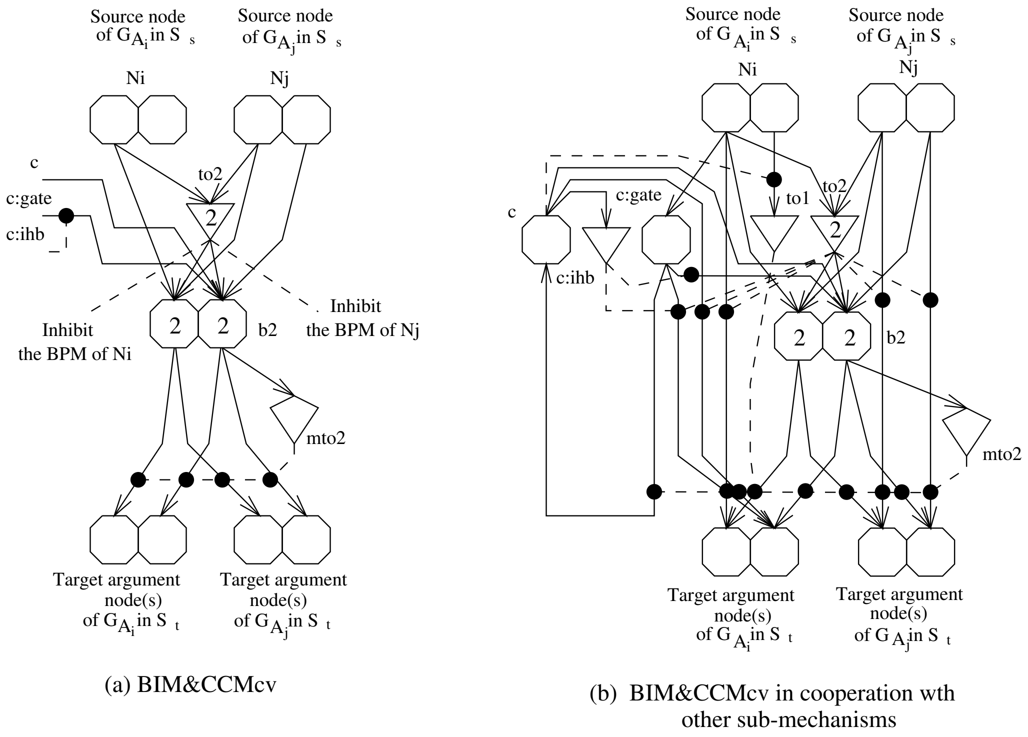


Figure 5.6: The structure of sub-mechanisms between a constant UAG and a variable UAG

signal to the BPMs for the two UAGs and, at the same time, sends an excitory signal to the binding node *b2*. Blocking the BPMs allows the initial bindings to propagate to the target argument nodes only through the binding node. Since the left and right elements of the binding node receive activation from those of the two source nodes, the binding node represents intermediate bindings when *to2* becomes active. Note that the right element of *b2* has two links coming from the constant UAG side, one coming from *c* and the other from *c:gate*. As explained in the constant consistency checking sub-mechanism, the right element of the binding node receives activation from the *c* constant node if it is activated at the beginning of inference, otherwise it receives activation from the *c:gate* node. Any intermediate bindings represented on the binding node are generated as the result of binding interaction between the two UAGs. The consistency of these bindings has to be checked next by the *mto2* node which serves as the CCM_{cv}. Note that the dashed link from *mto2* to one black dot near to the bottom of the network, which is connected to three other black dots by a dashed link. This is to reduce the complexity of a full connection from *mto2* to each black dot in the drawing. Intermediate binding propagation from the binding node to the target argument nodes occurs only if those bindings do not violate consistency enforced by the *mot2* node. When the *mot2* node is associated with two $BIM\&CCM$s at the same time, it can be used to ensure consistency across pairs of UAGs. For instance, if the constant UAG, G_{A_k}, is involved in two $BIM\&CCM$s, one with G_{A_i} and the other with G_{A_j} at the same time, any consistency violation occurring in either $BIM\&CCM$ with G_{A_k} will block the binding propagation from the source nodes to the target argument nodes through both $BIM\&CCM$s.

Figure 5.6b shows how these $BIM\&CCM_{cv}$ cooperate with other sub-mechanisms in the intermediate mechanism.

If G_{A_i} and G_{A_j} are isolated UAGs, the corresponding network will not have appropriate target argument nodes. However, the outputs of *mto2* will serve as a rule inhibitory signal to prevent a rule from firing.

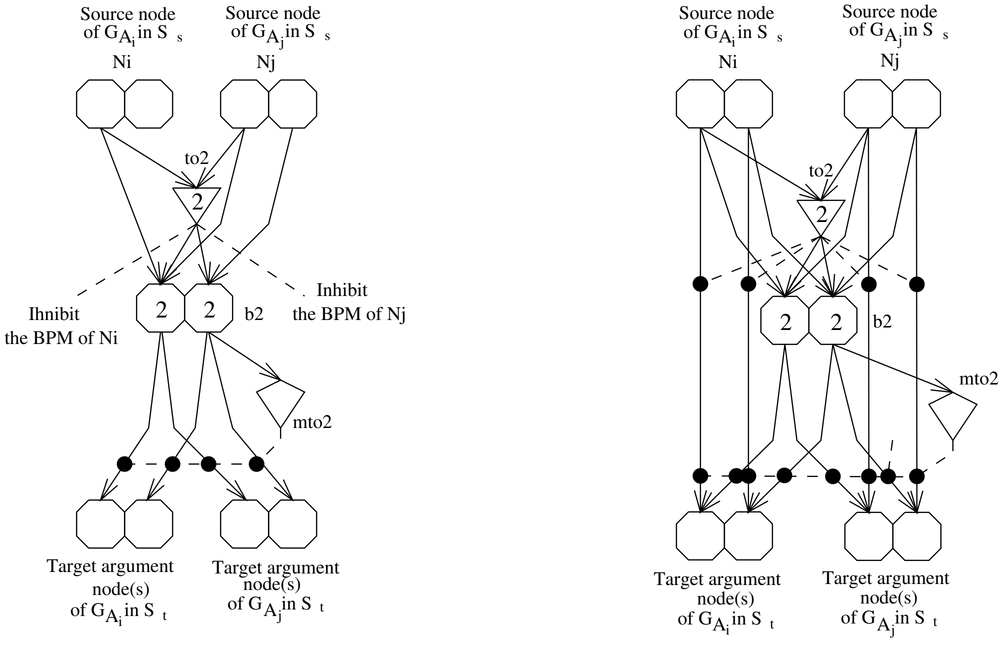
Sub-Mechanisms Between Variable UAGs

A *BIM* and a *CCM* between variable UAGs (a BIM_{vv} and a CCM_{vv}) are only needed when one of them has repeated arguments so that bindings generated by the repeated arguments interact with those produced by the argument(s) of the other UAG. If both variable UAGs have a single argument, these sub-mechanisms are not required because no binding interaction is needed between them. The pairs of UAGs which require the BIM_{vv} and the CCM_{vv} is determined by the following condition:

$$BIM\&CCM_{vv} = \{(G_{A_i}, G_{A_j}) |\ G_{A_i} \in S_s, G_{A_j} \in S_s, var(A_i), var(A_j), A_i \neq A_j,$$
$$size(G_{A_i}) > 1 \ or \ size(G_{A_j}) > 1\}.$$

Figure 5.7a illustrates the corresponding circuitry for these sub-mechanisms and Figure 5.7b depicts how this $BIM\&CCM_{vv}$ cooperate with BPMs.

When starting inference, if the same variable filler is assigned to argument nodes in G_{A_i} and G_{A_j}, both left elements of the source nodes become active. This activation



(a) BIM&CCMvv (b) BIM&CCMvv in cooperation with BPMs

Figure 5.7: The structure of sub-mechanisms between variable UAGs

then propagates to the *to2* node and makes it active. On becoming active, the *to2* node projects an inhibitory signal to block the BPMs and also sends an excitory signal to the binding node, *b2*. Then, the binding node represents intermediate bindings produced as the result of the binding interaction between these two source nodes. After consistency checking by the *mto2* node, the intermediate bindings are propagated to the target argument nodes. If the same variable filler is not assigned to the two different UAGs, *to2* will not be activated. As a result, the initial bindings are propagated directly to the target argument nodes through the BPMs.

Sub-Mechanisms Between Constant UAGs

The binding interaction between two constant UAGs refers to the situation where the same variable filler is assigned to the constant argument nodes belonging to two constant UAGs. This situation allows the same variable filler to get bound to two different constant arguments. The purpose of the binding interaction sub-mechanism between constant UAGs (BIM_{cc}) is to detect this situation and to prevent the rule from firing. Unlike the previously mentioned two $BIM\&CCM$s, this sub-mechanism does not require a CCM_{cc} because their consistency is checked by separate CCMs associated with each constant UAG. This allows the sub-mechanism between constant UAGs to be much simpler the other two cases. The pairs of UAGs which require this sub-mechanism is determined by the following condition:

$$BIM_{cc} = \{(G_{A_i}, G_{A_j}) |\ G_{A_i} \in S_s, G_{A_j} \in S_s, const(A_i), const(A_j), A_i \neq A_j\}.$$

The structure of a BIM_{cc} is demonstrated in Figure 5.8a.

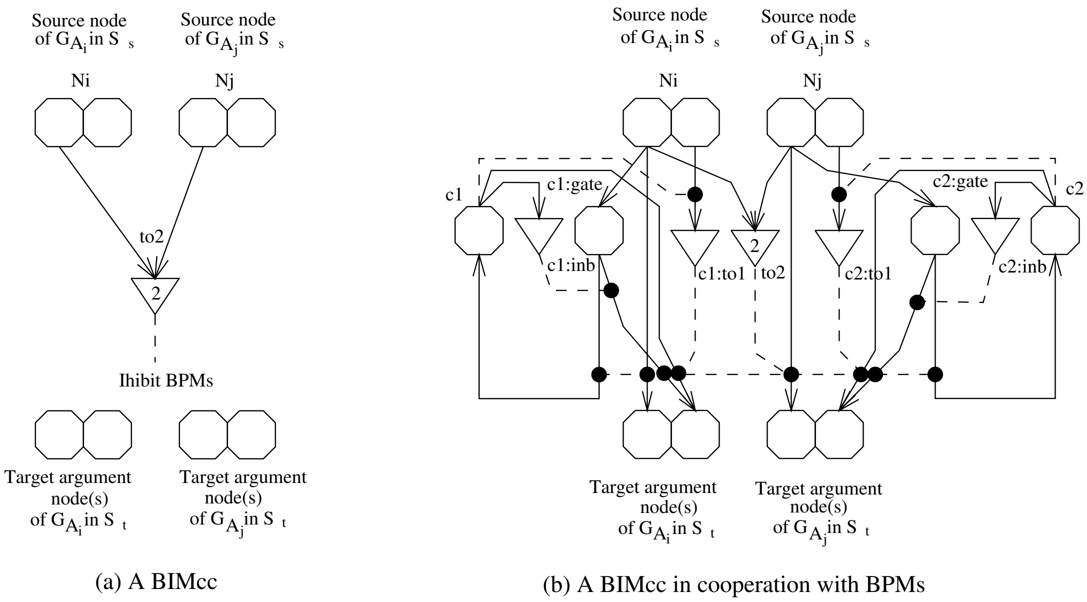


Figure 5.8: The structure of sub-mechanisms between constant UAGs

Again the source nodes, N_i and N_j, correspond to the two UAGs, G_{A_i} and G_{A_j} in S_s. As shown in the figure, the BIM_{cc} is represented only with the *to2* node. This node detects the situation where the same variable filler is assigned to the constant argument nodes of the two constant UAGs. When the *to2* node becomes active, its output is used as a rule inhibitory signal which blocks any binding propagation to the target arguments by other BPMs or BIM&CCMs. Figure 5.8b illustrates how the BIM_{cc}

cooperates with other sub-mechanisms. The names of the two UAGs are assumed to be G_{c1} and G_{c2}. Each constant UAG has the associated CCM_c. The inhibitory link between the *to2* node and one of the block dots are used to prevent the rule from firing if the *to2* node becomes active during inference.

5.3.4 An Example of Encoding a Rule

To demonstrate the entire rule encoding procedure, let us reconsider the following example rule:

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

Encoding the rule starts by constructing the base predicate assemblies which correspond to the predicates appearing in the rule. Then the intermediate mechanism is built between them, which involves the following procedure to determine the required sub-mechanisms.

First, the syntax of the rule is checked to obtain two sets of UAGs (S_s and S_t). From the example rule, we obtain

$$S_s = \{G_X, G_Y\}, \text{ where } G_X = \{p{:}arg_1, p{:}arg_2\},\ G_Y = \{p{:}arg_3\}$$
$$S_t = \{G_X, G_Y\}, \text{ where } G_X = \{q{:}arg_1\},\ G_Y = \{q{:}arg_2\}$$

Secondly, the number of arguments of each UAG in S_s is examined. The antecedent of the example rule has two UAGs: G_X which has two occurrences of the variable argument X and G_Y which has the single variable argument Y. Since $size(G_X) > 1$ and $size(G_Y) = 1$ only G_X requires the binding collection sub-mechanism (BCM):

$$BCM = \{G_X\}.$$

Thirdly, in order to determine the consistency checking sub-mechanisms, the types of arguments are checked. In the case of the example rule, we only need the CCM_v for the repeated argument X's because it does not have any constant argument. This CCM_v ensures the consistency condition that all variable fillers which get bound to these repeated arguments should be the same constant filler or free variable fillers. Thus we get

$$CCM_c = \{\},\ CCM_v = \{G_{X_s}\}.$$

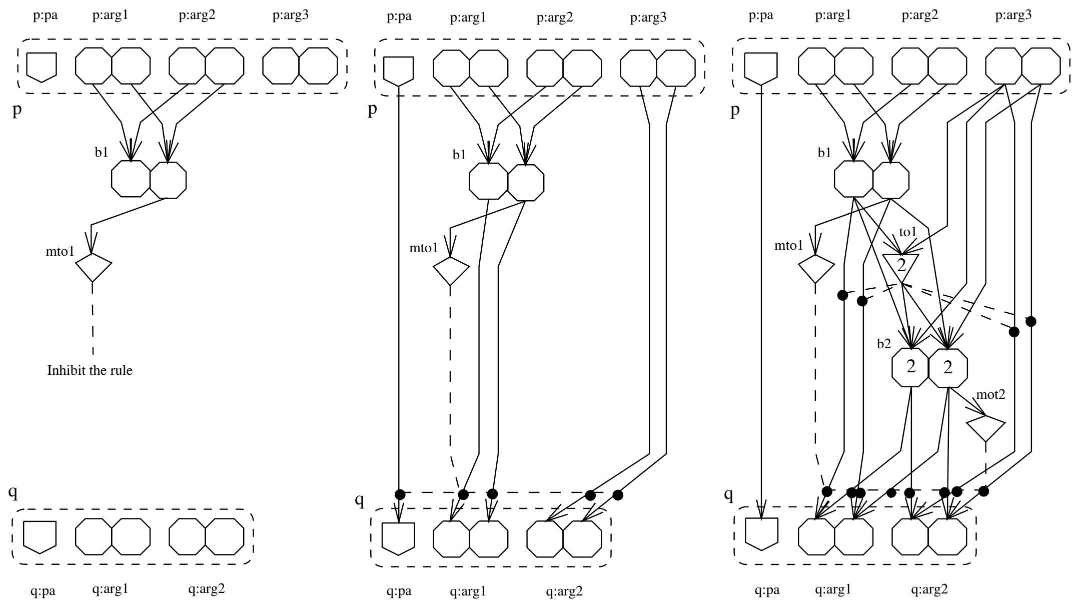
In the next stage, the matching between arguments in the antecedent and those in the consequent must be checked. This is carried out by examining each related UAG pair in S_s and S_t. Since both UAGs in S_s have their related UAGs in S_t, binding propagation sub-mechanisms (BPM) are needed between the first two arguments of p and the first argument of q and between the third argument of p and the second argument of q:

$$BPM = \{G_X, G_Y\}.$$

Finally, binding interaction and consistency checking between each pair of UAGs will be accommodated. As explained in the previous section, these sub-mechanisms between variable arguments ($BIM\&CCM_{vv}$) is needed for the pair of UAGs, G_{X_s} and G_{Y_s}:

$$BIM\&CCM_{cv} = \{\},\ BIM\&CCM_{vv} = \{(G_X, G_Y)\},\ BIM_{cc} = \{\}.$$

Figure 5.9c shows the complete network generated for the example rule. Two base predicate assemblies are shown at the top and bottom of the network and between them are the BCM, the CCM_v, the BPM_v, and the $BIM\&CCM_{vv}$. The binding node *b1* is used as the BCM, the *mto1* node as the CCM_v, and the rest as the BPM_v and the $BIM\&CCM_{vv}$. Although the complete network looks very complex, it is modularised. Figure 5.9a shows the network only with the BCM and the CCM_v and Figure 5.9b with the additional BPM_v. As can be seen, the complexity of the network mainly due to the binding interaction and consistency checking mechanisms between two UAGs. Rules which do not require this mechanism are encoded with a much simpler intermediate mechanism and therefore shorten the time taken for a step of inference on the network. Section 7.3.1 will deal with this issue in greater detail.



(a) BCM and CCMv (b) BCM, CCMv and BPMv (c) BCM, BPMv, CCMv and BIM&CCMvv

Figure 5.9: Building the intermediate mechanism for the rule $p(X, X, Y) \rightarrow q(X, Y)$

5.3.5 Encoding Rules with Multiple Predicate Antecedent

Until now, we have dealt with how to encode rules whose antecedent has only one predicate. However, there are rules whose antecedent consists of more than one predicate. An example of such a rule is as follows:

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

To obtain a right inference result from the r predicate, the two predicates p and q must be activated at the same time with their initial bindings. Because this rule is the same as the one shown in the previous sub-section except the antecedent is divided into two predicates, encoding the rule has to build the same intermediate mechanism but with an additional sub-mechanism which makes the rule fire only when the two antecedent predicates become active. This is achieved by inserting one τ-and element and n binding nodes if the rule has n target UAGs. Figure 5.10 depicts these new nodes and shows how these nodes are used in cooperation with existing sub-mechanisms.

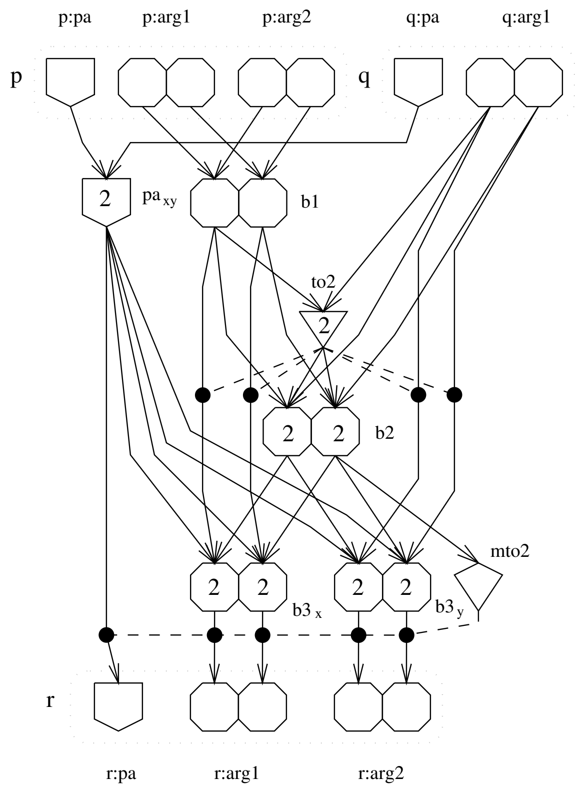


Figure 5.10: Encoding the rule $p(X, X) \wedge q(Y) \rightarrow r(X, Y)$ which has the multiple antecedent predicates

The node pa_{xy} is the τ-and element and two nodes $b3_x$ and $b3_y$ are the binding nodes inserted. The node pa_{xy} becomes active only if it receives two input signals from *p:pa* and *q:pa* at the same time. Consequently, this node is used to check if both predicate assemblies are active at the same time at the beginning of inference. Once it becomes active, its output is sent to the two bindings nodes ($b3$'s) to allow binding propagation from the source argument node (*q:arg*$_1$) or the binding node (*b2*), to the target argument nodes (*r:arg*$_1$ and *r:arg*$_2$). These binding nodes have the threshold 2 and will be activated on receiving inputs from the pa_{xy} and from one of other links at the same time. Therefore, these nodes are used to control the binding propagation sub-mechanism according to the situation whether pa_{xy} becomes active or not.

5.4 Intermediate Mechanisms for Facts

5.4.1 Some Considerations

Symbolically, an n-ary fact is represented by the fact name followed by n constant arguments. It has the same structure as that of an ordinary predicate except that it has pre-established bindings. One operation we can perform against encoded facts is posing a query predicate with constant or variable fillers to see if there is any fact which matches with the given query in the logical database. If there is a match, we expect a set of bindings between the constant arguments of the fact and variable fillers of the given query predicate. Thus, implementing a fact encoding mechanism in a connectionist manner requires us to provide a connectionist equivalent to this query processing. To demonstrate connectionist sub-mechanisms required to support this, let us consider the situation where the example fact, $p(a,a,b)$, is encoded into the logical database:

- posing the query predicate, $p(a,b,c)$, should fail to match with the example fact because the second and the third constant fillers are not the ones specified in the encoded fact. In general, posing a query predicate which has only constant fillers requires consistency checking within each UAG to check the consistency between each argument of the encoded fact and corresponding filler of the query predicate;

- posing the query predicate, $p(U,U,V)$, will result in the affirmative answer with the binding $\{U/a, V/b\}$. This indicates that posing a query predicate which has only variable fillers first requires consistency checking across UAGs which checks the situation where the same variable filler gets bound to two different constant arguments at the same time. If the query predicate does not violate consistency, bindings between the variable fillers and constant arguments should be represented in some way;

- posing the query predicate, $p(a,U,b)$, should also result in an affirmative answer with the binding {U/a}. This shows that posing a query predicate which has mixture of constant and variable fillers requires both types of consistency checking and binding representation mechanisms.

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

- it requires built-in connectionist mechanisms for consistency checking both within each UAG and across UAGs;
- unlike symbolic systems, the result of variable bindings should be explicitly represented on a network in the form of activation.

These conditions suggest that encoding a fact requires associated sub-mechanisms to check consistency and an additional sub-mechanism to represent result bindings. CASI's fact encoding mechanism uses a fact assembly and the intermediate mechanism for this purpose. The structure of the fact assembly is the same as that of the base predicate assembly. We use the suffix "$_f$" after predicate name to differentiate the fact assembly from the base predicate. The fact assembly of the given fact *p(a,b,c)*, for instance, is represented as $p_f\{pa[0], arg_1([0],[0]), arg_2([0],[0]), arg_3([0],[0])\}$. Therefore, constructing an SPN for a fact involves building the base predicate assembly, the fact assembly, and the intermediate mechanism between them (see Figure 4.2).

When a fact SPN is used for a step of inference, the base predicate assembly (as the source predicate assembly) is used to represent initial bindings and the fact assembly (as the target predicate assembly) to represent result bindings. The sub-mechanisms that will be embedded into the intermediate mechanism are used for binding collection, consistency checking, and binding propagation. Whenever a query predicate is posed for backward chaining, the initial binding instantiation procedure will set up bindings between filler nodes of the query predicate and the argument nodes of the base predicate assembly. These bindings will then propagate through the intermediate mechanism where their consistency is checked. If they do not violate any consistency condition, the activation reaches to the fact assembly and represents the result of unification over its argument nodes.

During the fact encoding procedure, all facts which have the same predicate name and arity share the same base predicate assembly with which all their fact assemblies are associated. This allows parallelism on the SPNs encoding those facts when used for inference. Whenever a base predicate assembly becomes active by posing a query predicate, all fact assemblies associated with it become candidates to be fired. Fact assemblies which can unify with the presented query predicate are activated in parallel.

The following summarises the procedure for encoding each fact:

PROCEDURE: *encoding a symbolic fact*

Step 1: compute S_s and S_t for a given fact;

Step 2: build base predicate assemblies corresponding to the fact predicate if it does not exist in CLDB;

Step 3: build a fact assemblies for the given fact;

Step 4: build the intermediate mechanism in such a way that

(1) for each UAG, G_{A_i} in S_s, which has repeated arguments build a sub-mechanism for binding collection;

(2) for each UAG, G_{A_i} in S_s, build sub-mechanisms for consistency checking within each UAG and across UAGs;

(3) for each UAG, G_{A_i} in S_s, build a sub-mechanism for binding propagation

As can be seen, the same UAG concept is used to decide necessary sub-mechanisms for the intermediate mechanism. Since the base predicate assembly is considered as the source predicate assembly and the fact assembly as the target predicate assembly, sets of UAGs obtained from the example fact, *p(a,a,b)*, are:

$$S_s = \{G_a, G_b\} \text{ where } G_a = \{p{:}arg1, p{:}arg2\},\ G_b = \{p{:}arg3\},$$
$$S_t = \{G_a, G_b\} \text{ where } G_a = \{p_f{:}arg1, p_f{:}arg2\},\ G_b = \{p_f{:}arg3\}.$$

The following sub-sections describe how to determine the necessary sub-mechanisms to build the intermediate mechanism for each type of a fact based on these sets of UAGs.

5.4.2 Binding Collection Sub-Mechanism

Each UAG of a fact has either a single argument or repeated arguments. Like the BCM used within each UAG when encoding a rule, the binding collection sub-mechanism for each UAG of the fact (BCM_f) is only needed when the UAG has repeated arguments as follows:

$$BCM_f = \{G_{A_i} | \; G_{A_i} \in S_s, size(G_{A_i}) > 1\}.$$

Constructing the BCM_f for the UAG which has repeated arguments is done by inserting a binding node. The *b4* node shown in Figure 5.11a is used for this purpose.

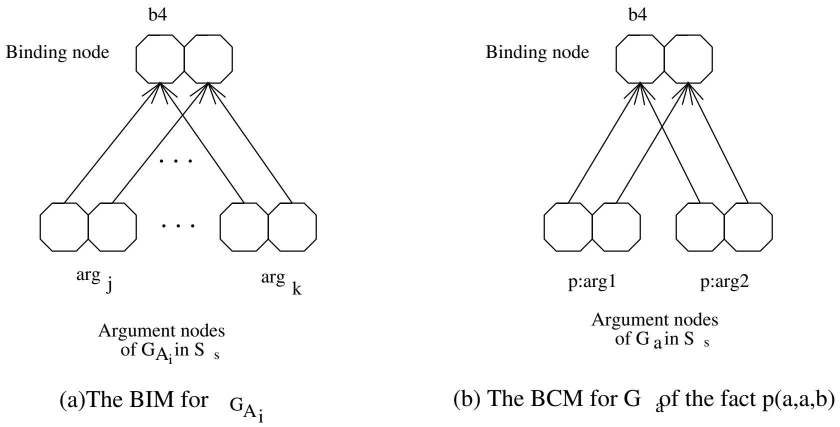


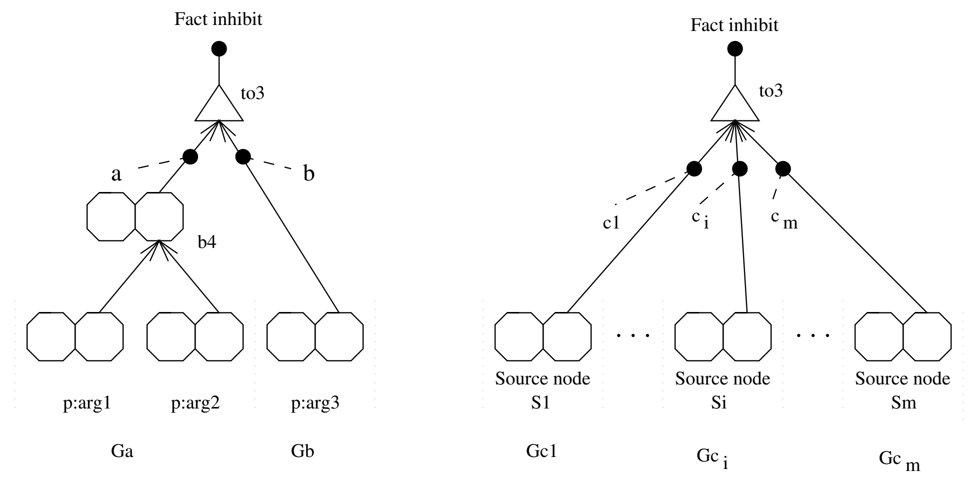
Figure 5.11: The structure of a binding collection sub-mechanism

If we consider the example fact, *p(a,a,b)*, G_a has repeated arguments and G_b a single argument. The binding node *b4* is only needed for G_a as shown in Figure 5.11b. Since all the argument nodes in G_a are connected to *b4*, any constant and variable bindings on *p:arg1* and *p:arg2* will be propagated to the *b4* node after initial binding instantiation.

5.4.3 Consistency Checking Sub-Mechanisms

Fillers that can be assigned to argument nodes of the encoded fact can be either constants or variables. Thus, two types of sub-mechanisms are necessary to force consistency within each UAG and across UAGs.

Firstly, the consistency checking sub-mechanism within each UAG ($CCM1_f$) makes sure that any constant filler(s) assigned to the argument(s) of the UAG is the same as the one specified as the constant name of the UAG. Since all arguments of an encoded fact are constants, this sub-mechanism is required for all UAGs of the fact. The $CCM1_f$ for the given example fact is shown in Figure 5.12a.



(a) The CCM1f for the example fact p(a,a,b)

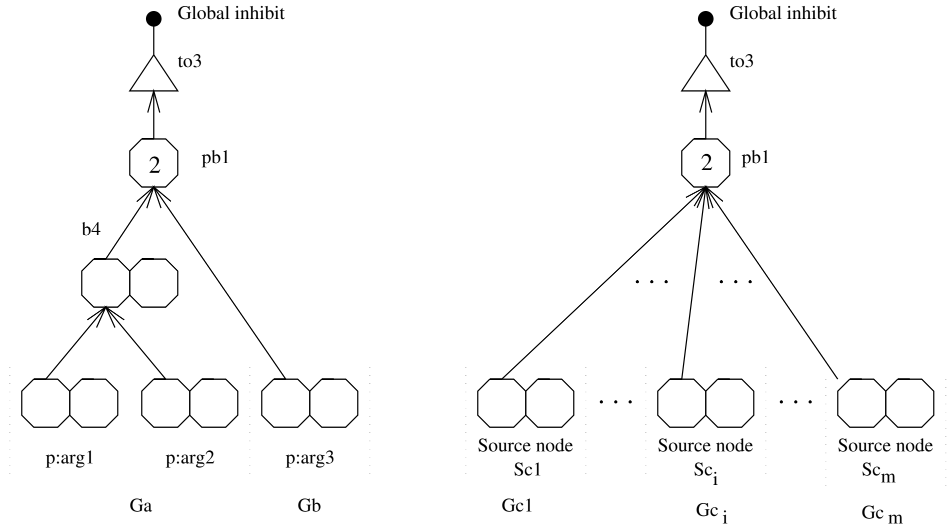
(b) The CCM1f for a fact with m UAGs

Figure 5.12: The structure of a consistency checking sub-mechanism within each UAG

In the figure, the node *b4* is the source node of G_a and the node *p:arg*$_3$ that of G_b. The right elements of the binding node and *p:arg*$_3$ are connected to *to3* node by links which are modified by the signal coming from the constant nodes *a* and *b*. The *to3* and these links serve as the $CCM1_f$. Whenever an argument node in G_a or G_b gets bound to any constant filler other than *a* or *b*, *to3* will be activated. The activation of *to3* blocks the binding propagation from the base predicate assembly *p* to the fact assembly p_f. This ensures the consistency condition of the example fact.

