Order Number 9130380

Neural Petri nets and their applications to combinatorial games

Tien, Jonathan Yeh, Ph.D.

City University of New York, 1991

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

A

NEURAL PETRI NETS AND THEIR APPLICATIONS TO COMBINATORIAL GAMES

by

JONATHAN YEH TIEN

A dissertation submitted to the Graduate Faculty in Mathematics in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1991

This manuscript has been read and accepted for the Graduate Faculty in Mathematics in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

5/10/91
Date

Michael Anshel
Chair of Examining Committee

May 10, 1991
Date

Executive Officer

Michael Anshel

Harvey Cohn

Burton Randol

Supervisory Committee

The City University of New York

Abstract

NEURAL PETRI NETS AND THEIR APPLICATIONS TO COMBINATORIAL GAMES

by

JONATHAN YEH TIEN

Advisor: Professor Michael Anshel

Neural Petri Nets (NPN), a new mathematical modeling structure is defined in this paper. The goal of this study is to investigate the manner in which neural Petri nets automata can be applied to play combinatorial games under varying informational constraints. The research focuses on positional games of a very simple type but with increasing complex rules of play. Algorithms are developed for simulating these automata and analysis of certain special cases are presented.

Acknowledgements

I would like to express my deep appreciation to my dissertation advisor Professor Michael Anshel. His academic guidance and constant encouragement over the years made the completion of this study possible. I also would like to thank Professor Harvey Cohn and Burton Randol for their valuable feedbacks, and William Breitmayer for proof reading the final version.

Table of Contents

Chapter 1

INTRODUCTION

There are primarily two kinds of computers in this world at an abstract level: one is the modern digital computer which is also called Von Neumann machine because is based on Von Neumann model; the other one is human brain. People seem to be amazed how powerful the modern digital computers are, and do not realize that biological computers - the brain and the nervous systems of human beings, have existed for millions of years, and they are extremely effective in processing sensory information and controlling interactions of each other with their environment. Tasks such as reaching for an apple, recognizing a shape or remembering things associated with a particular event are computations just as much as addition and multiplication are. The fact that biological computation is so effective suggests that it may be possible to obtain similar capabilities in artificial devices based on the design principles of neural system.

The neural nets(NN) theory builds on a long history of efforts to capture the principles of biological computation in mathematical models, which began with the pioneering study of neurons as logical devices by McCulloch and Pitts in

1943. Artificial neural net models have been studied for many years in the hope of achieving human-like performance in the fields of speech and image recognition. These models are composed of many nonlinear computational elements operating in parallel arranged in the patterns reminiscent of biological neural nets.

On the other hand, as a later comer, Petri net theory has developed considerably from its beginnings with Dr. Petri's 1962 Ph.D. dissertation. After more than twenty years of deep investigation of the theoretical problems associated with PN, today PN can be regarded as a formal structure which a well-assesed theory has been developed and a wide range of application fields have been identified. Because of its design, PN is an very effective modeling tool for the description and analysis of concurrency and synchronization in parallel systems such as the digital computers.

Many extensions have been added to the basic PN model in order to facilitate the use of PN in different application fields. In particular, timed PN models can be used for performance analysis of computer systems by introducing the notion of time. When random variables are used to specify the time behavior of the model, timed PN is called stochastic PN(SPN). The recent study of SPN by Molloy([Mol85], [Mol87]) showed that SPN are isomorphic to continuous-time Markov chains, and an algorithm was constructed to do the

actual conversion. By using this scheme, we can quantitatively analyze computing system performance issues, such as response time delay and system capacity.

Although the Von Neumann computer is effective at number crunching, but it performs poorly in tasks that emulate the natural information processing which humans handle routinely using their computer - brain. Since biological brains are working examples of massively parallel, densely interconnected, self-organizing computational networks, they become an ideal candidate in future computer architecture. Recent technological advances in VLSI(very large scale integration) and computer aided design mean that it is now much easier to build massively parallel machines. This has contributed to a new wave of interest in models of computation that are inspired by neural nets rather than the formal manipulation of symbolic expressions.

In 1985, Zargham and Tyman [ZT85] examined the basic elements of brain, the neurons, the dynamics of the neural processing, and proposed a model called Neural Petri Nets based on the Petri Nets concept, to serve as a template for computer architecture. The neural Petri nets(NPN) they defined are primarily an extended PN in the field of neurophysiology as a study for the combination of the two models. More recent research work in this arena has been done by Habib and Newcomb [HN90] utilizing the timed Petri nets to

model the digital neuron type processors.

Inspired by all these exciting developments, this paper proposes a definition of neural Petri net (NPN) with its natural extension, stochastic neural Petri nets(SNPN) models as further study in this area by combining the neural nets and stochastic Petri nets concepts. Because of the definition, NPN inherit the advantages from both NN and PN. Furthermore, some restrictions are also discussed on NPN to make it more useful. Future application of such models would be in the fields of neural computing, neurophysiology, artificial intelligence, etc.. In the arena of theory and analysis, we show that there is a fast algorithm with polynomial complexity to find performance throughput bounds for SNPN, thus show its value in theoretical computer science and mathematical modeling and analysis. On the other hand, we utilize NPN to construct intelligent algorithms to play positional games. This gives audiences a demonstration how NPN can be applied in a real application.

Chapter 2

NEURAL NETS

1. Background and Definition

Neural nets are mathematical structures developed to model biological computation processes such as the natural information processing that human brains handle daily. The most basic processing elements are nerve cells or neurons. The principal architectural feature of these neurons relevant to its processing characteristics are those of layering, modularity, dense interconnections, and distribution of input processing. Neurons are complex, but even a highly simplified model of neuron, when it is connected with others in an appropriate network, can do significant computation.

A biological neuron receives information from other neurons through synaptic connections and passes on the signals to as many as thousands other neurons. This signal is known as a post synaptic potential (PSP). There are two types of PSPs: excitatory post synaptic potential (EPSP) and inhibitory synaptic potential (IPSP). An EPSP causes the receiving neuron to have more positive value and to increase the probability of generating a new potential.

On the other hand, an IPSP has the opposite effect.

Artificial neural net models [Lip87] attempt to achieve good performance via dense interconnection of simple computation elements. In this aspect, neural net structure is based on our present understanding of biological nervous systems. Instead of performing a program of instructions sequentially as in the Von Neumann computer, neural net models explore many competing hypotheses simultaneously using massively parallel nets composed of many computational elements connected by links with variable weights.

Figure 1. shows the structure of a neural net element - the neuron. A neuron y has n input neurons $x_1, x_2, ..., x_n$ in this case. Each input neuron has either value "+1" to represent EPSP or "-1" to represent IPSP. The output of neuron y is computed by forming a weighted sum of n inputs and passing the result through a filter function. The weights $\{w_i\}$ determines the input distribution. For example, if we want decrease the stength of the input from x_i, w_i can be set to a smaller number so $w_i x_i$ will play a smaller role in the total input before being summed into the filter function. Topologically, this can also be thought as locating neuron x_i very far from neuron y so the effect becomes minimal.

A node is characterized by an internal threshold θ and by the type of filter

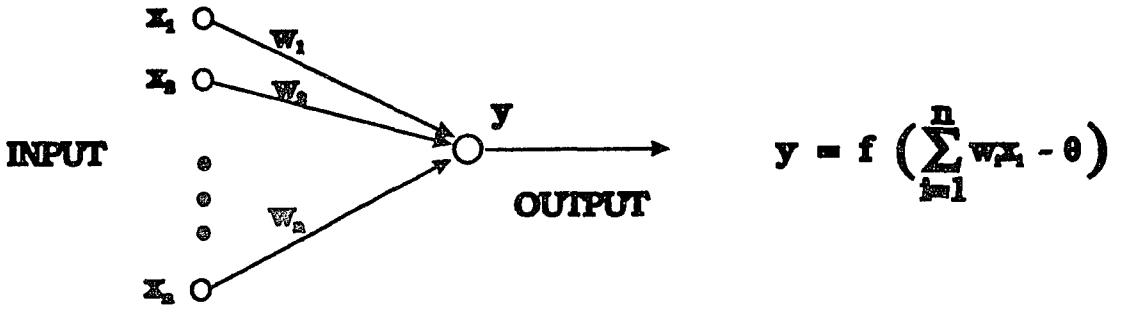


Figure 1. A typical neuron and its computational operation.

functions associated with it. The internal threshold θ gives the characteristic of this node. If θ is large, it certainly needs large input value to balance it out (to activate it). In the biological sense, this can be thought of as a "lazy" (or "heavy") neuron, and it needs more pus from its input neurons to generate a post synaptic potential. The final output value of a node is calculated by filter functions which in general are nonlinear. Figure 2. illustrates three most common types of nonlinearities used as filter functions: binary, threshold logic, and sigmoid nonlinearities. More complex filter functions may include integration or other types of time dependencies, or more complex mathematical operations than summation.

As an example, assume there are only two input nodes to node y, x_1 = 1(EPSP) and x_2 = -1(IPSP). The weights are assigned as $w_1 = 1$, $w_2 = 0.5$, threshold $\theta = 0$, and a binary function is used as the filter function. The computation is y = f(1 - 0.5) = 1. As one can see, the final output of node y represents an EPSP since the input from x_2 is surpressed by the weight w_2.

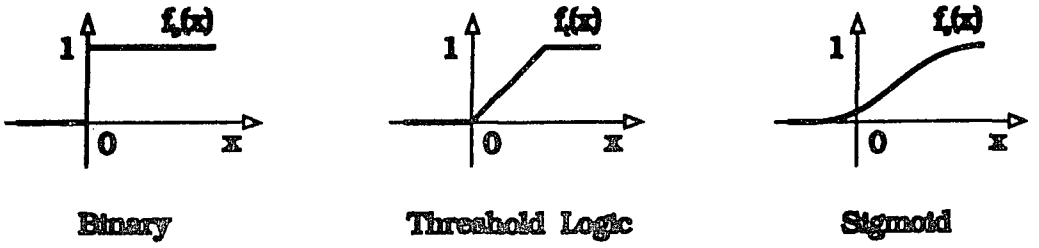


Figure 2. Three commonly used nonlinearity functions

A neural net model is specified by the net topology, node characteristics, and learning rules. These learning rules specify an initial set of weights and indicate how weights should be adapted during use to improve performance. Both design procedure and learning rules are the topic of much current research.

Most neural net algorithms also adapt connection weights across time interval to improve performance based on the current results. Adaptation or learning is a major focus of neural net research. The ability to adapt and continue learning is essential in areas such as speech recognition where training data is limited and new speakers, new words, new phrases and new environments are continuously encountered.

Development of detailed mathematical models began with the pioneering investigations of neurons as logical devices by McCulloch and Pitts more than 40 years ago. In the 1960's Rosenblatt and Widrow created "adaptive neurons" and simple networks that can learn. More recent work by Hopfield[Hop82],

Grossberg, and others attempted to model more closely the behavior of real neurons in computational networks and to develop the mathematics and architecture for extracting features from patterns, for classifying patterns and for studying "associative memory". In "associative memory" structure, pieces of a stored memory can be used to retrieve the entire memory.

2. Hopfield Net

Many neural net models were built through the years of research and development. As an example, one of the well known models, the Hopfield net illustrates neural net design.

This net has n nodes containing binary filter functions and binary inputs labeled x's and outputs labeled y's taking on values 1 and 0. The output of each node is fed back to all other nodes via weights denoted w_{ij}. Given several pre-defined patterns, which are called exemplars $\{x^1, x^2, ..., x^M\}$, for the Hopfield net to memorize, the weight assignment depends upon the initial exemplar pattern values. An unknown binary input pattern is applied at time zero and the net then iterates until convergence, that is when the node outputs remain unchanged. The output is the pattern produced by these nodes after convergence. The following is the algorithm describing the net operation.

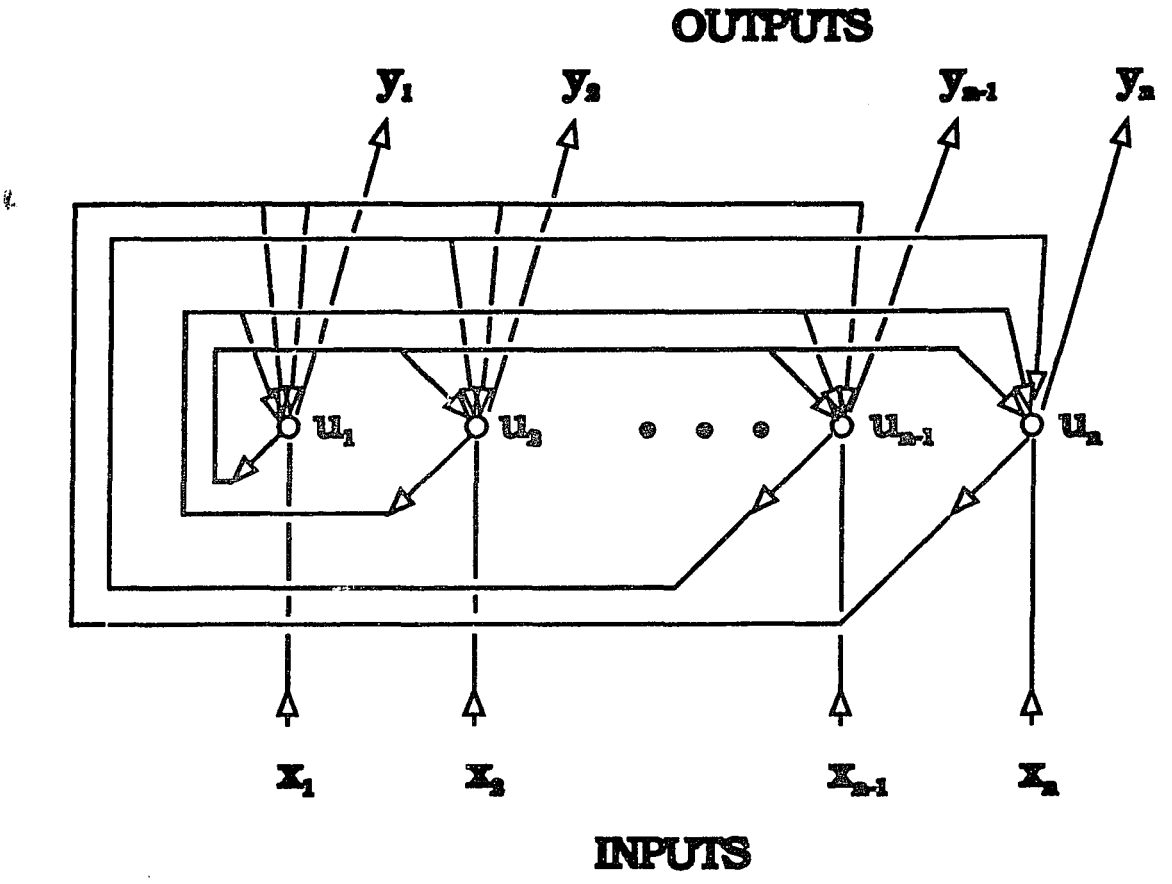


Figure 3. A Hopfield neural net.

Hopfield Net Algorithm

Step 1. Assign Connection Weights

$$w_{ij} = \begin{cases} \sum_{i=1}^{M} (2x_i^s - 1) * (2x_j^s - 1), & i \neq j \\ 0, & i = j, \quad 1 \leq i, j \leq n \end{cases}$$

In this formula w_{ij} is the connection weight from node i to node j. $\{x^s_k\}$,

which is binary, can be 1 or 0 is exemplar x^s. M exemplars and n nodes are assumed.

Step 2. Initialize with Unknown Input Pattern

$$\mu_i(0) = x_i, \quad 1 \le i \le n$$

Where $\mu_i(\tau)$ is the output node i at time τ and x_i which can be 1 or 0 is element i of the input pattern.

Step 3. Iterate Until Convergence

$$\mu(\tau+1) = f\left(\sum_{i=1}^{n} w_{ij}(2\mu(\tau) - 1)\right), \qquad 1 \le i \le n$$

The function f is the binary filter function from Figure 2. . The process is repeated until node outputs remain unchanged with further iteration. The outputs of the nodes then represent the exemplar pattern that best matches the unknown input.

As indicated in the above algorithm, weights are set using the given exemplar patterns for all classes first. Then an unknown pattern is imposed on the net at time zero by forcing the output of the net to match the unknown pattern. Following this initialization, the net iterates in discrete time steps

using the given formula. The net is considered to have converged when outputs no longer change on successive iterations. The pattern specified by the node output after convergence is the net output.

Hopfield and the others have proven that this net converges when the weights are symmetric ($w_{ij} = w_{ji}$) and node outputs are updated asynchronously using the formulas above.

3. Associative Memory

We know that a computer have physical memory which can store information and return it to a user when recalled. However, at the time of retrieval, the user must give the computer an exact information as what the user wants. Any error will fail to retrieve the stored memory. Many of us encounter this type of phenomena daily. An ideal memory should have some level of error correction ability. Associative memory is a structure that allows users to retrieve a stored item from memory by providing only partial information.

As an well-known example, the Hopfield net is often used for constructing associative memory (see Figure 3.). In R. P. Lippmann's [Lip87] example, eight patterns: 0, 1, 2, 3, 4, 6, ♦, 9 are chosen as exemplars. Each image is

divided in 120 pixels and each is given a value 1 if this pixel is black, 0 if not. So a total 120 nodes are constructed in Hopfield net to accommodate these inputs. According to the above algorithm 14400 weights ware trained to recall these patterns. The pattern "3" was corrupted by randomly reversing each bit independently from 1 to 0 and vice versa with a probability of 0.25, then applied at the time zero to the net. Lippmann showed that the net has converged after six iterations and the digit "3" is produced by the net as output.

While the Hopfield net is very useful, it has some limitations. Questions are still been researched like the number of patterns the net can remember, i.e the capacity of the net; or if the net will converge to the pattern one wants or converges to an unknown pattern, and under what condition. The memory capacity problem has been studied extensively. Mceliece et al[MPRV87] determined that the upper bound for Hopfield net capacity is O(n/(4log n) where n is the size of the net. This means that if m fundamental memories are chosen at random, the maximum asymptotic value of m in order that every one of the m memories is exactly recoverable is n/(4log n).

Chapter 3

PETRI NETS

Petri nets (PN) [Pet81] are an effective modeling tool for the description and analysis of concurrency and synchronization in parallel systems exhibiting the cooperative actions of different entities. Petri nets were introduced by C. A. Petri in 1962. Theoretical problems associated with Petri nets have been deeply investigated since then. Today Petri nets can be regarded as a formal structure for which a well-assessed theory has been developed and a wide range of application fields have been identified.