In general, if there are m UAGs in a given fact, the $CCM1_f$ will have the circuitry as shown in Figure 5.12b. The source node is the associated binding node (used as the BCM_f) if a UAG has repeated argument, otherwise the argument node is considered as the source node.

Secondly, the consistency checking sub-mechanism across UAGs ($CCM2_f$) in encoding



(a) The CCM2f for the example fact p(a,a,b) (b) The CCM2f for a fact with m UAGs

Figure 5.13: The structure of a consistency checking sub-mechanism across UAGs

a fact is used to ensure that the same variable filler does not get bound to more than one constant at the same time. This sub-mechanism is needed among all UAGs of the fact. The corresponding $CCM2_f$ is depicted in Figure 5.13a. The $CCM2_f$ is associated with the same source nodes, *b4* (for G_a) and *p:arg3* (for G_b). All the left elements of the source nodes are connected to a special π-btu element, called *pb1*, whose threshold is 2. This node becomes active if it receives more than one spike signal at the same phase of any oscillation cycle. Since each UAG has only one connection to *pb1*, this corresponds to the procedure to check if argument nodes in different UAGs get bound to the same variable filler. If this is the case, *pb1* will become active and then *to3* by *pb1*. The activation of *to3* eventually stops the activation flow from the base predicate to the fact assembly during inference. The $CCM2_f$ for a fact which has m UAGs is shown in Figure 5.13b.

As shown in Figure 5.12 and 5.13, both sub-mechanisms, the $CCM1_f$ and the $CCM2_f$, share the same source node and *to3* as a common part of their sub-mechanisms. Therefore, these two sub-mechanisms can be merged together as shown in Figure 5.14. On the left hand side are two separated sub-mechanisms and on the right hand side is the

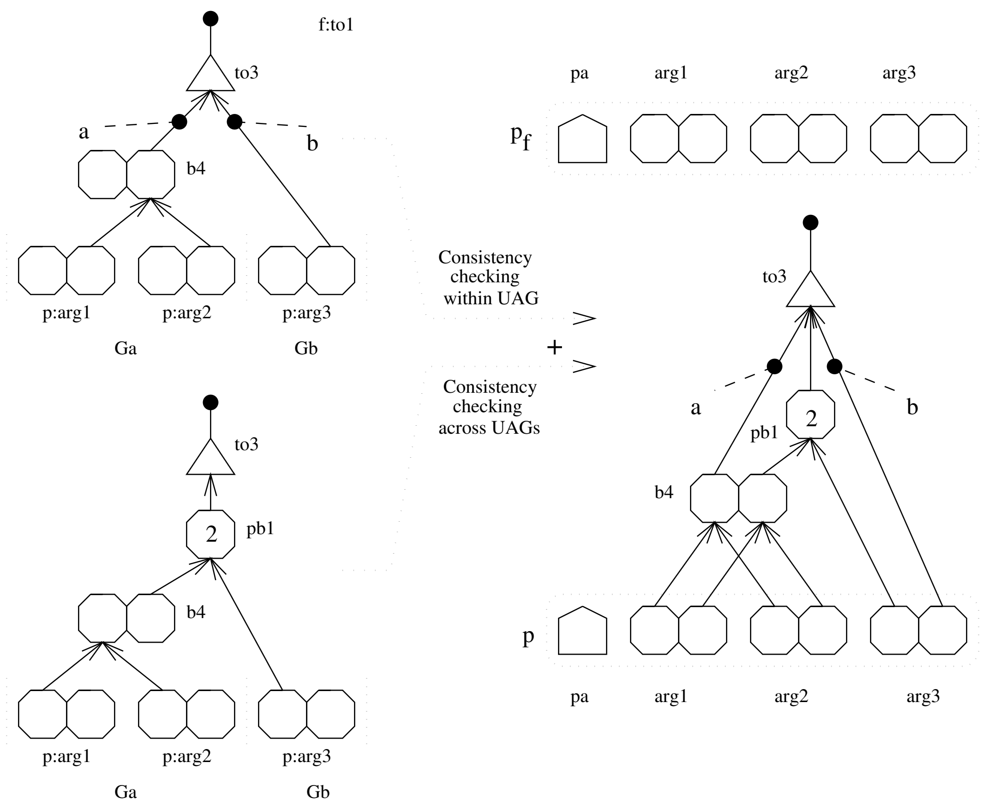


Figure 5.14: The merged consistency checking sub-mechanisms

merged network for consistency checking. The merged sub-mechanism are associated with the base predicate assembly p. When any of these two embedded sub-mechanisms detect the violation of the consistency during inference, the node *to3* will be activated and the output of *to3* is then used to inhibit binding propagation from the base predicate assembly to the fact assembly. If the query predicate satisfys the consistency conditions forced by these sub-mechanisms, the initial bindings between the query predicate and the base predicate assembly will propagate to the fact assembly, which will represent the result bindings. A more detailed sub-mechanism for binding propagation will be explained in the next subsection.

5.4.4 Binding Propagation Sub-Mechanisms

The binding propagation sub-mechanism (BPM_f) provides a path for binding propagation from the base predicate assembly to the fact assembly. According to the size of a related UAG in S_t, the structure of the BPM_f for each UAG requires a different set of links for the circuitry.

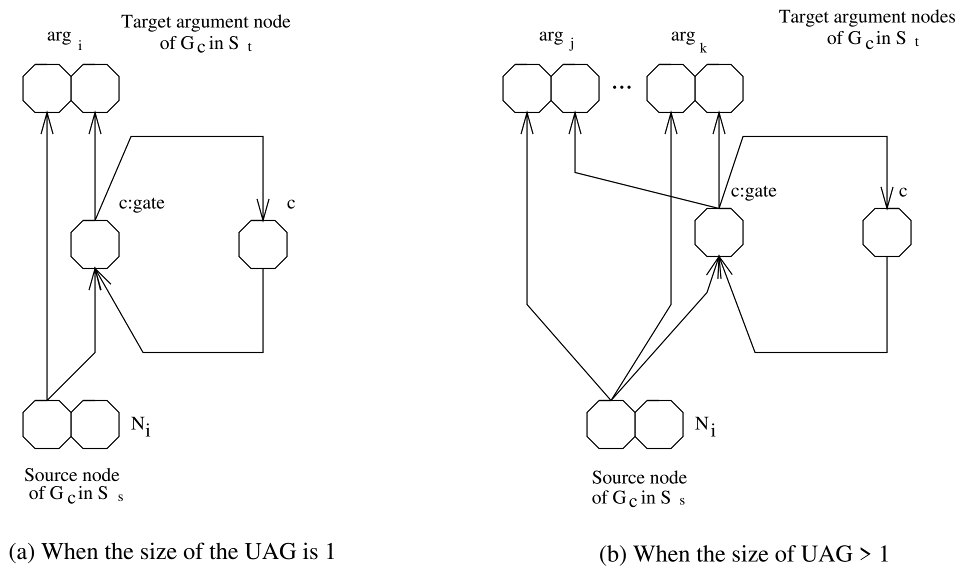


Figure 5.15: The structure of sub-mechanisms for the binding propagation

Figure 5.15 demonstrates two different variations of BPM_fs. The basic structure of the BPM_f for the UAG which has a single argument is shown in Figure 5.15a. Whenever the source node, N_i of G_c, gets bound to a variable filler, the variable binding will be propagated to the gate node of the constant (*c:gate*) and the left element of the target argument node of G_c in S_t. The output of *c:gate* then activates the constant node c and the right element of the target argument node. As a result, the presented variable filler node, the constant node, and the left and right elements of the target argument node are activated in the same phase. This represents the result of unification between the variable filler and the constant.

The BPM_f for a UAG which has repeated arguments can be treated in the same way except it requires additional sets of links from the source node to the target argument nodes according to the number of the target argument nodes. As shown in Figure 5.15b, the BPM_f is used to propagate the variable binding from the left element of the source node to all the left elements of the target argument nodes, as well as to the right elements of the target argument nodes through the *c:gate* node. The output of the gate also propagates to the constant node c to produce the desired unification result.

5.4.5 An Example of Encoding a Fact

Let us go back to the example fact, *p(a,a,b)*. Since the base predicate and the fact predicate of the given fact are $p(arg_1,arg_2,arg_3)$ and $p_f(arg_1,arg_2,arg_3)$, S_s and S_t of the example fact are:

$$S_s = \{G_a, G_b\} \text{ where } G_a = \{p\text{:}arg1, p\text{:}arg2\},\ G_b = \{p\text{:}arg3\},$$
$$S_t = \{G_a, G_b\} \text{ where } G_a = \{p_f\text{:}arg1, p_f\text{:}arg2\},\ G_b = \{p_f\text{:}arg3\}.$$

Based on this information, encoding the example fact firstly involves creating the base predicate assembly p and its corresponding fact assembly p_f. Then the intermediate mechanism which involves the binding collection sub-mechanisms, the consistency checking sub-mechanisms, and the binding propagation sub-mechanisms are built.

By applying the condition for deciding the binding collection sub-mechanism (BCM_f), we can obtain the UAG as follows:

$$BCM_f = \{G_a\}.$$

Since all UAGs require the $CCM1_f$, the $CCM2_f$, and the BPM_f, we can obtain

$$CCM1_f = \{G_a, G_b\},$$
$$CCM2_f = \{G_a, G_b\},$$
$$BPM_f = \{G_a, G_b\}.$$

The intermediate mechanism of the SPN for the fact is then constructed by putting all the required sub-mechanisms together. Figure 5.16 illustrates the structure of the fact

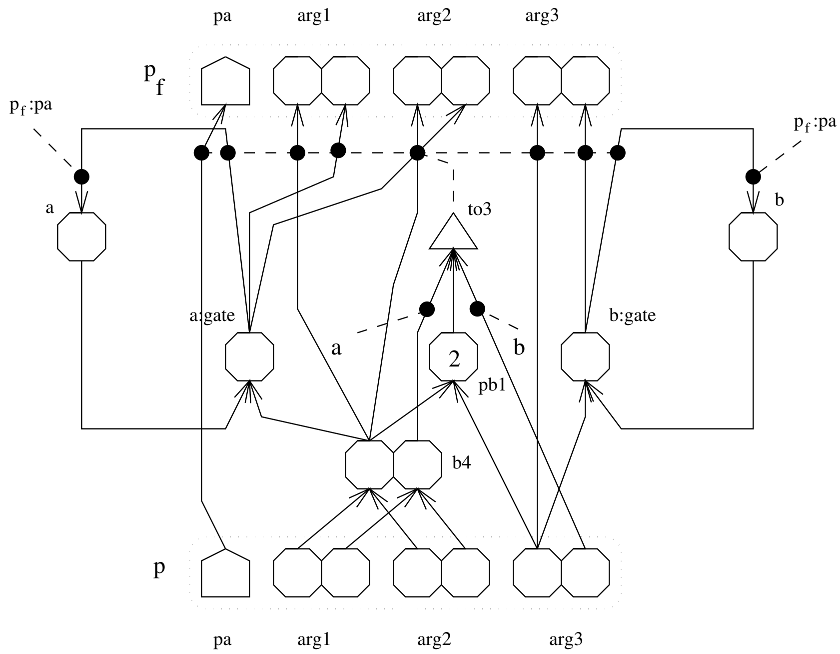


Figure 5.16: Encoding of the fact, $p(a, a, b)$

SPN. We can see the fact assembly p_f and its base predicate assembly p. Between them are the binding collection sub-mechanism, the binding propagation sub-mechanisms, and the consistency checking sub-mechanisms. The binding node *b4* is used as the BCM_f for the repeated arguments a's. The modified links between $b4_{right}$ and *to3*, and between $p{:}arg3_{right}$ and *to3*, are used for $CCM1_f$ to check consistency checking within each UAG. The nodes, $b4_{left}$, $p{:}arg3_{left}$, *pb1*, and *to3* and related links between them serve as the $CCM2_f$ which checks consistency across UAGs. When a posed query violates any of these consistency conditions, *to3* will be activated and this will prevent the fact assembly from becoming active. Otherwise, the initial bindings will propagate from the base predicate assembly to the fact assembly on which the result bindings will be represented. The nodes *a:gate* and *b:gate* are used to activate the right arguments nodes of the fact assembly and their corresponding constant node at the same time when variable fillers are assigned to the argument nodes of the base predicate assembly. Note that links to the two constant nodes are modified by the inhibitory signal coming from p_f*:pa*. This makes both constant nodes to be activated only once in relation to the fact assembly p_f during a chain of inference.

5.4.6 Clarifying Bindings Among Multiple Encoded Facts

Since CASI's fact encoding mechanism provides uniformity in representing rules and facts, argument nodes of each fact assembly can easily represent bindings generated during query processing. CASI's fact encoding mechanism therefore provides information that can be used to disambiguate bindings even when multiple facts are encoded under the same predicate name.

Suppose we have encoded the following two facts which share the same predicate name into CASI's CLDB:

$$p_{f1}(a,b,c),\ p_{f2}(a,a,b).$$

Posing the query $p(U,V,W)$ to CASI first performs initial phase allocation to the fillers, $U[1]$, $V[2]$, $W[3]$, and activation of corresponding argument nodes of the p predicate assembly. This results in the set of initial bindings $\{U/p{:}arg1,\ V/p{:}arg2,\ W/p{:}arg3\}$. Because these bindings do not violate consistency conditions, they propagate to the fact assemblies, p_{f1} and p_{f2}, through the binding propagation sub-mechanisms. Consequently, the constant nodes and their gate nodes for each fact and argument nodes of each fact assembly are activated in the following phases:

$$a=[1,2],\ b=[2,3],\ c=[3],$$

$$p_{f1}{:}\{pa[*],\ arg1([1],[1,2]),\ arg2([2],[2,3]),\ arg3([3],[3])\},$$
$$p_{f1}{:}a\_gate[1,2],\ p_{f1}{:}b\_gate[2,3],\ p_{f1}{:}c\_gate[3],$$
$$p_{f2}{:}\{pa[*],\ arg1([1,2],[1,2]),\ arg2([1,2],[1,2]),\ arg3([3],[2,3])\},$$
$$p_{f1}{:}a\_gate[1,2],\ p_{f1}{:}b\_gate[2,3],$$

Since the oscillation states of the constant nodes represent both unification results with p_{f1} and p_{f2} at the same time, we need the activation states of the constant gate nodes to distinguish the unification result obtained from p_{f1} from the obtained from p_{f2}. Therefore by observing the temporal patterns of the argument nodes of each fact assembly and their corresponding constant gate nodes, we can obtain the answers to the given query. For example, from the first argument node of p_{f1} oscillating in p_{f1}*:arg1([1],[1,2])* and its corresponding constant gate node oscillating in p_{f1}*:a_gate[1,2]*, the binding $\{a/U\}$ is obtained and also from the second and third argument nodes and their corresponding constant gate nodes, the bindings $\{b/V\}$ and $\{c/W\}$. In a similar way, from the oscillation states of the arguments of the second fact assembly and their corresponding constant gate nodes, the following result can be obtained:

$$\{a/U,\ a/V,\ b/W\}.$$

5.5 Performing Inference

5.5.1 Types of Inference

Two types of inferences are possible in the proposed connectionist inference architecture: forward chaining and backward chaining. Each type of inference is started by presenting or posing the predicate to the system. This input predicate has the form, *pred_name*$(t_1,\ t_2,\ \ldots,\ t_n)$, where t_i's are either constants or existentially quantified variables. We can think of each input predicate as being part of a description of a real situation. For instance, the sentence, "Mary bought car7", may be represented by the predicate, *buy(mary, car7)*. Also "John owns something" by the predicate, *own(john,U)*. In forward chaining, presenting these predicates corresponds to telling CASI the initial description and deriving its consequences. In backward chaining, on the other hand, these predicates are considered as queries for which we ask CASI to find proper answers. The above two example predicates are therefore interpreted as the sentences, "Did Mary buy car7?" and "What does John own?".

Suppose CASI encodes the following example rules and a fact into CLDB:

$$buy(X,Y) \rightarrow own(X,Y),$$
$$own(X,Y) \rightarrow can\_sell(X,Y),$$
$$own(mary,car7).$$

Presenting the predicate, *buy(mary, book1)* to CASI for forward chaining will automatically produce a series of the dynamic predicates *own(mary, book1)* and *can_sell(mary, book1)* over the SPNs which encode these example rules. This is the procedure for deriving the fact "Mary can sell book1" by being told "Mary bought book1".

On the other hand, posing the predicate, *can_sell(mary, car7)*, to CASI for backward chaining will produce a *yes* answer because of the fact that CASI already knew "Mary owns car7". In this case, CASI considers the input predicate as the question "Can mary sell car7?". If the query predicate which contains variable such as *can_sell(mary,U)* is posed to CASI, the system should produce not only the affirmative answer but also the unification result between the constant of a fact matched and the variable filler U.

5.5.2 Inference Examples

Forward chaining is started by presenting a predicate to the network. Presenting a predicate involves specifying the argument bindings between the presented predicate and its corresponding base predicate assembly in the network. If we consider the network shown in Figure 5.9c, which encodes the following rule:

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

presenting the predicate, *p(a,U,U)*, to the network will set up initial variable 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[1], U[6],
p:{*pa[*], arg1([0],[1]), arg2([6],[0]), arg3([6],[0])*},

where the numbers specify the particular phases. Once these initial variable bindings are set up, these bindings are then propagated to the target predicate assembly through the intermediate mechanism between them. As inference continues, the result of the binding collection from *p:arg1* and *p:arg2* will be represented on the *b1* node by activating it as *b1([6],[1])*, and this state is used again for binding interaction with *p:arg3* and activates the *b2* node in *b2([6],[1])*. Since the presented predicate does not violate any consistency condition of a rule, these initial bindings propagate to the *q* predicate assembly and activate its argument nodes as follows:

$$q{:}\{pa[*],\ arg1([6],[1]),\ arg2[6],[1])\}.$$

The right elements of the argument nodes represent the constant bindings and the left elements the variable bindings. Consequently, the result of inference, *q(a,a)*, and the result of unification during the inference, {a/U}, are represented. The detailed procedure for this forward chaining is demonstrated in Appendix A.2.

In a similar way, backward chaining is also initiated by supplying a predicate to the network but in this case we interpret the predicate as a query. Posing a query predicate sets up the initial bindings between the fillers of the query predicate and the corresponding base predicate assembly (the consequent predicate assembly). The result of inference is then obtained by propagating these initial bindings to the antecedent predicate assembly or 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 produced during inference.

The final result of backward chaining is then obtained by observing the states of fact assemblies activated during the propagation of activation on the network. If there is any fact assembly activated, this is considered as the affirmative answer "*yes*" (with appropriate bindings) to the given query predicate, otherwise the answer is considered as "*no*".

An example of the backward chaining inference can be seen by posing the query predicate, $q(a,U,V)$, to the network shown in Figure 5.16 which encodes the example fact *p(a,a,b)*. The initial bindings, {*a/p:arg1, U/p:arg2, V/p:arg3*} will be set up by activating the filler nodes and their corresponding argument nodes in the following phases:

$$a[1],\ U[4],\ V[7],$$
$$p{:}\{pa[*],\ arg1([0],[1]),\ arg2([4],[0]),\ arg3([7],[0])\}$$

These initial bindings then propagate through the intermediate mechanism consisting of consistency checking and binding propagation sub-mechanisms. Since the posed query does not violate any consistency conditions, these bindings propagate to the fact asembly. Consequently, the argument nodes of the fact assembly and their corresponding constant gate nodes will be activated as follows:

$$p_f\text{:}\{pa[*],\ arg1([0],[1,4]),\ arg2([4],[1,4]),\ arg2([7],[7])\},$$
$$p\_f1\text{:}a\_gate[1,4],\ p\_f1\text{:}b\_gate[7].$$

The constant bindings are represented on the fact assembly, $\{a/p_f\text{:}arg_1,\ a/p_f\text{:}arg_2,\ b/p_f\text{:}arg_3\}$, by in-phase oscillation between the constant gate nodes (*p_f1:a_gate* and *p_f1:b_gate*) and the argument nodes of the fact assembly. This represents that the encoded fact, *p(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 $\{U/a, V/b\}$. Appendix A.3 demonstrates the detailed steps of this backward chaining.

5.6 Summary

Encoding a rule and a fact into their corresponding SPNs involves creating the source and target predicate assemblies and building an intermediate mechanism between them. The previous chapter has shown how to build the source and target predicate assemblies using our basic neuron elements and this chapter has described how to construct the intermediate mechanism.

The concept of unifying argument groups (UAGs) is used to categorise arguments appearing the source and target predicates. The structure of an SPN for a rule and a fact is decided on the basis of the size and the type of each UAG, and the forms of matching between the source and target UAGs. As can be seen section 5.5.2, the intermediate mechanisms designed for the example rule and fact provides the required sub-mechanisms to complete the symbolic inference over the SPNs encoding them.

One rule encoding procedure which is not described in this chapter is how to encode unbalanced rules – rules which have isolated UAGs in the target predicate. Unlike balanced rules, an unbalanced rule requires a special treatment called binding generation. Thus, an additional sub-mechanism has to be inserted into the intermediate mechanism. The next chapter deals in more detail with the procedure for encoding unbalanced rules.

Also the rules and facts used in this chapter have only variable or constant arguments. They are not allowed to have any other types of arguments such as structured terms. The next chapter also illustrates the possible encoding procedure to handle rules and facts which have structured terms as arguments.

Chapter 6

Extensions

6.1 Overview

This chapter describes the possible extension of CASI' knowledge encoding mechanism to accommodate the following:

- unbalanced rules
- rules and facts involving structured terms

An unbalanced rule is a rule which has isolated UAG(s) in the target predicate. Encoding this type of rule requires a special sub-mechanism, called a binding generation sub-mechanism. The first part of the this chapter describes a necessary condition to decide which type of a rule needs this sub-mechanism. The detailed structure of the binding generation sub-mechanism will then be demonstrated using an example.

The rest of the chapter deals with possible extensions of the knowledge encoding mechanism to encode rules and facts involving structured terms. This involves the representation of a structured term, an extension of the initial binding instantiation procedure, and introduction of special sub-mechanisms required in encoding both balanced and unbalanced rules and facts involving structured terms.

6.2 Encoding Unbalanced Rules

Encoding unbalanced rules requires a special sub-mechanism called a binding generation sub-mechanism (BGM) which has to interact with an external connectionist mechanism. The condition to decide which rule requires a BGM is given as follows:

$$BGM = \{G_{A_i} \,|\, G_{A_i} \notin S_s, G_{A_i} \in S_t\}$$

where G_{A_i} is defined as an isolated UAG in the target predicate.

Let us refer to the following rule:

$$trangle(X) \rightarrow no\_of\_sides(X, three).$$

The S_s and S_t are

$$S_s = \{G_X\} \text{ where } G_X = \{\textit{triangle:arg}_1\}$$
$$S_t = \{G_X,\ G_{three}\} \text{ where } G_X = \{\textit{no\_of\_sides:arg}_1\},\ G_{three} = \{\textit{no\_of\_sides:arg}_2\}$$

Since G_three appears only in S_t, this rule is unbalanced. If the predicate *triangle(a)* is presented to the SPN encoding the example rule for forward chaining, the network must produce *no_of_sides(a,three)* as a result of inference. If we follow the detailed inference steps over the SPN, presenting the predicate first activates the constant node, *a[0]*, and the corresponding argument node, *triangle:arg*$_1$*([0],[0])*, in the following phase to set up the initial binding:

$$\textit{a[1]},$$
$$\textit{triangle:}\{\textit{pa[*],arg}_1\textit{([0],[1])}\}.$$

When this binding propagates to the predicate assembly *no_of_sides*, the first argument node will be activated in *no_of_sides:arg*$_1$*([0],[1])* by the binding propagation sub-mechanism. This only represents the binding, $\{\textit{a/no\_of\_sides:arg}_1\}$, and leaves the other binding needed $\{\textit{three/no\_of\_sides:arg}_2\}$ unspecified in the form of phases. The expected set of bindings in this inference are $\{\textit{a/no\_of\_sides:arg}_1,\ \textit{three/no\_of\_sides:arg}_2\}$.

To represent the second binding using phases, the model has to allocate an unused phase to both the constant node *three[0]* and the right element of the argument node *no_of_sides:*$arg_{2_{right}}$*[0]*. This requires

- an external phase manager (EPM) which monitors available phases;
- a binding generation sub-mechanism which request the external mechanism an available phase when the source predicate assembly becomes active.

The structure of the EPM will not be dealt with in this thesis. One possible structure of the EPM could be similar to the Free Phase Map described in Aaronson (1991).

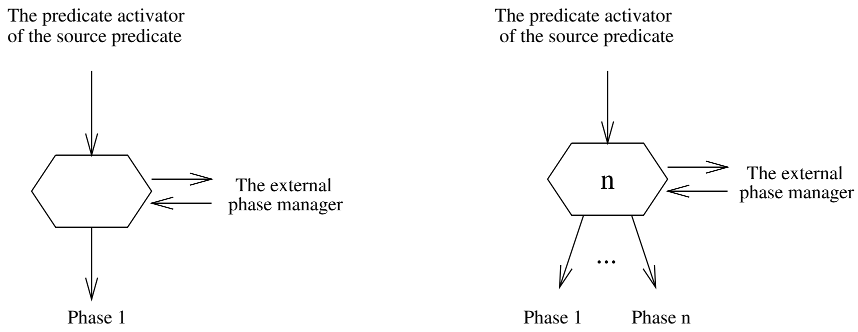
The structure of the binding generation sub-mechanism, introduced to encode unbalanced rules, depends on the number of UAGs in the target predicate which requires binding generation during inference. The most important components of this sub-mechanism will be the component, called the *phase requester*, which requests a number of available phases to the EPM. The following subsections will describe the structure of the phase requester and how unbalanced rules are encoded using the binding generation sub-mechanism to be introduced.

6.2.1 The Phase Requester

The required functional behaviour of the phase requester can be summarised as follows:

- the phase requester needs to be activated by the source predicate when it becomes active;
- the phase requester needs to communicate with EPM to request available phases and to receive them;
- if there are more than one isolated UAG in the target predicate, the phase requester has to ask the EPM for the proper number of phases.

The graphical representation of the behaviour of the phase requester is shown in Figure 6.1. The phase requester is drawn with a hexagon. If the target predicate has



(a) When requesting one unused phase

(b) When requesting n unused phases

Figure 6.1: The graphical representation of the behaviour of the phase requester

only one isolated UAG, the phase request asks the EPM for one phase. On receiving the signal from the predicate activator of the source predicate assembly, the phase requester sends the request to the EPM and asks for a free phase. After receiving the free phase from the EPM, the phase requester propagates this phase to the argument node(s) of the isolated UAG.

In general, if the target predicate has n isolated UAGs, the phase requester has to ask the EPM for n free phases. The number inside the hexagon indicates the number of phases that the phase requester has to ask.

6.2.2 Encoding Rules with a Single Isolated UAG

Let us consider the following two rules:

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

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

The S_s and S_t of the first rule are

$$S_s = \{G_X\},\ where\ G_X = \{p{:}arg_1\},$$
$$S_t = \{G_X, G_Y\},\ where\ G_X = \{q{:}arg_1\},\ G_Y = \{q{:}arg_2\},$$

and those of the second rule are

$$S_s = \{G_X\},\ where\ G_X = \{p{:}arg_1\},$$
$$S_t = \{G_X, G_a\},\ where\ G_X = \{q{:}arg_1\},\ G_a = \{q{:}arg_2\}.$$

The isolated UAG of the first rule, G_Y in S_t, is the variable UAG and therefore requires variable binding generation during inference. On the other hand, the isolated UAG of the second rule, G_a in S_t, is the constant UAG and requires constant binding generation.

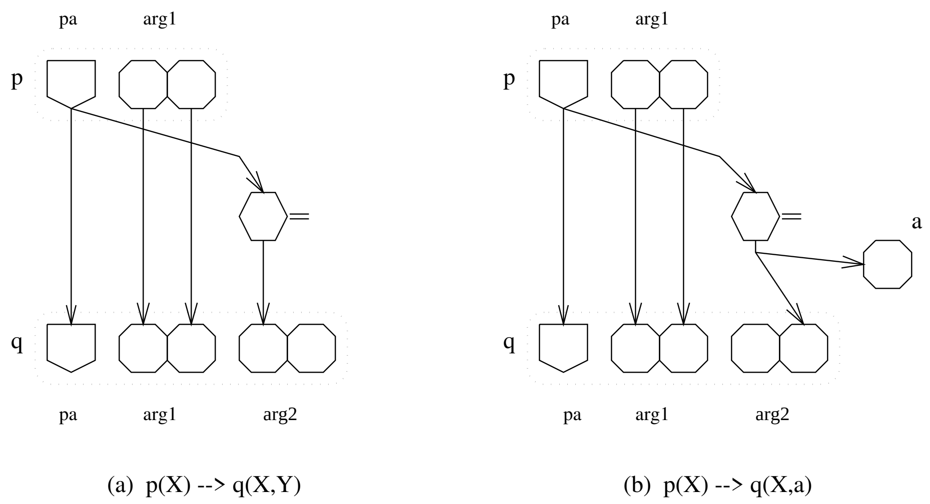


Figure 6.2: Encoding the rules which have one isolated UAG in the target predicate

The SPNs which correspond to the two example rules are shown in Figure 6.2. Figure 6.2a demonstrates how the phase requester is used to generate a variable binding. The output of the phase requester is connected to the $q{:}arg2_{left}[0]$. Thus the propagation of the new binding from the phase requester to $q{:}arg2_{left}[0]$ will set up a variable binding on $q{:}arg_2$. Now we demonstrate how the phase requester is used to generate a constant binding. The output of the phase request is sent to the right element of the target argument node and the corresponding constant node at the same time to achieve required constant binding. Figure 6.2b illustrates this situation.

6.2.3 Encoding Rules with More Than One Isolated UAGs

The following is a rule which has more than one isolated UAGs:

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

The S_s and S_t of the given rule are

$$S_s = \{G_X\},\ where\ G_X = \{p{:}arg_1\},$$
$$S_t = \{G_X, G_Y, G_a\},\ where\ G_X = \{q{:}arg_1\},\ G_Y = \{q{:}arg_2\}\ G_a = \{q{:}arg_3\}.$$

Since G_Y and G_a in S_t are isolated, they need BGMs:

$$BGM = \{G_Y, G_a\}.$$

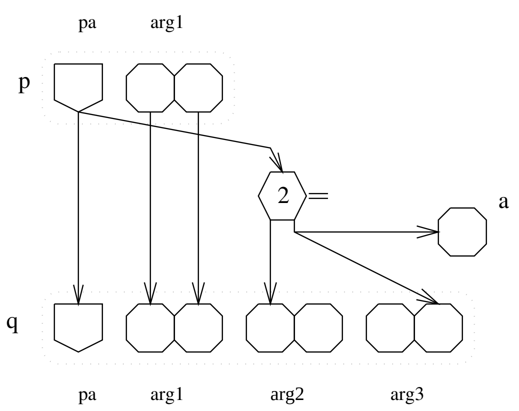


Figure 6.3: The intermediate mechanisms for a repeated argument

Figure 6.3 depicts the corresponding SPN of the given rule. The number shown on the phase requester indicates the number of free phases that the phase requester has to ask. Since there are two isolated UAGs in the target predicate, the phase requester has to ask two phases at the same time. The generated phases then propagate to $q{:}arg_{1_{left}}[0]$ and $q{:}arg_{2_{right}}[0]$ and $a[0]$ to set up the variable binding and the constant binding on each target argument node.

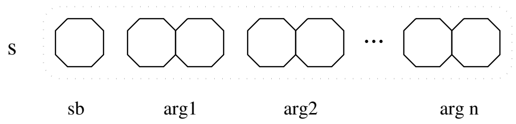
6.3 Dealing with Structured Terms

Allowing a structured term as an argument of a rule and a fact extends the expressive power of CASI. Although Ajjanagadde (1990) proposed a provisional connectionist mechanism which encodes rules involving structured terms based on temporal synchrony, his solution does not specify how it can deal with consistency checking and unification during inference. Neither does his system provide any mechanism to encode facts with structured terms. The following subsection shows how rules and facts involving structured terms are represented into the corresponding SPNs and how these networks are used for inference.

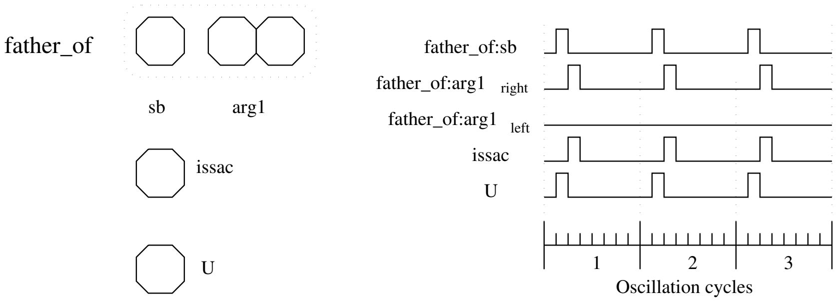
6.3.1 Representing Structured Terms

Since we consider Ajjanagadde's representation of structured term [Ajjanagadde (1990)] to be elegant, we adopt his representation in CASI. A structured term s of n arguments, $s(arg_1, arg_2, \ldots, arg_n)$, is represented using an assembly of nodes consisting of a *structured term binder* (sb) and n arguments nodes as shown in Figure 6.4a. The n argument nodes are the same as ordinary argument nodes used in an n-ary predicate assembly. The structured term binder is a single π-btu node and is used to refer to the whole structured term. If any variable node becomes active in-phase with this binder, this situation indicates that the variable is bound to the structured term.

A dynamic structured term, $s(F_1, F_2, \ldots, F_n)$, is represented by activating a node representing each filler F_i and the corresponding argument node of the structured assembly in the same phase. If F_i is a variable filler, a variable binding needs to be set between them and if F_i is a constant filler, a constant binding. An example of binding involving a structured term can be seen in Figure 6.4b. The in-phase oscillation between *issac[2]* and *father_of:*$arg_{1_{right}}$*[2]* represents the instantiated structured term *father_of(issac)* and the in-phase oscillation between *U[1]* and *father_of:sb[1]* the binding $\{$*father_of(issac)/U*$\}$.



(a) A structured assembly having n arguments



(b) The phasic representation of the binding {father_of(issac)/U}

Figure 6.4: The intermediate mechanisms for a repeated argument

6.3.2 Inference Involving Structured Terms

When a structured term is used as a filler for any argument of the source predicate, a special mechanism is not necessary to accommodate the structured term. However, the binding instantiation procedure has to be changed to activate a structured term in appropriate phases when a predicate involving the structure term is presented. Suppose we have rule whose corresponding SPN is shown in Figure 6.5.

$$buy(X, Y) \rightarrow own(X, Y).$$

If the predicate, *buy(father_of(andrew),camera2)*, is presented to the SPN which encodes the example rule, the predicate, *own(father_of(andrew),camera2)*, must be obtained as a result of inference. In order to get the correct result, the initial binding instantiation procedure has to be extended as follows:

- for each filler appearing in the predicate presented,

  Step 1: if the filler is a structured term,

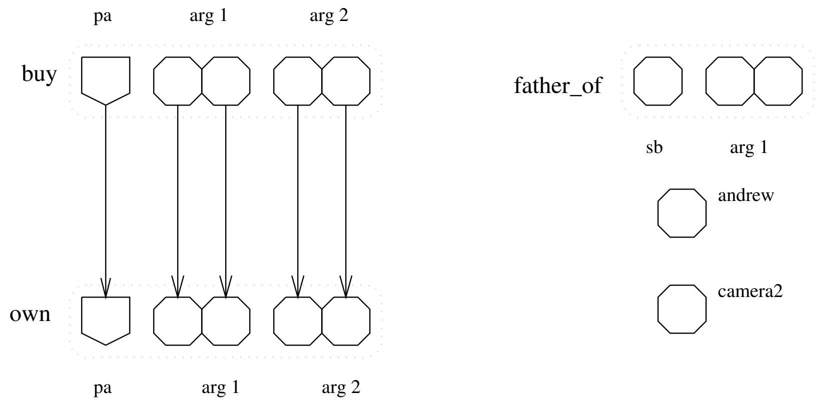


Figure 6.5: Inference involving the function term, father_of(andrew)

- firstly create the corresponding structured assembly (if it does not exists in the CLDB) and activate its binder in one of the available phases. To activate each argument node of the structured term, apply this procedure recursively;
- secondly, activate the right element of the corresponding argument node of the base predicate assembly in the same phase allocated to the binder.

Step 2: if the filler is not a function term, create a filler node and activate the filler node and the corresponding argument node of the base predicate assembly in such a way that:

- if the filler is a constant, activate both the filler node and the right elements of the argument nodes;
- if the filler is a variable, activate both the filler node and the left elements of the argument nodes.

Applying this extended initial binding instantiation procedure to the SPN shown in Figure 6.5 produces in the initial bindings as follows:

$$\textit{father\_of}{:}\{sb[1], arg_1([0],[2])\},\ andrew[2],\ camera2[3],$$
$$buy{:}\{pa[*], arg_1([0],[1]), arg_2([0],[3]).$$

When these initial bindings are propagated to the target predicate assembly, they activate the *own* predicate assembly in

$$own{:}\{pa[*], arg_1([0],[1]), arg_2([0],[3]).\}$$

This activation represents the desired result, *own(father_of(andrew),camera2)*.

Therefore, if a structured term is used as a constant filler, pre-established SPNs can accommodate the structured term without adding any special sub-mechanism except using the new binding procedure to set up initial bindings between the source predicate assembly and the presented predicate involving structured terms.

6.3.3 Encoding Balanced Rules with Structured Terms

If the antecedent of a balanced rule has a structured term whose argument name is also used as an argument of the consequent, a special binding propagation sub-mechanism is necessary to encode the rule. When encoding this type of balanced rule, the knowledge compiler should detect this situation and insert an additional sub-mechanism to support proper binding propagation between them. The following rule exemplifies this situation:

$$greater(age\_of(X), fifty) \rightarrow on\_pension(X).$$

The structured term *age_of(X)* is used as the first argument of the *greater* predicate. Since the argument of *age_of(X)* appears in the target predicate, the initial binding involving this argument needs to be propagated to the related argument of the target predicate through the binding propagation sub-mechanism during inference. However, this situation is different from the one demonstrated when encoding an ordinary rule where the BPM directly connects between argument nodes of a UAG in the source predicate and those of the related UAG in the target predicate. Encoding the example rule requires a modification of the BPM to represent the relationships between the argument of the structured term used as the first argument of the *greater* predicate and the argument of the *on_pension* predicate to provide a binding path between them.

The information needed to decide this special sub-mechanism can be obtained from S_s and S_t of the given rule. If we consider the above rule, S_s and S_t are

$$S_s = \{G_{X_{age\_of}}, G_{fifty}\}$$
$$\text{where } G_{X_{age\_of}} = \{\mathit{greater{:}arg}_1\},\ G_{fifty} = \{\mathit{greater{:}arg}_2\},$$
$$S_t = \{G_X\} \text{ where } G_X = \{\mathit{on\_pension{:}arg}_1\}\ .$$

The notation, $G_{X_{age\_of}}$ in S_s, represents that the UAG, G_X, is obtained from the structured term *age_of* which appears as an argument of the source predicate. The more detailed information, $G_{X_{age\_of}} = \{\mathit{greater{:}arg}_1\}$, tells that the function *age_of* is used as the first argument of *greater* predicate. Based on the two sets of UAGs obtained, all sub-mechanisms required to build the intermediate mechanism are decided as follows:

$$BCM = \{\},\ CCM_c = \{G_{fifty}\},\ CCM_v = \{\},$$
$$BIM\&CCM_{cv} = \{(G_{X_{age\_of}}, G_{fifty})\},\ BIM\&CCM_{vv} = \{\},\ BIM_{cc} = \{\},$$
$$BPM = \{G_{X_{age\_of}}\}.$$

The intermediate mechanism of the SPN encoding the example rule thus requires:

- the CCM_c for the second argument of the *greater* predicate because it is the constant argument;
- the $BIM\&CCM_{cv}$ between the argument of the structured term *age_of* and the second argument of the *greater* predicate;
- the BPM between the argument of the structured term *age_of* and that of the *on_pension* predicate;
- additionally, a sub-mechanism which specifies the relation between the first argument of the *greater* predicate and the structured term, *age_of*.

Figure 6.6 depicts the SPN encoding the example rule.

On the top right and bottom of the SPN show two predicate assemblies corresponding to the *greater* and *on_pension* predicates. On the top left is the structured assembly

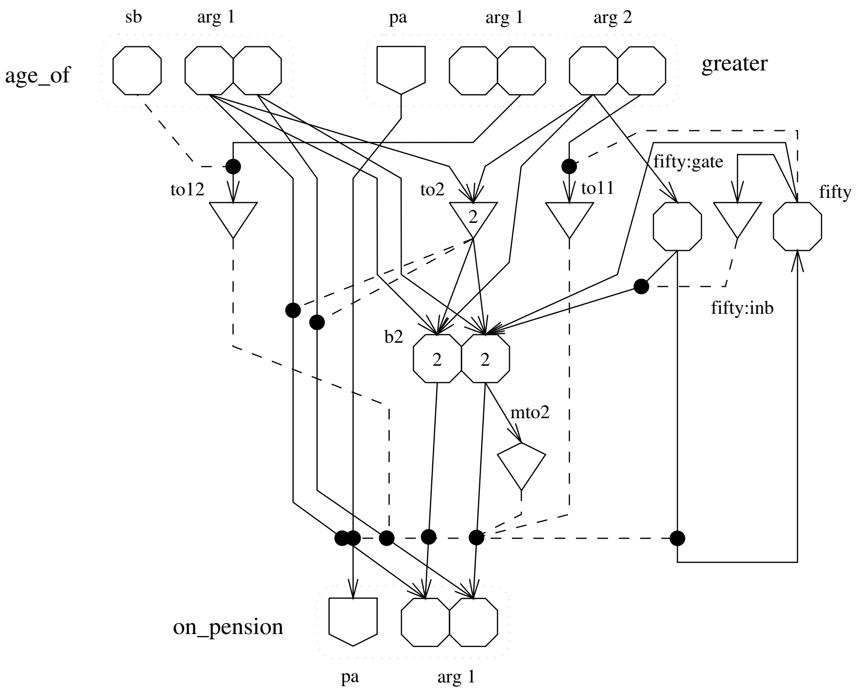


Figure 6.6: Encoding of the rule, *greater(age_of(X),fifty)* $\rightarrow$ *on_pension(X)*

corresponding to *age_of(X)*. The intermediate mechanism is illustrated in the middle of the SPN. The node *to11*, the constant node *fifty[0]* and its associated nodes are used as the CCM_c for the second argument of the *greater* predicate. The nodes *to2*, *b2*, *mto2* and associated links serve as the $BIM\&CCM_{cv}$ between the first argument, the structured term, and the second argument of the *greater* predicate. The links between *age_of:arg*$_1$*([0],[0])* and *on_pension:arg*$_1$*([0],[0])* provide the BPM between them. Finally, an additional node, *to12*, and associated links ensure the relationship between the first argument of the *greater* predicate and the structured term. If the first argument is bound to any filler other than the structured term, *age_of*, *to12* becomes active and projects an inhibitory signal to stop the inference.

If the predicate, *greater(age_of(tom),fifty)*, is presented to this SPN to demonstrate a step of inference, the source predicate assembly, the function assembly, and involved constant filler nodes are activated in the following phases by the extended initial binding instantiation procedure:

$$\textit{age\_of:}\{\textit{sb[1]},\textit{arg}_1\textit{([0],[2])}\},\ \textit{tom[2]},\ \textit{fifty[3]},$$
$$\textit{greater:}\{\textit{pa[*]},\textit{arg}_1\textit{([0],[1])},\textit{arg}_2\textit{([0],[3])}\}.$$

As inference continues, the binding on *age_of:*arg_1*([0],[2])* propagates to the argument of the *on_pension* predicate. The nodes *to11* and *to12* are not activated during the inference because both the *greater:*$arg_{1_{right}}$*[0]* and *greater:*$arg_{2_{right}}$*[0]* elements get bound to proper fillers *age_of(tom)* and *fifty*. As a result, the *on_pension* predicate assembly will be activated in

$$\textit{on\_pension:}\{\textit{pa[*]},\textit{arg}_1\textit{([0],[2])}\}.$$

This indicates the result of inference *on_pension(tom)*.

If the predicate, *greater(age_of(U),fifty)*, is presented to the network, the initial bindings will be

$$\textit{age\_of:}\{\textit{sb[1]},\textit{arg}_1\textit{([2],[0])}\},\ \textit{U[2]},\ \textit{fifty[3]},$$
$$\textit{greater:}\{\textit{pa[*]},\textit{arg}_1\textit{([0],[1])},\textit{arg}_2\textit{([0],[3])}\}.$$

After binding propagation, the target predicate assembly will be activated

$$\textit{on\_pension:}\{\textit{pa[*]},\textit{arg}_1\textit{([2],[0])}\},$$

from which the result *on_pension(U)* is obtained. This result can be interpreted as "U is on a pension".

During these two inferences, the binding propagation from *age_of:*arg_1*([0],[0])* to *on_pension:*arg_1*([0],[0])* is achieved through the direct links between them. However, if the predicate, *greater(age_of(U), U)*, is presented to the network, the SPN shows a different behaviour. Since the argument of the structured term and the second argument of the *greater* predicate are bound to the same variable filler, the initial bindings, in this case, will be

$$\textit{age\_of:}\{\textit{sb[1]},\textit{arg}_1\textit{([2],[0])}\},\ \textit{U[2]},$$
$$\textit{greater:}\{\textit{pa[*]},\textit{arg}_1\textit{([0],[1])},\textit{arg}_2\textit{([2],[0])}\}.$$

In the next inference cycle, *to2* becomes active because of the two inputs coming from *age_of:*$arg1_{left}$*[2]* and *greater:*$arg2_{left}$*[2]*. The node *fifty:gate[0]* also becomes active by the input coming from *greater:*$arg1_{left}$*[2]*, and the same activation also propagates to the constant node *fifty[0]*. The activation of *to2* blocks the BPM and allows the initial bindings to be propagated only through the *b2* node. When the binding node *b2* becomes active in *b2([2],[2])*, these intermediate bindings propagate to the *on_pension* predicate and activate it in

$$on\_pension{:}\{pa[*], arg_1([2],[2])\}.$$

Since the constant node *fifty[0]* also becomes active in the second phase during inference by the signal coming from *fifty:gate[2]*, the result obtained is interpreted as *on_pension(fifty)* with the binding $\{fifty/U\}$. This result may be seem strange but this refers to the situation where the name of the person is the same as the word representing the number "50". As long as they are represented with the same entity name, CASI uses one entity node to represent them. When encoding rules and facts, CASI analyses them only syntactically and does not consider any semantics involved. Thus CASI treats the above rule syntactically the same as the rule, $p(f(X), fifty) \rightarrow q(X)$ where presenting the predicate, $p(f(U), U)$ produces the result, $q(fifty)$, with the binding $\{fifty/U\}$.

6.3.4 Encoding Unbalanced Rules with Structured Terms

If the example rule shown in the previous section is seen from the backward chaining point of view, the rule becomes unbalanced. Because the source predicate becomes the target predicate and vice versa when a rule is used to construct the corresponding SPN to support backward chaining, the sets of UAGs of the example rule are now as follows:

$$S_s = \{G_X\} \ where\ G_X = \{on\_pension{:}arg_1\},$$
$$S_t = \{G_{X_{age\_of}}, G_{fifty}\}$$
$$\text{where } G_{X_{age\_of}} = \{greater{:}arg_1\},\ G_{fifty} = \{greater{:}arg_2\}.$$

The given rule has an isolated UAG, G_{fifty}, in S_t and is therefore unbalanced. Since the given rule has neither any constant UAG nor any variable UAGs which have repeated arguments in S_s, it does not require any sub-mechanisms to build the intermediate mechanism except the BPM:

$$BCM = \{\},\ CCM_c = \{\},\ CCM_v = \{\},$$
$$BIM\&CCM_{cv} = \{\},\ BIM\&CCM_{vv} = \{\},\ BIM_{cc} = \{\},$$
$$BPM = \{G_X\}.$$

By applying the condition to decide the BGM shows that the rule requires one BGM because of the isolated UAG, G_{fifty} in S_t, as follows:

$$BGM = \{G_{fifty}\}.$$

Encoding this unbalanced rule needs the BGM and a modification of the BPM to deal with the structured term involved. Figure 6.7 illustrates the corresponding SPN encoding the example rule to support backward chaining.

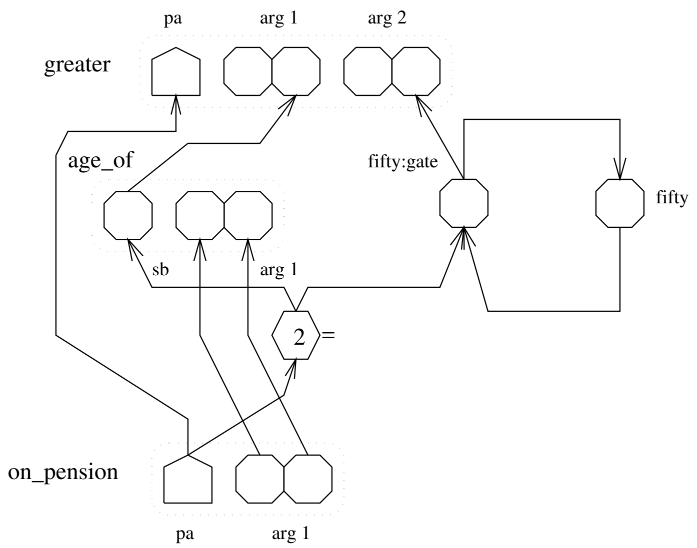


Figure 6.7: The SPN which encodes the rule, *greater(age_of(X),fifty)* $\rightarrow$ *on_pension(X)*, to support backward chaining

On the right hand side of the figure shows the BGM for the second argument of the *greater* predicate and on the left hand side the modified BPM between the argument of the *on_pension* predicate and the first argument of the *greater* predicate. The difference is that the outgoing links of the *on_pension:*arg_1*([0],[0])* are connected to the argument of the structured term *age_of* and the outgoing link of *age_of:sb[0]* is connected to *greater:*$arg_{1_{right}}$*[0]*. These links provide path for the binding propagation from the source predicate assembly to the target predicate assembly.

During backward chaining, two phases have to be generated: one for the binder of the structured term *age_of* and the other for the constant node *fifty[0]*. The phase requester will get these phases from the external phase manager when it is activated by the predicate activator of the *on_pension* predicate assembly. Therefore, posing the query predicate, *on_pension(U)*, will activate *on_pension* predicate assembly and the variable *U* in the flowing phases:

$$on\_pension{:}\{pa[*], arg_1([1],[0])\},\ U[1].$$

After generation of two phases, the assembly representing the structured term *age_of* and the node *fifty:gate[0]* will be activated as follows:

$$age\_of{:}\{sb[2], arg_1([1],[0])\},\ fifty{:}gate[3],$$

assuming that the second phase is generated and assigned to *age_of:sb[0]* and the third phase to *fifty:gate[0]*. When the propagation of bindings are completed, the predicate assembly *greater* and the constant node *fifty[0]* will be activated in

$$greater{:}\{pa[*], arg_1([0],[2]), arg_2([0],[3])\},\ fifty[3]\ .$$

This will be interpreted as the intermediate query *greater(age_of(U),fifty)* and will be used for further inference to find if there is any encoded fact which satisfys this intermediate query.

6.3.5 Encoding Facts with Structured Terms

To complete backward chaining with a rule which has a structured term, a fact involving structured terms should be able to be encoded into the CLDB. In the previous section, for example, the generated intermediate predicate *greater(age_of(U),fifty)* can be satisfied only if any fact such as *greater(age_of(tom),fifty)* is pre-encoded into the CLDB. Therefore, CASI has to provide a proper encoding mechanism for facts involving structured terms.

If we use the fact, *greater(age_of(tom),fifty)*, as an example, after constructing the source and target predicate assemblies the fact encoding procedure needs to decide the necessary sub-mechanisms to build the intermediate mechanism. Since the base predicate and the fact predicates of the example fact are represented as $greater(arg_1,arg_2)$ and $greater_f(arg_1,arg_2)$, their sets of UAGs are

$$\begin{aligned} S_s &= \{G_{tom_{age\_of}}, G_{fifty}\} \\ \text{where } G_{tom_{age\_of}} &= \{greater{:}arg_1\},\ G_{fifty} = \{greater{:}arg_2\}, \\ S_t &= \{G_{tom_{age\_of}}, G_{fifty}\} \\ \text{where } G_{tom_{age\_of}} &= \{greater_f{:}arg_1\},\ G_{fifty} = \{greater_f{:}arg_2\}. \end{aligned}$$

The example fact does not need the BCM_f because no UAG in S_s has repeated arguments therefore

$$BCM_f = \{\}.$$

By applying the conditions to decide other sub-mechanisms for consistency checking and binding propagation, we obtain

$$\begin{aligned} CCM1_f &= \{G_{tom_{age\_of}}, G_{fifty}\}, \\ CCM2_f &= \{G_{tom_{age\_of}}, G_{fifty}\}, \\ BPM_f &= \{G_{tom_{age\_of}}, G_{fifty}\}. \end{aligned}$$

Building the intermediate mechanism for the example fact is carried out in the same way that used in encoding an ordinary fact except a slight modification of the binding propagation sub-mechanism to handle the structured term used as an argument. Figure 6.8 illustrates how the example fact can be encoded.

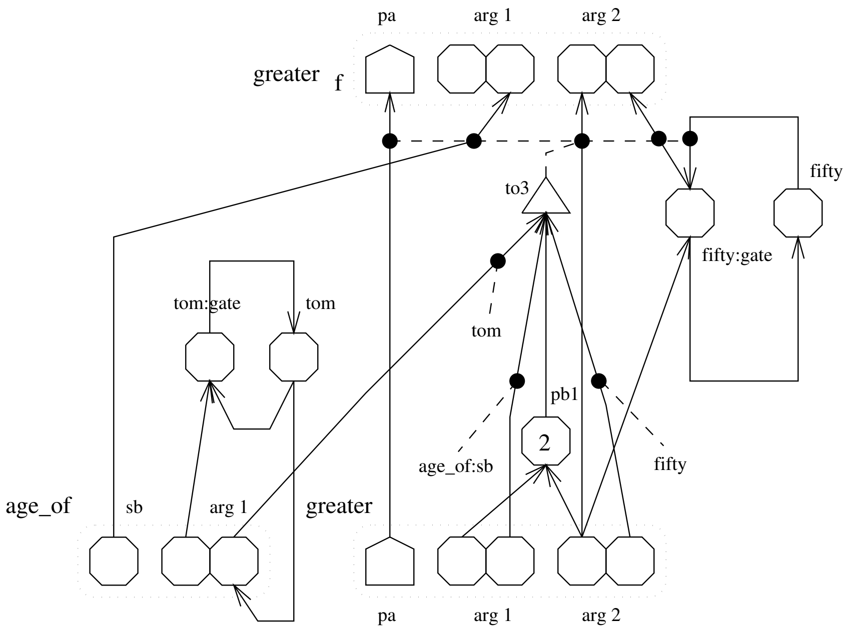


Figure 6.8: The SPN which encode the fact, *greater(age_of(tom,fifty))*

At the bottom of the network shows the node *pb1* used for consistency checking across UAGs (the $CCM2_f$) and modified links from *greater:arg*$_{1_{right}}$*[0]* and *greater:arg*$_{2_{right}}$*[0]* to *to3* are used to check consistency within each UAG (the $CCM1_f$). Note that the link between *greater:arg*$_{1_{right}}$*[0]* and *to3* is modified by the inhibitory link coming from *age_of:sb[0]*. This makes sure that the first argument of the *greater* predicate to get bound to the structured term *age_of*. Between *greater:arg*$_2$ and *greater*$_f$*:arg*$_2$ has a normal binding propagation sub-mechanism including the links to the constant node *fifty[0]*. However, the binding propagation between *greater:arg*$_1$ and *greater*$_f$*:arg*$_1$ has been modified because of the structured term *age_of(tom)*. The constant node *tom[0]* and its associated nodes are attached to the argument of the structured term to represent a constant binding on the argument node. Actual binding propagation to *greater*$_f$*:arg*$_1$ is carried out by the link between the node *age_of:sb[0]* and *greater*$_f$*:arg*$_{1_{right}}$*[0]*.

To demonstrate a step inference involving this SPN, let us start the inference with the following initial binding setup which were obtained from the previous backward chaining:

$$\textit{age\_of}:\{\textit{sb[2]},\textit{arg}_1\textit{([1],[0])}\},\ \textit{U[1]},\ \text{fifty[3]},$$
$$\textit{greater}:\{\textit{pa[*]},\textit{arg}_1\textit{([0],[2])},\textit{arg}_2\textit{([0],[3])}\}.$$

Since the right elements of the argument nodes of the *greater* predicate assembly are active in the same phase with *age_of:sb[2]* and *fifty[3]* this is considered as the state that the predicate *greater(age_of(U),fifty)* is presented to CASI.

When this initial activation propagates to the intermediate mechanism, they do not activate *to3*. Therefore there is no inhibitory signal from *to3* to stop the inference. The activation of $\textit{age\_of:arg}_{1_{left}}\textit{[1]}$ first propagates to *tom:gate[0]* and then to the constant node *tom[0]*, activating both of them in the first phase. This is the the same phase allocated to the variable node U at the beginning of the previous inference. In addition, the activation of *age_of:sb[2]* also propagates to the $\textit{greater}_f\textit{:arg}_{1_{right}}\textit{[0]}$ to make it active in the second phase.

At the end of binding propagation, the assembly representing the structured term *age_of* and the fact assembly $\textit{greater}_f$ will be activated in the following phases:

$$\textit{age\_of}:\{\textit{fb[2]},\textit{arg}_1\textit{([1],[1])}\},\ \textit{tom[1]},$$
$$\textit{greater}_f:\{\textit{pa[*]},\textit{arg}_1\textit{([0],[2])},\textit{arg}_2\textit{([0],[3])}\}.$$

From the in-phase oscillation between *tom[1]* and the argument of $\textit{age\_of:arg}_1\textit{([1],[1])}$, the binding {tom/U} is obtained and the activation of the fact assembly indicates that the encoded fact *greater(age_of(tom),fifty)* is matched with the posed query predicate. Consequently, the query *on_pension(U)* in the previous subsection is succeeded with the binding {tom/U}.

6.4 Summary

This chapter has described how CASI can be extended to deal with unbalanced rules and rules and facts involving structured terms. Encoding unbalanced rules required additional sub-mechanisms: the phase requester and the external phase manager (EPM). The necessity of these sub-mechanisms is due to the fact that all the bindings have to be represented in the form of phases in CASI. Extending the knowledge encoding mechanism to accommodate rules and facts involving structured terms also required the modification of binding propagation sub-mechanisms. To setup proper bindings on SPNs encoding these rules and facts, the initial binding instantiation procedure had to be extended to handle the structured terms involved.

Chapter 7

Experiments and Evaluation

7.1 Overview

This chapter describes the evaluation of CASI. The first part portrays experiments involving detailed behaviour of CASI as a compiler of rules and facts in first-order Horn clause expressions. Each experiment is carried out by encoding example rules and facts into a network and performing inference over the network. A test set of rules and fact are chosen to test a specific sub-mechanism of CASI.

The second part will analyse the time and space complexities of CASI. The time complexity is measured on the basis of oscillation cycles taken for each type of inference. The space complexity is analysed by counting the number of nodes required to construct an SPN for each type of rule or fact.

The last part of this chapter compares CASI with other similar connectionist inference systems. Architectural differences as well as advantages and disadvantages of CASI will be illustrated with reference to those of other systems.

7.2 Experiments with CASI

The experiments are carried out to test CASI's knowledge encoding and inference mechanisms. Each test set of rules and facts are selected to test one of the functional abilities of CASI summarised as follows:

- representation of dynamic bindings including constant bindings and variable bindings;
- various types of consistency checking not only within UAG but also across UAGs;
- inference involving various types of rules;
- representation of unification results during inference;
- forward chaining and backward chaining.

When each set of rules and facts are given to CASI, its knowledge compiler translates them into the corresponding SPNs. The SPNs are then used to perform a symbolic style of inference when a predicate with various combination of fillers is presented to initiate inference over the SPNs. The results of inference are obtained by observing predicate assemblies activated during inference. To help the interpretation of these result, CASI's user interface provides the corresponding symbolic representation of these activations. The type of inference used in each experiment is forward chaining unless otherwise specified. Appendix B also shows additional experiments with six more rules which are not dealt with in this chapter.

7.2.1 Representation of Dynamic Bindings

Objectives:

Representing dynamic bindings involves representing constant bindings and variable bindings. The purpose of this experiment is to test if CASI correctly represents constant and variable bindings over an SPN which encodes a simple basic rule. The experiment also tests if represented bindings correctly propagate from the source predicate assembly to the target predicate assembly to achieve a step of inference.

The sample rule used:

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

The test predicates used:

P1: *p(a,b)* - to test representation of constant bindings and their propagation;

P2: *p(U,V)* - to test representation of variable bindings and their propagation;

P3: *p(U,a)* - to test representation and propagation of both bindings at the same time.

The results:

```
(P1) predicate presented: p(a,b)
  . phase allocation - a[1],b[6]
  . initial bindings - p:{pa[*],arg1([0],[1]),arg2([0],[6])}
  . inference result - q:{pa[*],arg1([0],[1]),arg2([0],[6])}
    (symbolic meaning) by right elements - q(a,b)
  . inference cycles taken - 1 cycle

(P2) predicate presented: p(U,V)
  . phase allocation - U[1],V[6],
  . initial bindings - p:{pa[*],arg1([1],[0]),arg2([6],[0])}
  . inference result - q:{pa[*],arg1([1],[0]),arg2([6],[0])}
    (symbolic meaning) by left elements - q(U,V)
  . inference cycles taken - 1 cycle

(P3) predicate presented: p(U,a)
  . phase allocation - U[1],a[6],
  . initial bindings - p:{pa[*],arg1([1],[0]),arg2([0],[6])}
  . inference result - q:{pa[*],arg1([1],[0]),arg2([0],[6])}
    (symbolic meaning) by right elements - q(0,a)
                       by left elements - q(U,0)
  . inference cycles taken - 1 cycle
```

The result of each inference was extracted from the output of CASI and slightly modified before it was put in here to help readability. Each number stands for a specific phase and the number "0" is used to indicate the inactive state of a π-btu element. As mentioned earlier, the symbol "*" is used to present activation throughout the entire oscillation cycle. The predicate activators (*pa*'s) are represented with this symbol.

Since this is the simplest type of rule, it took only one oscillation cycle to obtain the answer for each predicate presented. (P1) shows the representation of constant bindings and (P2) variable bindings. In (P3), a constant and a variable binding are represented together. As designed, a constant binding was represented using the right element of an entity node and a variable binding the left element of an entity node. The results of inference were obtained by automatic propagation of initial bindings from the antecedent predicate assembly to the consequent predicate assembly on the SPN.

7.2.2 Consistency Checking Within UAG

This experiment is to test consistency checking sub-mechanisms within each UAG. Two types of consistency checking mechanisms will be explored: a consistency checking sub-mechanism for a constant UAG (CCM_c) and that of a variable UAG which has repeated argument (CCM_v).

For a Constant UAG

Objectives:

When a rule whose antecedent has a constant argument is encoded, a constant consistency checking sub-mechanism (CCM_c) is added into the intermediate mechanism. This sub-mechanism forces consistency between the constant argument and its corresponding filler during inference. This experiment explores the behaviour of the CCM_c.