The success of PN is mainly due to the simplicity of the basic mechanism of the model, which is, however, paid for with the complexity of describing large systems. Many extensions have been added to the basic PN model by several authors in order to facilitate the use of PN in different application fields. Many authors extended PN models by introducing the notion of time: timed PN models can be used to analyze the quantitative performance of systems. When random variables are used to specify the time behavior of the model, timed PN are called stochastic Petri nets (SPN). It can be shown that SPN are, under certain conditions, isomorphic to homogeneous Markov chains (MC).

The use of SPN [Mol87] as a modeling tool for evaluating computer systems (especially multiprocessor systems) is extremely attractive. The capability of describing simultaneously both concurrency and synchronization seems very interesting when the needed level of detail makes queuing network models inadequate. Moreover, it is possible to obtain the state transition rate diagram of the associated MC automatically from the SPN description of the system. The latter can often be derived from a simple analysis of the system behavior, whereas the direct characterization of the system in terms of a MC can be by far more complex.

The structure of a standard PN is a bipartite graph that comprises a set of places P, a set of transitions T, and a set of directed arcs A. In the graphical representation of PN, places (the nodes of PN) are drawn as circles and transitions as bars. Arcs connect transitions to places and places to transitions. A place is an input to a transition if an arc exits from the place to the transition and is an output of a transition if an arc exits from the transition to the place. The set of arcs can be partitioned into the sets of transition input arcs A_I and transition output arcs A_O. Tokens which are contained in the places are drawn as black dot. The state of a marked PN is defined by the number m_i of tokens contained in each place $p_i \in P$, and the PN state is usually called the Petri net marking: $M = (m_1, m_2, ..., m_n)$.

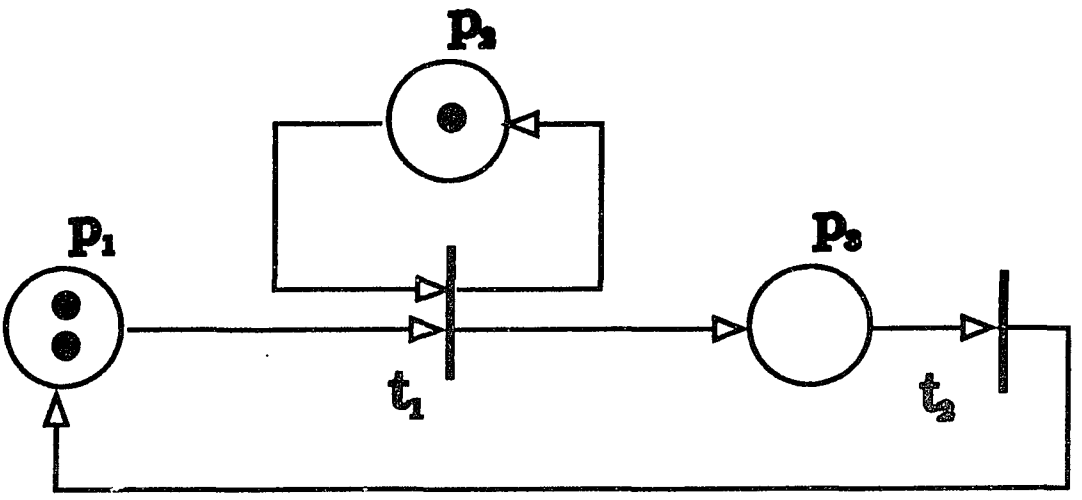


Figure 4. . A marked PN with 2 transitions and 3 places. The marking is (2, 1, 0).

A Petri net executes according to the following rules.

a. A transition is enabled when all of its input places contain at least one token.

b. An enabled transition can fire. And when it does fire, one token is removed from each input place and placing one token in each output place of the transition.

c. Each firing of a transition modifies the distribution of tokens in the places and thus produces a new marking for the PN.

Using the PN in Figure 4. as an example, transition t_1 is enabled at this

time since there are tokens in places p_1 and p_2. After t_1 fires, a token in p_1 has been removed, and both places p_2 and p_3 contain 1 token.

The formal definition of a marked PN can be given as the following:

$PN = \{P, T, A, M_0\}$, where,

$P = \{p_1, p_2, ..., p_n\}$, is a finite non-empty set of labelled places.

$T = \{t_1, t_2, ..., t_m\}$, is a finite non-empty set of labelled transitions.

$A \subset \{P \times T\} \cup \{T \times P\}$, is a relation. It represents a set of arcs where each arc is either from a place to a transition or from a transition to a place.

$M^0 = \{m_1^0, m_2^0, ..., m_n^0\}$, is a set of non-negative numbers representing the initial marking, i.e. number of tokens in each place.

Let $\#(p_a, t_b)$ equal to 1 if an arc from place p_a to transition t_b exits and 0 otherwise; similarly $\#(t_a, p_b)$ is equal to 1 if an arc from transition t_a to place p_b exits and 0 otherwise. Then a transition t_j is enabled if

$$m_i \geq \#(p_i, t_j) \qquad \text{for } \forall\, p_i \in P,$$

and the result of the firing t_j is a new marking $M' = (m_1', m_2', ..., m_n')$ defined as

$$m_i' = m_i - \#(p_i, t_j) + \#(t_j, p_i) \qquad \text{for } \forall\, p_i \in P,$$

In the marked PN in Figure 4. , the initial marking $M^0 = (2,1,0)$ enables only transition t_1. A new marking M' = (1,1,1) is reached after the firing of t_1. In this case, both transition t_1 and t_2 are enabled and can fire independently. The firing of t_1 results a new marking M" = (0,1,2) and firing of t_2 makes the system return to the initial marking.

The PN execution allows a sequence of markings $\{M^1, M^2, ...\}$ and a sequence of firing transitions $\{t_1, t_2, ...\}$ to be defined. The firing of t_1, enabled in M^0, changes the marking of the PN from M^0 to M^1 and so on. A marking M' is said to be reachable from marking M if there exists a sequence of transition firings that moves the PN state from M to M'. Furthermore, the reachability set $R(M^0)$ of a PN is defined as the set of markings that are reachable from M^0. Extensive study has been performed in this area, since reachability is one of the most important problems for PN analysis. Several author have published articles on the complexity and decidability of the reachability set of PN [Lam88]. It has been shown that the reachability problem is decidable but the complexity of the problem is at least exponential.

Probabilistic performance models try to represent the behavior of complex deterministic systems by means of stochastic processes. The possibility of merging the capability of PN to describe synchronization and concurrency with a stochastic model is a very attractive way to obtain performance estimates of

complex computing systems.

Stochastic Petri nets(SPN) are obtained by associating each transition in a PN an exponentially distributed random variable that expresses the delay from the enabling to the firing of the transition [BG85]. On the other hand, given an SPN, the associated PN is obtained by disregarding the transition timing. Formally, a Stochastic Petri nets can be defined as the following:

$$SPN = (P, T, A, M^0, \Lambda)$$

where P, T, A, and M^0 are as in (1) and $\Lambda = \{\lambda_1, \lambda_2, ..., \lambda_m\}$ is the set of finite transition rates associated with the PN transitions for exponentially distributed firing times. These values may be obtained by determining the average firing delay ($d_i=1/\lambda_i$) when the transition is enabled in isolation from the rest of the Stochastic Petri net.

Molloy showed that, due to the memoryless property of the exponential distribution of firing delays, SPN are isomorphic to continuous-time Markov chains. In particular, a bounded SPN (namely $\exists\, K \in N$, such that $|M'| \leq K$, for $\forall M' \in R(M^0)$) can be shown to be isomorphic to a finite MC. The MC associated with a given SPN can be obtained following these rules:

1. The MC state space S corresponds to the reachability set $R(M^0)$ of the SPN ($M^i \leftrightarrow i$).

2. The transition rate from state i (corresponding to marking M^i) to state j (corresponding to marking M^j) is

$$q_{ij} = \sum_{k \in T_{ij}} \lambda_k$$

where T_{ij} is the set of transitions enabled by marking M^i, whose firing generates M^j.

By using these rules, it is possible to write an algorithm that uses the SPN description to automatically derives state transition rate matrix of the isomorphic continuous time MC.

One of the most important properties of SPN is ergodicity of markings [FN89]. The marking process is said to be ergodic if there is a finite positive vector M^* such that:

$$M^* = \lim_{\tau \to +\infty} E(M(\tau)) = \lim_{\tau \to +\infty} \int_0^\tau \frac{M(s)}{\tau} < +\infty$$

where $M(\tau)$ denotes the SPN marking at time τ and M^* is the steady state mean marking.

Furthermore, if an SPN is ergodic, it is possible to compute the steady state probability distribution of markings of the Markov chains. Therefore, by using MC analysis, it is possible to obtain quantitative estimates of SPN behavior from the steady state distribution.

Chapter 4

NEURAL PETRI NETS

1. Definitions

Two distinctive models were discussed in previous sections so far: neural net and Petri net. They are very powerful tools, but in different application fields. The question that comes to mind is would be possible to build a model which combines the advantages from both neural and Petri nets? To answer this question, M. R. Zargham and M. Tyman[ZT85] introduced the term neural Petri net (NPN) in 1985 at the International Workshop on Timed Petri Nets. In that paper, they analyzed the neuron structure in detail and map it the elements in Petri nets. A NPN definition was given to describe neuron activities in a PN-like structure.

After studying many publications in the related field, we provide here a different approach to defining neural Petri net (NPN) and its natural extension - stochastic neural Petri nets (SNPN). The definition of these models is developed by combining both neural nets and stochastic Petri nets models.

The following is a formal definition of NPN:

A marked Neural Petri Net with initial marking M^0 is a system NPN = $(P, T, A, H, W, \Theta, V, M^0)$, where:

$P = \{p_1, p_2, ..., p_n\}$,

is a finite non-empty set of labelled places and potentially containing tokens.

$T = \{t_1, t_2, ..., t_m\}$,

is a finite non-empty set of labelled transitions.

$A \subset A_I \cup A_O$,

is a relation. It represents a set of arcs where each arc is either from a place to a transition or from a transition to a place. $A_I \subset \{P \times T\}$ is the set of input arcs (from a place to a transition) and $A_O \subset \{T \times P\}$ is the set of output arcs (from a transition to a place).

$W = \{w_{ij}\}$, $1 \leq i \leq n$ and $1 \leq j \leq m$,

is a set of weights associated with input arcs A_I.

$H = \{h_1, h_2, ..., h_m\}$,

is a set of filtering functions associated with transitions T. They determine the values for output tokens and are binary functions.

$\Theta = \{\theta_1, \theta_2, ..., \theta_m\}$,

is the set of thresholds for H.

V: is a set of values $\{v_1, v_2\} = \{1, 0\}$ for the tokens, which represent the two types of post synaptic potentials of neurons: "1" for excitatory post synaptic potential(EPSP) and "0" for inhibitory post synaptic potential(IPSP).

$M^0 = (m^0_1, m^0_2, ..., m^0_n)$,

is the initial NPN state called initial marking, i.e. number of tokens in each place. Furthermore, a marking $M = M_\bullet \oplus M_o$, where $M_\bullet$ the marking of value "1" tokens and M_o is the marking for value "0" tokens.

In the above model places represent neurons; transitions represent the process of information transaction between the neurons; and tokens in the places represent the information (synaptic potential) in the neurons. Since post synaptic potentials can be either excitatory or inhibitory, there are two classes of tokens distinguished by values: value "1" (using dot "•" in graphic presentation) representing excitatory post synaptic potential (EPSP) and value "0" and (using "o" in graphic presentation) representing inhibitory post synaptic potential (IPSP).

The firing of a transition corresponds to the generation of an action potential in the associated output places. As a consequence of the firing of a transition, tokens appear in each output place of the transition. The value of these tokens

is determined by the filter function H which is associated with transitions T. Similarly to PN definition, let $\#(p_a, t_b)$ be 1 if there is an arc from place p_a to transition t_b and 0 if not; let $\#(t_a, p_b)$ be 1 if there is an arc from place t_a to transition p_b and 0 otherwise and $M = \{m_1, m_2, ..., m_n\}$ is the current marking. Then, a transition $t_j \in T$ is said to be enabled if

$$m_i \geq \#(p_i, t_j), \qquad \text{for } \forall p_i \in P$$

and the result of firing t_j is a new marking $M' = \{m_1', m_2', ..., m_n'\}$, where

$$m_i' = m_i - \#(p_i, t_j) + \#(t_j, p_i) \qquad \text{for } \forall p_i \in P$$

In other words firing transition t_j removes a token from all input places and puts a new token in all output places. The value of new tokens is calculated by the formula:

$$v = h_j \left(\sum_{i=1}^{n} w_{ij} * (2x_i - 1) - \theta_j\right)$$

where, $\{x_i\}$ are the values of the tokens transition t_j removed from places $\{p_i\}$.

Figure 5. represents an example of NPN. Besides places $P = (p_1, p_2, p_3, p_4, p_5)$, transitions $T = (t_1, t_2)$ and arcs A are indicated in the graph, the filtering function $H = (h_1, h_2)$ is binary with threshold $\Theta = (0, 0)$, i.e.

$$h_i(x) = \begin{cases} 1, & \text{if } x \geq 0, \\ 0, & \text{otherwise,} \end{cases} \qquad \text{for } i=1, 2.$$

Weight matrix W is:

$$W = \begin{pmatrix} 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

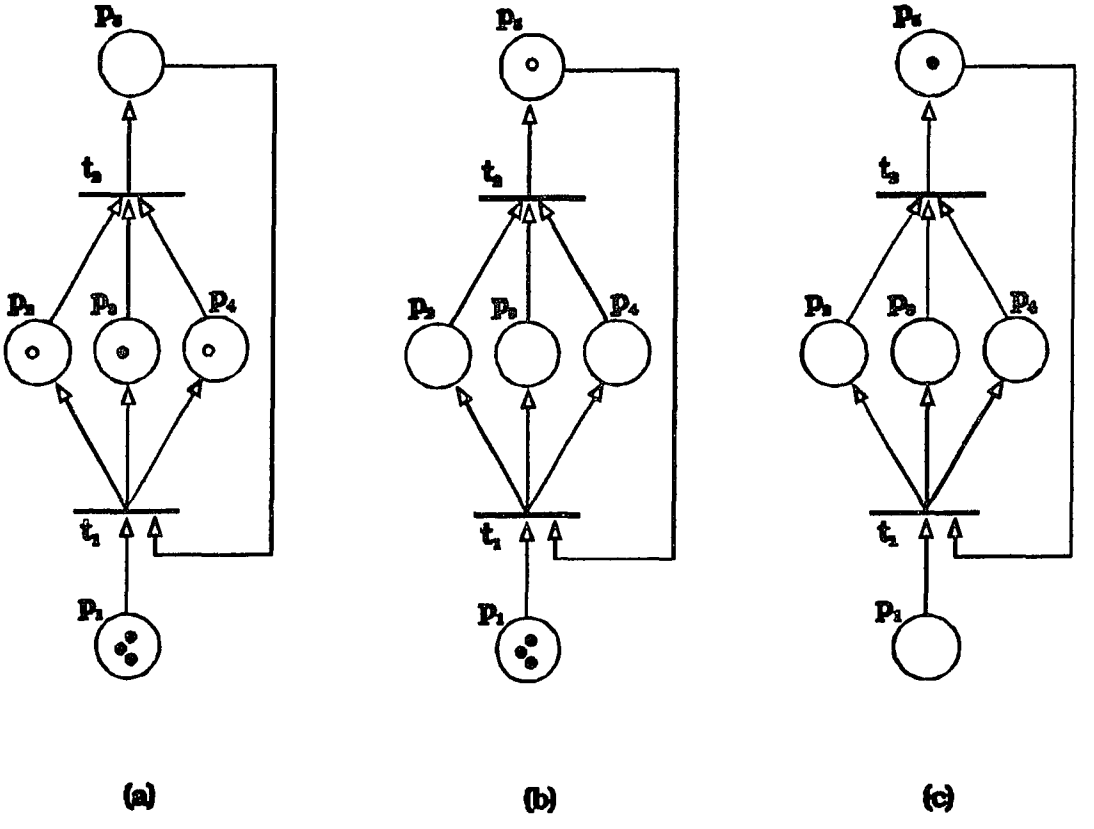


Figure 5. A marked NPN with 2 transitions and 4 places.

There are two token values V = {1, 0}. Tokens with value "1" are represented by dot and tokens with value "0" are represented by circle. A marking of

a place p is m = (a, b) with the first coordinate a representing the number of tokens of value "1" and second coordinate representing the number of tokens of value "-1". The initial marking (indicated in sub-graph (a)) is is $M^0 = (m^0_1, m^0_2, m^0_3, m^0_4, m^0_5) = ((3,0), (0,1), (1,0), (0,1), (0,0))$.

Sub-graph (a) is the initial status with enabled transition t_2, and (b) is the result of firing t_2. The firing of t_2 removes the tokens from p_2, p_3, p_4 and adds a token in p_5. The token value is determined by

$$h_2 (W\times(2P-1) - \Theta)_2 = h_2 (-1 +1 -1) = 0.$$

Now, transition t_1 is enabled. Firing t_1 puts a token in p_2, p_3, p_4 with value "1" determined by $h_1 (W\times(2P-1) - \Theta)_1 = h_1 (-1*(-1)) = 1$, and removes one token from place p_1. Transition t_2 is now enabled. Repeating this process, the final state will be reached when no transition is enabled. This state is represented by sub-graph (c) where there is only one token in place p_5 with value "1".

One may think of this NPN as a simple computing device such that t_2 makes output based on the input values from p_2, p_3, p_4 and feeds them back as input. This iterate process is controlled by p_1, and the number of tokens in place p_1 decides the number of iterations this device can do (3 in this case). The final output is in place p_5 which has a token with value "1".

2. Stochastic Neural Petri Nets

As a natural extension, the concept of stochastic neural Petri nets(SNPN) can be developed almost the same way as for stochastic Petri nets by associating each transition in a NPN with an exponentially distributed random variable that express the firing delay from the enabling to the firing of the transition. Formally, a stochastic neural Petri nets model can be defined as the following:

SNPN = (P, T, A, H, W, Θ, V, M^0, Λ), where P, T, A, W, H, Θ, V, and M^0 are the same as in NPN definition, and

$\Lambda = (\lambda_1, \lambda_2, ..., \lambda_m)$ is the set of finite transition rates associated with NPN transitions for exponentially distributed firing times. These values may be obtained by determining the average firing delays ($\{d_i=1/\lambda_i\}$) when the transition is enabled and in isolation from the rest of the SNPN.