The sample rule used:

$$p(a) \rightarrow q(a)$$

The test predicates used:

P1: *p(a)* - to test the consistency checking sub-mechanism (CCM_c) when the given constant does not violate consistency. The completion of inference by the binding propagation sub-mechanism (BPM) is expected;

P2: *p(b)* - to test the consistency checking sub-mechanism when the given filler violate consistency. The inference should not be completed;

P3: *p(U)* - to test unification between the variable filler and the constant argument when a variable filler is assigned to the argument.

The results:

```
(P1) predicate presented: p(a)
  . phase allocation - a[1]
  . initial bindings - p:{pa[*],arg1([0][1])}
  . inference results - q:{pa[*],arg1([0][1])}
    (symbolic meaning) by right element - q(a)
  . inference cycles taken - 3 cycles

(P2) predicate presented: p(b)
  . phase allocation - U[1]
  . initial bindings - p:{pa[*],arg1([1][0])}
  . inference results - none
    (symbolic meaning) - no result

(P3) predicate presented: p(U)
  . phase allocation - U[1]
  . initial bindings - p:{pa[*],arg1([1][0])}
  . inference results - q:{pa[*],arg1([1][1])},
                        p_q:a_gate[1]
    (symbolic meaning) by right element - q(a)
                       by left element -  q(U)
                       unification result - {a/U}
  . inference cycles taken - 3 cycles
```

At first test, (P1), the filler a satisfied the constant consistency condition, therefore the initial binding $\{a/p{:}arg_1\}$ propagated to the target predicate and produced the desirable result *q(a)*. In the last test, the variable filler U was presented to the argument node and this resulted in the activation of the argument of q predicate in *arg1([1][1])*. Since the left element is active in-phase with the variable node *U[1]* and the right element in-phase with the constant gate node of the given rule, *p_q:a_gate[1]*, this represent the result *q(a)* and the result of unification $\{a/U\}$. The second test did not produce any results since the presented constant filler b is not the same as the one specified as the constant argument. This inconsistency was detected by the CCM_c and a step of inference was stopped by it.

For Variable UAGs Having Repeated Arguments

Objectives:

Repeated variable arguments of a rule have to get bound to the same constant fillers or free variable filler in order not to violate consistency of a variable UAG. This experiment is carried out to test whether the variable consistency checking sub-mechanism (CCM_v) performs proper consistency checking during inference.

The sample rule used:

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

The test predicates used:

P1: *p(a,a)* - to test the CCM_v when the same constant fillers are assigned to the repeated variable arguments. The completion of inference is expected;

P2: *p(a,U)* - to test the CCM_v when a constant filler and a variable filler are assigned to the repeated variable arguments. The completion of inference with the representation of unification result between the constant and the variable filler is expected;

P3: *p(U,V)* - to test the CCM_v when two different variable fillers are assigned to the repeated variable arguments. The completion of inference with the representation of unification result is expected;

P4: *p(a,b)* - to test the CCM_v if it stops the inference when two different constant fillers are assigned to the repeated variable arguments.

The results:

```
(P1) predicate presented: p(a,a)
  . phase allocation - a[1]
  . initial bindings - p:{pa[*],arg1([0][1]),arg2([0][1])}
  . inference result - q:{pa[*],arg1([0][1])}
    (symbolic meaning) by right element - q(a)
  . inference cycles taken - 3 cycles

(P2) predicate presented: p(a,U)
  . phase allocation - a[1],U[6]
  . initial bindings - p:{pa[*],arg1([0][1]),arg2([6],[0])}
  . inference result - q:{pa[*],arg1([6][1])}
    (symbolic meaning) by right element - q(a)
                       by left element -  q(U)
                       unification result - {a/U}
  . inference cycles taken - 3 cycles

(P3) predicate presented: p(U,V)
  . phase allocation - U[1],V[6]
  . initial bindings - p:{pa[*],arg1([1][0]),arg2([6],[0])}
```

```
    . inference result - q:{pa[*],arg1([1,6][0])}
      (symbolic meaning) by left element - q(U)
                         unification result - {U/V}
    . inference cycles taken - 3 cycles

  (P4) predicate presented: p(a,b)
    . phase allocation - a[1],b[6]
    . initial bindings -  p:{pa[*],arg1([0][1]),arg2([0],[6])}
    . inference result - none
      (symbolic meaning) - no result
```

When the same constant was presented to the repeated argument, the CCM_v did not became active as can be seen in (P1). Thus, the initial bindings propagated to the argument of q predicate and produced the desirable result *q(a)*. During the second test, two fillers, a and U, are unifiable. Thus the binding collection sub-mechanism (BCM) first produced intermediate binding, $\{a/U\}$, without checking consistency. Since no consistency violation was found, the intermediate binding was propagate to the argument node of q predicate assembly through the binding propagation sub-mechanism (BPM). Consequently both constant and variable bindings were represented at the same time from which the unification result, $\{a/U\}$, can be obtained. In the third test, on the other hand, both fillers presented to the repeated arguments were variables. Both argument nodes of the p predicate therefore represented only variable bindings and those bindings were propagated to the q predicate assembly, resulting *q(U)* with the unification result $\{U/V\}$ after three oscillation cycles. The last test demonstrated the active role of the CCM_v. When two different constant fillers were presented to the repeated arguments, this sub-mechanism detected the inconsistency, and blocked the propagation of the bindings from the source predicate assembly to the target predicate assembly.

7.2.3 Consistency Checking Between a Pair of UAGs

If arguments belonging to different UAGs get bound to the same variable filler, binding interaction (limited unification) occurs between a pair of UAGs. This binding interaction unifys pre-established bindings in all argument nodes of UAGs involved but this is limited unification because it does not involves any consistency checking. Thus an additional consistency checking sub-mechanism is needed. The intermediate bindings generated by the binding collection sub-mechanism (BCM) have to be checked how their consistency by this sub-mechanism before to be propagated to the target predicate assembly. Since CASI considers consistency checking only between a pair of UAGs, three types of binding interaction and consistency checking sub-mechanisms ($BIM\&CCM$s) are required to carry out this task:

- the $BIM\&CCM_{cv}$ - between a constant UAG and a variable UAG,
- the $BIM\&CCM_{cc}$ - between variable UAGs,
- the BIM_{cc} - between constant UAGs.

Between a Constant and Variable UAGs

Objectives:

To test the binding interaction and consistency checking sub-mechanisms between a constant UAG and a variable UAG ($BIM\&CCM_{cv}$). These sub-mechanisms are activated only when the same variable filler is assigned to both UAGs. Otherwise, the initial bindings propagate to the target predicate assembly through the binding propagation sub-mechanism (BPM).

The sample rule used:

$$p(a, X, X) \rightarrow q(a, X)$$

The test predicates used:

P1: *p(a,U,b)* - to test binding collection and consistency checking within the repeated variable UAG;

P2: *p(U,a,U)* - to test binding interaction and consistency checking within the repeated variable UAG and across UAGs;

P3: *p(U,U,U)* - to test unification within the constant UAG and binding interaction across UAGs;

P4: *p(U,b,U)* - to test the situation where the $BIM\&CCM_{cv}$ detects consistency violation during inference.

The results:

```
(P1) predicate presented: p(a,U,b)
  . phase allocation - a[1],U[4],b[7]
  . initial bindings - p:{pa[*],arg1([0][1]),arg2([4][0]),arg3([0][7])}
  . inference result - q:{pa[*],arg1([0][1]),arg2([4][7])}
    (symbolic meaning) by right elements - q(a,b)
                       by left elements - q(0,U)
                       unification result - {b/U}
  . inference cycles taken - 5 cycles

(P2) predicate presented: p(U,a,U)
  . phase allocation - a[1],U[6]
  . initial bindings - p:{pa[*],arg1([6][0]),arg2([0][1]),arg3([6][0])}
  . inference result - q:{pa[*] arg1([6][1]),arg2([6][1])}
    (symbolic meaning) by right elements - q(a,a)
                       by left elements - q(U,U)
                       unification result - {a/U}
  . inference cycles taken - 5 cycles

(P3) predicate presented: p(U,U,U)
  . phase allocation - U[1]
  . initial bindings - p:{pa[*],arg1([0][1]),arg2([0][1]),arg3([0][1])}
  . inference result - q:{pa[*],arg1([1][1]),arg2([1][1])}
                       p_q:a_gate[1]
    (symbolic meaning) by right elements - q(a,a)
                       by left elements - q(U,U)
                       unification result -  {a/U}
  . inference cycles taken - 5 cycles

(P4) predicate presented: p(U,b,U)
  . phase allocation - U[1],b[6]
  . initial bindings - p:{pa[*],arg1([1][0]),arg2([0][6]),arg3([1][0])}
  . inference result - none
    (symbolic meaning) - no result
```

The first test shows binding collection and consistency checking within the UAG which has repeated arguments. Since two fillers a and U assigned to the repeated variable arguments do not violate consistency, the inference produced the expected result, *p(a,b)*, with the binding $\{b/U\}$. In case of (P2), (P3), and (P4), the same variable filler was assigned to argument nodes of both the constant UAG and the variable UAG which has repeated arguments. Therefore, the $BIM\&CCM_{cv}$ became active during inference. The difference is that the presented predicates, (P2) and (P3), did not violate consistency but (P4) did violate consistency. These inferences first performed binding collection within the argument nodes belong to the variable UAG which has repeated arguments and unification within the constant UAG and then binding interaction across them. Note that the result of (P4) was represented by activating both the target predicate assembly q and the gate node of the constant a to produce the unification result $\{a/U\}$. This is because no constant filler was assigned to the repeated arguments of the source predicate assembly. In case of (P4), consistency violation occurred between two UAGs because they share the same variable filler U and the filler was bound to two different constants a and b. The consistency checking part of the $BIM\&CCM_{cv}$ detected this and blocked the inference. Consequently no result was obtained.

Between Two Variable UAGs

Objectives:

To explore the binding interaction and consistency checking sub-mechanism between variable UAGs ($BIM\&CCM_{vv}$). A rule which requires these sub-mechanisms should have at least one variable UAG which has repeated arguments in the source predicate.

The sample rule used:

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

The test predicates used:

P1: *p(a,a,b,b)* - to test consistency checking for repeated variable arguments within each UAG;

P2: *p(U,V,U,V)* - to test binding interaction across two variable UAGs;

P3: *p(a,U,U,U)* - to test binding collection and consistency checking within each UAG and binding interaction across two UAGs when only one variable filler is involved;

P4: *p(U,V,a,U)* - to test binding collection and consistency checking within each UAG and binding interaction across two UAGs when two variable fillers are involved;

P5: *p(U,a,U,b)* - to test binding collection and consistency checking within each UAG and binding interaction across two UAGs when the given predicate violates consistency during inference.

The results:

```
(P1) predicate presented: p(a,a,b,b)
  . phase allocation - a[1],b[6]
  . initial bindings - p:{pa[*],arg1([0][1]),arg2([0][1]),
                                 arg3([0][6]),arg4([0][6])}
  . inference result - q:{pa[*],arg1([0][1]),arg2([0][6])}
    (symbolic meaning) by the right elements - q(a,b)
  . inference cycles taken - 5 cycles

(P2) predicate presented: p(U,V,U,V)
  . phase allocation - U[1],V[6]
  . initial bindings - p:{pa[*],arg1([1][0]),arg2([6][0]),
                                 arg3([1][0]),arg4([6][0])}
  . inference result - q:{pa[*],arg1([1,6][0]),arg2([1,6][0])}
    (symbolic meaning) by the left elements -  q(U,U)
                       unification result - {U/V}
  . inference cycles taken - 5 cycles

(P3) predicate presented: p(a,U,U,U)
  . phase allocation - a[1],U[6]
  . initial bindings - p:{pa[*],arg1([0][1]),arg2([6][0]),
                                 arg3([6][0]),arg4([6][0])}
  . inference result - q:{pa[*],arg1([6][1]),arg2([6][1])}
    (symbolic meaning) by right elements - q(a,a)
                       by left elements - q(U,U)
                       unification result- {a/U}
  . inference cycles taken - 5 cycles

(P4) predicate presented: p(U,V,a,U)
  . phase allocation - U[1],a[4],V[7]
  . initial bindings - p:{pa[*],arg1([1][0]),arg2([7][0]),
                                 arg3([0][4]),arg4([1][0])}
  . inference result - q:{pa[*],arg1([1,7][4]),arg2([1,7][4])}
    (symbolic meaning) by right elements - q(a,a)
                       by left elements - q(U,U)
                       unification result- {a/U,U/V}
  . inference cycles taken - 5 cycles

(P5) predicate presented: p(U,a,U,b)
  . phase allocation - U[1],a[4],b[7]
  . initial bindings - p:{pa[*],arg1([1][0]),arg2([7][0]),
                                 arg3([0][4]),arg4([1][0])}
  . inference result - none
    (symbolic meaning) - no result
```

The first test simply demonstrates the propagation of the initial bindings to the target predicate assembly after binding collection and consistency checking within each UAG. The CCM_c for each UAG was not activated because a pair of constant fillers assigned to each repeated arguments are the same. The result *q(a,b)* was produced without involving binding interaction across UAGs. The rest of the test predicates, on the other hand, required not only binding collection and consistency checking within each UAG but also binding interaction across the two UAGs because the same variable filler was assigned to both UAGs at the same time. The test predicate (P2) has only variable fillers thus produced the unification result, $\{U/V\}$, on the target predicate. The predicates, (P3) and (P4), are basically the same except the number of variable fillers involved. The predicate (P3) has one variable filler, U, and the predicate (P4) two variable fillers, U and V. Their inference results are the same but the unification results are different: $\{a/U\}$ for (P3) and $\{a/U, U/V\}$ for (P4). The last predicate (P5) was chosen to cause a consistency violation. During binding interaction across two UAGs, the consistency checking part of the $BIM\&CCM_{vv}$ detected this violation and blocked the inference.

Between Two Constant UAGs

Objectives:

To test the binding interaction sub-mechanism between two constant UAGs (BIM_{cc}). If the same variable filler is assigned to argument nodes belonging to two constant UAGs at the same time, the BIM_{cc} has to detect this and prevent the rule from firing. An additional consistency checking sub-mechanism (CCM_{cc}) is not necessary in this case because the BIM_{cc} performs the necessary task for consistency checking between two constant UAGs.

The sample rule used:

$$p(a,b) \rightarrow q(a,b)$$

The test predicates used:

- P1: *$p(a,b)$* - to test the CCM for each UAG. The completion of inference is expected because the constant fillers assigned to constant arguments are the same as the constant argument names;
- P2: *$p(U,V)$* - to test unification within each constant UAG;

P3: *p(a,c)* - to test the CCM associated with each UAG. The CCM for the second argument should be able to detect the consistency violation during inference;

P4: *p(U,U)* - to test unification within each UAG and binding interaction across constant UAGs. Any binding interaction occurred between constant UAGs leads consistency violation during inference.

The results:

```
(P1) predicate presented: p(a,b)
  . phase allocation - a[1],b[6]
  . initial bindings - p:{pa[*],arg1([0][1]),arg2([0][6])}
  . inference result - q:{pa[*],arg1([0][1]),arg2([0][6])}
    (symbolic meaning) by right elements - q(a,b)
  . inference cycles taken - 4 cycles

(P2) predicate presented: p(U,V)
  . phase allocation - U[1],V[6]
  . initial bindings - p:{pa[*],arg1([1][0]),arg2([6][0])}
  . inference result - q:{pa[*],arg1([1][1]),arg2([6][6])}
                      p_q:a_gate[1],p_q:b_gate[6]
    (symbolic meaning) by right elements - q(a,b)
                       by left elements - q(U,V)
                       unification results - {a/U,b/V}
  . inference cycles taken - 4 cycles

(P3) predicate presented: p(a,c)
  . phase allocation - b[1],a[6]
  . initial bindings - p:{pa[*],arg1([0][1]),arg2([0][6])}
  . inference result - none
    (symbolic meaning) - no result

(P4) predicate presented: p(U,U)
  . phase allocation - U[1]
  . initial bindings - p:{pa[*],arg1([1][0]),arg2([1][0])}
  . inference result - none
    (symbolic meaning) - no result
```

The first two tests were carried out to observe proper propagation of the initial bindings to the target predicate. The first test (P1) was straight forward since the presented constant fillers were the same as the name of the constant constant arguments. The CCMs for the constant UAGs did not project an inhibitory signal to stop the binding propagation. Thus this step of inference produced the result *q(a,b)*. During the second test, (P2), the initial variable bindings propagated not only to the target predicate assembly but also to the gate nodes of both constants, *p_q:a_gate* and *p_q:b_gate*. The in-phase oscillation between these constant gate nodes and the variable fillers, U and V, represented the results of unification required. The inference for the third test predicate (P3) was not successful because the CCM for the constant argument b became active during inference. The last test predicate (P4) was also failed because of the BIM_{cc} across the two constant UAGs detected the situation where the variable filler U gets bound to the two different constant arguments a and b at the same time. Consequently, the BIM_{cc} became active and blocked the inference.

Across more than two UAGs

Objectives:

If a rule has more than two UAGs, the result of the binding interaction between two UAGs may require further binding interaction with other UAGs. This experiment explores how $BIM\&CCM$s carry out binding interaction and consistency checking when more than two UAGs are involved.

The sample rule used:

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

The test predicates used:

- P1: $p(a,b,U,c,V)$ - to test binding collection and consistency checking within each UAG;
- P2: $p(U,U,U,U,U)$, P3: $p(a,b,U,a,U)$ - to test binding interaction across two UAGs and consistency checking after binding interaction;
- P4: $p(U,a,V,U,V)$, P5: p(U,b,V,U,V) - to test binding interaction across more than two UAGs at the same time;
- P6: $p(U,V,W,U,V)$ - to test binding interaction across more than two UAGs when all fillers are variables.

The results:

```
(P1) predicate presented: p(a,b,U,c,V)
  . phase allocation - a[1],b[3],U[5],c[7],V[9]
  . initial bindings - p:{pa[*],arg1([0][1]),arg2([0][3]),arg3([5][0]),
                              arg4([0][7]),arg5([9][0])}
  . inference result - q:{pa[*],arg1([0][1]),arg2([5][3]),arg3([9][7])}
    (symbolic meaning) by right elements - q(a,b,c)
                       by left elements - q(0,U,V)
                       unification results - {b/U,c/V}
  . inference cycles taken - 5 cycles

(P2) predicate presented: p(U,U,U,U,U)
  . phase allocation - U[1]
  . initial bindings - p:{pa[*],arg1([1][0]),arg2([1][0]),arg3([1][0]),
                              arg4([1][0]),arg5([1][0])}
  . inference result - q:{pa[*],arg1([1][1]),arg2([1][1]),arg3([1][1])}
                       p_q:a_gate[1]
    (symbolic meaning) by right elements - q(a,a,a)
                       by left elements - q(U,U,U)
                       unification results - {a/U}
  . inference cycles taken - 5 cycles

(P3) predicate presented: p(a,b,U,a,U)
  . phase allocation - b[1],a[4],U[7]
  . initial bindings - p:{pa[*],arg1([0][4]),arg2([0][1]),arg3([7][0]),
                              arg4([0][4]),arg5([7][0])}
  . inference result - none
    (symbolic meaning) - no result

(P4) predicate presented: p(U,a,V,U,V)
  . phase allocation - a[1],U[4],V[7]
  . initial bindings - p:{pa[*],arg1([4][0]),arg2([0][1]),arg3([7][0]),
                              arg4([4][0]),arg5([7][0])}
  . inference result - q:{pa[*],arg1([4,7][1]),arg2([4,7][1]),arg3([4,7][1])}
    (symbolic meaning) by right elements - q(a,a,a)
                       by left elements - q(U,U,U)
                       unification results - {a/U,U/V}
  . inference cycles taken - 5 cycles

(P5) predicate presented: p(U,b,V,U,V)
  . phase allocation - b[1],U[4],V[7]
  . initial bindings - p:{pa[*],arg1([4][0]),arg2([0][1]),arg3([7][0]),
                              arg4([4][0]),arg5([7][0])}
  . inference result - none
    (symbolic meaning) - no result

(P6) predicate presented: p(U,V,W,U,V)
  . phase allocation - W[1],U[4],V[7]
  . initial bindings - p:{pa[*],arg1([4][0]),arg2([7][0]),arg3([1][0]),
                              arg4([4][0]),arg5([7][0])}
  . inference result - q:{pa[*],arg1([4,7][4]),arg2([1,4,7][0]),
                              arg3([1,4,7][4])}
                       p_q:a_gate[4]
    (symbolic meaning) by right elements - q(a,0,a)
                       by left elements - q(U,U,U)
```

```
                        unification results - {a/U,U/V,V/W}
  . inference cycles taken - 5 cycles
```

When the first predicate (P1) was presented, the fillers assigned to the repeated arguments did not violate consistency and there was no binding interaction across UAGs. Therefore, the predicate *q(a,b,c)* with the result of unification, $\{b/U, c/V\}$, was obtained as the result of inference. The second and third predicates, (P2) and (P3), required binding interaction between two UAGs of the source predicate. When the second predicate is presented to CASI, the intermediate binding generated between the variable filler U and the constant UAG, a, propagated to another UAGs and this produced the result *q(a,a,a)* with the unification result $\{a/U\}$. However, the third test predicate did not produce any results because two repeated argument UAGs sharing the same variable filler were bound to two different constant fillers and this caused consistency violation during inference. The next predicates, (P4) and (P5), demonstrated more complicated binding interaction among the three source UAGs. For the forth predicate, the last UAG (G_Y) shares the same variable filler with the first and the second UAGs (G_a and G_X). Thus the inference needed binding interaction between the first and the third UAGs and between the second and the third UAGs including consistency checking. The binding interaction between the first and the second UAGs is done indirectly through the third UAG. Since all the initial bindings did not violate consistency conditions, the desirable result is obtained. However, when the predicate (P5) was presented, consistency violation occurred because of the constant binding generated by the second UAG (G_X) conflicted with another constant binding generated between the first and third UAGs. Consequently, the inference is blocked by the CCM after binding interaction across UAGs. Finally, the last predicate (P6) also required binding interaction among the source UAGs during inference. Since it has only variable fillers, the unification results have to be represented on the target predicate assembly. The result of this inference obtained by the target argument nodes was *q(a,0,a)* with the unification result $\{a/U, U/V, V/W\}$ and did not produce the desirable constant binding for the second argument of the q predicate. This was found to be one of limitations that CASI's current knowledge encoding mechanism has. More details about this limitation will be discussed in Section 8.2.2.

7.2.4 Rules with Multiple Antecedent Predicates

Objectives:

When a rule whose antecedent consists of more than one predicate is encoded into an SPN, the consequent of the rule should be activated only if all the antecedent predicates are activated at the same time with proper initial bindings. Partial activation of the antecedent predicates should not allow firing of the rule. This experiment explores an additional sub-mechanism inserted into the intermediate mechanism to ensure this condition. To make a comparison easier, this experiment uses the rule that was used in the previous experiment. The rule has been slightly modified in such a way that all the arguments of the antecedent are divided into three antecedent predicates.

The sample rule used:

$$p(a, X) \wedge q(X) \wedge r(Y, Y) \rightarrow s(a, X, Y)$$

The test predicates used:

- P1: $p(a, b) \wedge q(U) \wedge r(c, V)$ - to test binding collection and consistency checking within each UAG;
- P2: $p(U, U) \wedge q(U) \wedge r(U, U)$, P3: $p(a, b) \wedge q(U) \wedge r(a, U)$ - to test binding interaction across two UAGs and consistency checking after binding interaction;
- P4: $p(U, a) \wedge q(V) \wedge r(U, V)$, P5: $p(U, b) \wedge q(V) \wedge r(U, V)$ - to test binding interaction across more than two UAGs at the same time;
- P6: $p(U, V) \wedge q(W) \wedge r(U, V)$ - to test binding interaction across more than two UAGs when all fillers are variables;
- P7: $p(a, b) \wedge q(b)$ - to test the sub-mechanism for multiple antecedent predicates, where only parts of antecedent predicates are active.

The results:

```
(P1) predicate presented: p(a,b) ^ q(U) ^ r(c,V)
```

```
  . phase allocation - a[1],b[3],U[5],c[7],V[9]
  . initial bindings - p:{pa[*],arg1([0][1]),arg2([0][3])}
                       q:{pa[*],arg1([5][0])}
                       r:{pa[*],arg1([0][7]),arg2([9][0])}
  . inference result - s:{pa[*],arg1([0][1]),arg2([5][3]),arg3([9][7])}
    (symbolic meaning) - by right elements - s(a,b,c)
                         by left elements - s(0,U,V)
                         Unification results - {b/U,c/V}
  . inference cycles taken - 5 cycles

(P2) predicate presented: p(U,U) ^ q(U) ^ r(U,U)
  . phase allocation - U[1]
  . initial bindings - p:{pa[*],arg1([1][0]),arg2([1][0])}
                       q:{pa[*],arg2([1][0])}
                       r:{pa[*],arg1([1][0]),arg2([1][0])}
  . inference result - s:{pa[*],arg1([1][1]),arg2([1][1]),arg3([1][1])}
                       p_q_r_s:a_gate[1]
    (symbolic meaning) by right elements - s(a,a,a)
                       by left elements - s(U,U,U)
                       Unification results - {a/U}
  . inference cycles taken - 5 cycles

(P3) predicate presented: p(a,b) ^ q(U) ^ r(a,U)
  . phase allocation - b[1],a[4],U[7]
  . initial bindings - p:{pa[*],arg1([0][4]),arg2([0][1])}
                       q:{pa[*],arg1([7][0])}
                       r:{pa[*],arg1([0][4]),arg2([7][0])}
  . inference result - none
    (symbolic meaning) - no result

(P4) predicate presented: p(U,a) ^ q(V) ^ r(U,V)
  . phase allocation - b[1],U[4],V[7]
  . initial bindings - p:{pa[*],arg1([4][0]),arg2([0][1])}
                       q:{pa[*],arg1([7][0])}
                       r:{pa[*],arg1([4][0]),arg2([7][0])}
  . inference result - s:{pa[*],arg1([4,7][1]),arg2([4,7][1]),arg3([4,7][1])}
    (symbolic meaning) by right elements - s(a,a,a)
                       by left elements - s(U,U,U)
                       unification results - {a/U,U/V}
  . inference cycles taken - 5 cycles

(P5) predicate presented: p(U,b) ^ q(V) ^ r(U,V)
  . phase allocation - b[1],U[4],V[7]
  . initial bindings - p:{pa[*],arg1([4][0]),arg2([0][1])}
                       q:{pa[*],arg1([7][0])}
                       r:{pa[*],arg1([4][0]),arg2([7][0])}
  . inference result - none
    (symbolic meaning) - no result

(P6) predicate presented: p(U,V) ^ q(W) ^ r(U,V)
  . phase allocation - W[1],U[4],V[7]
  . initial bindings - p:{pa[*],arg1([4][0]),arg2([7][0])}
                       q:{pa[*],arg1([1][0])}
                       r:{pa[*],arg1([4][0]),arg2([7][0])}
```

```
  . inference result - s:{pa[*],arg1([4,7][4]),arg2([1,4,7][0]),
                                 arg3([1,4,7][4])}
                       p_q_r_s:a_gate[4]
    (symbolic meaning) by right elements - s(a,0,a)
                       by left elements - s(U,U,U)
                       unification results - {a/U,U/V,V/W}
  . inference cycles taken - 5 cycles

(P7) predicate presented: p(a,b) ^ q(b)
  . phase allocation - a[1],b[6]
  . initial bindings - p:{pa[*],arg1([0][1]),arg1([0][6])}
                       q:{pa[*],arg2([0][6])}
  . inference result - none
    (symbolic meaning) - no result
```

As expected, the results obtained in this experiment were the same as those obtained in the previous experiment except for the additional test predicate (P7). The last test predicate demonstrated that this example rule fires only if all of three antecedent predicate assemblies become active without violation of consistency conditions. As can be seen, activation of only some of the antecedent predicates did not produce any results.

7.2.5 Rules with Isolated UAGs in The Antecedent

Objectives:

This experiment explores an SPN encoding a rule which has isolated UAGs in the source predicate. If an isolated UAG is a constant UAG or a variable UAG which has repeated arguments, a consistency checking sub-mechanism is required for that isolated UAG but it does not require a binding propagation sub-mechanism (BPM).

The sample rule used:

$$p(X,Y) \wedge q(Y,Z) \rightarrow r(X,Z)$$

The test predicates used:

P1: $p(a,b) \wedge q(b,c)$, P2: $p(a,b) \wedge q(c,d)$ - to test binding propagation sub-mechanisms when constant fillers are assigned with and without consistency violation for the isolated UAG (G_Y) in the source predicate;

P3: $p(U,U) \wedge q(a,V)$, P4: $p(U,a) \wedge q(U,U)$ - to test binding collection and consistency checking within each UAG and binding interaction between two UAGs involving the isolated UAG;

P5: $p(U,V) \wedge q(U,V)$ - to test binding interaction across UAGs with only variable fillers;

The results:

```
(P1) predicate presented: p(a,b) ^ q(b,c)
  . phase allocation - a[1],b[4],c[7]
  . initial bindings - p:{pa[*],arg1([0][1]),arg2([0][4])}
                       q:{pa[*],arg1([0][4]),arg2([0][7])}
  . inference result - r:{pa[*],arg1([0][1]),arg2([0][7])}
    (symbolic meaning) by right elements - r(a,c)
  . inference cycles taken - 5 cycles

(P2) predicate presented: p(a,b) ^ q(c,d)
  . phase allocation - a[1],b[3],c[5],d[7]
  . initial bindings - p:{pa[*],arg1([0][1]),arg2([0][3])}
                       q:{pa[*],arg1([0][5]),arg2([0][7])}
  . inference result - none
    (symbolic meaning) - no result

(P3) predicate presented: p(U,U) ^ q(a,V)
  . phase allocation - U,[1],a[4],V[7]
  . initial bindings - p:{pa[*],arg1([1][0]),arg2([1][0])}
                       q:{pa[*],arg1([0][4]),arg2([7][0])}
  . inference result - r:{pa[*],arg1([1][4]),arg2([7][0])}
    (symbolic meaning) by right elements - r(a,0)
                       by left elements - r(U,V)
                       unification results - {a/U}
  . inference cycles taken - 5 cycles

(P4) predicate presented: p(U,a) ^ q(U,U)
  . phase allocation - a[1],U[6]
  . initial bindings - p:{pa[*],arg1([6][0]),arg2([0][1])}
                       q:{pa[*],arg1([6][0]),arg2([6][0])}
  . inference result - r:{pa[*],arg1([6][1]),arg2([6][1])}
    (symbolic meaning) by right elements - r(a,a)
                       by left elements - r(U,U)
                       unification results - {a/U}
  . inference cycles taken - 5 cycles

(P5) predicate presented: p(U,V) ^ q(U,V)
  . phase allocation - U[1],V[6]
  . initial bindings - p:{pa[*],arg1([1][0]),arg2([6][0])}
                       q:{pa[*],arg1([1][0]),arg2([6][0])}
  . inference result - r:{pa[*],arg1([1,6][0]),arg2([1,6][0])}
    (symbolic meaning) by left elements - r(U,U)
                       unification results - {U/V}
```

The example rule has the isolate UAG (G_Y) which has repeated variable arguments (Y's). This UAG requires the associated consistency checking sub-mechanism when encoded. The first and second predicates were simply to test binding propagation from the source predicate assembly to the target predicate assembly when the fillers assigned to the isolated UAG did and did not violate consistency. As can be seen, the first inference produced the result, *r(a,c)*, because the initial bindings on the argument nodes of the isolate UAG did not violate consistency. The second predicate, on the other hand, did not complete the inference because of consistency violation. The predicate (P3) required binding interaction between the isolated UAG (G_Y) and the first UAG (G_X) and the predicate (P4) needed extra binding interaction between isolated UAG and the third UAG (G_Z). When the SPN is tested using these two predicates, the desirable results were obtained as the results of the inference. As for the last predicate (P5), it has only variable fillers and consequently produced only variable bindings as the result of unification.

7.2.6 Backward Chaining

A Set of Rules and Facts

Objectives:

This experiment is to explore CASI's inference mechanism when it is used to perform backward chaining. To carry out the experiment, a set of rules and facts are translated into SPNs and then various query predicates, representing specific types of questions, are posed to CASI. CASI's inference mechanism then performs backward chaining over the network and gives answers to those queries. The basic example rules and facts used in this experiment are taken from Shastri & Ajjanagadde (1993).

The sample rules and facts used:

$$give(X,Y,Z) \rightarrow own(X,Y)$$
$$buy(X,Y) \rightarrow own(X,Y)$$
$$own(X,Y) \rightarrow can\_sell(X,Y)$$
$$give(john, mary, book1)$$

$$give(jane, tom, computer2)$$

$$buy(john, car3)$$

$$own(mary, ball4)$$

The test queries used:

Q1: *give(jane,tom,U)* - corresponds to the question, "What did Jane give Tom?". Finding an object *U* is the purpose of this inference;

Q2: *own(mary,U)* - is considered as the question, "What does Mary own?". The aim of this inference is finding an object that Mary owns;

Q3: *own(U,car3)* - indicates the question, "Who owns Car3?". A name of a person is expected to get bound to the variable *U* if there is any fact matched during inference;

Q4: *can_sell(john,pen5)* - can be interpreted as the question, "Can John sell Pen5?". Unlikely other queries, this query require *yes* or *no* answer as the result of inference;

Q5: *can_sell(U,V)* - means that "Who can sell what?". All the facts which match this query are expected to be active during inference.

The results:

```
(Q1) query posed: give(jane,tom,U)
  . phase allocation - jane[1],tom[4],U[7]
  . initial bindings - give:{pa[*],arg1([0][1]),arg2([0][4]),arg3([7][0])}
  . inference result - give_f2:{fa[*],arg1([0][1]),arg2([0][4]),arg3([7][7])}
                       give_f2:computer2_gate[7]
    (symbolic meaning) by right elements - f2: give(jane,tom,computer2)
                       by left elements - f2: give(0,0,U)
                       unification results - {computer2/U}
  . inference cycles taken - 3 cycles

(Q2) query posed: own(mary,U)
  . phase allocation - mary[1],U[6]
  . initial bindings - own:{pa[*],arg1([0][1]),arg2([6][0])}
  . inference result1 - own_f1:{fa[*],arg1([0][1]),arg2([6][6])}
                        own_f1:ball4_gate[6]
    (symbolic meaning) by right elements - own(mary,ball4)
                       by left elements - own(0,U)
```

```
                        unification results - {ball4/U}
   . inference cycles taken - 3 cycles for own_f1
   . inference result2 - give_f1:{fa[*],arg1([0][0]),arg2([0][1]),
                                       arg3([6][6])}
                         give_f1:book1_gate[6]
     (symbolic meaning) by right elements - give(0,mary,book1)
                        by left elements - give(0,0,U)
                        unification results - {book1/U}
   . inference cycles taken - 4 cycles for give_f1

 (Q3) query posed: own(U,car3)
   . phase allocation - U[1],car3[6]
   . initial bindings - own:{pa[*],arg1([1][0]),arg2([0][6])}
   . inference result - buy_f1:{fa[*],arg1([1][1]),arg2([0][6])}
                        buy_f1:john_gate[1]
     (symbolic meaning) by right elements - buy(john,car3)
                        by left elements - buy(U,0)
                        unification results - {john/U}
   . inference cycles taken - 4 cycles

 (Q4) query posed: can_sell(john,pen5)
   . phase allocation - john[1],pen5[6]
   . initial bindings - can_sell:{pa[*],arg1([0][1]),arg2([0][6])}
   . inference result - none
     (symbolic meaning) no result (considered as 'no' answer)

 (Q5) query posed: can_sell(U,V)
   . phase allocation - U[1],V[6]
   . initial bindings - can_sell:{pa[*],arg1([1][0]),arg2([6][0])}
   . inference result1 - own_f1:{fa[*],arg1([1][1]),arg2([6][6])}
                         own_f1:mary_gate[1],own_f1:ball4_gate[6]
     (symbolic meaning) by right elements - own(mary,ball4)
                        by left elements - own(U,V)
                        unification results - {mary/U,ball4/V}
   . inference cycles taken - 4 cycles for own_f1
   . inference result2 - give_f1:{fa[*],arg1([0][0]),arg2([1][1]),
                                       arg3([6][6])}
                         give_f1:mary_gate[1],give_f1:book1_gate[6]
     (symbolic meaning) by right elements - give(0,mary,book1)
                        by left elements - give(0,U,V)
                        unification results - {mary/U,book1/V}
   . inference cycles taken - 5 cycles for give_f1
   . inference result3 - give_f2:{fa[*],arg1([0][0]),arg2([1][1]),
                                       arg3([6][6])}
                         give_f2:tom_gate[1],give_f2:computer2_gate[6]
     (symbolic meaning) by right elements - give(0,tom,computer2)
                        by left elements - give(0,U,V)
                        unification results - {tom/U,computer2/V}
   . inference cycles taken - 5 cycles for give_f2
   . inference result4 - buy_f1:{fa[*],arg1([1][1]),arg2([6][6])}
                         buy_f1:john_gate[1],buy_f1:car3_gate[6]
     (symbolic meaning) by right elements - buy(john,car3)
                        by left elements - buy(U,V)
                        unification results - {john/U,car3/V}
```

```
. inference cycles taken - 5 cycles for buy_f1
```

The given query (Q1) corresponding to the question "Who owns Car3?" produced the answer *computer2*. This answer was deduced from the unification result, {*computer2/U*}. When the second query was posed to find answers for "What does Mary own?", the answers were *ball4* and *book1* by the facts matched, *own(mary,ball4)* and *give(john, mary, book1)*. These answers were obtained by the unification results, {*ball4/U*} and {*book1/U*}. Note that a unification result produced by one encoded fact has to be interpreted independently from those obtained by other encoded facts so that each result is interpreted independently. Without this distinction, the two independent results, {*ball4/U*} and {*book1/U*}, will cause a consistency violation because one variable filler gets bound to two different constants at the same time. In a similar fashion, the result obtained from the query (Q3) is *john*. However, the query (Q4) did not generate any result because no fact was matched during inference. As for the final query (Q5), all facts encoded were matched and consequently produced sets of unification results from which the answers to the query can be obtained. To disambiguate each set of result bindings from others, CASI has to refer to the activation states of the constant gate nodes associated with each constant node of the fact SPN.

A Set of Facts Having the Same Predicate Name

Objectives:

When encoded, facts which share the same predicate name and arity are associated with the same base predicate assembly. According to a type of fillers involved, a posed query can activate more than one fact assemblies at the same time. CASI's fact encoding mechanism is designed to disambiguate one active fact from others in a connectionist manner. The purpose of this experiment is to confirm the behaviour of CASI in dealing with this situation.

The sample facts used:

f1: p(a,b,c)

f2: p(a,a,b)

f3: p(a,a,a)

The test queries used:

Q1: *p(a,a,b)* - to test consistency checking sub-mechanism within each UAG when the given fillers do not violate consistency;

Q2: *p(a,c,b)* - to test consistency checking sub-mechanism within each UAG when the given fillers violate consistency;

Q3: *p(a,U,a)*, Q4: *p(a,U,V)* - to test constant checking both within each UAG and across UAGs;

Q5: *p(U,U,V)*, Q6: *p(U,V,W)* - to test consistency checking across UAGs;

The results:

```
(Q1) query posed: p(a,b,c)
  . phase allocation - a[1],b[4],c[7]
  . initial bindings - p:{pa[*],arg1([0][1]),arg2([0][4]),arg3([0][7])}
    (symbolic meaning) - p(a,b,c)
  . inference result - p_f3:{fa[*],arg1([0][1]),arg2([0][4]),arg3([0][7])}
    (symbolic meaning) by right elements - f3: p(a,b,c)
  . inference cycles taken - 3 cycles

(Q2) query posed: p(a,c,b)
  . phase allocation - a[1],c[4],b[7]
  . initial bindings - p:{pa[*],arg1([0][1]),arg2([0][4]),arg3([0][7])}
    (symbolic meaning) - p(a,c,b)
  . inference result - none
    (symbolic meaning) - no result (considered as 'no' answer)

(Q3) query posed: p(a,U,a)
  . phase allocation - U[1],a[6]
  . initial bindings - p:{pa[*],arg1([0][6]),arg2([1][0]),arg3([0][6])}
    (symbolic meaning) - p(a,U,a)
  . inference result - p_f1:{fa[*],arg1([1][1,6]),arg2([1][1,6]),
                                    arg3([1][1,6])}
                       p_f1:a_gate[1,6]
    (symbolic meaning) by right elements - f1: p(a,a,a)
                       by left elements - f1: p(U,U,U)
                       unification results - f1: {a/U}
  . inference cycles taken - 4 cycles

(Q4) query posed: p(a,U,V)
  . phase allocation - a[1],U[4],V[7]
  . initial bindings - p:{pa[*],arg1([0][1]),arg2([4][0]),arg3([7][0])}
```

```
    (symbolic meaning) - p(a,U,V)
  . inference result1 - p_f3:{fa[*],arg1([0][1]),arg2([4][4]),arg3([7][7])}
                         p_f3:b[4]_gate,p_f3:c_gate[7]
    (symbolic meaning) by right elements - f3: p(a,b,c)
                       by left elements - f3: p(0,U,V)
                       unification results - f3: {b/U,c/V}
  . inference cycles taken - 3cycles for p_f3
  . inference result2 - p_f1:{fa[*],arg1([4,7][1,4,7]),arg2([4,7][1,4,7])
                                     arg3([4,7][1,4,7])}
                         p_f1:a_gate[1,4,7]
    (symbolic meaning) by right elements - f1: p(a,a,a)
                       by left elements - f1: p(U,U,U)
                       unification results - f1: {a/U,U/V}
  . inference cycles taken - 4 cycles for p_f1
  . inference result3 - p_f2:{fa[*],arg1([4][1,4]),arg2([4][1,4]),
                                     arg3([7][7])}
                         p_f2:a_gate[1,4],p_f2:b_gate[7]
    (symbolic meaning) by right elements - f2: p(a,a,b)
                       by left elements - f2: p(U,U,V)
                       unification results - f2: {a/U,b/V}
  . inference cycles taken - 4 cycles for p_f2

(Q5) query posed: p(U,U,V)
  . phase allocation - U[1],V[6]
  . initial bindings - p:{pa[*],arg1([1][0]),arg2([1][0]),arg3([6][0])}
    (symbolic meaning) - p(U,U,V)
  . inference result1 - p_f1:{fa[*],arg1([1,6][1,6]),arg2([1,6][1,6]),
                                     arg3([1,6][1,6])}
                       p_f1:a_gate[1,6]
    (symbolic meaning) by right elements - f1: p(a,a,a)
                       by left elements - f1: p(U,U,U)
                       unification results - f1: {a/U,U/V}
  . inference cycles taken - 4 cycles for p_f1
  . inference result2 - p_f2:{fa[*],arg1([1][1]),arg2([1][1]),arg3([6][6])}
                         p_f2:a_gate[1],p_f2:b_gate[6]
    (symbolic meaning) by right elements - f2: p(a,a,b)
                       by left elements - f2: p(U,U,V)
                       unification results - f2: {a/U,b/V}
  . inference cycles taken - 4 cycles for p_f2

(Q6) query posed: p(U,V,W)
  . phase allocation - U[1],V[4],W[7],
  . initial bindings - p:{pa[*],arg1([1][0]),arg2([4][0]),arg3([7][0])}
    (symbolic meaning) - p(U,V,W)
  . inference result1 - p_f3:{fa[*],arg1([1][1]),arg2([4][4]),arg3([7][7])}
                         p_f3:a_gate[1],p_f3:b_gate[4],p_f3:c_gate[7]
    (symbolic meaning) by right elements - f3: p(a,b,c)
                       by left elements - f3: p(U,V,W)
                       unification results - f3: {a/U,b/V,c/W}
  . inference cycles taken - 3cycles for p_f3
  . inference result2 - p_f1:{fa[*],arg1([1,4,7][1,4,7]),arg2([1,4,7][1,4,7])
                                     arg3([1,4,7][1,4,7])}
                       p_f1:a_gate[1,4,7]
    (symbolic meaning) by right elements - f1: p(a,a,a)
```

```
                        by left elements - f1: p(U,U,U)
                        unification results - f1: {a/U,U/V,V/W}
    . inference cycles taken - 4 cycles for p_f1
    . inference result3 - p_f2:{fa[*],arg1([1,4][1,4]),arg2([1,4][1,4]),
                                     arg3([7][7])}
                         p_f2:a_gate[1,4],p_f2:b_gate[7]
      (symbolic meaning) by right elements - f2: p(a,a,b)
                        by left elements - f2: p(U,U,W)
                        unification results - f2: {a/U,U/V,b/W}
    . inference cycles taken - 4 cycles for p_f2
```

One of encoded facts became active when the first query was posed but none when the second query was posed. This is because the constant fillers of the first query are the same as the constant argument names of the first fact matched and did not violate any constant consistency condition forced by the CCM_f, whereas the second filler of the second query (b) did not have any matched fact. The queries (Q3) and (Q4) also produced the desirable result. Only one fact was matched when the query (Q3) is posed but three facts when the query (Q4) is posed. Since these queries involved variable fillers, unification results involving these variable bindings were represented on the fact assemblies as the result of inference. The last queries, (Q5) and (Q6), contain only variable fillers and inferences involving those queries are expected to activate any fact assemblies representing the matched facts during inference with the representation of the unification result on their argument nodes. Two facts were matched for the query (Q5) and three facts for the query (Q6). As can be seen in this experiment, CASI's mechanism disambiguates a result obtained from one matched fact from those obtained from others using the active phases of an argument node, constant nodes, and their associated gate nodes.

7.3 Computational Analysis of CASI

7.3.1 Time Complexity

Each SPN encoding a rule consists of the source predicate assembly and target predicate assembly and the intermediate mechanism between them. The intermediate mechanism is composed of several layers of nodes. A number of these layers in each intermediate mechanism depends on a type of rule to be encoded. It takes one oscillation cycle to propagate activation from one layer to all the nodes on adjacent layer. In order to achieve a step of inference, the activation of the source predicate assembly has to go through all the layers of the intermediate mechanism to reach the target predicate assembly. Therefore, a time taken for a step of inference on an SPN can be represented with reference to the number of oscillation cycles used for that inference. If l_r is the number of layers of an intermediate mechanism, which we will call a *length* of a rule SPN, the value of l_r – in other words, the number of oscillation cycles needed to achieve a step of inference – varies from one type of rules to others as follows:

```
. rule type1   : p(X,Y) --> q(X,Y)
  cycles needed: 1 oscillation cycle (lr = 1)
                   1 for binding propagation

. rule type2   : p(a) --> q(a)
  cycles needed: 2 oscillation cycles (lr = 2)
                   1 for constant consistency checking for the constant
                     arguments
                   1 for binding propagation

. rule type3   : p(X,X) --> q(a,X)
  cycles needed: 3 oscillation cycles (lr = 3)
                   1 for binding collection within repeated variable UAG
                   1 for consistency checking for the repeated
                     arguments
                   1 for binding propagation

. rule type4   : p(a,X) --> q(a,X)
  cycles needed: 4 oscillation cycles (lr = 4)
                   2 for binding interaction across UAGs
                   1 for consistency checking after binding interaction
                     across UAGs
                   1 for binding propagation

. rule type5   : p(a,X,X) --> q(a,X)
  cycles needed: 5 oscillation cycles (lr = 5)
                   1 for binding collection within repeated variable UAG
                   2 for binding interaction across UAGs
                   1 for consistency checking after binding interaction
                     across UAGs
                   1 for binding propagation
```

The simplest rule (type1) only requires one oscillation cycle for a step of inference, whereas a rule which requires binding collection within UAG and binding interaction across UAGs needs 5 oscillation cycles.

If one SPN is connected with other SPNs by sharing a common predicate assembly, the inference result obtained from the first SPN is used for the next step of inference over the connected SPNs directly. In this case initiating an inference leads to a chain of inference over all SPNs serially connected each other. The more SPNs are connected serially, the longer time is required to spread the activation over the SPNs. In the worst case, if all rules to be encoded have serial relationships each other such as $p_1() \rightarrow p_2()$, $p_2() \rightarrow p_3()$, $p_3() \rightarrow p_4()$, ..., their corresponding SPNs will form a network which looks like a serial inferential chain. If n rules are encoded, a diameter of all connected SPNs $d = \sum_{i=1}^{n} l_{r_i}$, where l_{r_i} is the maximum length of a rule SPN (when all rules are type5). In the best case, if all SPNs are separated, the diameter of a SPN d is equal to l_r, where l_r is the minimum length of a rule SPN (when all rules are type1) .

The following two sub-sections describe the time complexities measured by the number of oscillations cycles needed for each type of inference.

Forward Chaining

In the best case, forward chaining takes 1 oscillation cycle. This is the situation where CASI's CLDB encodes only a set of simple rules (type1 as can be seen above) which do not share any common predicates among them. The time taken for a chain of inference in forward chaining is d where $d = l_r$, i.e. depending on the number of oscillation cycles needed for an individual SPN. In case of the basic rule (type1 rule), l_r is equal to 1. Thus total time taken for inference in the best case is 1 oscillation cycle.

In the worst case, it takes $5n$ inference cycles to complete a chain of inference if n rules are encoded. This is the case where all input rules are serially related each other and all rules have more than one repeated argument UAGs where l_{r_i} becomes the maximum length of a rule. This is the case when all n rules are type5. Since $d = \sum_{i=1}^{n} l_{r_i}$ and $l_{r_i} = 5$, d is $5n$. This shows that forward chaining, in the worst case, requires $5n$ oscillation cycles for a step of inference.

Another factor that affects the time complexity is the number of phases in each oscillation cycle. The larger the number of phases CASI adopts to represent the constant and variable entities, the longer it takes for a chain of inference. This makes the time taken for a chain of inference also to be linearly proportional to the number of phases in each oscillation cycle.

If the number of phases is fixed, the time complexity of CASI is therefore $\mathbf{O}(n)$, where n is the number of rules encoded. This indicates that, in forward chaining style of inference, the time taken for a chain of inference is linearly proportional to the number of rules to be encoded. Therefore, the performance of CASI's reasoning mechanism does not dramatically decrease as the number of rules to be encoded increases.

Backward Chaining

If a set of rules and facts are given to be encoded to support backward chaining, they are translated into a set of rule SPNs and fact SPNs. When rules are encoded, a rule SPN is either connected to or separated from others according to its dependencies with them. In a similar way, a fact SPN is either separated from a rule SPN or associated with a rule SPN whose name and arity are the same as the base predicate assembly of the fact SPN. Let us consider the following example rule and facts:

$$p(X) \rightarrow q(X),$$
$$p(a),\ q(b),\ r(c).$$

When encoded, the fact SPN for *p(a)* will be associated with the predicate assembly representing *p(X)* and the fact SPN for *q(b)* the predicate assembly representing *q(X)*. The fact SPN for *r(c)*, however, will be built separately from the rule SPN because it does not have any shared predicate with the given rule.

When a query predicate is posed, CASI activates the base predicate assembly corresponding to the given query predicate to initiate backward chaining. The procedure to obtain the first answer to the given query is summarised as follows:

1. if no base predicate assembly which corresponds to the given query predicate is encoded in CLDB, stop the inference;
2. when the base predicate is activated, if it has any associated fact assemblies, propagate the initial bindings to those fact assemblies. The answer to the query is obtained from the activation states of fact assemblies became active;

3. if the activated base predicate does not have any associated fact assembly, propagate the initial bindings to the adjacent predicate assemblies connected through links. Then propagate bindings from those predicate assemblies directly to their associated fact assemblies, if any, to obtain the answer;

4. continue the step 3 until there are no more predicate assemblies to be activated.

To describe the time taken for a step of backward chaining, it is necessary how many oscillation cycles are required to activate fact assembly once its base predicate assembly is activated. Let l_f to be a length of a fact SPN which corresponds to the number of oscillation cycles needed to propagate activation from the base predicate assembly to the fact assembly, the value of l_f varies according to a type of a fact as follows:

```
. fact type1   : p(a,b,c)
  cycles needed: 3 oscillation cycles (lf = 3)
                   2 for consistency checking
                   1 for binding propagation to the fact assembly

. fact type2   : p(a,a,b)
  cycles needed: 4 oscillation cycles (lf = 4)
                   1 for binding collection within UAG
                   2 for consistency checking
                   1 for binding propagation to the fact assembly
```

In the best case, obtaining the first answer to the given query requires only 3 oscillation cycles. This refers to the situation where a fact which does not have repeated arguments (type1 fact) is encoded in CLDB and the given query predicate directly matches with this encoded fact. Posing the query predicate activates the base predicate assembly of its corresponding SPN and then, after l_r oscillation cycles, the fact asembly becomes active. Since the fact does not have repeated arguments (type 1), $l_r = 3$.

In the worst case, the first answer is obtained after $5n+4$ cycles, where n is the number of rules encoded. This occurs if n rules are serially related each other and a fact which has repeated argument (type 2) is associated with the antecedent predicate assembly of the first rule. In order to obtain the answer, the consequent predicate of the last rule has to be activated first and the initial bindings propagate through n rule SPNs to the antecedent predicate assembly of the first rule and then to the fact assemblies associated with it. It takes $5n$ (which is equal to $\sum_{i=1}^{n} l_{r_i}$, where $l_{r_i} = 5$) for the initial bindings to reach the antecedent predicate assembly of the first rule. Additional 4 inference cycles (which is equal to l_f of the encoded fact) are needed to propagate those bindings to the fact assembly associated with the antecedent predicate assembly of the first rule. The total time taken for a step of inference, in the worst case, is therefore $5n + 4$. This shows that the time complexity of backward chaining is also $\mathbf{O}(\mathrm{n})$, which is linearly proportional to the number of rules encoded.

7.3.2 Space Complexity

CASI uses a different number of neuron elements to build an SPN for each rule and fact. The factors that affects the number of neuron elements are as follows:

- the number of UAGs in the source and target predicate;
- the type of each UAG: a constant UAG or a variable UAG;
- the number of arguments in each UAG: single or repeated.

The number of neuron elements needed to build each rule and fact SPN is different according to their types. The following subsections analyse space complexities, in the worst case, in encoding rules and facts.

In Encoding Rules

Encoding each rule firstly requires two predicate assemblies to represent the source and target predicates. If both of them has k arguments, this requires 2 τ-and nodes for the predicate activators and $2k$ entity nodes ($4k$ π-btu elements) for their argument nodes. The intermediate mechanism to be built between them will consist of simple links if the given rule is of the basic type (which has only single variable arguments) or the number of sub-mechanisms otherwise. In the latter case, CASI decides a required number of sub-mechanisms according to a syntactic condition that the given rule enforces. The total number of neuron elements for the intermediate mechanism can be obtained by figuring out the number of required sub-mechanisms and the number of neuron elements needed to construct each sub-mechanism. The following is the number of neuron elements required for each sub-mechanism:

. the binding collection sub-mechanism (BCM):

2 π-btu elements (*b1*)

. the consistency checking sub-mechanism for a constant UAG (CCM_c):

1 τ-or (*to1*)

. the consistency checking sub-mechanism for a variable UAG (CCM_v):

1 multiphase τ-or (*mto1*)

. the binding interaction and consistency checking sub-mechanisms:

for $BIM\&CCM_{cv}$ and $BIM\&CCM_{vv}$

1 τ-or (*to2*),

2 π-btu elements (*b2*),

1 multiphase τ-or (*mto2*)

for BIM_{cc}

1 τ-or (*to2*)

. binding propagation sub-mechanism (BPM):

for a constant UAG

2 π-btu elements (*for a constant itself*),

1 τ-or (*for a constant ihb node*),

1 π-btu element (*for a constant gate node*)

for a variable UAG

none

Consider a rule whose source and target predicate consist of k arguments. If the k arguments of the source predicate are categorised into m distinct UAGs, the total number of neuron elements required will be at a maximum (the worst case) when all m UAGs are distinct repeated variable UAGs because the $BIM\&CCM_{vv}$ will require 4 neuron elements to be built between any two UAGs. The number of $BIM\&CCM_{vv}$s is calculated by the formula

$$\frac{m!}{i!(m-i)!}$$

which specifies the number of ways of combining i UAGs out of m UAGs (${}_mC_i$) without order. The number of ways selecting pairs of UAGs out of m UAGs is therefore

$$\frac{m!}{2(m-2)!} = \frac{m^2 - m}{2}$$

because $i = 2$. As can be seen, the number of $BIM\&CCM_{vv}$s increases quadratically as the number of UAGs of a rule increases. Figure 7.1 shows the relationship between number of $BIM\&CCM$s with reference to the number of UAGs of a rule.

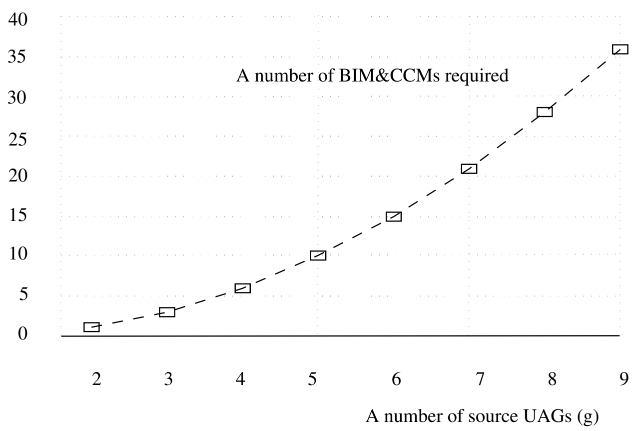


Figure 7.1: The relationship between the number of source UAGs and the number of $BIM\&CCM$s required

The total number of neuron elements required to encode one rule can be analysed as follows:

- 2 τ-or elements and $4k$ π-btu elements are needed for the predicate activators and argument nodes of the source and target predicate assemblies;
- $2m$ π-btu elements are required to build BCM for m UAGs;
- m multiphase τ-or elements are required for the CCM_v;
- no neuron element is needed for the BPM_v;
- $(m^2 - m)$ π-btu elements, $(m^2 - m)/2$ τ-or elements, and $(m^2 - m)/2$ multiphase τ-or elements are needed for the $(m^2 - m)/2$ $BIM\&CCM_{vv}$.

The sum of all of these neuron elements are $2m^2 + m + 4k + 2$. This demonstrates that the space complexity of CASI in encoding n rules is $\mathbf{O}(nm^2)$ and is quadratic in the number of UAGs in the source predicate of the rules.

In Encoding Facts

Encoding a fact requires less number of neuron elements because it does not require $BIM\&CCM$s between UAGs. When a fact is given to be encoded, the fact is represented using two predicate assemblies, one for based predicate assembly and the other fact assembly. Then, the intermediate mechanism has to be built between them. The number of neuron elements needed to build each sub-mechanism of the intermediate mechanism is as follows:

. the binding collection sub-mechanism (BCM_f):
  2 π-btu elements (*b3*)
. the consistency checking sub-mechanism (CCM_f):
  1 τ-or (*pb3*),
  1 π-btu element (*pb1*)
. the binding propagation sub-mechanism (BPM_f):
  for right elements of the argument nodes:
    2 π-btu elements (*for a constant itself*),
    1 τ-or (*for a constant ihb node*),
    1 π-btu element (*for a constant gate node*)
  for the left elements of the argument nodes
    none

For a given fact which has k arguments and m UAGs whose size is greater than 1, the total number of neuron elements needed to encode the fact is analysed as follows:

- *2* τ-and and *4k* π-btu elements (*2k* entity nodes) are necessary for the base and fact predicate assemblies;
- *2m* π-btu elements for the BCM_f;

- one τ-or and one π-btu elements are required for the CCM_f (there are share by all UAGs);
- $2m$ π-btu elements, m τ-or elements, and m π-btu element for the BPM_f.

Therefore the total number of neuron elements needed to encode a fact is $6m + 4k + 4$. Thus, the space complexity in encoding n facts, in the worst case, is $\mathbf{O}$(nm), which is linear to the number of UAGs in the source predicate of the facts.

In each rule and fact, the relationship between the number of arguments (k) and the number of UAGs (m) is $k \geq 2m$ if all the UAGs have repeated arguments because a repeated UAG has at lease 2 arguments. Since CASI needs one phase to represent entities in a distinct UAG dynamically, the number of UAGs involved in a chain of inference can not exceed the number of phases allowed. A more detailed discussion about this will be described in Section 8.2.1.

7.4 Comparison with Similar Systems

7.4.1 Comparison with SHRUTI

Since CASI's basic dynamic binding mechanism is inspired by Shastri & Ajjanagadde's temporal synchrony approach, it is interesting to compare their connectionist system, SHRUTI, with CASI.

The most important difference is found in a behaviour of a basic neuron elements. SHRUTI restricted their ρ-btu element to carrying only a single phase signal per oscillation cycle but CASI's π-btu element allows multiple phase signals per cycle and propagates these in synchrony with the driving inputs. Thus, a π-btu element is a generalised ρ-btu element.

The next difference lies in the structure of the entity node they adopt. SHRUTI uses a single ρ-btu element to represent an entity but CASI uses a pair of π-btu elements. Consequently, SHRUTI's dynamic binding mechanism represents only constant bindings explicitly on ρ-btu elements and leaves variable bindings unspecified – by tagging all of them inactive states throughout the entire oscillation cycle. SHRUTI does not therefore differentiate one variable binding from others. Its ability to replicate symbolic style of inference is fundamentally limited in this respect because this is a basic, and fundamentally important, feature of symbolic inference. CASI, on the other hand, is capable of representing both variable and constant bindings on entity nodes using temporal synchronous activity between them.

Thirdly, SHRUTI's rule and fact encoding mechanisms do not consider all the syntactic conditions that symbolic rules and facts may impose when encoding them. In particular, it cannot provide any information with which to build sub-mechanisms for carrying out binding interaction and consistency checking across UAGs. This also limits SHRUTI's unification ability only within each UAG. However, CASI can represent groups of unifying arguments and can perform unification across them with consistency checking.

Fourthly, SHRUTI's backward chaining differentiates wy-type query processing from yes-no type. In case of wh-type query processing, SHRUTI requires two separate stages to obtain the desirable result. This requires SHRUTI to activate the source predicate assembly twice for wh-type query processing (see Section 3.3.3): once to activate all fact nodes[1] which will be matched with the query predicate during the first stage; and again to active constant nodes which are to be matched with variable fillers of the query predicate during the second stage. Whereas CASI's fact encoding mechanism does not differentiate wh-type query from yes-no query and, thus, does not require different network configurations to support them. In CASI, these two query procedures are merged into the common fact encoding mechanism (this eliminates the need of latch nodes used by SHRUTI). When a wh-type query is presented, CASI's mechanism automatically propagates initial activation of the variable filler to its corresponding argument node of a fact assembly matched during inference. Consequently, the result of unification between the variable filler and constant nodes will be represented on the fact assemblies matched without an additional stage of query processing, but they must be collected from their distributed locations in the network.