Lets look at an example of SNPN. The following graph represent a SNPN with 3 transitions, 5 places and initial marking $M^0 = (m^0_1, m^0_2, m^0_3, m^0_4, m^0_5)$ = ((3,0), (0,2), (0,0), (0,0), (0,1)). Similar to the last NPN example, we have T = (t_1, t_2, t_3), P = $(p_1, p_2, p_3, p_4, p_5)$ and arcs A are indicated in the graph, the filtering function H = (h_1, h_2, h_3) is binary with threshold Θ =(0, -2, 1), i.e.

$$h_i(x) = \begin{cases} 1, & \text{if } x \geq 0, \\ 0, & \text{otherwise,} \end{cases} \qquad \text{for } i=1, 2, 3.$$

Weight matrix W is:

$$W = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

The transition rates are $\Lambda = (\lambda_1, \lambda_2, \lambda_3) = (1, 2, 1)$. This means that transition t_1 and t_3 can process 1 tokens per unit time and t_2 can process 2 token per unit time.

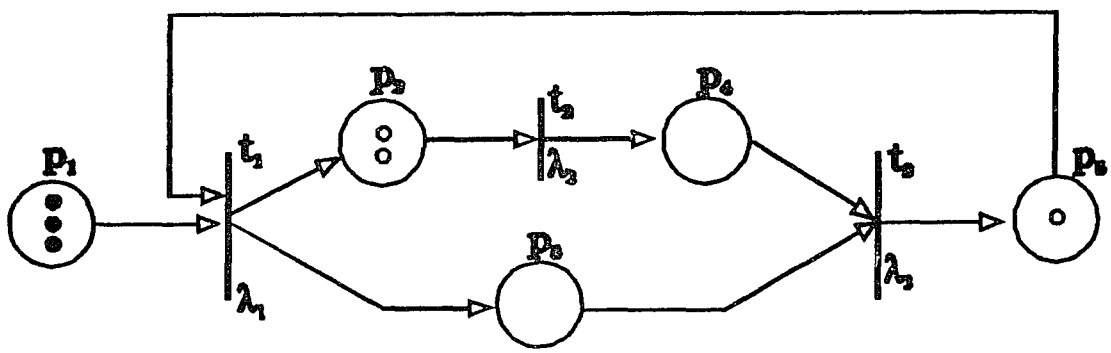


Figure 6. An example of SNPN with 3 transitions and 5 places.

The execution of SNPN is some what different than NPN because the time and transition rate factors. The following list is the status of this SNPN at different times:

Time 0: Both transitions t_1 and t_2 are enabled and ready to fire. The current marking is ((3,0), (0,2), (0,0), (0,0), (0,1)).

Time 1: p_1, p_2, p_5 lost tokens and p_2, p_3 gained tokens due to firing t_1 and t_2. Transitions t_2 and t_3 are enabled. The current marking is ((2,0), (0,0), (0,1), (2,0), (0,0)).

Time 2: p_2, p_3, p_4 lost tokens and p_4, p_5 gained tokens due to firing t_2 and t_3. Transitions t_1 is enabled. The current marking is ((2,0), (0,0), (0,0), (1,0), (0,1)).

Time 3: p_1, p_5 lost tokens and p_2, p_3 gained tokens due to firing t_1. Transitions t_2 and t_3 are enabled. The current marking is ((1,0), (0,1), (0,1), (1,0), (0,0)).

Time n: Continuing this firing routine, all three transitions are enabled alternately and fired until there is no token in place p_1, the SNPN reaches its final state (no more enabled transitions). The final marking is ((0,0), (0,0), (0,0), (1,0), (0,1)).

Place p_1 is serves as a control place and the tokens are served as control tokens. The number of tokens in p_1 decides the number of feedbacks allowed from p_5 to t_1. One might already notice t_2 moved 2 tokens from time 0 to time 1, since t_2 has a faster transition rate than t_1 and t_3, and resulted extra tokens being accumulated in place p_4 before t_4 can remove them. This represents the queuing effect in a computing system.

3. SNPN Properties

The concepts of reachability problem, ergodic property and liveness etc. of stochastic neural Petri nets can be defined and discussed in the similar fashion as SPN, because the basic structure and firing rules are kept the same. The study of these theoretical problems will require hughe amount of research effort and time.

Chapter 5

USING SNPN FOR MODEL ANALYSIS

From the studies of both neural nets and stochastic Petri nets models, we know stochastic neural Petri nets would be a very useful modeling tool for computer systems, especially large scale parallel computing systems. Now it is time to do some analysis on the SNPN we just created to see its capability as a mathematical model.

A good model should represent the basic characteristics of the actual system, and be able to analyze mathematically providing quantified information beyond the obvious. For example, queuing network models can take inputs such as service times at each node of a computer system to produce useful information such as user response times for the commands they key in the system. Let us use SNPN to represent a computing network and see what is can provide us.

1. Computer System Performance Issues

One of the most important concerns in managing computer system is that it

has "adequate" performance [MBC86]. Definition of "adequate performance" may be explicitly or implicitly given, and will be determined largely by the intended functions. System throughput (i.e. how many jobs the system can process in a unit time) is one of the most critical issues of performance.

The most direct approach to evaluation of system performance is to measure directly, either with hardware dedicated to do measurements or with code embedded in software to obtain performance estimates or a combination of hardware and software. For example, one can run an industrial standard TP benchmark, a workload to simulate ATM (automated teller machine) banking application in a on line transaction processing environment, on a computer system and obtain performance data such as throughput (total transactions per second) and response time (time delay between a transaction send and transaction receive, this can be thought as how long a user has to wait for bank machine to respond).

Unfortunately, this approach only works at post production time. It is too late when a system is found to have "inadequate performance". There are numerous examples of systems which have gone through their entire development and have totally unacceptable performance when complete. If one ends up with unacceptable performance with a reasonable amount of hardware, then the only options are to abandon the system entirely (which

happens too frequently) or to go through redesign and redevelopment phases until the system is acceptable. Either of these options is much more expensive than a design and development process which explicitly considers performance. This forces people to consider the computer performance issues at the pre-production stage.

Direct measurement is not feasible in the design and development stages of the computer system. The system is not measurable if it is not operational. Considering one can only work with what one has, modeling should be used when measurement is impossible. The basic idea is to devise a model that captures the main factors determining system performance, determine performance measures in the model and use these measures from the model as estimates of performance of the actual system. Depending on how we plan to determine performance measures in the model, the model may seem very abstract relative to the actual system or may be a very detailed representation of the actual system. Generally, the more detailed the model, the less manageable it is and the more human and machine expense will go into obtaining its performance measures. However, very abstract and seemingly simplistic models can provide relatively accurate estimates of system performance.

2. Throughput Bounds for SNPN

Assuming we have SNPN to model a computer system, the computer throughput issue is then automatically translated into a SNPN throughput issue. Determining throughput boundaries becomes a very important subject in SNPN study. First of all, let us define the throughput for a SNPN:

> The throughput of a SNPN is the number of tokens leaving a user-specified transition in per unit time.

Since the throughput concept only deals with the flow of total number of tokens in a SNPN, the values associated with tokens need not to be considered. So we can ignore the factors that are related to value calculation functions H = $(h_1, h_2, ..., h_m)$, and concentrate on the topology of SNPN.

Intuitively speaking, no system can perform infinitely fast, so the throughput of any SNPN must have upper bounds. In fact, $\Lambda = (\lambda_1, \lambda_2, ..., \lambda_m)$ constitutes a set of upper bounds for the SNPN. But this may not be a good bound (a good or tight bound is close to the actual throughput). The difficulty is how to find a tight upper bound? One approach is to analyze the associated MC, therefore obtain throughput estimates of the SNPN. However, from the previous sections we know that this process needs to solve the reachability set problem

(in order to find the MC state space), which is a problem of at least exponential complexity. The discussion on tightness of upper bounds is beyond the scope of this paper.

The next step is to find fast bounds for SNPN throughput in steady state instead of enumerating the reachability set, i.e. construct an algorithm with polynomial complexity to determine throughput bounds. The class of SNPN under consideration are ergodic, bounded, and live, because all realistic systems are finite in nature and we are interested in the steady state behavior of systems that are in continuous operation.

Let F represent a vector of throughputs for the m transitions in the SNPN:

$$F = (f_1, f_2, ..., f_m), \text{ where } f_j \text{ is the token flow rate of } t_j \in T$$

The aim here is to determine boundaries for F.

Let us examine a place p_i of SNPN in steady state. According the assumption of ergodicity of SNPN, the number of tokens in this place p_i, m_i^*, is constant.

Since tokens can not be created or destroyed by a place in SNPN, token flow

in and out of the place, averaged over an infinitely long period of time, must balance in a bounded net. So, we have the following balance equations:

$$\sum_{j=1}^{m} \#(t_j\ ,\ p_i)*f_j = \sum_{j=1}^{m} \#(p_i\ ,\ t_j)*f_j, \qquad \forall p_i \in P \quad \textit{where}$$

$$\#(t_j\ ,\ p_i) = \begin{cases} k, & \textit{if } \exists \textit{ an output arc of multiplicity } k \textit{ from } t_j \textit{ to } p_i\ , \\ 0, & \textit{otherwise;} \end{cases}$$

$$\#(p_i\ ,\ t_j) = \begin{cases} k, & \textit{if } \exists \textit{ an input arc of multiplicity } k \textit{ from } p_i \textit{ to } t_j\ , \\ 0, & \textit{otherwise;} \end{cases}$$

Define

Forward incidence matrix $C^+ = \{ a_{ij} \}$, where $a_{ij} = \#(t_j, p_i)$ and

Backward incidence matrix $C^- = \{ b_{ij} \}$, where $b_{ij} = \#(p_i, t_j)$

for $1 \le i \le n$ and $1 \le j \le m$.

Furthermore, the incidence matrix C for SNPN is defined as:

$$C = C^+ - C^-$$

Then the set of balance equations among the tokens flowing through places of the net can be represented in the following matrix form:

$$C*F = 0$$

This together with SNPN transition rates Λ certainly will result a set of upper bounds for SNPN (in fact Λ, itself is a set of upper bounds). The complexity of this algorithm is clearly polynomial, since it is just to solve a set of linear equations.

Let us use the SNPN in Figure 7. to illustrate this method. This SNPN consist of 9 places (p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9) and 6 transitions (t_1, t_2, t_3, t_4, t_5, t_6) with transition rates (λ_1, λ_2, λ_3, λ_4, λ_5, λ_6) = (2, 4, 4, 4, 1, 5). The initial marking is M^0 = ((1,1), 0, 0, 0, 0, 0, 0, 0, 0). As it was discussed above, the value of tokens is not of concern for throughput issues, so the value function H and its related variables are not even considered.

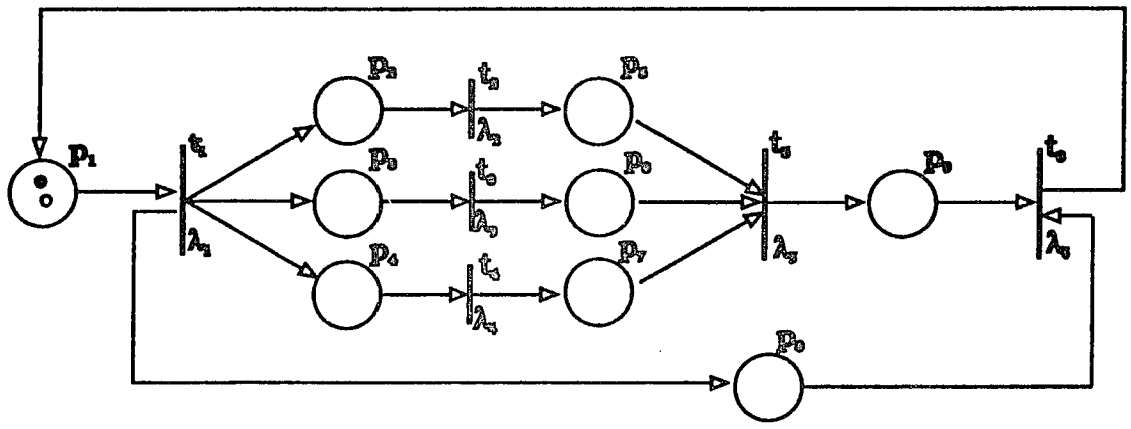


Figure 7. A SNPN with 6 transitions and 9 places.

Figure 7. can be thought as an example of a simple parallel computing system. Place p_1 represents the location of terminals where user send in job

requests. Transition t_1 divides the user work into several segments, in this case three, and stores them in p_2, p_3, p_4. A job control code is also generated in p_8. Transition t_2, t_3, t_4 are responsible for doing the data processing needed in parallel. Transition t_5 is the unit that assembles the results from these parallel processors to make on output. At last t_6 does the final check to assure the integrity of the output data by taking inputs from p_8 (the job control code) and p_9 (processed data), then the final results are sent back to user at place p_1. A job that a user submits is represented as a token in place p_1, and the job is considered done when the token travels through the system and comes back to place p_1. Transition rates $(\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5, \lambda_6) = (2, 4, 4, 4, 1, 5)$ indicate the number of tokens a transition fires on average, and this represents the processing power of this transition unit. Let us assume that all transitions use processing units of the same power. Transition rate $\lambda_1 = 2$ means t_1 can process 2 jobs per unit time. Transitions t_2, t_3, t_4 process 4 jobs per unit time because each of them only deals one small job (roughly 1/3 of a job t_1 has). Transition t_5 can process only one job per unit time because the cost of integrating results from all sub-processors. Transition t_6 has high output of 5 jobs per unit time simply because error checking is not a heavy job to do.

The system throughput which is the number of tokens through the system per unit time is actually the total number of tokens through p_1 per unit time. This is the same rate as the number of tokens passing through t_1, because we

are considering the steady state of the SNPN. The problem is to find a fast throughput bound for t_1.

Using the metheds disscuss above and the topology of the SNPN structure illustrated in Figure 7. we can derive the following matices:

$$\textit{Forward incidence matrix } C^{+} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

and,

$$\textit{Backward incidence matrix } C^{-} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Then, the incidence matrix C for this stochastic neural Petri net is:

$$C^+ - C^- = \begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & -1 \end{pmatrix}$$

Let $F = (f_1, f_2, f_3, f_4, f_5, f_6)$, where f_i is the token flow rate for transition t_i, for $1 \le i \le 6$. Then the following equations are established from the flow balance equations $C^*F = 0$ in steady state:

$$\begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & -1 \end{pmatrix} \times \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The following relations can be deduced by solving the above equations:

$f_1 = f_2 = f_3 = f_4 = f_5 = f_6$

It is clear that f_1 (the system throughput) has an upper bound of 2 because $f_1 \leq \lambda_1$. Furthermore, since all job flow rates are equal (from the above relation), f_1 is bounded by the smallest transition rate. So, the system has a throughput upper bound of 1 job per unit time ($f_1 = f_5 \leq \lambda_5 = 1$), which might not have been obvious before the analysis.

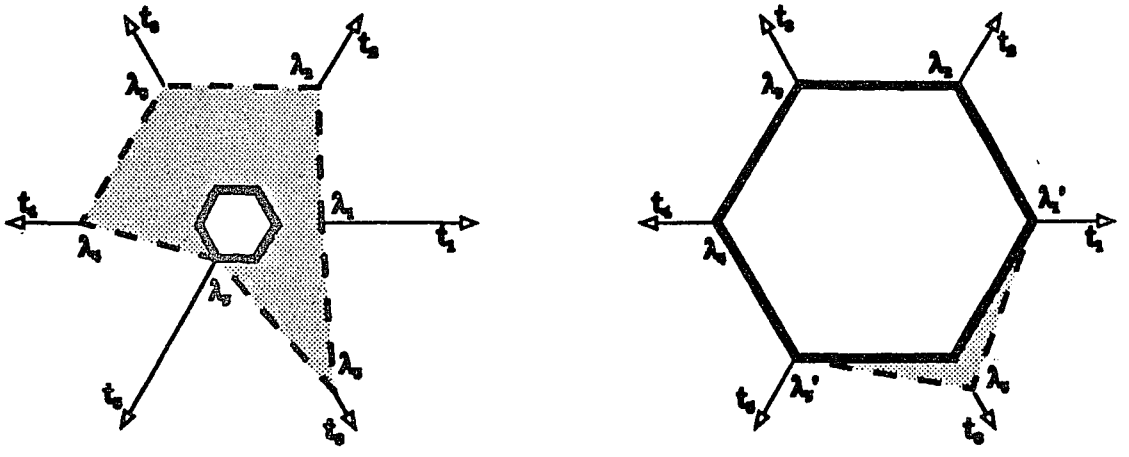
3. Bottleneck Analysis

A computer system consists of many components which work together to accomplish user tasks. The inter communication and dependencies can be very complex. Because of system architecture and other factors, the system can experience performance slowdowns due to resource exhaustion on one or few of its components. These components are often called bottlenecks.

In the system above, transition t_1 has throughput of 1 job per unit time, but it has capacity of 2 jobs per unit time because it is given that $\lambda_1 = 2$. Intuitively speaking, computing device t_1 has not been fully utilized. The reason that system throughput has upper bound 1 is that transition t_5 has capacity of 1 ($\lambda_5 = 1$) and is fully utilized ($f_5 = \lambda_5$). If we increase λ_5 to 1.5, the system throughput will be 1.5. This shows that t_5 becomes a bottleneck for the whole system due to its transition rate, and system throughput will increase

by just boosting the processing power of this transition. On the other hand, transition t_6 is used only 20 percent (f_6/λ_6), and is certainly a waste.

Finding and eliminating bottlenecks, therefore increasing system output at minimum cost is one of the challenging tasks in the computer industry. The following graph shows SNPN can play an important role in this arena.



(a) Before Improvement (b) After Improvement

Figure 8. Eliminating System Bottlenecks.

Each direction from the center represents a transition (sub-computing component) in the above computing system. The length from center to each corner of the dashed box represents transition rate which is the capacity of this component, and the distance from center to the bold box (the inner box) represents throughput limits. So the enclosed dash line connecting $\{\lambda_i\}$ forms

capacity boundary. The box of bold lines is determined by the SNPN topology and represents the actual throughput limits of $\{t_i\}$. The bold box which can change size only proportionally are bounded by the dash box, because throughput can not exceed capacity. Once they touch, the bold box can not enlarge anymore and becomes an actual throughput upper bound for this system.

In Figure 8. , the area enclosed in bold box indicates throughput limitation, and the area enclosed in dash box indicates system capacity, thus the area between these two boxes which is shaded indicates the capacity this system has that has not been utilized. Reducing bottlenecks and improving efficiency is therefore to minimize this shaded area. Graph (a) show the current system performance status. As one can see clearly the system capacity has not been utilized much and there is a lot of room for improvements (i.e shaded area to be reduced). There are many ways to reduce the shaded area depending on individual circumstances. A simple way is to increase the transition rate on all transitions to 5, and system will have a throughput of 5 jobs per unit time, and an utilization of 100 percent on all transitions. While this sounds good, it requires to upgrade the five of the six components. One might be able to build a new system more economically with these supplies. A more efficient way is to bring up the processing power of transition t_1 and t_5 to 4 jobs per unit time, thus system has throughput of 4 jobs per unit time. Graph (b) showes

the new system status after this improvement with new transition rate values $\lambda_1' = 4$, $\lambda_6' = 4$, and same values for the rest. The shaded area relative to system throughput which is the area enclosed by bold box has been greatly reduced while only two out of six transitions have been upgraded.

Chapter 6

NPN RESTRICTIONS AND POSITIONAL GAMES

1. Restrictions on NPN

According the current NPN definition, the firing of transitions is allowed as long as all input places have tokens, and lacks the ability to discriminate among the values of their inputs. In practice, programming for instance, there are many conditional decision processes having input value dependencies. To enhance NPN models simulating these processes, a restriction is added to its transitions effecting their ability to fire thus increasing sensitivity to their input values. The enhanced models are called restricted neural Petri nets (RNPN).

The definition of a RNPN can be given as a system RNPN = (NPN, R), where NPN is regular neural Petri net, and R is a set of firing restrictions associated with transition set T. The selection R can vary a great deal and therefore effect the properties of RNPN. For our purpose in this article, only the following firing restriction logic is used:

SYNC: The transition is enabled when all its input tokens have the same value.

∅: No restriction. The transition is enabled when there are tokens in all of its input places.

¬SYNC: This is the negation of SYNC, i.e., the transition is enabled when its input token values are not all the same.

Based on this definition, NPN becomes a special case of RNPN with restriction R = ∅. Furthermore one can apply these restrictions to stochastic neural Petri nets to define restricted stochastic neural Petri nets (RSNPN). These logical restrictions increase the flexibility of transitions, therefore make RNPN a very good simulation model. Many examples of RNPN are included in chapter 7.

2. Positional Games

So far, we have concentrated on RNPN definitions and analysis, and we know that RNPN is a great model and has a lot of potential. Questions remain is this a realistic tool to use, how do we use it? As we all know that often times it is not easy to implement the glorious theory. For RNPN, the definition and analysis is a science, to design NPN for a real scenario can be an art. In next few chapters, we will demonstrate its applicability by

designing RNPN for a real application.

RNPN can be applied in many fields. As an example, positional games are chosen for modeling demonstrations. A standard definition of positional games is the following:

a. There are two players.
b. There are several, usually finitely many positions.
c. There are clearly defined rules that specify the moves that either player can make.
d. Both players move alternately.
e. The rules are such that play will always come to an end, so there can be no game with unlimited repetition of moves.
f. Conclusion criterions are clearly defined, so at the end a game result can be reached: one player wins (the other loss) or a draw (no one wins).

There are many positional games, such as: Tic-Tac-Toe, Chess, Checkers, Go, Go-Moku, etc. Some are more difficult to play than the others due to the design of its positions and rules, but they all have the same characteristics.

Mathematicians and computer scientists have been studying these games for

years, and as a result, there are good computer programs to play these games. Some programs play at a professional level such as Chess, but others can only reach amateur level such as Go. Among the games, Chess has probably been studied most extensively. Chess has especially been treated as a challenge in the field of artificial intelligence. Many articles have been published and computer programs been developed to play chess at a professional level.

As a practical example, a well known game, tic-tac-toe is used here as our "positional game". The following chapter will analyze the game and design a RNPN model for it, subsequently construct a computer algorithm to play Tic-Tac-Toe. This algorithm finally is implemented in "C" code and can be applied on personal computers. Although many computer programs exist for these games, most of them are rule based. The RNPN model gives a different approach and provides a solution to combine rule base and pattern recognition. Kriegspiel version of tic-tac-toe is also analyzed to show how a RNPN model can be applied in the field of incomplete information games with a referee.

Chapter 7

RNPN DESIGNS FOR TIC-TAC-TOE GAME

1. The Game

Everyone knows how to play tic-tac-toe, since it is a simple and classical game. The game board consists of nine positions, situated in an three by three matrix formation. Two players alternatively put their own stones on an unoccupied position, until either one player wins the game or all positions are taken. A win is determined by the player who has three stones in a row in any direction. If the board is full and no winner is found, then a drawn decision is concluded.

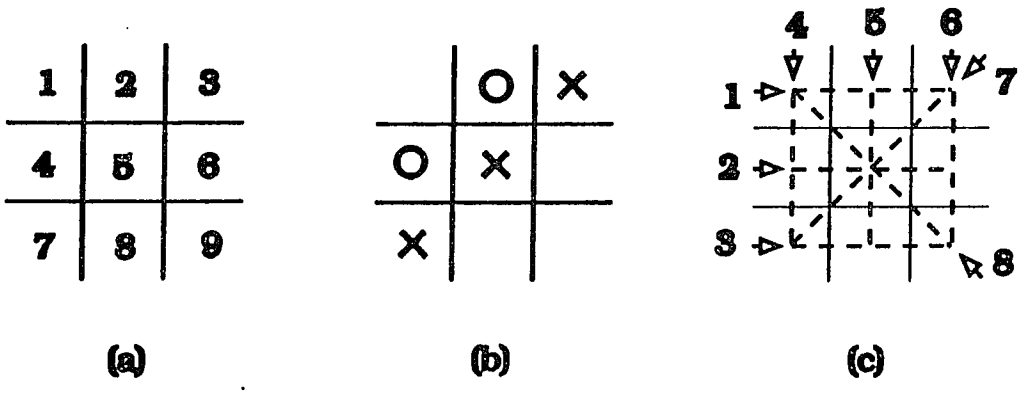


Figure 9. Tic-Tac-Toe board and its eight lines

For future reference the nine positions of a tic-tac-toe board are labeled sequentially from 1 to 9 as shown in graph (a). Assume player 1 uses "X"s and player 2 uses "O"s on the board. Graph (b) shows a partially occupied board with player 1 as the winner(three "X"s in a row). Graph (c) indicates the eight lines (dash lines) that can possibly form a winning configuration. These lines are also labeled from 1 to 8 which is associated with a line by an arrow for future RNPN design. When board position i or line j is mentioned in rest of this chapter, it is referring the numbering convention in graph (a) and (c).

Although the game itself is rather simple, the actual analysis is not trivial. E. R. Berlekamp[BER85] et. al. and many mathematicians have given detailed analysis for this game and its extension.

2. High Level Logic Diagram

To limit the complexity of this model, it is a good idea to divide the model to several components and model them separately. First of all, there must be a game board which contains the current information of game status such as which positions are occupied, by whom, and which positions are available. A decision unit(DU) controls the game. Its functions include informing players when it is their turn to make a move, making sure no one moves twice, and

determining the outcome of a game, etc. The model structure for both players is identical, since their access to information and possible moves are equal.

Each player has two components: an execution unit(EU) and a pattern recognition unit(PRU). The execution unit is responsible for interfacing with the game board; the decision unit and pattern recognition unit are responsible for generating good moves.

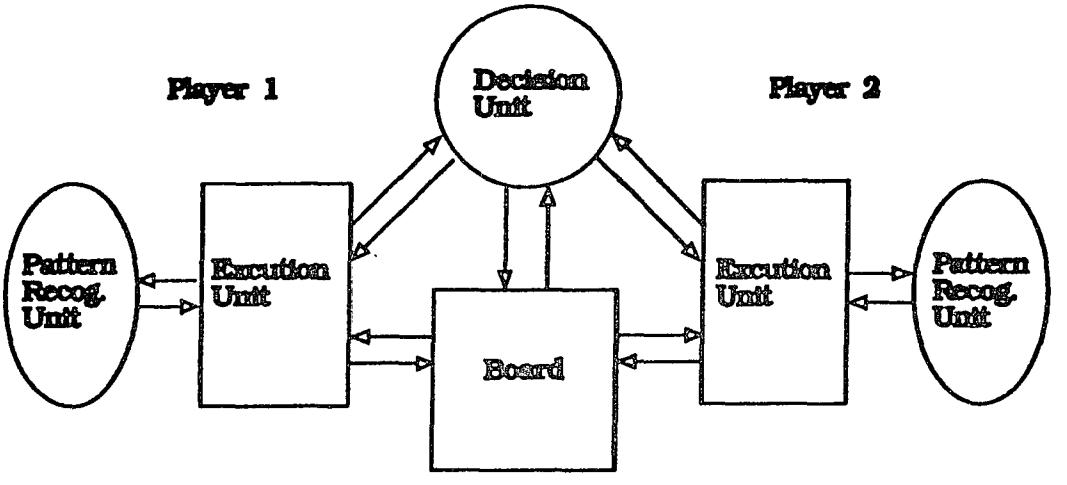


Figure 10. Job Flow Diagram

The logic connection between these components are illustrated in Figure 10.. The operation starts at DU. DU decides who should move, and then activates that player, say player 1. Now the EU of player 1 takes the board information and passes to its PRU for response. After the PRU gives a response, the EU makes a move on the board so the board information is updated, then gives the control back to DU. DU now examines the board to see if the game has ended(

win, loss or drawn). If not, the control is passed to the other player. The other player executes the same way as the first player, then passes control back to EU. Continuing this loop, the game will eventually end since there are only a finite number of positions. It is clear that the operation of this model portrays a real game scenario.

3. The Game Board

As we all know, each position on a tic-tac-toe board can have three values: X, O, and empty. Since a token in each place has only 2 distinct values in RNPN definition, we need more than one place in RNPN to represent a position on the board. Figure 11. is a graphic representation of the RNPN model for a board position. It has 8 places and 5 transitions.

Place p_1 is an indicator for occupancy of this board position. A token in place p_1 (called control token) indicates this position is available, and occupied otherwise. Places p_1, p_2, p_3 and transition t_1 (the items in the dashed box) form a control device for information updating the position which is under supervision of two places: p_1 and board update control. The device is enabled when control tokens exit in both p_1 and the board control place which means this position is available and needs to be update. The actual board position

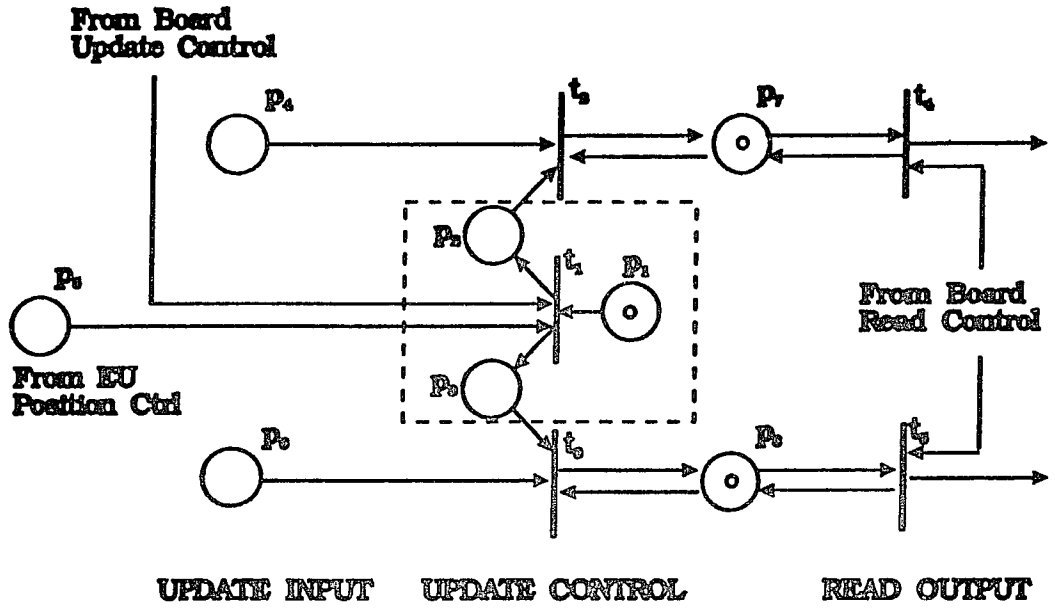


Figure 11. NPN representation for a position on the board.

value is hold in places p_7 and p_8. A "X" is represented by p_7=(1,0), p_8=(0,1); "o" represented by p_7=(0,1), p_8=(1,0) and empty position is represented by p_7=(0,1), p_8=(0,1) which is the current RNPN marking in Figure 11. . Places p_4 and p_6 hold an update value (a move that a player wants make). The update engine consists of transitions t_2 and t_3 and is enabled when update tokens are present in p_4, p_6 and control tokens are present in other connected places. Firing the update engine takes away the update tokens in p_4 and p_6, old tokens in p_7 and p_8 (current board position value), the control tokens in p_2 and p_3, and then puts new tokens in p_7 and p_8. Now this board position has new value. Notice t_2 and t_3 are enabled and disabled synchronously due to the design of the control device. This maintains the integrity of the information pairs that represent one board position. Transitions t_4 and t_5 form a read out engine in the sense

that they send out a copy of current position information if a token from board read control is present.

Formally, a RNPN net can be defined as a system B(board position) = (P, T, A, H, W, Θ, V, R) with V = {0, 1} (the token values), R = {∅} (no firing restriction on transitions), and

$P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$,

$T = \{t_1, t_2, t_3, t_4, t_5\}$,

Arcs set A are shown in Figure 11. ,

$H = \{h_1, h_2, h_3, h_4, h_5\}$ with $\Theta = \{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5\}$,

where h_i is the binary function with threshold $\theta_i = 0$ for i = 1, 2, 3, 4, 5.

The weight matrix is:

$$W = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

If we assume that $X = (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ is the input from places P to transitions T and $Y = (y_1, y_2, y_3, y_4, y_5)$ is the output from transition set T,

then $Y = H(\langle W, X\rangle - \Theta)^{\dagger}$.

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix} = H \left(\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \right) = \begin{pmatrix} 0 \\ x_4 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix}$$

The sole purpose of transition t_1 is to control t_2 and t_3, so that the token output from t_1 always has value 0, independent from any input token value, i.e. $y_1 = 0$. The transitions t_2 and t_3 perform the board update function and their output token values should be the same as the value in p_4 and p_6 respectively, i.e. $y_2 = x_4$ and $y_3 = x_6$. Lastly, t_4 and t_5 should copy out the current board value, so the output token is the same as board position value, i.e. $y_4 = x_7$ and $y_5 = x_8$.

Considering that a tic-tac-toe game board is composed of nine positions, the RNPN for a board is simply the union of nine BP's. Figure 12. shows the RNPN architecture for a game board. Each of the nine positions has the same

---

† All variable matrices used in matrix calculations are in the form of their transposes throughout this chapter. For example, X is a 1 by 7, and is used as a 7 by 1 matrix in calculations.

topology as the above BP, with its own sub-index to represent actual board positions sequentially. Each node is labeled by using letter b(for board position) followed by node name with its board position order as superscript and its own element order as subscript. For example, the third transition in board position BP_8 is labeled as bt_3^9.

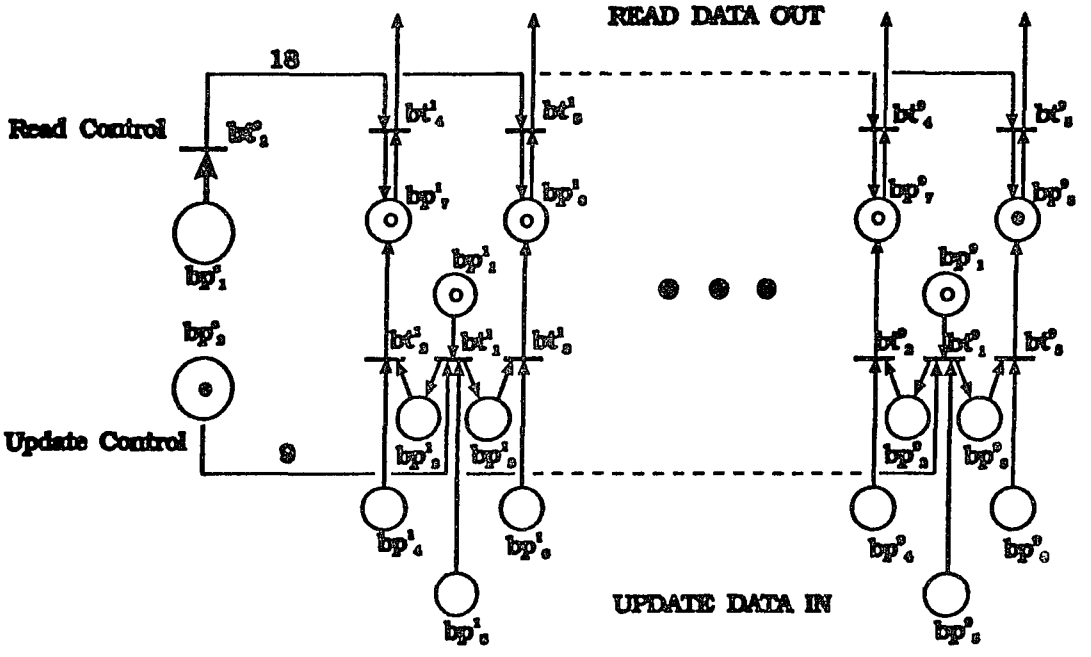


Figure 12. NPN Representation For Tic-Tac-Toe Board.

In addition to these board positions, there are two board control places bp_1^c, p_2^c and a control transition t_1^c. Place p_2^c controls the board update activity by controlling each control unit of the BP's. Board updating can only occur when there is a token present in p_2^c. Once a BP has updated, the token is removed so that no more activity can occur. This controls when an update can occur

and inhibits more than one position update at a time. Place p_1^c is the read control place. A token in p_1^c causes transition t_1^c to fire which sends control tokens to all position read engines. As a result, a copy of current board information data is made out.