Fifthly, time taken for forward chaining in SHRUTI is proportion to d, where d is the

[1] In fact, it is the *latch* node associated with each fact which maintains activation even after the first stage of query processing. This activation persists until the second stage of wh-type query is finished.

maximum diameter of a network. If n rules are serially represented in the network, $d = \sum_{i=1}^{n} l_{r_i}$, where l_{r_i} is a length of each rule involved. In the worst case, $l_r = 3$ in SHRUTI, therefore, the time taken for a step of inference involving n rules is $3n$. This is slightly faster than that of CASI's (which is $5n$). In backward chaining, to process a yes-no type query involving n rules, SHRUTI needs approximately $3n + 1$ because $d = \sum_{i=1}^{n} l_{r_i} + l_f$ and $l_{r_i} = 3$ and $l_f = 1$ (see Section 4 of Shastri & Ajjanagadde (1993)). Since wh-type of query processing requires two phase processing the time taken for a chain of inference will be approximately double the previous case, i.e $6n+2$. Whereas CASI needs $5n+4$ oscillation cycles regardless of a type of queries for backward chaining.

A comparison in terms of the number of neuron elements needed is that CASI generally requires a larger number of elements in encoding rules and facts than SHRUTI. In order to encode a simple rule (type1), CASI needs the double the number of neuron elements than SHRUTI because CASI's entity node consists of a pair of π-btu elements whereas SHRUTI's entity nodes are composed of a single ρ-btu element. However, it is difficult to compare the number of neuron elements needed to encode rules which require a special treatment during inference such as unification across different UAGs because SHRUTI does not provide any mechanism to encode such rules. The additional neuron elements required for CASI to encode this type of rule are

- one entity node to build BCM for a variable UAG which has repeated arguments;
- one τ-or element to build CCM_c for a constant UAG;
- one τ-or element, one entity node, one multiphase τ-or element to build $BIM\&CCM$ across UAGs.

Finally, SHRUTI provides extensions to support multiple instantiation of predicates and a connectionist representation of an IS-A hierarchy which can cooperate with SHRUTI's reasoner. CASI cannot currently deal with multiple instantiation of predicates. However, Entity categorisation using an IS-A hierarchy and defeasible rules described in SHRUTI can be supported in CASI without requiring special purpose additional mechanisms. In this case, CASI translates defeasible rules and ISA hierarchy definitions into the form of ordinary rules and facts and encodes them using the standard translation algorithm. This approach is in substantial agreement with that of Ajjanagadde (1994).

7.4.2 Comparison with CONSYDERR

Sun's CONSYDERR, consists of two sub-systems one of which uses distributed representation, the other localist representation. The localist sub-system can deal with various knowledge representation issues such as consistency checking, unification, etc.. The major difference between the localist sub-system of CONSYDERR and CASI lies in the functional requirements of nodes in a network. CASI uses only one type of predicate assembly which consists of a τ-and element and entity nodes. The behaviour of the predicate assembly is quite transparent because, apart from temporal behaviour, each element of entity nodes performs very simple processing (summing of weighted inputs and thresholding over temporal phases). CONSYDERR, however, uses three different types of assemblies. Ordinary assemblies compute weighted sums of inputs and pass along bindings for the variables; OR assemblies compute maximum of inputs and select one set of bindings out of many based on who is the winner; and Complex assemblies perform consistency checking, unification, and also binding generation when necessary (see p. 106 of Sun(1992)). Although it does not use temporal phases, the need for high computational ability of the assemblies and the use of abstraction which hides network details is beyond simple summation with thresholding and so demands much more sophisticated neural elements than CASI requires.

Another difference is that CASI's dynamic variable binding mechanism uses the temporal synchrony approach, thus the number of distinct entities allowed in each inference is constrained by the number of phases in one oscillation cycle. However, CONSYDERR uses numeric values to tag different entities. Therefore it can accommodate larger number of distinct entities in each inference. Moreover, CONSYDERR's inference can deal with a measure of degree of belief. The antecedent predicate assembly of a network can be instantiated with a certain confidence value and the output obtained from the consequent predicate can also represent the degree of confidence in a numeric form. In CASI, on the other hand, argument nodes are activated as binary values (1 or 0) and a result of inference is also represented by those values.

Finally, CASI can carry out both types of inference: forward chaining and backward chaining but CONSYDERR's architecture only supports forward chaining.

7.4.3 Comparison with ROBIN

Another similar approach to CASI can be found in ROBIN proposed by Lange & Dyer (1989a, 1989b). The first difference is that ROBIN's network is based on semantic network in which each node representing a concept is connected to others with links representing special relationships between them but CASI's network is based on SPNs which are constructed by direct compilation of rules and facts.

Dynamic binding mechanisms in both systems share many similarities. Both systems represent a dynamic binding between a variable and a concept by making nodes representing them share the same activation value or pattern. ROBIN activates the binding node associated with a value node and the signature associated with a concept node in the same activation pattern (called a signature) and CASI activates the variable and concept nodes in the same phase to establish the dynamic binding between the variable node and the concept node. The difference is that ROBIN can maintain large number of distinct dynamic bindings using numerical signatures. The number of distinct bindings in CASI is constrained by the number of phases allowed.

ROBIN can represent a degree of confidence by different activation values but it cannot deal with key knowledge representation issues such as consistency checking and unification. Only forward chaining is supported in ROBIN.

7.5 Summary

This chapter first explored CASI's ability, as a connectionist compiler, with the series of experiments. In each experiment, an example rule was translated into the corresponding SPN and various types of predicates are presented to test how well CASI replicates symbolic styles of inferences over the networks. The results of experiments showed that the SPNs produced by CASI's knowledge encoding mechanism successfully replicated various types of symbolic inferences. CASI was able to represent symbolic components of rules and facts in a connectionist manner. It's dynamic binding mechanism was able to represent both constant bindings and variable bindings. An intermediate mechanism of each SPN performed necessary subtasks to achieve symbolic inference such as consistency checking, unification and binding propagation. However, the experiments show that CASI has a problem in representing a result of inference on the consequent predicate assembly when a rule which requires binding interaction across more than two UAGs during inference is used for a step of inference.

A time taken for a chain of inference in CASI is proportional to the number of rules encoded. The number of neuron elements required to encode a rule is exponential to the number of UAGs in the antecedent of the rule. When encoding a fact, the number of neuron elements needed is proportional to the number of UAGs of the fact.

In comparison to other similar connectionist inference systems, CASI's inference mechanism provides more expressive power than SHRUTI and ROBIN by providing connectionist sub-mechanisms to handle the various knowledge representation issues. CASI's SPNs have more transparent network architecture than CONSYDERR and requires neuron elements with simple computational ability.

Chapter 8

Conclusions and Future Directions

8.1 Summary

Any connectionist model which aspires to replicate symbolic inference based on first order Horn clause expressions should be able to represent the following in a connectionist manner:

- representation of symbolic knowledge components;
- representation of dynamic bindings involving constant and variable bindings;
- representation of strategies to be used for drawing inferences;

This thesis proposes a connectionist architecture for symbolic inference (CASI) which provides connectionist mechanisms for all these requirements to achieve a standard symbolic style of inference.

8.1.1 Representing Knowledge Components

Representing symbolic knowledge components involves representing symbols and structured knowledge such as n-ary predicates.

To represent symbols in a connectionist manner, CASI uses entity nodes. An entity node consists of a pair of π-btu elements and is used as a variable node, a constant node, or a binding node. A symbolic constant or variable is represented using this entity node. Because of the temporal oscillatory property of a π-btu element, when an entity node is used as a binding node it can represent several variable and constant bindings in one oscillation cycle at the same time, which allows limited multiple instantiations. In biological systems, an entity node does not need to be a single neuron cell – it might find a counterpart in an ensemble of neurons or a small part of the brain – but biological plausibility is not the main theme of this thesis.

The n-ary predicates are represented using an assembly of nodes, called a predicate assembly, which is composed of one τ-and element and n entity nodes. The τ-and nodes is used to indicate the activation state of the predicate and entity nodes n arguments of the predicate. The same structure of predicate assemblies is used to represent all the predicates appearing in input rules and facts. The use of entity nodes as the arguments of a predicate assembly allows CASI easily represents the results of inference and unification on each argument node.

8.1.2 Representing Dynamic Bindings

Based on the structured representation of symbols and predicates, CASI represents variable bindings and constant bindings using synchronous activity among entity nodes. A variable binding is represented by activating a variable filler node in phase with the left element of a binding node, a constant binding by activating a constant filler node in phase with the right element of the binding node. Multiple instantiations on a binding node allow several variable bindings and constant bindings involving the same argument at the same time. This property of an entity node efficiently represents the result of inference and unification. The number of bindings involved in dynamic bindings during a chain of inference is only bounded by the number of phases in each oscillation cycle not by the number of entities involved.

8.1.3 Representing Inference Strategies

One of difficulties in achieving symbolic inference in a connectionist manner is how to achieve some strategies needed to draw inferences. CASI provides connectionist equivalents to two basic symbolic inference procedures: matching and substitution.

The match procedure has been implemented by the initial binding instantiation procedure and a consistency checking sub-mechanism of an intermediate mechanism. Since the initial binding instantiation procedure simply sets up dynamic bindings between a presented predicate and the corresponding base predicate assembly of an SPN, this sub-mechanism performs necessary consistency checking. The substitution procedure, on the other hand, is implemented by a binding propagation sub-mechanism which is embedded into the intermediate mechanism. The effect of symbolic substitution is achieved by propagating the initial bindings from the source predicate assembly to the target predicate assembly through this sub-mechanism.

Since different types of rules and facts require different structure for their intermediate mechanisms, the difficulties in building intermediate mechanisms lie in how to decide a required set of sub-mechanisms according to various types of rules and facts. CASI's knowledge compiler provides a set of algorithms for this task. On receiving input rules and facts, the knowledge compiler examines each rule and fact to figure out types of arguments in the source predicate and argument matching between the source and the target predicates. Through this procedure, the knowledge compiler builds an intermediate mechanism which performs two standard symbolic inference procedures.

Once SPNs are constructed from the input rules and facts, inference in CASI is automatic. When a predicate is presented to a network, CASI starts inference by setting up initial bindings between fillers appearing in the presented predicate and arguments nodes of its base predicate assembly. Since the initial variable instantiation procedure does not involve any form of consistency checking, the intermediate mechanism does this and provides a binding propagation path from the source predicate assembly to the target predicate assembly. The result of inference is obtained by observing the filler nodes and predicate assemblies activated during inference.

8.1.4 Application of CASI

One possible application of CASI will be translating symbolic specifications to a sub-symbolic level. Although specifications expressed in a symbolic style are usually more convenient to read and manipulate, some domains may demand specifications at a sub-symbolic level (for faster run times or to enable inference using simpler components).

This creates an extra layer in the software lifecycle, with the symbolic specification acting as an intermediary between high-level requirements and sub-symbolic specification. CASI, proposed in this thesis, could be used as an automatic translation mechanism from a symbolic specification language (a subset of Horn clause logic) to a sub-symbolic language (For more details see [Robertson et al (1994)]). This could allow users to work at the symbolic level and have their descriptions *compile* to the sub-symbolic level.

Secondly, CASI may be used as a connectionist reasoner for a knowledge-based system adopting a production system architecture. When rules and facts describing knowledge of a specific domain are given to CASI, it encodes the input rules and facts into CLDB in the form of SPNs [Park & Robertson (1996b), Park et al. (1994), Park et al. (1996)]. Thus, CASI's CLDB corresponds to the rule memory (long-term memory) of the production system. During inference CASI maintains activation states (oscillation states) between entities and argument nodes of predicate assemblies participating a chain of inference. Therefore, these activation states over SPNs during inference corresponds to the working memory (short-term memory) of a production system. Finally, CASI implements the inference engine part of the production system into the structure of SPNs as an embedded connectionist mechanism. An initialisation of a certain predicate assembly of SPNs automatically produces a result of inference by propagation the initial activation throughout the network.

Finally, CASI may be used as a part of cognitive model to explore various aspects of human cognitive processes. This possibility will be discussed in Section 8.3.2.

8.1.5 Concluding Remarks

CASI translates a significant subset of Horn clause into a connectionist representation, called structured predicate networks (SPNs), which may be executed very efficiently (not only in forward chaining but also in a backward chaining style). The major contributions of CASI is summarised as follows:

- it provides a connectionist representation of symbolic components in first-order calculus;

- it extends the temporal synchrony solution to the dynamic binding problem to represent both variable and constant bindings efficiently;
- it introduces UAG concept to analyse each type of symbolic rules and facts;
- it provides the connectionist equivalents to the two basic symbolic inference procedures, matching and substitution.
- it provides a set of algorithms which automatically decides necessary sub-mechanisms to build an intermediate mechanism when encoding a rule or a fact;

CASI is a step in achieving symbolic inference in a connectionist manner. It's knowledge encoding mechanism provides a way of representing not only a group of unifying arguments but also many groups of such arguments involving consistency checking by various sub-mechanisms adopted [Park & Robertson (1995), Park & Robertson (196b), Park et al. (1996)]. This extends the expressive power of a connectionist inference system. However, in order to have the full expressive power of first-order Horn clause expressions, CASI will have to be extended to tackle some of its limitations described in the next section. Currently, no connectionist system has provided convincing solutions to these constraints and 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.

8.2 Limitations

As a compiler for first-order Horn clause expressions CASI has the following limitations.

8.2.1 Fundamental Limitations

The most clear constraint of CASI is the limitation of the number of distinct entities allowed to participate in dynamic bindings simultaneously. This was one of constraints that the original temporal synchrony approach has. The temporal synchrony approach requires each entity to occupy one distinct phase in a period of oscillation to avoid cross-talk. Thus the maximum number of different entities allowed at a time cannot exceed the number of phases. Since CASI's dynamic binding mechanism adopts this approach, this fundamental limitation remains. As described in Section 3.5.1, Shastri & Ajjanagadde suggested an appropriate number of phases on the basis of biological and psychological measurements. According to their estimation, the number of entities referenced by the dynamic bindings is five or less. It is interesting that this estimation coincides a commonly accepted measure of short-term memory capacity, 7 ± 2 [Miller (1956)] and this merits further investigation.

8.2.2 Unification Across More Than Two UAGs

During the experiments, CASI shows one limitation in processing unification across more than two UAGs. This situation was detected when the predicate, *p(U, V, W, U, V)*, is presented to the SPN encoding the following rule:

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

To decide the necessary sub-mechanisms to build the intermediate mechanism, two sets of UAGs for the given rule have to be obtained as follows:

$$S_s = \{G_a, G_X, G_Y\}$$
where $G_a = \{p{:}arg1\}$, $G_X = \{p{:}arg2, p{:}arg3\}$, $G_Y = \{p{:}arg4, p{:}arg5\}$,
$$S_t = \{G_a, G_X, G_Y\}$$
where $G_a = \{q{:}arg1\}$, $G_X = \{q{:}arg2\}$, $G_Y = \{p{:}arg3\}$.

The required sub-mechanisms of the intermediate mechanism are

$$BCM = \{G_X, G_Y\},$$
$$CCM_c = \{G_a\}, CCM_v = \{G_X, G_Y\},$$
$$BIM\&CCM_{cv} = \{(G_a, G_X), (G_a, G_Y)\},$$
$$BIM\&CCM_{vv} = \{(G_X, G_Y)\},$$
$$BIM_{cc} = \{\},$$
$$BPM = \{G_a, G_X, G_Y\}.$$

Once the SPN is built based on this analysis, a step of inference over the SPN will be initiated by presenting the predicate to CASI, which sets up the initial bindings between the fillers and arguments of the source predicate assembly. Presenting the predicate, *p(U,V,W,U,V)*, sets up the initial bindings on the source predicate assembly of the SPN as follows:

$$p\text{:}\{pa[*],arg1([4],[0]),arg2([7],[0]),arg3([1],[0]),arg4([4],[0]),arg5([7],[0])\},$$

where phase allocation to the fillers is *W[1], U[4], V[7]*. This corresponds to the same effect that the variable filler U is assigned to the G_a, and two variable filler V and W to the G_X, and the rest two fillers U and V to G_Y. Consequently, the pair of UAGs, (G_a,G_Y), share the variable filler, U, and the pair of UAGs, (G_X,G_Y), the variable filler, V.

Since the given rule has three $BIM\&CCM$s between the following pairs of UAGs:

1. the $BIM\&CCM_{cv}$ between G_a and G_X,
2. the $BIM\&CCM_{cv}$ between G_a and G_Y,
3. the $BIM\&CCM_{vv}$ between G_X and G_Y,

presenting the predicate therefore leads activation of the second $BIM\&CCM_{cv}$ and the third $BIM\&CCM_{vv}$ during inference. The first $BIM\&CCM_{cv}$ remains inactive because two UAGs, G_a and G_X, do not share the same variable filler. When the initial bindings are propagated through these mechanisms, they become active in parallel and generate the following intermediate bindings

- by the second $BIM\&CCM_{cv}$, the intermediate bindings, {a/U,U/V}, are generated and propagated to the first and third argument nodes of the q predicate assembly;
- by the third $BIM\&CCM_{vv}$, the intermediate bindings, {V/W,U/V}, are produced and propagated to the second and third argument nodes of the q predicate.

When binding propagation is finished, the argument nodes of the q predicate assembly become active in

q:{pa[*],arg1([4,7],[4]),arg2([1,4,7],[0]),arg3([1,4,7],[4])},
p_q:a_gate[4].

The activation states of the right elements of the argument nodes represent the result of inference, *q(a,0,a)*, and that of the left elements of the argument nodes, *q(U,U,U)*. From these, the unification result, $\{a/U, U/V, V/W\}$, is obtained.

This result shows that the right element of the second argument node does not properly represent the desirable constant binding, $\{a/q{:}arg_2\}$. Although the desirable result can be inferred from the unification result, the right element of the second argument node did not become active during inference. This is because CASI's SPNs are designed to carry out only single-phase unification between each pair of UAGs, it cannot therefore deal with unification which requires more than one-phase computation. This situation occurs when a UAG is unifiable with another UAG indirectly through other UAGs. In the case of the example, when the predicate, *p(U,V,W,U,V)*, is presented, the UAG, G_a, is unifiable with the UAG, G_X, indirectly through the UAG G_Y. The unification between the two UAGs, G_a and G_X, must be done after unification between the two UAGs, G_X and G_Y to produce desirable result. Thus, this requires one more phase to complete all the unification procedure involved. This additional unification step can be achieved by inserting another set of $BIM\&CCM$s between the first layer of $BIM\&CCM$s and the argument nodes of the q predicate assembly. When the current outputs of the inference go through this additional layer of $BIM\&CCM$s, the desirable result will be produced, which will make the right element of the second argument node in-phase with the constant filler a as follows:

q:{pa[*],arg1([1,4,7][4]),arg2([1,4,7][4]),arg3([1,4,7][4])},
p_q:a_gate[4].

In general, if there are m UAGs in the antecedent of a rule or a fact, this type of inference architecture requires $m-1$ layers of $BIM\&CCM$s to perform full unification among all the UAGs.

8.2.3 Multiple Instantiations of Predicates

The ability to handle multiple instantiations of the same predicate allows the system to deal with more complex inferential dependencies including symmetric and transitive inferences and bounded recursion. Although CASI allows multiple instantiations of an entity node consisting of a pair of π-btu elements, it does not deal with multiple instantiations of predicate assemblies directly. This means that the mechanism cannot reason with the rules such as $sibling(x, y) \rightarrow sibling(y, x)$, $less\_than(x, y) \wedge less\_than(y, z) \rightarrow less\_than(x, z)$ because these rules force the same predicate to become active more than once with different sets of dynamic bindings during a step of inference. In CASI, all predicates which share the same predicate name and arity are represented using the same predicate assembly.

One solution to multiple instantiations of predicates based on temporal synchrony was introduced in Shastri & Ajjanagadde (1993). In their solution, they limited the number of predicates participating in multiple instantiations during reflexive reasoning to be 3 because of additional time and space costs associated with it. However, we argue that this estimation due to the necessity of keeping these resources within bounds in the context of reflexive reasoning may not be reasonable. In the following two paragraphs, for example, it is difficult to answer who likes whom after reading the first paragraph without repetition but it is not the case in the second paragraph. Both paragraphs have four multiple instantiations and there is not much difference in the number of distinct entities involved in both paragraphs. The first paragraph has six distinct entities and the second paragraph five distinct entities.

Paragraph1: *Susan likes Tom. John likes Lisa. Tom likes Mary. Clara likes Tom.*
Paragraph2: *Susan likes Tom. Susan likes Lisa. Susan likes Mary. Susan likes Clara.*

This example shows that the sequence of information plays an important role in human cognition [Stenning et al (1988), Stenning & Levy (1988)]. If the number of multiple instantiations that human can handle simultaneously is bounded mainly by the cost of time and space, it should be equally difficult to understand both the above paragraphs but this is not the case. This may imply that the human limitation in dealing with multiple instantiations is bounded by the measure of short-term memory capacity and other factors such as an active role of the sequence of information rather than the cost of time and space. It may be also possible that there is a mechanism in the human brain which is capable of handling of multiple instantiations in a more efficient way. Following this direction requires CASI to be extended to cooperate with yet another connectionist modules which control the sequence of information during inference or to adopt higher order predicate logic.

8.2.4 Further Limitations

In addition to the limitations discussed so far, there remains a couple of knowledge representation issues which should be tackled further to improve CASI's expressive power as a connectionist inference system. These include:

- allowing negated predicates in rules and facts;
- allowing recursive definitions in rules.

Extending CASI's knowledge encoding mechanism to handle negated predicates will enable CASI represents logical expressions like

$$\text{John is not a student: } \neg \textit{student(john)};$$
$$\text{Jacob is not father of Abraham: } \neg \textit{father(jacob,abraham)};$$
$$\text{No member is allowed to enter: } \neg member(X,Y) \rightarrow \neg enter(X,Y).$$

This would enhance CASI's expressive power a step further to approach the expressive power of first-order Horn clause expressions. One way of achieving this would be adopting a solution proposed by Shastri & Grannes (1995) to allow negated predicates in SHRUTI. They modified the structure of predicate assemblies by introducing two collectors: one indicates positive assertion of a predicate and the other negative assertion of a predicate. They also proposed how contradictions and inconsistencies are dealt with in a connectionist manner.

Encoding recursively defined rules is difficult to achieve in a connectionist manner because each step of recursion of a rule has to be represented by a finite network and there is no information about how many steps are needed to complete a recursive inference in advance. However, a limited number of recursion my be allowed if we can provide connectionist mechanisms which perform the following sub-tasks:

- multiple instantiations of predicate assemblies;
- a mechanism to control a depth of recursions to avoid an indefinite loop;
- a mechanism to store and retrieve intermediate results.

No convincing connectionist mechanism has yet been proposed to carry out these tasks.

8.3 Future Directions

Apart from some extension needed to overcome the limitations of CASI described in the previous subsection, future work related to CASI involves the following: dealing with degrees of confidence, exploring CASI as a part of a cognitive model, and finding more efficient implementation of CASI.

8.3.1 Dealing With Degrees of Confidence

Since CASI cannot represent degrees of confidence on SPNs during inference, some work may be needed to extend CASI in this direction. One way of allowing degrees of confidence into CASI's inference mechanism would be to use different confidence activation values rather than using binary values when instantiating the source predicate assembly. This may require additional connectionist mechanisms which carry out merging confidence values according to "and" or "or" relationship between rules involved. Adjusting weights of links may be considered as another way of representing degrees of confidence on SPNs or combination of both.

Another way of representing degrees of confidence may be using the degrees of oscillation between nodes involved in a binding. When two entity nodes become active in the same phase to set up a binding, the degree of tightness in phase locking between them may be considered as a degree of confidence to support that binding. The more closely related nodes oscillate in tighter phase locking and the more remotely related nodes oscillate in looser phase locking. This is possible since the degree of synchronisation between node depends on the strength of connections between them [Baddeley (1992), Cairns et al (1993)]. However, building such a network model would require an additional connectionist mechanism supported by well established mathematical analysis.

8.3.2 Exploration as A Cognitive Model

Until now, CASI has been explored mainly as a compiler of first-order Horn clause expressions. However, CASI also proposes many interesting behaviours as a part of models for human cognitive processes: such as how language in the form of rules and facts may be understood in a connectionist manner; how much working memory space is needed to perform a chain of inference involving specific rules and facts, etc..

As a part of a connectionist language understanding model, CASI might be used as a reasoning module which performs inferences (in either a backward or forward chaining fashion) based on the knowledge it encodes. In this case, CASI may need to cooperate with other models such as

- a module which performs natural language preprocessing;
- a module which carries out focus maintenance;
- a module which sets up and controls overall inference plans.

Since CASI deals with first-order Horn clause expressions, a language preprocessing module would need to connect the natural language inputs to the input expressions that CASI takes. When a natural language style of inputs is given, this module has to parse the natural language input and provides inputs for CASI's inference instantiation procedure. These inputs may involve information about how entities appearing in a sentence can be translated into a series of predicates which consist of predicate names and entities as arguments.

When one predicate assembly is activated to initiate a chain of inference, the initial bindings are propagated to all the predicate assemblies connected to the initial predicate assembly. This means that CASI, as a reasoning module, only controls the adequacy of argument-fillers and leaves the activation control issues untouched. In implementing forward inferencing, for instance, there exists the danger that there are too many available inferences. People perform some forward inferencing on the guess that the same patterns occur. But they do not carry them vary far because the certainty of the inferences quickly falls below some threshold of plausibility [Ezquerro, J. & Iza, M. (1995)]. CASI's inferential behaviour is unfocused. Thus, an additional module is needed to maintain focus of inference.

In addition, language understanding might involve a planning module which cooperates with the focus maintenance module to maintain continuous inference. This module recognises current intermediate results of inference and starts subsequent chains of inference.

Secondly, as a connectionist reasoner, CASI might be used to investigate the following aspects of human cognitive process:

- it might suggest a plausible size of working memory used for a chain of inference because its inference involves activation of all the constant and variable entities at the beginning of inference;
- it can be used to estimate time taken for backward chaining style of inference or inference involving unification across groups of unifying arguments; CASI suggests that a wh-type of inference takes the same time as a yes-no type of inference because it does not use two-phase query processing for wh-type queries as is the case in SHRUTI;
- its architecture might suggest that why certain types of rules (which represent special types of inference) are more difficult to encode than others; such as rules which require multiple instantiations, rules which have recursive definitions, and unbalanced rules which require a special mechanism for binding generation.

8.3.3 More Efficient Implementation of CASI

CASI was implemented as a simulator using Prolog. Some features of the simulator were adopted from those of Rochester Connectionist Simulator (RCS) [Goddard et al (1988)]. Although CASI's behaviour has been successfully simulated using the Prolog based simulator, there are some ways of improving CASI's performance.

Firstly, enhancement of its performance can be achieved by implementing the reasoning module of CASI using more efficient programming language such as C. CASI is composed of several modules: the parsing module, the network building module, the reasoning module, and the user interface module. Among them, the reasoning module requires the most computing power because it performs the basic computation for all neuron elements involved in a chain of inference. Thus implementing this module using more efficient programming language, the performance of CASI will be improved.

Secondly, reimplementation of CASI on parallel machines or hardware platforms dedicated to neural network systems will greatly improve CASI's performance. In this case, dividing CASI's modules and further splitting of the task of the reasoning module to fit into the structure of a parallel machine and a dedicated hardware platform will be necessary.

Finally, if four basic neuron elements used to build SPNs can be implemented using corresponding components of hardware, CASI might be built directly on hardware. This will be the procedure implementing precompiled SPNs on hardware. The most difficult issue in this case, however, will be how temporal property of each neuron element is implemented as a hardware component. Although some neuron elements which have an oscillatory temporal behaviour have been proposed by König & Schillen (1991), Somers & Kopell (1993) and Wang (1995), a connectionist inference mechanism which uses these elements has not been proposed yet.

8.3.4 User Interface

As can be seen in Appendix A and B, CASI has a simple, text-based user interface. To provides more user friendly environment, CASI's current user interface may need to be reimplemented to have a X-window based user interface. This graphical user interface will give users the ability to explore CASI in more convenient ways.

Bibliography

Aaronson, J. (1991), Dynamic fact communication mechanism: a connectionist interface, *Proceedings of the 13th Conference of the Cognitive Science Society*, Hillsdale, NJ, Lawrence Erlbaum Associates Publishers, pp 643 – 647.

Ackley, D. H., Hinton, G. E., & Sejnowski, T. J. (1985), A learning algorithm for Boltzmann machines, *Cognitive Science*, Vol. 9, pp 147 – 169.

Ajjanagadde, V. (1990), Reasoning with function symbols in a connectionist system, *Proceedings of the 12th Conference of the Cognitive Science*, Hillsdale, NJ, Lawrence Erlbaum Associates Publishers, pp 285 – 292.

Ajjanagadde, V. & Shastri, L. (1991), Rules and variables in neural nets, *Neural Computation*, Vol. 3, pp 121 – 134.

Ajjanagadde, V. (1994), Unclear distinctions lead to unnecessary shortcomings: Examining the rule vs fact, role vs filler, and type vs predicate distinctions from a connectionist representation and reasoning perspective, *Proceedings of the 12th National Conference on Artificial Intelligence*, Menlo Park, CA, AAAI Press, pp 846 – 851.

Alexandre, F., Burnod, Y., Guyot, F. & Haton, J-P. (1991), The cortical column: a new processing unit for multilayered networks, *Neural Networks*, Vol. 4, No. 1, pp 15 – 26.

Baddeley, R. J. (1992), Phase coherence for solution likehood estimation in neural networks, *Technical Report CCCN-11*, Centre for cognitive and computational neuroscience, University of Stirling, Scotland.

Barnden, J. (1984), On short-term information processing in connectionist theories, *Cognition and Brain Theory*, Vol. 7, No. 2, pp 25 – 59.

Barnden, J. (1989), Neural-net implementation of complex symbol-processing in a mental model approach to syllogistic reasoning, *Proceedings of The 11th International Joint Conference on Artificial Intelligence*, San Mateo, CA, Morgan Kaufmann Publishers Inc. pp 568 – 573.

Barnden, J. (1992), Connectionism, structure-sensitivity, and systematicity: Refining the task requirements, *MCCS-92-227*, Computing Research Laboratory, New Mexico State University, NM.

Barnden, J. & Srinivas K. (1991), Encoding techniques for complex information structures in connectionist systems, *Connection Science*, Vol. 3, No. 2, pp 269 – 315.

Bartfai, G. (1996), Inferring hierarchical categories with ART-based modular neural network, *Proceedings of ECAI-96 Workshop on Neural Networks and Structured Knowledge*, pp 79 – 86.

Bechtel, W. & Abrahamsen, A. (1991), *Connectionism and the mind*, Cambridge, MA, Basil Blackwell Inc..

Bibel, W. (1987), *Automated Theorem Proving*, Braunschweig, Vieweg Verlag, Second edition.

Browne, A. & Pilkington, J. (1994a), Unification using a distributed representation, *Association for Computing Machinery Special Interest Group Bulletin on Artificial Intelligence*, Vol. 5, No. 2, pp 5 – 7.

Browne, A. & Pilkington, J. (1994b), Variable binding in a neural network using a distributed representation, *Proceedings of European Symposium on Artificial Neural Networks*, pp 199 – 204.

Bundy, A. (1983), *The Computer Modelling of Mathematical Reasoning*, Orlando, FL, Academic Press Inc..

Cairns, D. E., Baddeley, R. J. & Smith L. S. (1993), Constraints on synchronising oscillator networks, *Neural Computation*, Vol. 5, pp 260 – 266.

Chalmers, D. J. (1990), Syntactic transformations on distributed representation, *Connections Science*, Vol. 2, No. 1 & 2, pp 53 – 62.

Chang, C. L. & Lee, R. C. (1973), *Symbolic Logic and Mechanical Theorem Proving*, New York, NY, Academic Press Inc..

Chrisman, L. (1991), Learning recursive distributed representations for holistic computation, *Connection Science*, Vol. 3, No. 4, pp 345 – 365.

Das, S, Giles, C. L. & Sun, G. Z. (1993), Using prior knowledge in a NNPDA to learn context-free language, In S. J. Hanson, J. D. Cowan, C. L. Giles (Eds.), *Advances in Neural Information Processing Systems 5*, San Mateo, CA, Morgan Kaufmann Publishers Inc., pp 65 – 72.