Formally, the RNPN net for a tic-tac-toe board can be defined as BU (board unit) = (P, T, A, W, H, Θ, V, R), with Θ = {0}, V = {0, 1}, R = {∅}, H is a set of binary functions associated with transition set T, arcs A indicated in Figure 12. , and

$$P = \{p_1^c, p_2^c\} \cup \bigcup_{i=1}^{9} BP_i \,, \quad \textit{where } BP_i \textit{ is the set of places for position } B_i$$

$$T = \{t_1^c\} \cup \bigcup_{i=1}^{9} BT_i \,, \quad \textit{where } BT_i \textit{ is the set of transitions for position } B_i$$

$$W = (0, 0) \oplus \bigoplus_{i=1}^{9} BW_i \,, \quad \textit{where } BW_i \textit{ is the set of weights for filter function set}$$

With the board position well defined by RNPN (BP), we now have a complete RNPN definition for the board. As one can see, this RNPN represents not only the 9 positions on the board, but also certain amount of control knowledge, such as the mechanism to supply current copies of board information when it is called upon and update control for only one position at a time. So, what we have is actually a somewhat smart board.

4. Decision Unit

This structure controls the game flow. Its function includes checking the board after every move to see if a game conclusion (win, lose or drawn) is reached and notifying a player when it is their turn to make a move. DU has to perform much more in the case of Kriegspiel version of the games which will be discussed more in later chapters.

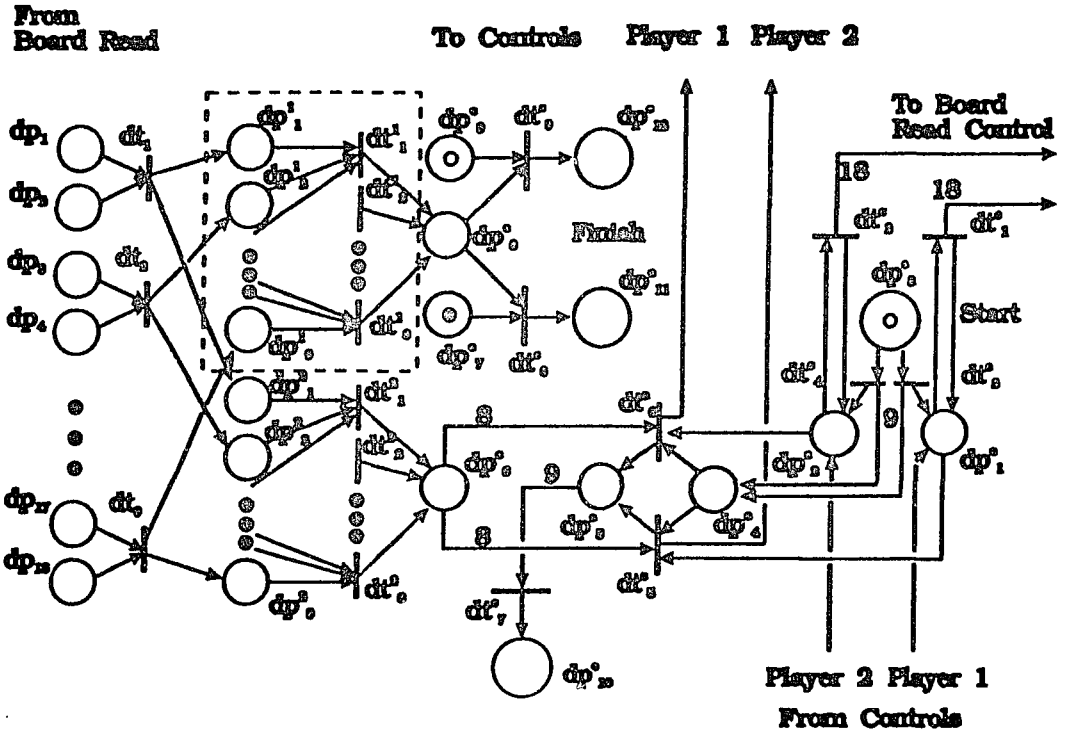


Figure 13. NPN representation for Decision Unit

Figure 13. gives the graphical representation of RNPN design for DU. This RNPN has total of 48 places and 34 transitions which are divided into several group P^c, P^0, P^1, P^2, T^c, T^0, T^1, T^2 for logical clarity and with

$P^c = \{dp^c_1, dp^c_2, ..., dp^c_{10}\}$,

$P^0 = \{dp_1, dp_2, ..., dp_{18}\}$,

$P^1 = \{dp^1_1, dp^1_2, ..., dp^1_9\}$,

$P^2 = \{dp^2_1, dp^2_2, ..., dp^2_9\}$,

$T^c = \{dt^c_1, dt^c_2, ..., dt^c_9\}$,

$T^0 = \{dt_1, dt_2, ..., dt_8\}$,

$T^1 = \{dt^1_1, dt^1_2, ..., dt^1_8\}$,

$T^2 = \{dt^2_1, dt^2_2, ..., dt^2_8\}$.

Most of the connections between places and transitions are clearly illustrated, except for the ones between P^1, T^1 and P^2, T^2 due to the difficulties of drawing complex connections in this small two dimensional space. Numerical value "k" along arcs implies that there are k arcs. Because the topological structure which represents the eight possible winning lines on the tic-tac-toe board are identical, it is enough to look at just one of them in detail, which is indicated by the dashed box. There are nine places, P^1(representing the nine board positions sequentially) and eight transitions, T^1(representing the eight winning lines) in this dashed box. Since we know that the arc set A^1

is a relation and $A^1 \subseteq P^1 \times T^1$, the connection in the dashed box can be clearly defined in the following matrix:

$$(a_{ij}) = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

where, 1 *implies there is an input* arc *from place* dp_i^1 *to transition* dt_j^1, *and* *no* arc *otherwise, for* $1 \le i \le 9$, $1 \le j \le 8$.

The RNPN definition of decision Unit is DU = (P, T, A, H, W, Θ, V, R), with H is the binary functions for T, Θ = {0}, V = {0, 1}, arc set A is indicated in Figure 13. and the above relation matrix, and

$P = P^c \cup P^0 \cup P^1 \cup P^2$

$= \{dp^c_1, dp^c_2, ..., dp^c_{10}, dp_1, dp_2, ..., dp_{18}, dp^1_1, dp^1_2, ..., dp^1_9, dp^2_1, dp^2_2, ..., dp^2_9\}$

$T = T^c \cup T^0 \cup T^1 \cup T^2$

$= \{dt^c_1, dt^c_2, ..., dt^c_9, dt_1, dt_2, ..., dt_9, dt^1_1, dt^1_2, ..., dt^1_8, dt^2_1, dt^2_2, ..., dt^2_8\}$,

$W = W^c \oplus W^0 \oplus W^1 \oplus W^2$,

where W^j is the weight matrix for transition set T^i taking inputs from place set P^j, with

$$W^0 = \begin{pmatrix} 1 & -1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & -1 & \dots & 0 & 0 \\ \vdots & \vdots & & \vdots & & & \\ 0 & 0 & 0 & 0 & \dots & 1 & -1 \end{pmatrix}_{9\times 18},$$

$$W^1 = W^2 = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{pmatrix},$$

and $W^c = (0)$ because T^c deals only with control tokens, their value is not our concern(which can be set to 0).

Because transition set T^1, $\{dt^c_8, dt^c_9\}$ can fire only when all input tokens have the same value, and T^0, T^2 can fire only when all input tokens do not have the same value, these transitions are restricted. Let R^j be the restriction set on transition set T^j, dr^a_b be firing restriction on transition dt^a_b, with possible restrictions $\{\varnothing, SYNC, \neg SYNC\}$ being the ones described in chapter six, then $R = R^c \cup R^0 \cup R^1 \cup R^2$, where,

$R^c = \{dr^c_1, dr^c_2, ..., dr^c_9\}$

$= \{\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, SYNC, SYNC\}$,

$R^0 = \{dr_1, dr_2, ..., dr_9\}$

$= \{\neg SYNC, \neg SYNC, \neg SYNC, \neg SYNC, \neg SYNC, \neg SYNC, \neg SYNC, \neg SYNC, \neg SYNC\}$,

$R^1 = \{dr^1_1, dr^1_2, ..., dr^1_8\}$

$= \{SYNC, SYNC, SYNC, SYNC, SYNC, SYNC, SYNC, SYNC\}$,

$R^2 = \{dr^2_1, dr^2_2, ..., dr^2_8\}$

$= \{\neg SYNC, \neg SYNC, \neg SYNC, \neg SYNC, \neg SYNC, \neg SYNC, \neg SYNC, \neg SYNC\}$,

Assuming that $X = (X^c, X^0, X^1, X^2)$ represents input token values from place set $P = (P^c, P^0, P^1, P^2)$ and $Y = (Y^c, Y^0, Y^1, Y^2)$ represents output token values from transition set $T = (T^c, T^0, T^1, T^2)$, where X^j represents input values for P^j and Y^j represents output values from T^j with $j \in \{c, 0, 1, 2\}$, the values of output tokens from all transitions in EU can be calculated by:

$$Y = H(\langle W, X\rangle) = H\left(\begin{pmatrix} W^c & 0 & 0 & 0 \\ 0 & W^0 & 0 & 0 \\ 0 & 0 & W^1 & 0 \\ 0 & 0 & 0 & W^2 \end{pmatrix} \times \begin{pmatrix} X^c \\ X^0 \\ X^1 \\ X^2 \end{pmatrix}\right) = \begin{pmatrix} 0 \\ H(\langle W^0, X^0\rangle) \\ H(\langle W^1, X^1\rangle) \\ H(\langle W^2, X^2\rangle) \end{pmatrix}$$

Let us examine how does this net operates. The initialized DU is

represented in Figure 13. with a token in starting place dp^c_3. Assume player 1 makes the first move after a toss, DU starts operation by firing transition dt^c_4 which results in 1 token in dp^c_2 and 9 tokens in dp^c_4. Firing dt^c_2 sends a control token to the board read control place, and causes a copy of current board position information in place set P^0. Notice that a token is still in place dp^c_2 for future usage. The function of T^0 checks the board input to determine whether a position is occupied.

Based on the T^0 specification (enabled only when input token values are different), a token with value 1 (dot in graph) is fired by transition t^c_i (for $1 \le i \le 9$) if position i is occupied by "x" (inputs are (1, 0)), a token of value 0 (circle in graph) if it is occupied by "o" (inputs are (0, 1)), and no token output from this transition otherwise (inputs are (0, 0) and (1,1)).

Each place in P^1 and P^2 (P^1 and P^2 hold the same information) now has a token with value of 1 if there is a "x" on position i, with value 0 if there is a "o" on position i and no token otherwise. We know that the structure of T^1 represents the eight winning lines, and any transition in T^1 fires only when all of its inputs are the same. So, if there are three "x"s in a row, a transition in T^1 will produce a token with value "1", and if there are three "o"s in a row T^1 will produce a token with value "0" in place dp^c_8. On the other hand, T^2 fires when its inputs are not the same. As was mentioned before, the topology of

T^1 and T^2 are the same. We have a mutual exclusive scenario: Either there is a token in dp^c_8 or there are 8 tokens in dp^c_6.

Let us examine the last portion this DU. Suppose there is a token of value "1" in place dp^c_8 (there are three "x" in a row), then dt^c_8 (enabled when both inputs are 1) fires a token in dp^c_{11}, or a token of value "0" in dp^c_8, then dt^c_9 (enabled when inputs are the same) fires a token in dp^c_{12}. Either way the operation of DU comes to an end announcing player with "x"s a winner if there is a token in place dp^c_{11} or player with "o" the winner if a token is in place dp^c_{12}. If there is no token in dp^c_8 (no winner found), then dp^c_6 must contain 8 tokens which together with the token in dp^c_2 and dp^c_4 enables transition dt^c_6. Firing of dt^c_6 adds a token in dp^c_5 for accounting purpose and a control token to activate player 1.

After Player 1 finishes a play, a control token is passed back to dp^c_1 which enables dt^c_1. Firing dt^c_1 causes the DU to execute for the other player. At the end, if no winner is found, then transition dt^c_5 is fired in instead dt^c_6. This action adds a token in place dp^c_5 for accounting purposes and passes the execution to player 2. The token number decreases by 1 in place dp^c_4 which is initialized with 9 in the beginning and increases by 1 in place dp^c_5 (starts with 0) when a player is activated. As one can see, if no winner is found, DU alternates players until all the tokens in place dp^c_4 are gone. At this time dt^c_5

and dt^c_6 are both disabled and thus no player can move. The only live transition, dt^c_7, outputs a token in place dp^c_{10} to indicate a drawn is concluded.

5. Execution Unit

The function of execution unit(EU) is two-fold. First of all, it reads input from the board and passes two copies to the pattern recognition unit(PRU) to control the work flow. Secondly, it reads data back from PRU, does data checking and sends data to the board.

As indicated in Figure 14. , all devices are labeled with prefix letter "e" and the multiplicity of the arcs between places and transitions are labeled by a numerical value(18 for this case). The RNPN is activated when a control token is present in place ep^c_1. With this token, firing of transition et^c_1 causes the board to pass the current configuration to place set $P^0 = \{ep_1, ep_2, ..., ep_{18}\}$. Transition set $T^0 = \{et_1, et_2, ..., et_{18}\}$ passes one copy of the board configuration to PRU input 2 and 17 copies to PRU input 1 to insure the correct PRU operation. When the operation of PRU finishes, a new board configuration (the reply to the current board positions) is presented in place set $P^1 = \{ep^1_1, ep^1_2, ..., ep^1_{17}, ep^1_{18}\}$ and a control token is presented in ep^c_2 by PRU.

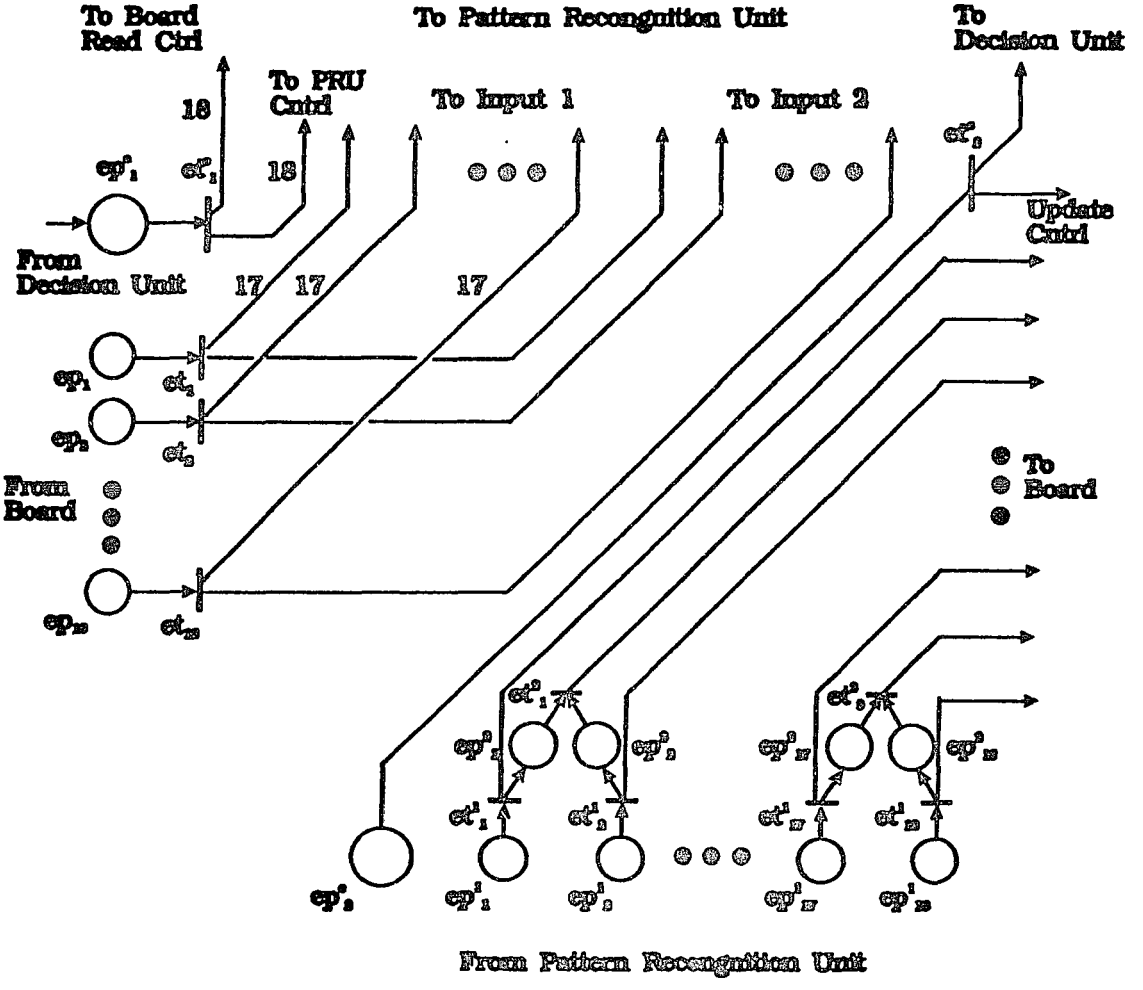


Figure 14. NPN design for the Execution Unit

It is the responsibility of transition set T^1 to provide a copy of this information in place set P^2, and a copy to board unit (BU). P^2 and T^2 form a checking device, and send a signal to BU, notifying it of the potential moves. To accomplish this goal, all transitions in T^2 are restricted with logical condition "¬SYNC". Taking et^2_1 as an example, it is enabled when tokens in places ep^2_1 and ep^2_2 have a different value, i.e the potential move to board position 1 is "X" (binary value (1, 0)) or "O" (binary value (0,1). Firing et^2_1

provides a control token to BU allowing updating to board position 1. On the other hand, if tokens in places ep^2_1 and ep^2_2 have the value, i.e (0, 0) or (1, 1), these combinations do not represent a predefined stone value, and are regarded as empty position or error information. Thus transition et^2_1 is disabled and no control token is to be passed to BU, and therefore board position 1 is kept untouched.

Formal definition of EU is EU = (P, T, A, H, W, Θ, V, R), with H is the binary functions for T, Θ = {0}, V = {0, 1}, arc set A is indicated in Figure 14.

$$P = P^c \cup P^0 \cup P^1 \cup P^2$$

$$= \{ep^c_1, ep^c_2, ep_1, ep_2, ..., ep_{18}, ep^1_1, ep^1_2, ..., ep^1_{18}, ep^2_1, ep^2_2, ..., ep^2_{18}\}$$

$$T = T^c \cup T^0 \cup T^1 \cup T^2$$

$$= \{et^c_1, et_1, et_2, ..., et_{18}, et^1_1, et^1_2, ..., et^1_{18}, et^2_1, et^2_2, ..., et^2_{18}\}$$

$$W = W^c \oplus W^0 \oplus W^1 \oplus W^2,$$

where W^j is the weight matrix for transition set T^i taking inputs from place set P^j, with

$$W^c = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix},$$

$$W^0 = W^1 = I = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}_{18\times 18},$$

$R = R^c \cup R^0 \cup R^1 \cup R^2$, where $R^c = \{\varnothing\}$, $R^0 = \{\varnothing\}$, $R^1 = \{\varnothing\}$ and

$R^2 = \{dr^2_1, dr^2_2, \dots, dr^2_8\} = \{\neg SYNC, \neg SYNC, \dots, \neg SYNC\}$

Assuming that $X = (X^c, X^0, X^1, X^2)$ represents input token values from place set P and $Y = (Y^c, Y^0, Y^1, Y^2)$ represents output token values from transition set T, where X^j represents input values for P^j and Y^j for output values from T^j with $j \in \{c, 0, 1, 2\}$, the values of output tokens from all transitions in EU can be calculated by:

$$Y = H(\langle W, X\rangle) = H\left(\begin{pmatrix} W^c & 0 & 0 & 0 \\ 0 & W^0 & 0 & 0 \\ 0 & 0 & W^1 & 0 \\ 0 & 0 & 0 & W^2 \end{pmatrix} \times \begin{pmatrix} X^c \\ X^0 \\ X^1 \\ X^2 \end{pmatrix}\right) = \begin{pmatrix} 0 \\ X^0 \\ X^1 \\ X^2 \end{pmatrix}$$

6. Pattern Recognition Unit

The function of the pattern recognition unit(PRU) is to generate a move by matching inputs from EU to the best next move in its "memory". The basic

structure of PRU is based on the Hopfield net. As discussed before, a Hopfield net can be used to construct associative memory, and after the net has been trained, one can retrieve the original information by giving only partial information. For the case of tic-tac-toe, PRU is trained to remember most board configurations. When a new board configuration is passed from EU, PRU uses this partial information to recall the best match in its memory and outputs the configuration to EU as the recommended reply to the move by the other player.

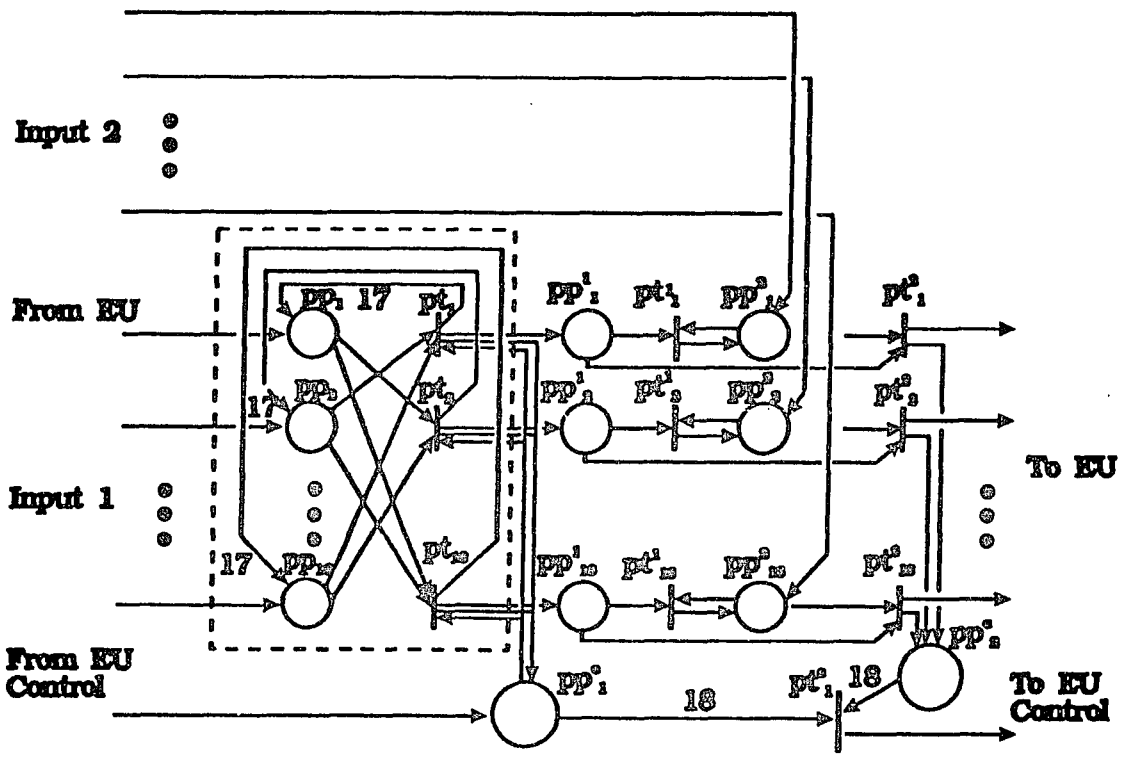


Figure 15. NPN Design For Pattern Recognition Unit

Figure 15. gives the topology of PRU. The dashed box indicates a Hopfield structure and the rest are control and supplemental mechanisms. The net operation in the dashed box needs some explanation because many lines are omitted due to the space limitation in the graph.

Place pp_1, pp_2, ..., pp_{18} take input 1 (a copy of input) from EU and then interact with transitions pt_1, pt_2, ..., pt_{18} in the following fashion: each place is a input place for all transitions except its corresponding transition and each place is a output place of its corresponding transition, i.e., there is an input arc from pp_i to pt_j and output arc from pt_i to pp_i for $1 \leq i, j \leq 18$ and $i \neq j$. This implies that each transition takes tokens from all places except its corresponding place and outputs to its corresponding place only. Places pp^1_1, pp^1_2, ..., pp^1_{18} hold a copy of output from transitions pt_1, pt_2, ..., pt_{18} (the Hopfield net). The function of transitions pt^1_1, pt^1_2, ..., pt^1_{18} is to replace the tokens in $\{pp^2_i\}$ with the tokens in $\{pp^1_i\}$. Places pp^2_1, pp^2_2, ..., pp^2_{18} are initialized with the tokens from input 2 (the other copy of input from EU) which have subsequently been changed by $\{pt^1_i\}$ to hold the tokens from $\{pp^1_i\}$. Transitions pt^2_1, pt^2_2, ..., pt^2_{18} are restricted with logical condition "SYNC" and are enabled only when all input tokens having the same value. A set of tokens which have the same values as the ones in $\{pp^2_i\}^{18}_1$ are produced and outputed to EU when transitions $\{pt^2_i\}^{18}_1$ fire. Meanwhile, 18 tokens are produced in place pp^c_2 which enables control transition pt^c_1.

A numerical value "k" appearing at a arc between places and transitions means this arc has multiplicity of k. All control devices are labeled with superscript "c". Place pt^c_1 holds 18 tokens when control token is passed in from EU, and these tokens together with the tokens in places $(pp_i)^{18}_1$ keep transition set $(pt_i)^{18}_1$ alive. Once there are 18 tokens presented in place pp^c_2 (output has been produced to EU), transition pt^c_1 is enabled. Firing pt^c_1 disables transition set $(pt_i)^{18}_1$ by removing tokens from pp^c_1 and enables the EU by passing a control token back to EU.

Let us look at the formal definition for this net. RNPN PRU is a system PRU = (P, T, A, H, W, Θ, V, R), with H is the binary functions for T, Θ = {0}, V = {0, 1}, arcs A indicated in Figure 15. , where:

$$P = P^c \cup P^0 \cup P^1 \cup P^2$$

$$= \{pp^c_1, pp^c_2, pp_1, pp_2, \ldots, pp_{18}, pp^1_1, pp^1_2, \ldots, pp^1_{18}, pp^2_1, pp^2_2, \ldots, pp^2_{18}\}$$

$$T = T^c \cup T^0 \cup T^1 \cup T^2$$

$$= \{pt^c_1, pt_1, pt_2, \ldots, pt_{18}, pt^1_1, pt^1_2, \ldots, pt^1_{18}, pt^2_1, pt^2_2, \ldots, pt^2_{18}\}$$

$R = R^c \cup R^0 \cup R^1 \cup R^2$, where

$R^c = \{\emptyset\}$,

$R^0 = \{\emptyset\}$,

$R^1 = \{\emptyset\}$,

$R^2 = \{pr^2_1, pr^2_2, \ldots, pr^2_{18}\} = \{SYNC, SYNC, \ldots, SYNC\}$

$W = W^c \oplus W^0 \oplus W^1 \oplus W^2$,

where W^j is the weight matrix for transition set T^j taking inputs from place set P^j, with

$$W^c = (0 \;\; 0) ,$$

$$W^1 = W^2 = I = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}_{18\times 18} ,$$

Weight matrix W^0 is critical for PRU operation and the values in this matrix determine the "memory sets" of PRU. The procedure to assign values to this matrix is discussed in the next section. Assuming that $X = (X^c, X^0, X^1, X^2)$ represents input token values from place set P and $Y = (Y^c, Y^0, Y^1, Y^2)$ represents output token values from transition set T, where X^j represents input values from P^j and Y^j for output values from T^j with $j \in \{c, 0, 1, 2\}$. Let $Z = (Z^c, Z^0, Z^1, Z^2)$, where $Z^c = X^c$, $Z^0 = 2*X^0 - 1$, $Z^1 = X^1$ and $Z^2 = X^2$, then the values of output tokens from all transitions can be calculated by:

$$Y = H\,(\langle W, Z\rangle) = H\left(\begin{pmatrix} W^c & 0 & 0 & 0 \\ 0 & W^0 & 0 & 0 \\ 0 & 0 & W^1 & 0 \\ 0 & 0 & 0 & W^2 \end{pmatrix} \times \begin{pmatrix} Z^c \\ Z^0 \\ Z^1 \\ Z^2 \end{pmatrix}\right) = \begin{pmatrix} 0 \\ H(\langle W^0, Z^0\rangle \\ Z^1 \\ Z^2 \end{pmatrix}$$

After weight assignment, the overall PRU operation can be characterized by the following algorithm:

Step 1. Initialize with unknown input pattern from EU at time 0

$\mu_i(0) = x_i, \quad 1 \le i \le 18$

Where $\mu_i(\tau)$ is the output for transition pt_i at time τ and x_i is binary value for element i of the input pattern. A copy of this pattern (x_i) is also held in place set P^2.

Step 2. Pattern generating process

$$\mu_i(\tau+1) = H\left(\sum_{j=1}^{18} w_{ij}(2\mu_j(\tau) - 1)\right), \qquad 1 \le i \le 18$$

where function H is the binary filter function and $W^0 = (w_{ij})_{18\times18}$ is the weight matrix for transition set T^0. This new pattern value set is held in place set P^1 and also fed back to P^0.

Step 3. Iterate until convergence

Repeat the pattern generating process until node outputs remain unchanged with further iteration. Check pattern generating process convergence by comparing the information sets in P^1 and P^2. If they are the same, an output set is made to EU by the means of firing T^2. Otherwise, replace P^2 with new pattern then go to step 2.

7. Learning Ability of PRU

It is known that neural net has ability to learn and this property is inherited by RNPN. The questions of how a net learns and how well it learns are very active fields of current research with many articles on its algorithms and complexities. In essence, a net can learn through changing its weights. In this PRU example, learning means the assignment of the weight matrix W^0.

Since the learning property is not the study objective, a very basic learning algorithm, known as slow learning, is employed to serve our purpose. Assume there are M patterns we would like PRU to remember: $\{x^1, x^2, ..., x^M\}$, where $x^j = (x^j_1, x^j_2, ..., x^j_{18})$ with $x^j_i \in \{0, 1\}$ for $1 \le i \le 18$ and $1 \le j \le M$. Slow learning is the procedure to assign values to the weight matrix $W^0 = (w_{ij})_{18\times18}$ according to:

$$w_{ij} = \begin{cases} \sum_{s=1}^{M} (2x_i^s - 1)\times(2x_j^s - 1), & i \neq j \\ 0, & i = j, \quad 1 \le i, j \le 18 \end{cases}$$

From now on, when a unknown pattern is presented to the input places, the operation of PRU will recall its memory and choose the closest pattern to the input as its output. This learning algorithm is slow and unsophisticated because one has to calculate all weights from scratch over again every time when there is change to the initial pattern set.

8. The Overall Definition and Algorithm

After defining the components, it is time to put them together. The RNPN definition for Tic-Tac-Toe is a system TIC = (P, T, A, H, W, Θ, V, R), with H is the binary functions for T, Θ = {0}, V = {0, 1}, with

$$P = BP \cup DP \cup EP \cup PP,$$

where BP is the place set of Board, DP is the place set of DU, EP is the place set of EU and PP is the place set of PRU. Similarly, $T = BT \cup DT \cup ET \cup PT$, where BT is the transition set of BU, DT is the transition set of DU, ET is the transition set of EU and PT is the transition set of PRU; $R = BR \cup DR \cup ER \cup PR$, where BR is the restriction set for BT, DR is the restriction set for DT, ER is the restriction set for ET and PR is the restriction set for transition set PT of PRU.

The formulation for the arc set between places and transitions is somewhat different than the previous formulation because there are additional arcs between the units besides the ones within the units. Let CA stands for the set of arcs between these units, then the arc set $A = CA \cup BA \cup DA \cup EA \cup PA$, where BA is the arc set of Board, DA is the arc set of DU, EA is the arcs set

of EU and PA is the arc set of PRU. Because the relation matrix for CA is too large (caused by too many nodes) to fit in this page, the vector form is used here to list the elements. Furthermore let's divide CA into few subsets: CA = $CA^b \cup CA^d \cup CA^e \cup CA^p$, where CA^b is the set of all input arcs to the Board, CA^d is the set of all input arcs to DU, CA^e is the set of all input arcs to EU and CA^p is the all input arc set for PRU. Each element in CA can be expressed as k(t, p) which means an arc with multiplicity k from transition t to place p. The following sets can be observed from the connection between the units:

$$CA^b = \{(dt_1^c, bp_1^c), (dt_2^c, bp_1^c), (et_1^c, bp_1^c), (et_2^c, bp_2^c)\} \cup \bigcup_{i=1}^{9} \{(et_{2i-1}^1, bp_4^1), (et_i^2, bp_5^1), (et_{2i}^1, bp_6^1)\}$$

$$CA^d = \{(et^c, dp_1^c), (et^c, dp_2^c)\} \cup \bigcup_{i=1}^{9} \{(bt_4^i, dp_{2i-1}), (bt_5^i, dp_{2i})\}$$

$$CA^e = \{(dt_5^c, ep_1^c), (dt_6^c, ep_1), (pt_1^c, ep_2^c)\} \cup \bigcup_{i=1}^{9} \{(bt_4^i, ep_{2i-1}), (bt_5^i, ep_{2i})\} \cup \bigcup_{i=1}^{18} \{(pt_i^2, ep_i^1)\}$$

$$CA^p = \{18(et_1^c, pp_1^c)\} \cup \bigcup_{i=1}^{18} \{18(et_i, pp_i), (et_i, pp_i^2)\}$$

Weight matrix definition can be expressed as W = BW ⊕ DW ⊕ EW ⊕ PW, while BW is the weight matrix for the BU, DW is the weight matrix for DU, EW is the weight matrix for EU and PW is the weight matrix for PRU. No

weight assignment for the arcs in CA because they are output arcs for transitions and the token values are determined only by the input values.

Assume that X = (BX, DX, EX, PX) represents input token values from place set P and Y = (BY, DY, EY, PY) represents output token values from transition set T, where BX and BY represent the input and output token values in BU; DX and DY represent the input and output token values in DU; EX and EY represent the input and output token values in EU; PX and PY represent the input and output token values in PRU.

Furthermore, Let Z = (BX, DX, EX, PZ), PZ = (X^c, $2X^0$-1, X^1, X^2) (see the discussion for PRU calculations), where X^c, X^0, X^1, X^2 are token input values from the place subsets of PRU.

Finally, values of all transition output tokens can be determined by:

$$Y = H(\langle W, Z\rangle) = H\left(\begin{pmatrix} BW & 0 & 0 & 0 \\ 0 & DW & 0 & 0 \\ 0 & 0 & EW & 0 \\ 0 & 0 & 0 & PW \end{pmatrix} \times \begin{pmatrix} BX \\ DX \\ EX \\ PZ \end{pmatrix}\right) = \begin{pmatrix} H(\langle BW, BX\rangle) \\ H(\langle DW, DX\rangle) \\ H(\langle EW, EX\rangle) \\ H(\langle PW, PZ\rangle) \end{pmatrix}$$

This is a well defined calculation, since we have already defined how to calculate H(<BW, BX>), H(<DW, DX>), H(<EW, EX>) and H(<PW, PZ>).

From the RNPN structure above, an overall computer algorithm can be derived to carry out the game operation. The algorithm below illustrates a scenario in which a computer plays the game against a human operator interactively.

Step 1. Naming conventions:

All positions and lines are numbered as in Figure 9.

Define a board position value as "1" if it is occupied by the computer and "-1" if it is occupied by the operator, "0" when it is empty;

$Z = \{z_1, z_2, ..., z_9\}$ represents game board position values with z_i for position i for $1 \le i \le 9$;

$X = \{x_1, x_2, ..., x_{18}\}$ is the binary representation of set Z;

$L = \{l_1, l_2, ..., l_8\}$ are used to represent the total value on the eight lines;

Step 2. Initialize the game:

Set move count mcount = 0;

Initialize variables $Z = \{0\}$, $L = \{0\}$;

Decide who goes first by negotiating with operator;

Set pflag = p, if operator wants to go first;

Step 3. Initialized weights for PRU:

$$w_{ij} = \begin{cases} \sum_{s=1}^{M} (2x_i^s - 1) \times (2x_j^s - 1), & i \neq j \\ 0, & i = j, \quad 1 \leq i, j \leq 18 \end{cases}$$

where $\{x^1, x^2, ..., x^M\}$ are the initial M patterns that PRU will remember with $x^j = (x^j_1, x^j_2, ..., x^j_{18})$ with $x^j_i \in \{0, 1\}$ for $1 \leq i \leq 18$ and $1 \leq j \leq M$.