Dolan, C. P. & Smolensky, P. (1989), Tensor product production system: A modular architecture and representation, *Connectionist Science*, Vol. 1, No. 1, pp 53 – 68.

Dreyfus, H. & Dreyfus, S. (1988), Making a mind versus modelling the brain: Artificial intelligence at a branch point, *Daedalus*, Winter, No. 1, pp 15 – 44.

Eckhorn, R. (1993), Dynamic bindings by real neurons: Arguments from physiology, neural network models and information theory, Open peer commentary on 'From simple associations to systematic reasoning: A connectionist representation of rules, variables, and dynamic bindings using temporal synchrony' by L. Shastri & V. Ajjanagadde, *Behavioural and Brain Sciences*, Vol. 16, No. 3, pp 457 – 458.

Edelman, G. (1992), *Bright air, brilliant fire, on the matter of the mind*, London, UK, Penguin.

Elman, J. L. (1989), Structured representations and connectionist models, *Proceedings of 11th conference of the Cognitive Science Society*, Hillsdale, NJ, Lawrence Erlbaum Associates Publishers, pp 17 – 25.

Feldman, J. A. (1982), Dynamic connections in neural networks, *Bio-Cybernetics*, Vol. 46, pp 27 – 39.

Feldman, J. A. (1989), Neural representation of conceptual knowledge, In L. Nadel, L. A. Cooper, P. Culicover & R. M. Harnish (Eds) *Neural connections, mental computation*, Cambridge, MA, The MIT Press, pp 68 – 103.

Feldman, J. A. & Ballard, D. H. (1982), Connectionist models and their properties, *Cognitive Science*, Vol. 6, No. 3, pp 205 – 254.

Fodor, J. A. & McLaughlin, B. P. (1990), Connectionism and the problem of systematicity: Why Smolensky's solution doesn't work, *Cognition*, Vol. 35, No. 2, pp 183 – 204.

Fodor, J. A. & Pylyshyn, Z. W. (1988), Connectionism and cognitive architecture: a critical analysis, In S. Pinkerr & J. Mehler (Eds.), *Connection and Symbols*, Cambridge, MA, The MIT Press, pp 3 – 71.

Gallant, S. I. (1988), Connectionist expert systems, *Communications of the Association for Computing Machinery*, Vol. 31, No. 2, pp 152 – 169.

Giles, C. L. & Omlin, C. W. (1996), Insertion and refinement of symbolic rules in dynamically driven recurrent neural networks, *Connection Science*, Vol. 5, No. 3&4, pp 307 – 337.

Goddard, N. H., Lynne, K. J. & Mintz, T. (1988), Rochester Connectionist Simulator, *Technical Report 233 (Revised)*, Computer Science Department, University of Rochester, NY.

Grant, B. W. (1991), EXCON: An Implementation of a Connectionist Expert System, *Internal BT report*, British Telecommunications, UK.

Güsgen, H. W. & Hölldobler, S. (1991), Connectionist inference systems, In B. Fronhöfer and G. Wrightson (Eds.), *Parallelization in Inference Systems*, Lecture Notes in Computer Science, Berlin, Springer-Verlag, pp 82 – 120.

Harris, C. L. & Elman, J. L. (1989), Representing variable information with simple recurrent networks, *Proceedings of the 11th conference of the Cognitive Science Society*, Hillsdale, NJ, Lawrence Erlbaum Associate Publishers, pp 635 – 644.

Hendler, J. A. (1989), Editorial: On the need for hybrid systems, *Connection Science*, Vol. 1, No. 3, pp 227 – 229.

Hilario, M. (1995), An overview of strategies for neurosymbolic integration, *Proceedings of ECAI-96 Workshop on Neural Networks and Structured Knowledge*, pp 1 – 6.

Hinton, G. E. (1984), Distributed representations, *CMU-CS-84-157*, Computer Science Department, Carnegie-Mellon University, PA.

Hinton, G. E., McClelland, J. M. & Rumelhart, D. E. (1986), Distributed representations, In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel Distributed Processing: Explorations in the microstructure of cognition*, Vol. 1, Cambridge, MA, The MIT Press, pp 77 – 109.

Hinton, G. E. & Sejnowski, T. J. (1986), Learning and relearning in Boltzmann machines, In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition*, Vol. 1, Cambridge, MA, The MIT Press, pp 282 – 317.

Hölldobler, S. H. (1990), A structured connectionist unification algorithm, *Proceedings of the National Conference on Artificial intelligence*, Menlo Park, CA, AAAI Press, pp 587 – 593.

Hölldobler, S. H. & Kurfeß, F. (1991), CHCL - A connectionist inference system, In B. Fronhöfer & G. Wrightson (Eds.), *Parallelization in Inference Systems*, Lecture Notes in Computer Science, Berlin, Springer-Verlag, pp 318 – 342.

Hopfield, J. J. (1982), Neural networks and physical systems with emergent collective computational abilities, *Proceedings of the National Academy of Sciences*, Vol. 79, pp 2554 – 2558.

Johnson-Laird, P. N. & Bara, B. G. (1984), Syllogistic inference, *Cognition*, Vol. 16, No. 1, pp 1 – 61.

König, P. & Schillen, T. B. (1991), Stimulus-dependent assembly formation of oscillatory responses: I. Synchronisation, *Neural Computation*, Vol. 3, 155 – 166.

Lallement, Y. & Alexandre, F. (1995), Cognitive aspects of neurosymbolic integration, *Proceedings of ECAI-96 Workshop on Neural Networks and Structured Knowledge*, pp 7 – 11.

Lange, T. E. & Dyre, M. G. (1989a), High–level inferencing in a connectionist network, *Connection Science*. Vol. 1, No. 2, pp 181 – 217.

Lange, T. E. & Dyer, M. G. (1989b), Dynamic, non-local role bindings and inferencing in a localist network for natural language understanding, In D. S. Tourentzky (Ed.), *Advances in neural Information Processing Systems I*, San Mateo, CA, Morgan Kaufmann Publishers Inc., pp 545 – 552.

Luger, G. F. & Stubblefield, W. A. (1989), *Artificial Intelligence and The design of Expert Systems*, Reading, MA, The Benjamin/Cummings Publishing Company, Inc..

Mani, D. R. & Shastri, L. (1991), Combining a connectionist type hierarchy with a connectionist rule-based reasoner, *Proceedings of the 13th Conference of the Cognitive Science Society*, Hillsdale, NJ, Lawrence Erlbaum Associates Publishers, pp 418 – 423.

Mani, D. R. & Shastri, L. (1993), Reflexive reasoning with multiple-instantiation in a connectionist reasoning system with a typed hierarchy, *Connection Science*, Vol. 5, No. 3&4, pp 205 – 242.

Manna, Z. & Waldinger, R. (1985), *The Logical Basis for Computer Programming*, Reading, MA, Addison-Wesley Publishing Company.

Markov, Z., Dichev, C. & Sinapova, L. (1990) The Net-Clause Language - a tool for describing network models, *Proceedings of the 8th Canadian Conference on Artificial Intelligence*, pp 33 – 39.

Markov, Z. (1992), An Approach to Concept Learning Based on Term Generalisation, *Proceedings of the 9th International Machine Learning Conference*, San Mateo, CA, Morgan Kaufmann Publishers Inc., pp 310 – 315.

Markov, Z. (1993), Inductive Inference in Networks of Relations, *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pp 256 – 277.

McCarthy, J. (1988), Epistemological Challenges for Connectionism, Open peer commentary on 'Proper Treatment of Connectionism' by P. Smolensky, *Behavioural and Brain Sciences*, Vol. 11, No. 1, p 44.

McCulloch, W. S. & Pitts, W. (1943), A logical calculus of the ideas immanent in nervous activity, *Bulletin of Mathematical Biophysics*, Vol. 5, pp 115 – 133.

Medsker, L. R. (1994), *Hybrid neural network and expert systems*, Boston, NY, Kluwer Academic Publishers.

Ezquerro, J. & Iza, M. (1995), A hybrid architecture for text comprehension: Elaborative inference and attentional focus, *Pragmatics & Cognitive*, Vol. 3, No. 2, pp 247 – 279.

Miller, G. A. (1956), The magical number seven, plus or minus two: Some limits on our capacity for processing information, *The Psychological Review*, Vol. 63, No. 2, pp 81 – 97.

Minsky, M. A. & Papert, S. (1969), *Perceptrons*, Cambridge, MA, The MIT Press.

Nenov, V. Dyer, M. (1993), Perceptually grounded language learning: Part 1 - a neural network architecture for robust sequence association, *Connection Science*, Vol. 5, No. 2, pp 115 – 138.

Nenov, V. Dyer, M. (1994), Perceptually grounded language learning: Part 2 - DETE: a neural/procedural model, *Connection Science*, Vol. 6, No. 1, pp 3 – 42.

Newell, A. & Simon, H. A. (1976), Computer science as empirical enquiry: symbols and search, *Communications of the Association for Computing Machinery*, Vol. 19, pp. 113 – 126, Also in J. Haugeland (Ed), *Mind Design*, Cambridge, MA, The MIT Press.

Orsier, B. & Labbi, A. (1995), NESSY3L: a NEuroSymbolic SYstem with 3 levels, *Working Notes of IJCAI-95 Workshop on Connectionist-Symbolic Integration*, pp 100 – 105.

Palm, G (1993), Making reasoning more reasonable: Event-coherence and assemblies, Open peer commentary on 'From simple associations to systematic reasoning: A connectionist representation of rules, variables, and dynamic bindings using temporal synchrony' by L. Shastri & V. Ajjanagadde, *Behavioural and Brain Sciences*, Vol. 16 No. 3, p 470.

Park, N. S. (1992), A connectionist representation of rules, facts and dynamic bindings of variables, *TP-15*, Department of Artificial Intelligence, The university of Edinburgh, Scotland.

Park, N. S. & Robertson, D. (1995), A localist network architecture for logical inference based on temporal synchrony approach to dynamic variable binding, *Proceedings of IJCAI-95 workshop on Connectionist-Symbolic Integration*, pp 63 – 68.

Park, N. S. and Robertson, D. (1996a), 'A connectionist representation of symbolic components, dynamic bindings and basic inference operations', *Proceeding of ECAI-96 Workshop on Neural Networks and Structured Knowledge*, pp 33 – 39.

Park, N. S., Robertson, D. (1996b), A localist network architecture for logical inference, In R. Sun, F. Alexandre (Eds.), *Connectionist Symbolic Integration*, Hillsdale, NJ, Lawrence Erlbaum Associates Publishers, in print.

Park, N. S., Robertson, D. & Stenning K. (1993), An extension of the temporal synchrony solution to dynamic variable bindings in a connectionist system, *RP93-666*, Department of Artificial Intelligence, The University of Edinburgh, Scotland.

Park, N. S., Robertson, D. & Stenning, K. (1994), Reasoning with limited unification in a connectionist rule-based system', *Electronic Nodes of ICLP-94 Workshop on Logic and Reasoning with Neural Networks*, S. Margherita Ligure, Italy.

Park, N. S., Robertson, D. & Stenning, K. (1995), An extension of the temporal synchrony approach to dynamic variable binding in a connectionist inference system', *Knowledge-Based Systems*, Vol. 8, No. 6., pp 345 – 358.

Park, N. S., Robertson, D. and Stenning, K. (1996), Symbolic knowledge encoding using a dynamic binding mechanism and an embedded inference mechanism, In D. Levine, B. Brown, and T. Shirey (Eds.), *Oscillation in Neural Systems*, Hillsdale, NJ, Lawrence Erlbaum Associates Publishers, in print.

Pereira Castro Jr., J. D. L. (1995), Acquiring semantic power in weightless neural network, *Proceedings of IEEE International Conference on Neural Networks*, pp 674-679.

Pereira Castro Jr., J. D. L. (1996), Semantic knowledge in general neural units: Issues of Representation, *Proceedings of ECAI-96 Workshop on Neural Networks and Structured Knowledge*, pp 17 – 24.

Pollack, J. B. (1988), Recursive auto-associative memory: Devising compositional distributed representations, *Proceedings of 10th conference of the Cognitive Science Society*, Hillsdale, NJ, Lawrence Erlbaum Associates Publishers, pp 33 – 39.

Pollack, J. B. (1990), Recursive distributed representations, *Artificial Intelligence*, Vol. 46, No. 1 & 2, pp 77 – 105.

Robertson, D., Park, N. S. & Agusti, J. (1994), Layered design of KBS from specification to hardware, *Proceedings of ECAI-94 Workshop on Formal Methods for Knowledge-Based Systems*, Amsterdam, Netherlands.

Rohwer, R. (1991), Neural networks for time-varying data, In F. Murtagh (Ed.), *Neural Networks for Statistical and Economic Data*, Luxembourg, Statistical Office of the European Communities, pp 59 – 70.

Rohwer, R., Grant, B. & Limb, P. R. (1992), Towards a connectionist reasoning system, *BT Technology*, Vol. 10, No. 3, pp 103 – 109.

Rosenblatt, F. (1962), *The principles of neurodynamics*, Washington, NY, Spartan Books.

Rumelhart, D. E. & McClelland, J. L. Eds. (1986), *Parallel distributed processing: Explorations in the microstructure of cognition*, Vol. 1, Cambridge, MA, The MIT Press.

Shastri, L. (1988), A connectionist approach to knowledge representation and limited inference, *Cognitive Science*, Vol. 12, pp 331 – 392.

Shastri, L. (1991), Relevance of connectionism to AI: A representation and reasoning perspective' In J. Barnden & J. Pollack (Eds.), *Advances in connectionist and neural computation theory*, Vol. 1, Ablex.

Shastri, L. (1992), Neurally motivated constraints on a working memory capacity of production system for rapid parallel processing, *Proceedings of the 14 Conference* *of the Cognitive Science Society*, Hillsdale, NJ, Lawrence Erlbaum Associates Publishers, pp 159 – 164.

Shastri, L. & Ajjanagadde, V. (1993), 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. pp 417 – 494.

Shasrti, L. & Feldman, J. A. (1986), Semantic nets, neural nets, and routines, In N. Sharkey (Ed), *Advances in Cognitive Science*, Chichester, NY, Ellis Harwood/John Wiley & Sons.

Shastri, L. & Grannes, D. J. (1995), Dealing with negated knowledge and inconsistency in a neurally motivated model of memory and reflexive reasoning, *TR-95-041*, International Computer Science Institute, Berkeley, CA.

Smith, B. C. (1985), Reflection and semantics in a procedural language, In R. J. Brachman and H. J. Levesque (Eds.), *Reading in Knowledge Representation*, San Mateo, CA, Morgan Kaufmann Publishers Inc., Chapter Three, pp 31 – 38.

Smolensky, P. (1987), On variable binding and the representation of symbolic structures in connectionist systems. *Technical Report CU-CS-355-87*, Department of Computer Science and Institute of Cognitive Science, University of Colorado, CO.

Smolensky, P. (1990), Tensor product variable binding and the representation of symbolic structure in connectionist networks, *Artificial Intelligence*, Vol. 46, pp 159 – 216.

Somers, D. & Kopell, N. (1993), Rapid synchronisation through fast threshold modulation, *Biological Cybernetics*, Vol. 68, pp 393 – 407.

Sperduti, A. (1994), Labeling RAAM, *Connection Science*, Vol. 6, No. 4, pp 429 – 459.

Sperduti, A., Starita, A. & Goller, C. (1995), Distributed representations for terms in hybrid reasoning systems, *Working Notes of IJCAI-95 Workshop on Connectionist-Symbolic Integration*, pp 75 – 80.

Sperduti, A. (1996), Neural networks for the classification of structures, *Proceedings of ECAI-96 Workshop on Neural Networks and Structured Knowledge*, pp 69 – 96.

Barnden, J. A. & Srinivas, K. (1990), Winner-takes-all-network: time-based versus activation-based mechanisms for various selection goals, *Proceedings of the IEEE International Symposium on Circuits and Systems*, NY, IEEE.

Stenning, K. & Levy, J. (1988), Knowledge-rich solutions to the binding problem: a simulation of some human computational mechanisms, *Knowledge-based Systems*, Vol. 1, No. 3, pp 181–217.

Stenning, K., Shepherd, M. & and Levy, J. (1988), On the construction of representations for individuals from descriptions in text, *Language and Cognitive Processes*, Vol. 3, No. 2, pp 129 – 164.

Stickel, M. E. (1987), An introduction to automated deduction, In W. Bibel & P. Jorrand (Eds.), *Fundamentals of Artificial Intelligence*, Berlin, Springer-Verlag, pp 75 – 132.

Sun, R (1992), On variable binding in connectionist networks, *Connection Science*, Vol. 4, No. 2, pp 93 – 124.

Sun, R. (1994), *Integrating Rules and Connectionism for Robust Commonsense Reasoning*, New York, NY, John Wiley and Sons.

Sun, G. Z., Giles, C. L., Chen, H. H. & Lee, Y. C. (1993), The neural network pushdown automaton: model, stack and learning simulations, *UMIACS-TR-93-77* & *CS-TR-3118*, Institute for Advanced Computer Studies, University of Maryland, MD.

Sun, R. & Peterson, T. (1995), A hybrid learning model for reactive sequential decision making, *Working Notes of IJCAI-95 Workshop on Connectionist-Symbolic Integration*, pp 18 – 24.

Tourentzky, D. S. (1990), BoltzCONS: Dynamic symbol structures in a connectionist network, *Artificial Intelligence*, Vol. 46, pp 5 – 46.

Touretzky D. S. & Hinton, G. E. (1988), A distributed connectionist production system, *Cognitive Science*, Vol. 12, No. 3, pp 423 – 466.

Towell, G. G., & Shavlik, J. W. (1993), Extracting refined rules from knowledge based neural networks, *Machine Learning*, Vol. 13, pp 71 – 101.

Towell, G. G., & Shavlik, J. W. (1994), Knowledge-based artificial neural networks, *Artificial Intelligence*, Vol. 70, pp 119 – 165.

Turner, R. (1984), *Logics for Artificial Intelligence*, Chichester, NY, Ellis Harwood/John Wiley & Sons.

von Neumann, J. (1956), Self-organizing of orientation sensitive sells in the striate cortex, *Kybernetik*, Vol. 14, pp 85 – 100.

von der Malsburg (1986), Am I thinking assemblies?, In G. Palm & A. Aertsen (Eds.), *Brain Theory*, Berlin, Springer-Verlag, pp 161 – 176.

Wang, D. L. (1995), Emergent synchrony in locally coupled neural oscillators, *IEEE Transaction Neural Networks*, Vol. 6, pp 941 – 948.

Winograd, S., & Cowan, J. (1963), *Reliable Computation in the Presence of Noise*, Cambridge, MA, The MIT Press.

Appendix A

Demonstration of Inference Procedures

This appendix demonstrates CASI's inference procedure with the example rule and fact described in Section 5.5.2. The first session shows forward chaining with the example rule, $p(X, X, Y) \rightarrow q(X, Y)$, and the second session backward chaining with the example fact, *p(a,a,b)*. Since CASI is an interactive simulator written in Prolog, encoding rules and facts and performing inference is carried out interactively with a text-based user interface. Should the readers refer to *User Manual* (Appendix C) for the detailed commands to operate CASI.

A.1 Starting CASI

Since CASI is implemented using Prolog, a standard Prolog interpreter is needed. CASI was implemented and simulated using Sicstus Prolog (Version 3.0) under UNIX and also in SWI-Prolog (Version 2.1.9) under LINUX. When the Prolog interpreter is invoked, the modules of the simulator have to be loaded first by typing "[load].". Then CASI is started by by typing "casi." as follows:

```
SICStus 3  #2: Fri Oct 6 16:26:21 BST 1995
| ?- [load].
{consulting ~namseog/phd/work/sim/load.pl...}
{consulting ~namseog/phd/work/sim/mesgs.pl...}
{~namseog/phd/work/sim/mesgs.pl consulted, 70 msec 6512 bytes}
{consulting ~namseog/phd/work/sim/interface.pl...}
{~namseog/phd/work/sim/interface.pl consulted, 280 msec 20928 bytes}
{consulting ~namseog/phd/work/sim/paser.pl...}
```

```
{~namseog/phd/work/sim/paser.pl consulted, 170 msec 14752 bytes}
{consulting ~namseog/phd/work/sim/compiler.pl...}
{~namseog/phd/work/sim/compiler.pl consulted, 1160 msec 64240 bytes}
{consulting ~namseog/phd/work/sim/reasoner.pl...}
{~namseog/phd/work/sim/reasoner.pl consulted, 490 msec 30160 bytes}
{~namseog/phd/work/sim/load.pl consulted, 2350 msec 186016 bytes}

yes
| ?- casi.

  WELCOME TO A CONNECTIONIST ARCHITECTURE FOR SYMBOLIC INFERENCE (CASI)
    @1996, NAM SEOG PARK, DEPT. OF ARTIFICIAL INTELLIGENCE, EDINBURGH

TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|:
```

A.2 Forward Chaining

This sample session demonstrate how CASI carries out forward chaining over the SPN encoding the following rule:

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

This example rule is loaded using the load command and compiled by the knowledge compiler. Once the SPN is built, one can present various types of predicates to CASI to obtain the result of inference. The following are some guidance to help understand the sample session:

- according to the Prolog naming convention, any name beginning with a lowercase alphabetic character is assumed to be a constant. All names beginning with uppercase letters are variable names.
- in processing of queries, a number n represents the nth phase of an oscillation cycle. The node names coming after "-" symbol are candidate nodes to be considered firing and the node names coming after "+" symbol are nodes fired during that period of oscillation. The initial bindings, after a predicate is presented, are represented at the beginning of inference after the symbol, "Q:+".

```
TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: l.

FILE NAME: rpxxyqxy.

load: p(X,X,Y) --> q(X,Y)

<MESG> LoadSuccess: rules and facts are loaded successfully

TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: c.

  RULE TO BE ENCODED:     p(X,X,Y) --> q(X,Y)

<MESG> FwNetBuilt: network for forward chaining has been built

TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: ss.

    Input knowledge  : 1 rules  0 facts
    Network type     : forward only
    Network status   : built

    Inference type   : forward
    Verbose mode     : off

TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: i.

ENTER QUERY: p(a,U,U).

  Q:+ <p:pa,[*]><a,[1]><p:arg1_r,[1]><U,[6]><p:arg2_l,[6]><p:arg3_l,[6]>
  1:- (q:pa,4)(bim1:p1p2_r,0)(bim1:p1p2_l,0)
      (bim2:p1p2_p3,1)(bim2:p1p2_p3_l,2)(q:arg2_l,4)
    + (bim1:p1p2_r[1])(bim1:p1p2_l[6])


  2:- (q:pa,3)(bim2:p1p2_p3,0)(bim2:p1p2_p3_l,1)
```

```
    (q:arg2_l,3)(ccmv:p1p2,0)(bim2:p1p2_p3_r,1)
    (q:arg1_r,3)(q:arg1_l,3)
  + (bim2:p1p2_p3[*])


3:- (q:pa,2)(bim2:p1p2_p3_l,0)(q:arg2_l,2)
    (bim2:p1p2_p3_r,0)(q:arg1_r,2)(q:arg1_l,2)
  + (bim2:p1p2_p3_l[6])(bim2:p1p2_p3_r[1])


4:- (q:pa,1)(q:arg2_l,1)(q:arg1_r,1)
    (q:arg1_l,1)(ccm2:p_X,0)(ccm2:p_Y,0)
    (q:arg2_r,1)
  +

5:- (q:pa,0)(q:arg2_l,0)(q:arg1_r,0)
    (q:arg1_l,0)(q:arg2_r,0)
  + (q:pa[*])(q:arg2_l[6])(q:arg1_r[1])
    (q:arg1_l[6])(q:arg2_r[1])

    q:{pa[*] arg1([6][1]) arg2([6][1])}
```

By the initial binding instantiation procedure, the argument nodes of the p predicate assembly became active in-phase with the filler nodes of the presented predicate predicate, *p(a,U,U)*. After 5 oscillation cycles, the argument nodes of the q predicate assembly became active as shown above. These activation states represent the result of inference, *q(a,a)*, and the result of unification, $\{a/U\}$, produced during inference.

A.3 Backward Chaining

Backward chaining is carried out in a similar fashion to forward chaining. The example fact used for this session is

$$p(a, a, b).$$

A step of inference is performed after the corresponding SPN is built from the example fact.

```
TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
```

```
                 < x>: EXIT                  <sv>: SET VERBOSE MODE
|: l.

FILE NAME: fpaab.

load: p(a,a,b)

<MESG> LoadSuccess: rules and facts are loaded successfully

TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: c.

  FACT TO BE ENCODED:     p(a,a,b)

<MESG> BuildBWNetOnly: no rule exists, only net for backward chaining

TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: ss.

    Input knowledge  : 0 rules  1 facts
    Network type     : backward only
    Network status   : built

    Inference type   : backward
    Verbose Mode     : off

TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: i.

ENTER QUERY: p(a,U,V).

  Q:+ <p:pa,[*]><a,[1]><p:arg1_r,[1]><U,[4]><p:arg2_l,[4]><V,[7]><p:arg3_l,[7]>
  1:- (p_f1:fa,3)(p_f1:a_gate,1)(bpmt:p_f1_12_l,2)
      (bpmt:p_f1_12_r,2)(p_f1:b_gate,1)(bpmt:p_f1_3_l,2)
      (bpmt:p_f1_3_r,2)(fbim:p_f1_12_r,0)(fbim:p_f1_12_l,0)
      (fccmv:p_f1,1)
    + (fbim:p_f1_12_r[1])(fbim:p_f1_12_l[4])


  2:- (p_f1:fa,2)(p_f1:a_gate,0)(bpmt:p_f1_12_l,1)
      (bpmt:p_f1_12_r,1)(p_f1:b_gate,0)(bpmt:p_f1_3_l,1)
      (bpmt:p_f1_3_r,1)(fccmv:p_f1,0)(fccm:p_f1,1)
```

```
  + (p_f1:a_gate[1,4])(p_f1:b_gate[7])


3:- (p_f1:fa,1)(bpmt:p_f1_12_l,0)(bpmt:p_f1_12_r,0)
    (bpmt:p_f1_3_l,0)(bpmt:p_f1_3_r,0)(fccm:p_f1,0)
    (a,1)(b,1)
  + (bpmt:p_f1_12_l[4])(bpmt:p_f1_12_r[1,4])(bpmt:p_f1_3_l[7])
    (bpmt:p_f1_3_r[7])


4:- (p_f1:fa,0)(a,0)(b,0)
    (p_f1:arg1_l,0)(p_f1:arg2_l,0)(p_f1:arg1_r,0)
    (p_f1:arg2_r,0)(p_f1:arg3_l,0)(p_f1:arg3_r,0)
  + (p_f1:fa[*])(a[1,4])(b[7])
    (p_f1:arg1_l[4])(p_f1:arg2_l[4])(p_f1:arg1_r[1,4])
    (p_f1:arg2_r[1,4])(p_f1:arg3_l[7])(p_f1:arg3_r[7])

    p_f1:{fa[*] arg1([4][1,4]) arg2([4][1,4]) arg3([7][7])}
       [a,a,b]
         - a[1,4] b[7]
         - p_f1:a_gate[1,4] p_f1:b_gate[7]

5:- (p_f1:a_gate,1)(p_f1:b_gate,1)
  +

6:- (p_f1:a_gate,0)(p_f1:b_gate,0)
  + (p_f1:a_gate[1,4])(p_f1:b_gate[7])


7:- (a,1)(bpmt:p_f1_12_r,0)(b,1)
    (bpmt:p_f1_3_r,0)
  + (bpmt:p_f1_12_r[1,4])(bpmt:p_f1_3_r[7])


8:- (a,0)(b,0)(p_f1:arg1_r,0)
    (p_f1:arg2_r,0)(p_f1:arg3_r,0)
  + (p_f1:arg1_r[1,4])(p_f1:arg2_r[1,4])(p_f1:arg3_r[7])
```

Posing the query predicate, $p(a,U,V)$, makes CASI set up the initial bindings between the filler nodes of the posed predicate and the argument nodes of the p predicate assembly. As inference cycles continue, the fact assembly (p_{f1}) which encode the fact, $p(a,a,b)$, became active after 4 oscillation cycles. The activation of the fact assembly indicates the affirmative answer to the posed query predicate and the activation states of the arguments nodes represent the results of the unification. Since the query predicate has variable fillers, the bindings obtained from the unification result is used to produce the values to the variable fillers. Since the unification result obtained from this inference is $\{a/U,b/V\}$, the values for the variable fillers (U and V) are a and b.

Appendix B

More Experiments with CASI

This appendix contains the results of more experiments carried out to test CASI's ability to support symbolic inference. Six rules which can be categorised into three groups are selected as follows:

- rules with repeated constant arguments

$$p(a, a) \rightarrow q(a)$$

$$p(a, a, b, b) \rightarrow q(a, b)$$

- rules with mixed types of arguments

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

$$p(a, a, b, b, X, X) \rightarrow q(a, b, X)$$

- rules with multiple antecedent predicates

$$p(X, Y) \wedge q(X, Y) \rightarrow r(X, Y)$$

$$p(a, X) \wedge q(X) \rightarrow r(X)$$

Each rule is encoded into CASI's CLDB. Then, various types of test predicates are presented to test CASI's ability to accommodate a symbolic style of inferences. The rest of this appendix illustrates the detailed results obtained from each experiment.