Step 4. Determine game status:

4.1 Evaluate the eight lines:

From definition, the line values can be calculated by the following matrix operation:

$$\begin{pmatrix} l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \\ l_7 \\ l_8 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \\ z_8 \\ z_9 \end{pmatrix}$$

4.2 Check winner:

If $|l_i| = 3$, for any i such that $1 \le i \le 8$, then let lv = l_i and go to Step 8;

4.3 Check end of game:

If mcount = 9 (board is full), go to Step 8;

Step 5. Let opponent move, if pflag = p:

5.1 Prompt current board information to screen and Let operator move;

5.2 Take the move on the board by update the board value set Z;

Update move count: mcount = mcount + 1;

Step 6. Pattern recognition process:

6.1 Convert board configuration to binary format:

$x_{2i-1} = 1$ if $z_i = 1$ and 0 otherwise,

$x_{2i} = 1$ if $z_i = -1$ and 0 otherwise, for $1 \le i \le 9$;

6.2 Initialize current board layout $\mu_i(0) = x_i$, $1 \le i \le 18$;

6.3 Pattern generating process:

$$\mu_i(\tau+1) = H\left(\sum_{j=1}^{18} w_{ij}(2\mu_j(\tau) - 1)\right), \qquad 1 \le i \le 18$$

where function H is the binary filter function.

6.4 Iterate until convergence:

Repeat pattern generating process until node outputs remain unchanged with further iteration.

6.5 Generate moves:

for i = 1 to 9 do

$y_i = \mu_{2i-1}(\tau) - \mu_{2i}(\tau)$;

Step 7. Update game board:

for i = 1 to 9 do

{ if $z_i = 0$ and $y_i \neq 0$, then do

$z_i = y_i$;

mcount = mcount + 1;

break this loop;

};

Goto Step 4;

Step 8. Game ends:

Announce Computer as winner, if lv = 3;

Announce Human operator as winner, if lv = -3;

Announce the game is a drawn, otherwise.

Chapter 8

KRIEGSPIEL GAME

1. The Game

The Kriegspiel version of tic-tac-toe has the same game board and play rules as the regular tic-tac-toe game. Unlike the regular tic-tac-toe games, The Kriegspiel version blocks the board from the players by providing information via a referee who supervises the game. Players make moves without knowing where the other player's moves are. The moves are made through referee according the following rules: if the position is already occupied, the player is notified to make another move until it is legal, then the move is made on the game board. This is an example of partial information games (oppose to complete information). The function of referee can be expanded further to provide players more information by allowing them to ask a set of questions. Such questions can be "how many positions are taken on line 2 ?" or "what is the total value of line 7 ?" etc.

2. Logical Components and Their RNPN Designs

Similar to the regular tic-tac-toe RNPN design process, the game model is divided to several components and model them separately: the board (KBU), decision unit(KDU) and execution unit(KEU) and pattern recognition unit(KPRU) for each player. Prefix "K" is added to each unit which stands for Kriegspiel components to distinguish them from the same ones of tic-tac-toe. The intention of this chapter is to give a high level description of model design, since a great detail of RNPN design was shown in last chapter.

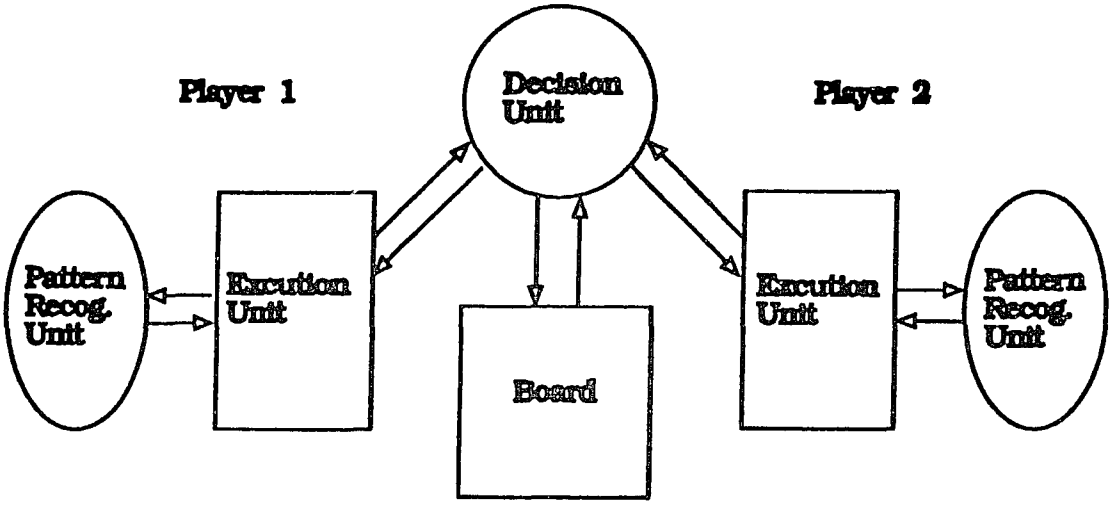


Figure 16. Job Flow Diagram For Kriegspiel Game

The logic connection between these components are illustrated in Figure 1. It is clear from this diagram that execution unit(KEU) does not have access to

game board like the regular tic-tac-toe. The operation starts at KDU. KDU decides who should move, then activate that player, say player 1.

Now the KEU of player 1 takes over, reads its copy of board information, passes it to its KPRU for response. After the KPRU gives a recommended move back, KEU request the move via KDU. KDU determines if the move is legal then passes the information back to the KEU. If the move is illegal, KEU is required to make moves until it becomes legal. When a legal move is made, KDU exams the board to see if the game has ended(win, loss or drawn), if not passes control to the other player. The other player execute the same way as the first player, then passes control back to KEU. Continuing this process, the game will end eventually since there are only finite number of positions.

Comparing these units with the last chapter, the function and model design of the board unit(KBU) and pattern recognition unit(KPRU), where are the same as BU and PRU respectively; the decision unit (KDU) performs the referee function (Kriegspiel) in addition to the regular DU functions; and the execution unit (KEU) makes moves via KDU instead directly to the board. To avoid repetition, only the KDU and KEU are discussed in this chapter.

3. Decision and Execution Units

Besides the role of DU, KDU performs Kriegspiel functions which include accepting moves from KEU, checking legality of these moves and informing KEU, updating board for legal moves. The RNPN design for KDU can be KDU = DU ∪ RU where RU is a referee unit to perform these Kriegspiel functions.

The structure of KEU can also be built on top of EU by making some changes. The function of KEU is similar to EU. When it's activated, KEU calls KPRU for recommended moves by inputing an assumed board configuration. The actual board configuration can not been read. Update mechanism outputs to KDU. When an update is sent to KDU, an acknowledgement is received back. If the feedback is legal, the move is made onto the KBU and it is the other player's turn. If the feedback is illegal, KEU is instructed to make an different move. Because the input for KPRU to generate moves is guessed board information, KEU has to have a device (named IU, the information unit) to store the current predicted board configuration which is modified through out the game for more accuracy. The information unit marks a position as its own when a move to the position is made successfully, or marks a position as its counterpart when a failure of move attempt to that position is acknowledged by KDU.

It is not necessary to show design details here because they can be produced by using the methodology of last chapter. Everyone also has their preference in designing. The RNPN definition of Kriegspiel tic-tac-toe game (KTIC) can be defined the same way as TIC when the RNPN definition of its components KBU, KDU, KEU, KPRU are given.

4. Execution Algorithm

Once again a computer algorithm is developed for software implementation of KTIC to carry out the game operation. The following algorithm illustrates a game played interactively by a computer against a human operator.

Step 1. Naming conventions:

All positions and lines of game board are numbered the same as in last chapter;

Define a board position value as "1" if it is occupied by the computer and "-1" if it is occupied by the operator, "0" if it is empty;

$Z = (z_1, z_2, ..., z_9)$ represents game board position values with z_i for position i for $1 \le i \le 9$;

$\underline{Z} = (\underline{z}_1, \underline{z}_2, ..., \underline{z}_9)$ represents the imaginary game board position

values that KEU maintains with $\underline{z}_i$ for position i for $1 \le i \le 9$;

$X = \{x_1, x_2, ..., x_{18}\}$ is the binary representation position values;

$L = \{l_1, l_2, ..., l_8\}$ are used to represent the total value on the eight lines;

Step 2. Initialize the game:

Set move count mcount = 0;

Initialize variables $Z = \{0\}$, $\underline{Z} = \{0\}$, and $L = \{0\}$;

Set line value lv = 0;

Decide who goes first by negotiating with operator;

Set pflag = p, if operator wants to go first;

Step 3. Initialized weights for PRU:

$$w_{ij} = \begin{cases} \sum_{s=1}^{M} (2x_i^s - 1) \times (2x_j^s - 1), & i \neq j \\ 0, & i = j, \quad 1 \le i, j \le 18 \end{cases}$$

where $\{x^1, x^2, ..., x^M\}$ are the initial M patterns that PRU will remember with $x^j = (x^j_1, x^j_2, ..., x^j_{18})$ with $x^j_i \in \{0, 1\}$ for $1 \le i \le 18$ and $1 \le j \le M$.

Step 4. Determine game status:

4.1 Evaluate the eight lines:

From definition, the line values can be calculated by the following matrix operation:

$$\begin{pmatrix} l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \\ l_7 \\ l_8 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \\ z_8 \\ z_9 \end{pmatrix}$$

4.2 Check winner:

if $|l_i| = 3$, for any i such $1 \le i \le 8$, let lv = l_i and go to Step 8;

4.3 Check end of game:

if mcount = 9 (board is full), go to Step 8;

Step 5. Let opponent move, if pflag = p:

5.1 Prompt current board information to screen and let operator move;

5.2 Make a move based on operator input:

If the position is taken, keep asking new moves until a valid one is made;

Update board configuration by changing value set Z;

Update move count: mcount = mcount + 1;

Set play flag pflag = m;

5.3 Check solution by going to Step 4;

Step 6. Pattern recognition process:

6.1 Convert guessed board information to binary format:

$x_{2i-1} = 1$ if $\underline{z}_i = 1$ and 0 otherwise,

$x_{2i} = 1$ if $\underline{z}_i = -1$ and 0 otherwise, for $1 \le i \le 9$;

6.2 Feed input to KPRU $\mu_i(0) = x_i, \quad 1 \le i \le 18$;

6.3 Pattern generating process:

$$\mu_i(\tau+1) = H\left(\sum_{j=1}^{18} w_{ij}(2\mu_j(\tau) - 1)\right), \qquad 1 \le i \le 18$$

where function H is the binary filtering function.

6.4 Iterate until convergence:

Have pattern generating process repeated until node outputs remain unchanged with further iteration.

6.5 Generate moves:

for i = 1 to 9 do

$y_i = \mu_{2i-1}(\tau) - \mu_{2i}(\tau)$;

Step 7. Update game board:

for i = 1 to 9 do

{ if $\underline{z_i}$ = 0 and y_i = 1, then do (move attempt)

if z_i = 0 then

$z_i = y_i$;

mcount = mcount + 1;

pflag = p;

goto Step 4;

otherwise

$\underline{z_i}$ = -1;

goto Step 6;

}

Step 8. Game ends:

Announce Computer as winner, if lv= 3;

Announce Human operator as winner, if lv= -3;

Announce the game is a drawn, otherwise.

Chapter 9

MODEL IMPLEMENTATIONS

1. Hardware Implementation

Recall the RNPN models in chapter 7, one can easily conclude the following advantages. First of all, RNPN models are graphical. The logical connections between nodes (or devices) are explicitly expressed in the graphs for all components. Secondly, computations at each node are clearly defined in terms of primitive calculation, such as addition, multiplication, binary function and simple logic. Thus these RNPN models can be implemented by hardware in a straight forward manner, especially by VLSI (very large scale integration) technology for which the RNPN models already provided physical layout and can be used as a design template with all nodes as sub-processors.

2. Software Implementation

Based on the algorithms developed in last two chapters, the computer program implementation for these RNPN models are constructed. These

programs are written in TURBO C (BORLAND International), and tested on a IBM AT compatible 386 Personal computer. As we all know the real world is less perfect than theory, the actual "C" program implementation has some necessary changes from the algorithm.

As is was discussed, this model relies on pattern recognition to generate recommended moves. The number of patterns that PRU can recognize is a function of the number of nodes PRU has. This is the capacity issue ([Bal88], [MPRV87]) for PRU and practice shows that the 18 nodes of PRU are not large enough to accommodate the game playing. In order to increase PRU capacity, an extension - interactive level for RNPN is used.

The idea of interactive level was mentioned as part of generalized neural network model[XT90]. Recall the RNPN model definition for PRU, the weight matrix W^0 from place set $P^0 = \{pp_1, pp_2, ..., pp_{18}\}$ to transition set $T^0 = \{pt_1, tp_2, ..., tp_{18}\}$ determines PRU's pattern recognition ability. Let's assume μ_j denotes the input token value in place pp_j for $1 \leq j \leq 18$, RNPN interactive level from P^0 to T^0 can be defined as the following:

Interactive Level 0:

The output token value of pt_i does not depend on its input places but just on its threshold value.

Interactive Level 1:

Input place influences the output token value of transition pt_i in the form

$$\sum_{j=1}^{18} w_{ij}\mu_j , \qquad where\ w_{ij} \in R.$$

Interactive Level 2:

Input places influence the output token value of transition pt_i in the form

$$\sum_{j=1}^{18}\sum_{k=1}^{18} w_{ijk}\mu_j\mu_k , \qquad where\ w_{ijk} \in R.$$

Interactive Level q:

Input places influence the output token value of transition pt_i in the form

$$\sum_{i_1=1}^{18}\cdots\sum_{i_q=1}^{18} w_{ii_1\cdots i_q}\mu_{i_1}\cdots\mu_{i_q} , \qquad where\ w_{ii_1\cdots i_q} \in R.$$

Let $v_i(\tau)$ denote the output token value of transition pt_i and $\mu_j(\tau)$ denote the token value of input place pp_j for $1 \le i, j \le 18$, when each transition fires randomly with interactive level q, its output token value is calculated

according the following equation:

$$
\begin{aligned}
&\nu_i(\tau) = \\
&H\Bigg(\frac{1}{q!}\sum_{i_1=1}^{18}\cdots\sum_{i_q=1}^{18} w_{ii_1\cdots i_q}(2\mu_{i_1}(\tau)-1)\cdots(2\mu_{i_q}(\tau)-1) \\
&+ \frac{1}{(q-1)!}\sum_{i_1=1}^{18}\cdots\sum_{i_{q-1}=1}^{18} w_{ii_1\cdots i_{q-1}}(2\mu_{i_1}(\tau)-1)\cdots(2\mu_{i_{q-1}}(\tau)-1) \\
&+ \cdots + \frac{1}{2}\sum_{i_1=1}^{18}\sum_{i_2=1}^{18} w_{ii_1i_2}(2\mu_{i_1}(\tau)-1)(2\mu_{i_2}(\tau)-1) + \sum_{i_1=1}^{18} w_{ii_1}(2\mu_{i_1}(\tau)-1) + w_i \Bigg),
\end{aligned}
$$

where H is the binary filter function: $H=\begin{cases}1 & x>0,\\ 0 & x\le 0.\end{cases}$ *and* w^i *is threshold of p*

The token value of place pp_i at time $\tau + 1$ is $\mu_i(\tau+1) = \nu_i(\tau)$. This process iterates as a function of time τ. An algorithm similar to the Hopfield algorithm can be generated to recall the closest pattern. The weight matrices $\{ w_{ii_1\cdots i_q} \}$ can be determined by the following equations given that $\{x^1, x^2, ..., x^M\}$

are the initial M patterns for PRU to remember with $x^j = (x^j_1, x^j_2, ..., x^j_{18})$ with $x^j_i \in \{0, 1\}$ for $1 \le i \le 18$ and $1 \le j \le M$.

$$
w_{ii_1\cdots i_q} = \begin{cases} 0, & i = i_1 = \cdots = i_q \\ \sum_{s=1}^{M}(2x_i^s-1)\times(2x_{i_1}^s-1)\times\cdots\times(2x_{i_q}^s-1), & \text{otherwise} \end{cases}
$$

where $1 \le i, i_1, ..., i_q \le 18$ *and* $1 \le q \le 17$.

It can be shown that the memory capacity of PRU with level q is $O(2^q)$ ([Bal88], [XT90]) with $q \leq N-1$, where N is number of nodes (in this case 18). It is easy to see that the current PRU design has interactive level 0 and 1 with $\{w_{ij}\}$ equal to W^0. Therefore, the memory capacity of PRU can be increased if its interactive level is increased, theoretically very large number.

On the other hand, the set of weights $\{ w_{i_1 \cdots i_q} \}$ grows exponentially when higher interactive level q is used. This results the number of program variables needed increases exponentially. Because most computers have physical limitations of their memory, an upper bound for number of variables can be specified exists and thus limits a higher interactive level can be actually implemented. It's been found that the maximum interactive level can be applied is 3 before the PC runs out of memory.

The "C" programs for both tic-tac-toe (TIC) and Kriegspiel tic-tac-toe (KTIC) are written and the TIC program attached in appendix for reference. In actual play, the TIC program shows very good performance, and it wins against average human operator. In order to test the performance of KTIC, a random choice program (plays its positions on the game board randomly) was used to play against KTIC. KTIC shows 85 percent winning record for more than a thousand games played.

Keep in mind that "slow learn" is the only knowledge base KTIC has at this time, and it can perform much better if more sophisticated leaning procedures are used.

Chapter 10

SUMMARY

The following reviews what has been accomplished so far. First of all, a new definition is given mathematically to a modeling structure called neural Petri nets (NPN) as an integration effort to combine neural net and Petri nets models. Secondly, NPN extensions, such as stochastic properties, and restrictions on transitions are added to make NPN more applicable. A chapter is devoted showing how to utilize stochastic neural Petri nets (SNPN) to analyze computing systems. Lastly, using positional games as example, this paper demonstrates several cases of actual designs of NPN in real life scenarios.

One important fact this paper illustrates is that NPN is a good modeling structure. It inherits the advantages of both neural net and Petri net. On one hand, it can perform intelligent functions like pattern recognition, and on the other hand, its mathematical structure is so clearly defined that analysis can be done easily and methodically.

This is only a preliminary study on this subject, and there are many future

studies to be done to make NPN a valid, useful automata. Study in areas such as theoretical analysis of NPN is very important. Such work discusses interesting aspects of NPN like liveness, reachability, and ergodicity. On the application side, chapter 5 demonstrates a fast way to obtain system throughput upper bounds. However, determining the tightness of these upper bounds(i.e., how close these bounds are to the smallest upper bound) remains a future study subject.

As an application of NPN, chapter 8 illustrates building NPN models for a simple Kriegspiel game(tic-tac-toe). The field of Kriegspiel game theory is still very much untouched. In fact, based on the conversation with Professor M. Anshel, well known game theory researchers like E.R. Berlekamp confirm that even the notion of the Kriegspiel games is still in formulating stage. After an electronic search in Science & Technology section of DIALOG databases, only four publications are found related with the terminology "Kriegspiel".