B.1 Rules with repeated constant arguments

B.1.1 Experiment 1

Sample rule used:

$$p(a,a) \rightarrow q(a)$$

Results:

```
(P1) query presented: p(a,a)
  . phase allocation - a[1]
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([0][1])}
  . inference result - q:{pa[*] arg1([0][1])}
    (symbolic meaning) by right elements - q(a)
  . inference cycles taken - 3 cycles

(P2) query presented: p(a,b)
  . phase allocation - a[1],b[6]
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([0][6])}
  . inference result - none
    (symbolic meaning) no result

(P3) query presented: p(a,U)
  . phase allocation - a[1],U[6]
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([6][0])}
  . inference result - q:{pa[*] arg1([6][1])}
    (symbolic meaning) by right elements - q(a)
                       by left elements - q(U)
                       unification results - {a/U}
  . inference cycles taken - 3 cycles

(P4) query presented: p(U,V)
  . phase allocation - U[1],V[6]
  . initial bindings - p:{pa[*] arg1([1][0]) arg2([6][0])}
  . inference result - q:{pa[*] arg1([1,6][1,6])}
                       p_q:a_gate[1,6]
    (symbolic meaning) by right elements - q(a)
                       by left elements - q(U)
                       unification results - {a/U,U/V}
  . inference cycles taken - 3 cycles
```

B.1.2 Experiment 2

Sample rule used:

$$p(a,a,b,b) \rightarrow q(a,b)$$

Results:

```
(P1) query presented: p(a,a,b,b)
  . phase allocation - a[1],b[6]
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([0][1])
                                arg3([0][6]) arg4([0][6])}
  . inference result - q:{pa[*] arg1([0][1]) arg2([0][6])}
    (symbolic meaning) by right elements - q(a,b)
  . inference cycles taken - 3 cycles

(P2) query presented: p(U,U,b,V)
  . phase allocation - U[1],b[4],V[7]
  . initial bindings - p:{pa[*] arg1([1][0]) arg2([1][0])
                                arg3([0][4]) arg4([7][0])}
  . inference result - q:{pa[*] arg1([1][1]) arg2([7][4])}
                       p_q:a_gate[1]
    (symbolic meaning) by right elements - q(a,b)
                       by left elements - q(U,V)
                       unification results - {a/U,b/V}
  . inference cycles taken - 3 cycles

(P3) query presented: p(U,U,U,U)
  . phase allocation - U[1]
  . initial bindings - p:{pa[*] arg1([1][0]) arg2([1][0])
                                arg3([1][0]) arg4([1][0])}
  . inference result - none
    (symbolic meaning) no result

(P4) query presented: p(U,a,U,U)
  . phase allocation - a[1],U[6]
  . initial bindings - p:{pa[*] arg1([6][0]) arg2([0][1])
                                arg3([6][0]) arg4([6][0])}
  . inference result - none
    (symbolic meaning) no result

(P5) query presented: p(U,V,U,V)
  . phase allocation - U[1],V[6]
  . initial bindings - p:{pa[*] arg1([1][0]) arg2([1][0])
                                arg3([6][0]) arg4([6][0])}
  . inference result - none
    (symbolic meaning) no result

(P6) query presented: p(U,a,U,b)
  . phase allocation - a[1],U[4],b[7]
```

```
  . initial bindings - p:{pa[*] arg1([4][0]) arg2([0][1])
                                arg3([4][0]) arg4([0][7])}
  . inference result - none
    (symbolic meaning) no result

(P7) query presented: p(U,b,c,c)
  . phase allocation - U[1],b[4],c[7]
  . initial bindings - p:{pa[*] arg1([1][0]) arg2([0][4])
                                arg3([0][7]) arg4([0][7])}
  . inference result - none
    (symbolic meaning) no result
```

B.2 Rules with mixed types of arguments

B.2.1 Experiment 3

Sample rule used:

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

Results:

```
(P1) query presented: p(a,U,a,V)
  . phase allocation - U[1],a[4],V[7]
  . initial bindings - p:{pa[*] arg1([0][4]) arg2([1][0])
                                arg3([0][4]) arg4([7][0])}
  . inference result - q:{pa[*] arg1([7][0])}
                       p_q:b_gate[5]
    (symbolic meaning) by left elements - q(V)
  . inference cycles taken - 5 cycles

(P2) query presented: p(a,b,b,c)
  . phase allocation - a[1],b[4],c[7]
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([0][4])
                                arg3([0][4]) arg4([0][7])}
  . inference result - q:{pa[*] arg1([0][7])}
    (symbolic meaning) by right elements - q(c)
  . inference cycles taken - 5 cycles

(P3) query presented: p(a,b,c,d)
  . phase allocation - a[1],b[3],c[5],d[7]
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([0][3])
                                arg3([0][5]) arg4([0][7])}
  . inference result - none
    (symbolic meaning) no result
```

```
(P4) query presented: p(U,U,b,c)
  . phase allocation - U[1],b[4],c[7]
  . initial bindings - p:{pa[*] arg1([1][0]) arg2([1][0])
                                arg3([0][4]) arg4([0][7])}
  . inference result - none
    (symbolic meaning) no result

(P5) query presented: p(U,b,V,U)
  . phase allocation - V[1],b[4],U[7]
  . initial bindings - p:{pa[*] arg1([7][0]) arg2([0][4])
                                arg3([1][0]) arg4([7][0])}
  . inference result - q:{pa[*] arg1([7][7])}
                       p_q:a_gate[7]
    (symbolic meaning) by right elements - q(a)
                       by left elements - q(U)
                       unification results - {a/U}
  . inference cycles taken - 5 cycles

(P6) query presented: p(U,U,U,U)
  . phase allocation - U[1]
  . initial bindings - p:{pa[*] arg1([1][0]) arg2([1][0])
                                arg3([1][0]) arg4([1][0])}
  . inference result - q:{pa[*] arg1([1][1])}
                       p_q:a_gate[1]
    (symbolic meaning) by right elements - q(a)
                       by left elements - q(U)
                       unification results - {a/U}
  . inference cycles taken - 5 cycles
```

B.2.2 Experiment 4

Sample rule used:

$$p(a, a, b, b, X, X) \rightarrow q(a, b, X)$$

Results:

```
(P1) query presented: p(a,U,b,V,V,V)
  . phase allocation - a[1],U[3],b[5],V[7]
  . initial bindings - p:{pa[*] arg1([7][0]) arg2([0][1]) arg3([0][5])
                                arg4([3][0]) arg5([7][0]) arg3([0][5])}
  . inference result - q:{pa[*] arg1([3][1]) arg2([7][5]) arg3([7][5])}
    (symbolic meaning) by right elements - q(a,b,b)
                       by left elements - r(U,V,V)
                       unification results - {a/U,b/V}
  . inference cycles taken - 5 cycles

(P2) query presented: p(U,U,V,V,U,U)
```

```
  . phase allocation - V[1],U[6]
  . initial bindings - p:{pa[*] arg1([6][0]) arg2([6][0]) arg3([0][1])
                                arg4([0][1]) arg5([6][0]) arg3([6][0])}
  . inference result - q:{pa[*] arg1([6][6]) arg2([1][1]) arg3([6][6])}
                       p_q:b_gate[1],p_q:a_gate[6]
    (symbolic meaning) by right elements - q(a,b,a)
                       by left elements - r(U,V,U)
                       unification results - {a/U,b/V}
  . inference cycles taken - 5 cycles

(P3) query presented: p(a,U,V,V,b,W)
  . phase allocation - a[1],U[3],V[5],b[7],W[9]
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([3][0]) arg3([5][0])
                                arg4([5][0]) arg5([0][7]) arg3([9][0])}
  . inference result - q:{pa[*] arg1([3][1]) arg2([5][7]) arg3([9][7])}
    (symbolic meaning) by right elements - q(a,b,b)
                       by left elements - r(U,V,W)
                       unification results - {a/U,b/V,b/W}
  . inference cycles taken - 5 cycles

(P4) query presented: p(a,U,V,V,V,W)
  . phase allocation - a[1],U[3],V[5],W[7]
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([3][0]) arg3([5][0])
                                arg4([5][0]) arg5([5][0]) arg3([7][0])}
  . inference result - q:{pa[*] arg1([3][1]) arg2([5,7][5]) arg3([5,7][5])}
                       p_q:b_gate[5]
    (symbolic meaning) by right elements - q(a,b,b)
                       by left elements - r(U,V,V)
                       unification results - {a/U,b/V,V/W}
  . inference cycles taken - 5 cycles

(P5) query presented: p(a,U,b,U,V,V)
  . phase allocation - a[1],b[3],U[5],V[7]
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([5][0]) arg3([0][3])
                                arg4([5][0]) arg5([7][0]) arg3([7][0])}
  . inference result - none
    (symbolic meaning) no result

(P6) query presented: p(a,U,b,V,U,V)
  . phase allocation - a[1],b[3],U[5],V[7]
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([5][0]) arg3([0][3])
                                arg4([7][0]) arg5([5][0]) arg3([7][0])}
  . inference result - none
    (symbolic meaning) no result

(P7) query presented: p(U,U,V,V,U,V)
  . phase allocation - U[1],V[6]
  . initial bindings - p:{pa[*] arg1([1][0]) arg2([1][0]) arg3([6][0])
                                arg4([6][0]) arg5([1][0]) arg3([6][0])}
  . inference result - none
    (symbolic meaning) no result

(P8) query presented: p(a,U,V,V,b,U)
  . phase allocation - a[1],V[3],b[5],U[7]
```

```
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([7][0]) arg3([3][0])
                                arg4([3][0]) arg5([0][5]) arg3([7][0])}
  . inference result - none
    (symbolic meaning) no result
```

B.3 Rules with multiple antecedent predicates

B.3.1 Experiment 5

Sample rule used:

$$p(a, X) \wedge q(X) \to r(X)$$

Results:

```
(P1) query presented: p(a,b) ^ q(b)
  . phase allocation - a[1],b[6]
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([0][6])}
                       q:{pa[*] arg1([0][6])}
  . inference result - r:{pa[*] arg1([0][6])}
    (symbolic meaning) by right elements - r(b)
  . inference cycles taken - 5 cycles

(P2) query presented: p(a,a) ^ q(b)
  . phase allocation - a[1],b[6]
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([0][1])}
                       q:{pa[*] arg1([0][6])}
  . inference result - none
    (symbolic meaning) no result

(P3) query presented: p(U,U) ^ q(U)
  . phase allocation - U[1]
  . initial bindings - p:{pa[*] arg1([1][0]) arg2([1][0])}
                       q:{pa[*] arg1([1][0])}
  . inference result - r:{pa[*] arg1([1][1])}
                       p_q:a_gate[1]
    (symbolic meaning) by right elements - r(a)
                       by left elements - r(U)
                       unification results - {a/U}
  . inference cycles taken - 5 cycles

(P4) query presented: p(U,V) ^ q(b)
  . phase allocation - U[1],V[4],b[7]
  . initial bindings - p:{pa[*] arg1([1][0]) arg2([4][0])}
                       q:{pa[*] arg1([0][7])}
  . inference result - r:{pa[*] arg1([4][7])}
```

```
    (symbolic meaning) by right elements - r(b)
                       by left elements - r(V)
                       unification results - {b/V}
  . inference cycles taken - 5 cycles

(P5) query presented: p(U,b) ^ q(U)
  . phase allocation - b[1],U[6]
  . initial bindings - p:{pa[*] arg1([6][0]) arg2([0][1])}
                       q:{pa[*] arg1([6][0])}
  . inference result - none
    (symbolic meaning) no result
```

B.3.2 Experiment 6

Sample rule used:

$$p(X,Y) \wedge q(X,Y) \rightarrow r(X,Y)$$

Results:

```
(P1) query presented: p(a,b) ^ q(a,b)
  . phase allocation - a[1],b[6]
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([0][6])}
                       q:{pa[*] arg1([0][1]) arg2([0][6])}
  . inference result - r:{pa[*] arg1([0][1]) arg2([0][6])}
    (symbolic meaning) by right elements - r(a,b)
  . inference cycles taken - 5 cycles

(P2) query presented: p(U,V) ^ q(U,V)
  . phase allocation - U[1],V[6]
  . initial bindings - p:{pa[*] arg1([1][0]) arg2([6][0])}
                       q:{pa[*] arg1([1][0]) arg2([6][0])}
  . inference result - r:{pa[*] arg1([1][0]) arg2([6][0])}
    (symbolic meaning) by left elements - r(U,V)
  . inference cycles taken - 5 cycles

(P3) query presented: p(a,b) ^ q(U,V)
  . phase allocation - a[1],b[3],U[5],V[7]
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([0][3])}
                       q:{pa[*] arg1([5][0]) arg2([7][0])}
  . inference result - r:{pa[*] arg1([5][1]) arg2([7][3])}
    (symbolic meaning) by right elements - r(a,b)
                       by left elements - r(U,V)
                       unification results - {a/U,b/V}
  . inference cycles taken - 5 cycles

(P4) query presented: p(a,a) ^ q(b,b)
  . phase allocation - a[1],b[6]
```

```
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([0][1])}
                       q:{pa[*] arg1([0][6]) arg2([0][6])}
  . inference result - none
    (symbolic meaning) no result

(P5) query presented: p(a,b) ^ q(U,U)
  . phase allocation - a[1],b[4],U[7]
  . initial bindings - p:{pa[*] arg1([0][1]) arg2([0][4])}
                       q:{pa[*] arg1([7][0]) arg2([7][0])
  . inference result - none
    (symbolic meaning) no result

(P6) query presented: p(U,U) ^ q(V,V)
  . phase allocation - U[1],V[6]
  . initial bindings - p:{pa[*] arg1([1][0]) arg2([1][0])}
                       q:{pa[*] arg1([6][0]) arg2([6][0])
  . inference result - r:{pa[*] arg1([1,6][0]) arg2([1,6][0])}
    (symbolic meaning) by right elements - r(U,U)
                       unification results - {U/V}
  . inference cycles taken - 5 cycles
```

Appendix C

CASI User Manual

C.1 Introduction

A Connectionist Architecture for Symbolic Inference (CASI) 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 a symbolic style of inferences over these networks.

CASI has been implemented using a Prolog version of a connectionist simulator. This simulator was built based on Rochester Connectionist Simulator (RCS)[1] and modified to adopt CASI's specialised functions.

The simulator consists of three major modules, the input parser, the knowledge compiler, and the reasoner. Their tasks in CASI are as follows:

the input parser: performs parsing over input rules and facts with syntactic error checking and prepare inputs for the knowledge compiler;

the knowledge compiler: builds SPNs from the inputs given by the rule parser;

the reasoner: carries out inference either in forward chaining style or backward chaining style over the SPNs built.

[1] Goddard, N. H., Lynne, K. J. & Mintz, T. (1988), Rochester Connectionist Simulator, *Technical Report 233 (Revised)*, University of Rochester.

C.2 Getting Started

C.2.1 Files Required

The following files contain the code for the simulator:

load.pl: The simple file used to load all the necessary files. This loader file contains following clauses:

```
:- [interface],
   [parser],
   [compiler],
   [reasoner],
   [mesgs],
   [utils].
```

interface.pl: A simple text-based user interface; contains a top level command line menu handler.

parser.pl: The input parser; contains a description of the grammar for the input language in DCG syntax of Prolog. It also contains the input handler.

compiler.pl: The network builder; contains all functions which are needed to build SPNs from input rules and facts. Each SPN can be built to support either forward chaining or backward chaining or both.

reasoner.pl: The reasoner; contains query processors which performs forward or backward chaining style of inferences over the SPNs built.

mesgs.pl: Message utilities; provides message utilities used by the simulator.

utils.pl: Other utilities; contains a description of predicates which are not supported by standard Prolog such as DEC-10 Prolog.

C.2.2 The Prolog

Version 3.0 of SICStus Prolog[2] was used to implement the simulator. SICStus Prolog follows the mainstream Prolog tradition in terms of syntax and built-in predicates, and is largely compatible with DEC-10 Prolog and Quintus Prolog[3].

Following predicates are used from the list library of SICStus Prolog to support some modules of the simulator:

- The knowledge compiler uses following library predicates:
  - *append(?Prefix,?Suffix,?Combined)*: is true when *Combined* is the combined list of the elements in *Prefix* followed by the elements in *Suffix*.
  - *nth(?N,?List,?Element)*: is true when *Element* is the *N*th element of *List*. The first element is numbered 1.
- The reasoner uses following library predicates:
  - *append(?Prefix,?Suffix,?Combined)*: see above description.
  - *nth(?N,?List,?Element)*: see above description.
  - *nth0(?N,?List,?Element)*: is true when *Element* is the *N*th element of *List*, counting the first element as 0.
  - *remove_duplicates(+List,?Pruned)*: is true when *Pruned* is the result of removing all identical duplicate elements in *List*.
  - *memberchk(+Element,+List)*: is true when *Element* is a member of List, but *memberchk/2* only succeeds once.
  - *sublist(?Sub,?List)*: is true when *Sub* contains some of the elements of *List*.

The implementation of these predicates are in the file *utils.pl*. When a Prolog interpreter which does not contain these predicates is used, it is required to load this file together with the main simulator codes.

---

[2] *SICStus Prolog User's Manual*, Release 3 #2, Swedish Institute of Computer Science, June 1995.

[3] *Quintus Prolog Reference Manual*, Version 10, Quintus Computer Systems, Inc, Mountain, View, 1987.

The simulator has been also successfully tested using SWI-prolog (Version 2.1.9) under LINUX environment with only built-in predicates.

C.2.3 Starting the Simulator

When SICStus Prolog is used to run the simulator, the following sequence of operations is needed to start the simulator:

- To run Prolog.
  % sicstus
- To load the necessary modules.
  ?- [load].
- T run the simulator.
  ?- casi.

C.3 Preparing Input

CASI takes symbolic rules and facts in first-order Horn clause expressions as defined in Chapter 4. To confirm the syntax of each rule and fact given, CASI uses a formal grammar on which the parsing process based as follows:

```
input -> fact, ``.''
        | rule, ``.''
fact -> grounded_pred
grounded_pred -> atom, ``('', const_args, ``)''
const_args -> atom
        | atom, const_args
rule -> pred_list, sps, ``-->'', sps, pred
pred_list -> pred
        | pred, ``^'', pred_list
        | pred, sps, ``^'', sps, pred_list
```

```
pred -> atom, ``('', arg_list, ``)''
arg_list -> argument
        | argument, arg_list
argument -> atom
        | var
        | number
atom -> lowercase
        | lowercase, characters
var -> uppercase
        | uppercase, characters
number -> digit
        | digit, number
characters -> char
        | char, characters
char -> uppercase
        | lowercase
        | digit
        | underbar
uppercase -> A-Z
lowercase -> a-z
digit -> 0-9
underbar -> _
sps -> whitespace
        | whitespace, sps
whitespace -> TAB
        | LINE_FEED
        | CARIDGE_RETURN
        | SPACE
```

The input rules and facts are expected to be in a file before they are parsed by the input parser. The following example rules and facts are given to illustrate the syntax of them:

```
give(X,Y,Z) --> own(Y,Z).
buy(X,Y) --> own(X,Y).
own(X,Y) --> can_sell(X,Y).
p(a,X,X,Y) --> q(a,X,Y).
p(X,Y) ^ q(X,Y) --> r(X,Y).

give(john,mary,book1).
give(jane,susan,ball2).
buy(john,car7).
own(mary,ball1).
p(a,b,b,c).
p(a,b).
q(c,d).
```

All variables in rules are assumed to be universally quantified. Note that the comment lines are not supported by the parser currently. Although most of the features are self-evident, some points are to be noted regarding the input syntax as follows:

- According to the Prolog naming convention, any name beginning with an lowercase alphabetic character is assumed to be a constant. All names beginning with uppercase letters are variable names. Names of predicates should be atoms, i.e. should begin with lowercase letters. Capitalisation of names should be consistently used – for example, `const_a` and `const_A` are different constants;
- A full stop (.) indicates the end of one rule or fact. It also indicates that more input is to follow. The end of inputs is indicated by *end_of_file* in input files;
- The input is therefore affected by the insertion of blanks, tabs or newlines to the position where the grammar doesn't allow them.

C.4 Running the Simulator

C.4.1 The Command Line Menu

Once the simulator is started it gives the following command line menu:

```
TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|:
```

Any commands can be selected from the command line to interact with CASI. The behaviour of the simulator for each command is as follows:

- *l* command: loads rules and facts from an input file given;
- *sn* command: sets network type, either for forward chaining or for backward chaining or for both;
- *sv* command: sets the verbose mode either *on* or *off*. The default is *off*. When the verbose mode is *on*, the knowledge compiler gives detailed information about the knowledge encoding process;
- *c* command: builds SPNs from the input rules and facts loaded;
- *sk* command: lists rules and facts encoded in a symbolic format;
- *ss* command: shows current status of the networks encoded such as numbers of rules and facts, a type of SPNs, and values of status variables;
- *pn* command: print detailed information about a neuron by its name, which involves the name of the neuron, threshold, forward and backward links, etc.;
- *si* command: sets types of inference, either forward chaining or backward chaining;
- *i* command: processes inference by presenting a predicate;
- *x* command: exits the simulator and returns to the Prolog top level.

C.4.2 Load Input Rules and Facts

Rules and facts are loaded into CASI by selecting the *l* command from the menu as follows:

```
TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: l.

FILE NAME: testfile.

load: p(X,Y) --> q(X,Y)
load: p(a,b)

<MESG> LoadSuccess: rules and facts are loaded successfully
```

On receiving a filename from the user, the parser scans input rules and facts to prepare for the network construction. With respect to reading rules and facts from a file, the following needs to be noted:

- syntax error(s) in an input file results in aborting the input process. The entire input read is completely abandoned. The error in the file must be fixed and the file should be reloaded;
- minimal error checking has been implemented with respect to the form of the rules and facts, so as to leave some room for further extension. For example, the parser does not report an error if an input rule contains isolated UAGs in the antecedent even if this type of rule is not yet supported by the current version of CASI.

The handler for the on-line insertion of rules and facts is not yet implemented for the command line menu. The loading rules and facts therefore has to be done only from the file.

C.4.3 Building SPNs

Before building SPNs, the user can select the types of networks to be built. This is done by selecting the *sn* command. The type of networks can be either forward chaining only or backward chaining only or both as follows:

```
TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: sn.

SET UP NETWORK TYPE <f>: FOR FORWARD CHAINING ONLY
                    <b>: FOR BACKWARD CHAINING ONLY
                    <a>: BOTH FOR FORWARD AND BACKWARD CHAINING
|: a.
```

If the input contains only a set of facts, the knowledge compiler automatically detects this and constructs SPNs only for backward chaining. If the input contains rules and facts, one should select a type of network before constructing the networks. The default setting is forward chaining only.

The loaded input rules and facts are encoded into the corresponding SPNs by selecting the *c* command from the menu as follows:

```
TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: c.

  RULE TO BE ENCODED:     p(X,Y) --> q(X,Y)

<MESG> FwNetBuilt: network for forward chaining has been built

  RULE TO BE ENCODED:     p(X,Y) --> q(X,Y)

  FACT TO BE ENCODED:     p(a,b)

<MESG> BwNetBuilt: network for backward chaining has been built
```

Note that we have chosen the *a* command from the previous command line to build SPNs for both forward chaining and backward chaining. Thus, two types of SPNs were built separately. The SPN for the fact is built only for backward chaining because it is used only for backward chaining.

To observe detailed rule and fact encoding procedure, the *sv* command can be selected. This command enables the user turn *on* or *off* the verbose mode for rule encoding procedure:

```
TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: sv.

SET VERBOSE MODE <o>: VERBOSE MODE ON
                 <f>: VERBOSE MODE OFF
|: o.
```

If the verbose mode is *on* before the compilation commend, the following compilation command prints out all the details involved in the encoding procedure such as information about creating neuron elements, setting up links and intermediate outputs generated etc.. Since the simulator does not have an associated debugger, this mode is very helpful to obtain useful information for the debugging when the rule or fact encoding procedure fails.

C.4.4 Getting Current States

There are two commands which provide useful information about CASI's connectionist logical database (CLDB): the command showing rules and facts encoded and the command showing network status and system variables. The *sk* command gives the user a list of rules and facts loaded into CASI in a symbolic format. The *ss* command prints out the number of rules and facts encoded, the current type of network, the current type of inference and the current states of the verbose mode.

```
TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: sk.
```

```
    RULES ENCODED
    p(X,Y) --> q(X,Y)

    FACTS ENCODED
    p(a,b)

TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: ss.
    Input knowledge  : 1 rules  1 facts
    Network type     : forward and backward
    Network status   : built

    Inference type   : forward
    Verbose mode     : on
```

C.4.5 Performing Inferences

Two types of inferences are possible in CASI: forward chaining and backward chaining. Each type of inference is started by presenting or posing the predicate to CASI. This input predicate has the form, $pred\_name(t_1, t_2, \ldots, t_n)$, where t_i's are either constants or existentially quantified variables. Each input predicate corresponds to a short description of a real situation. For instance, the sentence, "Mary bought car7", may be represented by the predicate, *buy(mary, car7)*. Also "John owns something" by the predicate, *own(john,U)*. In forward chaining, presenting these predicates correspond to telling CASI the initial description to start inference. In backward chaining, on the other hand, these predicates are considered as queries which ask CASI proper answers. The above two example predicates are therefore interpreted as the sentences, "Did Mary bought car7?" and "What does John own?".

Forward Chaining

When SPNs are built and the type of inference is set to forward chaining, a chain of inference can be carried out by selecting the *i* command. By presenting a predicate after this command, CASI initiates a chain of inference. The detailed inference procedure can be summarised as follows:

1. the inference initiator receives a predicate from the user;
2. the reasoner set up the initial bindings by activating each filler appearing in the presented predicate and its corresponding argument nodes of the source predicate assembly;
3. get the first candidate nodes which are connected from the argument nodes of the source predicate assembly by forward links;
4. for each candidate nodes to be fired, process the following computations according to their type
   (a) sum inputs,
   (b) compare with the threshold,
   (c) activate it if the sum is greater than or equal to the threshold;
5. accumulate next candidate nodes from fired nodes;
6. continue from the step 4 until no more candidate node exists for the next step of inference;
7. the answer of inference is obtained from the states of the predicate assemblies activated during inference.

The system is case sensitive, in that predicate names and constants should begin with lowercase in exactly the same way as they were typed in initially. The following demonstrates the CASI's inference procedure when the predicate *p(a,b)* is presented to the example rule

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

```
TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: i.

ENTER QUERY: p(a,b).
```

```
Q:+ <p:pa[*]><a,[1]><p:arg1_r,[1]><b,[6]><p:arg2_r,[6]>
1:- (q:pa,0)(q:arg1_r,0)(q:arg2_r,0)
  + (q:pa[*])(q:arg1_r[1])(q:arg2_r[6])

    q:{pa[*] arg1([0][1]) arg2([0][6])}
```

The states of node after the symbol "Q:+" represent the initial bindings between the fillers and their corresponding argument nodes. The above initial bindings shows the status where the constant node $a[\phi]$ becomes active in-phase with $p{:}arg1_{right}[\phi]$ and the constant node $b[\phi]$ in-phase with $p{:}arg2_{right}[\phi]$. These in-phase oscillations represent the initial bindings $\{a/p{:}arg_1,\ b/p{:}arg_2\}$. The symbol "*" in *p:pa[*]* is used to represent activation of the predicate activator throughout the entire oscillation cycle. Whereas, a number stands for a specific phase in an oscillation cycle.

In the next oscillation cycle, these initial bindings are propagated to the target predicate assembly and consequently the q predicate becomes active in

$$q{:}\{pa[*],arg1([0],[1]),arg2([0],[6])\}.$$

Since the right element of the first argument node is active in-phase with the constant node a and that of the second argument node in-phase with the constant node b, this represents the result of inference $q(a,b)$.

Backward Chaining

If SPNs are built to support both forward chaining and backward chaining, CASI's inference type can be set to backward chaining mode by the *si* command. This command allows a user to choose either a forward chaining or a backward chaining style of inference from the menu as follows:

```
TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: si.

SET UP INFERENCE TYPE <f>: FORWARD CHAINING
                      <b>: BACKWARD CHAINING
|: b.
```

When the backward chaining inference mode is set, backward inference can be initiated in the same fashion as shown in the forward chaining by selecting the *i* command from the menu and giving a query predicate. The detailed procedure of backward chaining is slightly different from that of forward chaining as follows:

1. the inference initiator receives a query predicate from the user;
2. the reasoner set up the initial bindings by activating each filler appearing in the posed predicate and its corresponding argument nodes of the source predicate assembly;
3. get the first candidate nodes which are connected from the argument nodes of the source predicate assembly by backward links;
4. for each candidate nodes to be fired, process the following computations according to their type
   (a) sum inputs,
   (b) compare with the threshold,
   (c) activate it if the sum is greater than or equal to the threshold;
5. accumulate next candidate nodes from fired nodes;
6. continue from the step 4 until no more candidate node exists for the next step of inference;
7. if any fact assemblies becomes active during inference, this is considered as *yes* answer to the given query. Otherwise, the answer is considered as *no*. If the query predicate contains a variable filler, the value of the variable filler is obtained from the result unification between this filler and a specific argument node of a fact assembly activated.

The following is an example procedure of backward chaining when the query predicate $q(a,U)$ is posed to the SPNs encoding the following rule and fact

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

```
TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: i.

ENTER QUERY: q(a,U).

      create neuron X  pi_btu  1
  Q:+ <q:pa,[*]><a,[1]><q:arg1_r,[1]><X,[6]><q:arg2_l,[6]>
  1:- (p:pa,0)(p_f1:a_gate,0)(p:arg1_r,0)
      (p:arg2_l,0)
    + (p:pa[*])(p:arg1_r[1])(p:arg2_l[6])

      p:{pa[*] arg1([0][1]) arg2([6][0])}

  2:- (p_f1:fa,2)(p_f1:a_gate,0)(p_f1_bpmt:p_f1_1_l,1)
      (p_f1_bpmt:p_f1_1_r,1)(p_f1:b_gate,0)(p_f1_bpmt:p_f1_2_l,1)
      (p_f1_bpmt:p_f1_2_r,1)(fccm:p_f1,1)(fccmv:p_f1,0)
    + (p_f1:a_gate[1])(p_f1:b_gate[6])


  3:- (p_f1:fa,1)(p_f1_bpmt:p_f1_1_l,0)(p_f1_bpmt:p_f1_1_r,0)
      (p_f1_bpmt:p_f1_2_l,0)(p_f1_bpmt:p_f1_2_r,0)(fccm:p_f1,0)
      (a,1)(b,1)
    + (p_f1_bpmt:p_f1_1_r[1])(p_f1_bpmt:p_f1_2_l[6])(p_f1_bpmt:p_f1_2_r[6])


  4:- (p_f1:fa,0)(a,0)(b,0)
      (p_f1:arg1_r,0)(p_f1:arg2_l,0)(p_f1:arg2_r,0)
    + (p_f1:fa[*])(a[1])(b[6])
      (p_f1:arg1_r[1])(p_f1:arg2_l[6])(p_f1:arg2_r[6])

      p_f1:{fa[*] arg1([0][1]) arg2([6][6])}
         [a,b]
           - a[1] b[6]
           - p_f1:a_gate[1] p_f1:b_gate[6]

  5:- (p_f1:a_gate,0)(p_f1:b_gate,0)
    + (p_f1:a_gate[1])(p_f1:b_gate[6])


  6:- (a,1)(p_f1_bpmt:p_f1_1_r,0)(b,1)
      (p_f1_bpmt:p_f1_2_r,0)
    + (p_f1_bpmt:p_f1_1_r[1])(p_f1_bpmt:p_f1_2_r[6])


  7:- (a,0)(b,0)(p_f1:arg1_r,0)
      (p_f1:arg2_r,0)
    + (p_f1:arg1_r[1])(p_f1:arg2_r[6])
```

After 4 oscillation cycles, the fact assembly p_{f_1} became active in

$$p_{f_1}\text{:}\{fa[*],\ arg1([0],[1]),\ arg2([6],[6])\}.$$

This represents the affirmative answer to the given query predicate. The value for the variable filler (U) is obtained from the result of unification, $\{a/U\}$. The following additional outputs are generated by the user interface

```
- a[1],b[6]
- p_f1:a_gate[1],p_f1:b_gate[6],
```

provides the user more detailed information about the symbolic constant arguments of the fact and the relationship between these arguments and the fillers of the posed predicate. The first line gives the names of two constant argument of the fact and their activation states and the second line the states of their associated gate nodes. The result of unification, $\{a/U\}$, is obtained by observing in-phase activation between p_{f1}*:b_gate[6]* and p_{f1}*:*$arg2_{right}$*[6]* and that between *U[6]* and p_{f1}*:*$arg2_{left}$*[6]*.

C.4.6 Printing Neurons

The information about the internal structure and status of a neuron element can be printed by selecting the *pn* command from the command line menu and giving a name of a neuron element. Each neuron is internally represented as a Prolog term

```
neuron(Name,[
        type: Type,
        potential: Threshold,
        flink_to: [List of pairs of to-neuron and delay],
        flink_from: [List of links],
        blink_to: [List of pairs of to-neuron and delay],
        blink_from: [List of links] ]).
```

Based on this internal structure of neuron, the *pn* command prints out all the detailed information about the neuron element including the link information for backward and forward chaining as follows:

```
TOP LEVEL COMMAND < l>: LOAD RULES AND FACTS   <sk>: SHOW RULES AND FACTS
                  < c>: KNOWLEDGE COMPILATION  <ss>: SHOW STATUS OF NET
                  < i>: START INFERENCE        <sn>: SET NET TYPE
                  <pn>: PRINT ONE NEURON       <si>: SET INFERENCE TYPE
                  < x>: EXIT                   <sv>: SET VERBOSE MODE
|: pn.

NEURON NAME: p:arg1_l.

  [NEURON] Name:p:arg1_l Type:pi_btu Potential:1 State:0 Value:0
           FWlink_to:[(q:arg1_l,0)]
           FWlink_from:[]
           BWlink_to:[(fccmv:p_f1,0),(p_f1:a_gate,0),(p_f1_bpmt:p_f1_1_l,1)]
           BWlink_from:[input((q:arg1_l,1000))]
```

C.4.7 Setting the Number of Phase

Initially, the number of phases per clock cycle is set to a default value of 10 in the file of *reasoner.pl* by the clause *maximum_phase(10).* This value can be changed only by modifying the value in the source program and reloading it.