- A technical report "A Programming and Problem Solving Seminar" by Van Wyk, C. and Knuth, D.E. 1979.

- "The Mathematical Gardner" edited by Klarner, D.A. 1981.

and they are found in the MATHSCI® (provided by the American Mathematical Society, Providence, RI).

- "A Program to referee Kriegspiel and Chess" by Buckholtz, T.J.; Wetherell, C.S. 1975.

- "A Director for Kriegspiel, a variant of Chess" by Buckholtz, T.J.; Booth, K.S.; Wetherell, C.S. 1972.
  and they are found in INSPEC (provided by the Institution of Electrical Engineers, London, England).

Appendix

"C" PROGRAM FOR RNPN MODEL TIC

The following is the C program code that implements RNPN model TIC playing regular tic-tac-toe game. This is an interactive version, i.e. the program assumes that its counterpart is an operator, and it prompts the current game board configuration on the screen after it moves then waits for a counter move from the operator.

Code Section

```
#include <stdio.h>
#include <dos.h>
#define D 18
#define limit 10
#define step 3
#define M  7                    /* number of samplers */

/* This program plays Tic-Tac-Toe */
/* This is a interactive version, it assumes that the computer */
/* plays against human operator */
```

```
main ()
{
        int line[8] = {0}, position[3][3] = {0}, post[3][3] = {0};
        int postx, posty, i, j, i1, j1, rot_count, rot_flag;
        int game = 0, board = 9, move = -1;
        char toss = 'n';

/* Start game, take center position */
        printf ("\n\n   Do you want move first?  type <y> or <n>  \n");
        scanf("%c", &toss);
        while ( toss != 'y' )
                {
                        if ( toss == 'n' )
                                break;
                        printf ("\n   Illegal input, please type <y> or <n>  \n");
                        scanf("%c", &toss);
                }

        if (toss == 'n')
                {
                        printf ("\n  Thanks for let me go first.  \n");
                        position[1][1] = 1;
                        board --;
                }
        move = -1;
```

```
while (game == 0) /* Wait response */
{
        if (move == -1 )
        {
/* print the current board to screen */
        printf("\n     This is the current position:\n\n");
        for (i = 0; i <= 2; i++)
                {
                        printf("          ");
                        for ( j = 0; j <= 2; j++)
                                printf ("  %d", position[i][j]);
                        printf ("\n");
                }
        /* Asking for response */
        printf("\n     Please respond by entering position of your move.\n");
        printf("     Example: 3 1 means a stone at row 3 and column 1...\n");
        scanf("%d %d", &postx, &posty);
        while (position[postx-1][posty-1] != 0)
                {
                        printf("\n     Space <%d, %d> is occupied,", postx, posty);
                        printf("  Please choose another move.\n");
                        scanf("%d %d", &postx, &posty);
                }

        /* update the board */
        position[postx-1][posty-1] = -1;
```

```
        move = 1;
        board--;
        goto status;
        }

/* Generate a move, i.e. a recommended configuration */
        /* Make a copy of the board */
        for (i = 0; i <= 2; i++)
                for ( j = 0; j <= 2; j++)
                        post[i][j] = position[i][j];
/* Check if there is a winning position */
        check (post, line);
        /* looking for the winner */
        for ( i = 0; i <= 7; i++)
                {
                        if ( line[i] == 2 )
                                {
                                        if ( i <= 2 )
                                                {
                                                        for ( j = 0; j <= 2; j++ )
                                                                post[i][j] = 1;
                                                }
                                        else if ( i <= 5 )
                                                {
                                                        for ( j = 0; j <= 2; j++ )
                                                                post[j][i-3] = 1;
```

```
                    }
                else if ( i == 6 )
                    {
                        for ( j = 0; j <= 2; j++ )
                            post[j][2-j] = 1;
                    }
                else
                    {
                        for ( j = 0; j <= 2; j++ )
                            post[j][j] = 1;
                    }
                goto update;
            }
        }

/* looking for the possible looser */
    check (post, line);
    for ( i = 0; i <= 7; i++)
        {
            if ( line[i] == -2 )
                {
                    if ( i <= 2 )
                        {
                            for ( j = 0; j <= 2; j++ )
                                post[i][j] = 1;
                        }
```

```
                    else if ( i <= 5 )
                        {
                            for ( j = 0; j <= 2; j++ )
                                post[j][i-3] = 1;
                        }
                    else if ( i == 6 )
                        {
                            for ( j = 0; j <= 2; j++ )
                                post[j][2-j] = 1;
                        }
                    else
                        {
                            for ( j = 0; j <= 2; j++ )
                                post[j][j] = 1;
                        }
                    goto update;
                }
            }
/* Always move to the center, if it is not occupied */
    if (post[1][1] == 0)
        {
            post[1][1] = 1;
            goto update;
        }

/* Call neural net for a recommendation */
```

```
/* Step one: center is occupied */
if (post[1][1] == -1)
        {
                neural(post);
                goto update;
        }

rot_count = 0;
if(post[1][0] == -1)
        rot_flag = 1;
else if (post[0][0] == -1)
        rot_flag = 2;
else
        rot_flag = 0;

/* symmetry reduction */
    if(post[1][0] ==-1 || post[0][1] == -1 || post[1][2] == -1 || post[2][1] == -1)
        {
                while (rot_flag != 1)
                        {
                                rotat(post);
                                if(post[1][0] == -1)
                                        rot_flag = 1;
                                rot_count++;
                        }
        }
```

```
        else if(post[0][0] == -1 || post[0][2] == -1 || post[2][2] == -1 || post[2][0] == -1)
                {
                        while (rot_flag != 2)
                                {
                                        rotat(post);
                                        if(post[0][0] == -1)
                                                rot_flag = 2;
                                        rot_count++;
                                }
                }

/*      printf("\n     This is the neural input:\n\n");
        for (i = 0; i <= 2; i++)
                {
                        printf("          ");
                        for ( j = 0; j <= 2; j++)
                                printf ("  %d", post[i][j]);
                        printf ("\n");
                }
*/
        neural(post);
        while (rot_count !=0 )      /* backward rotation */
                {
                        unrot(post);
                        rot_count--;
                }
```

```
/*
/* print out neural net output */
        printf("\n     This is the neural output:\n\n");
        for (i = 0; i <= 2; i++)
                {
                        printf("          ");
                        for ( j = 0; j <= 2; j++)
                                printf ("  %d", post[i][j]);
                        printf ("\n");
                }
*/

/* Looking for double whammies */
        check (post, line);
        for ( i = 0; i <= 7; i++)
                {
                        if (line[i] == 2)
                                goto update;
                }
        for (i = 0; i <= 2; i++)
                for ( j = 0; j <= 2; j++)
                        {
                                if (post[i][j] == 0 && post[i][0] + post[i][1] + post[i][2] == -1 &&
post[0][j] + post[1][j] + post[2][j] == -1)
                                        {
                                                for (i1 = 0; i1 <= 2; i1++)
```

```
                                        for ( j1 = 0; j1 <= 2; j1++)
                                            post[i1][j1] = position[i1][j1];

                                    post[i][j] = 1;
                                    goto update;
                                }
                        }

update:         /* Project feature board look like, protect exist board position */
        for ( i = 0; i <= 2; i++ )
                for ( j = 0; j <= 2; j++ )
                        {
                                if (position[i][j] != 0)
                                        post[i][j] = position[i][j];
                                if (position[i][j] == 0 && post[i][j] != 0)
                                        post[i][j] = 1;
                        }

/* Make a move          */
        for (i = 0; i <= 2; i++)
                for ( j = 0; j <= 2; j++)
                        if (position[i][j] == 0 && post[i][j] != 0)
                                {
                                        while (move == 1)
                                        {
                                                position[i][j] = post[i][j];
```

```
                                        move = -1;
                                        board--;
                                    }
                                }
        /* In the event no move was made by the above logics, we pick a random one */
        if (move == 1)
                {
                        for (i = 0; i <= 2; i++)
                                for ( j = 0; j <= 2; j++)
                                        if (position[i][j] == 0)
                                                {
                                                        while (move == 1)
                                                        {
                                                                position[i][j] = 1;
                                                                move = -1;
                                                                board--;
                                                        }
                                                }
                }

status: /* Determine win or loss situation */
        /* Compute possible win or loss line values */
        check (position, line);
        /* Checking game status */
        for ( i = 0; i <= 7; i++)
                {
```

```
                if ( line[i] == 3 )
                        {
                                printf ("\n   I won, sorry you lost, Please play again... \n");
                                game = 1;
                        }
                else if ( line[i] == -3 )
                        {
                                printf ("\n  Congratulations! you won the game! \n");
                                game = -1;
                        }
        }
if (board == 0 && game == 0)
        {
                game = 2;
                printf ("\n  We are even, Let's try another game! \n");
        }
/* print the final board to screen */
if ( game != 0)
        {
                printf("\n     This is the final position:\n");
                for (i = 0; i <= 2; i++)
                        {
                                printf("    ");
                                for ( j = 0; j <= 2; j++)
                                        printf ("  %d", position[i][j]);
                                printf ("\n");
```

```
                    }
                }

/* Go back to Loop, if board is still open and no winner */
}

}

check (post, line)      /* Compute total value on all lines */
int post[3][3], line[8];
{
        int i;
        for (i = 0; i <= 2; i++)
                line[i] = post[i][0] + post[i][1] + post[i][2];
        for (i = 3; i <= 5; i++)
                line[i] = post[0][i-3] + post[1][i-3] + post[2][i-3];
        line[6] = post[0][2] + post[1][1] + post[2][0];
        line[7] = post[0][0] + post[1][1] + post[2][2];
}

rotat(matrix) /* This program rotates a 3X3 matrix clockwise 90 degrees */
int matrix[3][3];
{
        int i, j, rot[3][3];
        for (i = 0; i <= 2; i++)
                for (j = 0; j <= 2; j++)
```

```
                    rot[i][j] = matrix[2-j][i];
        for (i = 0; i <= 2; i++)
                for (j = 0; j <= 2; j++)
                    matrix[i][j] = rot[i][j];
}

unrot(matrix) /* This program rotates a 3X3 matrix unti-clockwise 90 degrees */
int matrix[3][3];
{
        int i, j, rot[3][3];
        for (i = 0; i <= 2; i++)
                for (j = 0; j <= 2; j++)
                    rot[i][j] = matrix[j][2-i];
        for (i = 0; i <= 2; i++)
                for (j = 0; j <= 2; j++)
                    matrix[i][j] = rot[i][j];

}

/* This program implements the Generalized Neural structure */

neural (u)
int u[3][3];
{
        int w1[D][D] = {0}, w2[D][D][D] = {0};
```

```
        int i, j, l, m, n, i1, i2;
        int unew[D] = {0}, v[D] = {0};
        int utemp[D] = {0};
        int x = 0, temp1, temp2;
        /* The following threshold should be part of training */
        int w0[D] = { 0, 0, 228, 0, 208, 0, 0, 40, 0, 0, 0, 0, 188,  0, 0, 140, 92, 52 };

/* initial patterns "to be matched" */
        int z[M][D] = {0};
        int tempy[3][3], tempz[D];
        static int y[M][3][3] =     {
                                        {
                                                { 1,  0,  0 },
                                                { 0, -1,  0 },
                                                { 0,  0,  0 }
                                                        },
                                        {
                                                { 1,  0,  0 },
                                                { 0, -1,  0 },
                                                { 1,  0, -1 }
                                                        },
                                        {
                                                {-1,  1,  0 },
                                                { 0,  1,  0 },
                                                { 0,  0, -1 }
                                                        },
```

```
                {
                        {-1,  0,  0 },
                        { 0,  1,  0 },
                        { 0,  0,  1 }
                                },
                {
                        {-1,  0,  1 },
                        { 0,  1,  0 },
                        { 0, -1,  1 }
                                },
                {
                        { 1,  0,  0 },
                        {-1,  1,  0 },
                        { 0,  0,  0 }
                                },
                {
                        { 1,  0,  1 },
                        {-1,  1,  0 },
                        { 0,  0, -1 }
                                }
        };


/* translate board positions to 1 dimension arrays */
        for (I = 0; I <= M-1; I++)
                {
```

```
                for (i = 0; i <= 2; i++)
                        for (j = 0; j <= 2; j++)
                                tempy[i][j] = y[l][i][j];
                trans1 (tempy, tempz);
                for (i = 0; i <= D-1; i++)
                        z[l][i] = tempz[i];
        }

/* Set the weights */
        for (l = 0; l <= D-1; l++)
         for (i1 = 0; i1 <= D-1; i1++)
                for (i = 0; i <= M-1; i++)
                        w1[i][i1] =     w1[i][i1] + (2*z[l][i] - 1)*(2*z[l][i1] - 1);
        for (l = 0; l <= D-1; l++)
         for (i1 = 0; i1 <= D-1; i1++)
                 for (i2 = 0; i2 <= D-1; i2++)
                        for (i = 0; i <= M-1; i++)
                        w2[i][i1][i2] = w2[i][i1][i2] + (2*z[l][i] - 1)*(2*z[l][i1]-1)*(2*z[l][i2]-1);


/*      Set the Diagonal 0 */
/*      for (i = 0; i <= D-1; i++)
        {
                w1[i][i] = 0;
                w2[i][i][i] = 0;
        }
```

```
*/

/* print out the weight matrix */
/*        printf("\n   this is w1 data: \n");
          for (i = 0; i <= D-1; i++)
           for (i1 = 0; i1 <= D-1; i1++)
                    printf("  %d",     w1[i][i1]);
          printf("\n   this is w2 data: \n");
          for (i = 0; i <= D-1; i++)
           for (i1 = 0; i1 <= D-1; i1++)
                    for (i2 = 0; i2 <= D-1; i2++)
                             printf("  %d",     w2[i][i1][i2]);
*/

/* This portion is a neural net based on hopfield net topology */
/* make a new copy */
          trans1 (u, utemp);
          for (i = 0; i <= D-1; i++)
                    v[i] = utemp[i];
          m = 0;
          n = 0;

loop: while (m < step)     /* the beginning of neuron evaluation */
          {
/* calculate new values for all neurons according to weights */
                    for (i = 0; i <= D-1; i++)
```

```
        {
                x = 0;
                temp1 = 0;
                temp2 = 0;
                for (i1 = 0; i1 <= D-1; i1++)
                        temp1 = temp1 + w1[i][i1]*(2*v[i1]-1);
                for (i1 = 0; i1 <= D-1; i1++)
                        {
                                for (i2 = 0; i2 <= D-1; i2++)
                                temp2 = temp2 + w2[i][i1][i2]*(2*v[i1]-1)*(2*v[i2]-1);
                        }
                x = temp1/1 + temp2/2 + w0[i];
/*
/* here is the check point for individual threshold */
                if ( i == 3 )
                        printf ( "\n x %d is:  %d, ", i, x);
                if ( i == 14 )
                        printf ( " x  %d is:  %d ", i, x);
                if ( i == 16 )
                        printf ( " x  %d is:  %d ", i, x);
*/

                unew[i] = binary(x);
        }
```

```
/* update all neurons sync */
                for (i = 0; i <= D-1; i++)
                {
                        v[i] = unew[i];
                }
                m++;                            /*step count*/
                n++;                            /*total loop count*/
        }
/* find out the difference */
        x = 0;
        for (i = 0; i <= D-1; i++)
                x = x + (v[i]-utemp[i])*(v[i]-utemp[i]);
/* update u */
        for (i = 0; i <= D-1; i++)
                        utemp[i] = v[i];

        if (n >= limit)
                {
                        printf("\n WARNING:    Neural Net Looping For Too Long, Approximate\n");
                        goto end_neural;
                }
        /*convergence comparison*/
        if      (x > 0)
                {
                        m = 0;
                        goto loop;
```

```
                }
        else
                {
                        printf("\n  Neural Net Terminates OK!\n");
                }
        end_neural: ; /* end of neural */
        trans2 (utemp, u);
}

/* This is a modified binary function, output 0 when input is 0 */

binary(x)
int x;
{
        if (x >= 0)
                return (1);
        else
                return (0);
}

trans1 (b, utran)       /* this function converts 3X3 matrices to 1XD array */
int b[3][3], utran[D];
{
        int i, j;
        for (i = 0; i <= 2; i++)
                for (j = 0; j <= 2; j++)
```

```
                {
                        if (b[i][j] == 1)
                                {
                                        utran[2*(3*i + j)] = 1;
                                        utran[2*(3*i + j) + 1] = 0;
                                }
                        else if (b[i][j] == 0)
                                {
                                        utran[2*(3*i + j)] = 0;
                                        utran[2*(3*i + j) + 1] = 0;
                                }
                        else
                                {
                                        utran[2*(3*i + j)] = 0;
                                        utran[2*(3*i + j) + 1] = 1;
                                }
                }
}

trans2 (utran, b)       /* this function converts 1XD array to 3X3 matrices */
int b[3][3], utran[D];
{
        int i, j;
        for (i = 0; i <= 2; i++)
                for (j = 0; j <= 2; j++)
                        {
```

```
                    if (utran[2*(3*i + j)] == 1)
                        b[i][j] = 1;
                    else if (utran[2*(3*i + j) + 1] == 1)
                        b[i][j] = -1;
                    else
                        b[i][j] = 0;
                }
        }
        /* END OF PROGRAM */
```

Bibliography

[Bal88] Baldi, P. "Neural Networks, Orientations of the Hypercube, and Algebraic Threshold Functions", *IEEE Transactions on Information Theory*, May 1988, Vol. 34, No. 3.

[BCG82] Berlekamp, E.R; Conway, J.H.; Guy, R.K. *"Winning Ways", Vol.2*, Academic Press, London, England, 1982

[BG85] Bruell, S.C.; Ghanta, S. "Throughput Bounds for Generalized Stochastic Petri Net Models", *Proceedings of the International Workshop on Timed Petri Nets*, Torino, Italy, 1985, pp 250-261.

[FN89] Florin, G.; Natkin, S. "Necessary and Sufficient Ergodicity Condition for Open Synchronized Qeueing Networks", *IEEE Transactions on Software Engineering*, April, 1989, Vol. 15, pp 367-379.

[HN90] Habib, M.K.; Newcomb, R.W. "Neuron Type Processor Modeling Using a Timed Petri Net", *IEEE Transaction on Neural Networks*, December 1990, Vol. 1, No. 4.

[Hop82] Hopfield, J.J. "Neural Networks and Physical systems with emergent collective computational abilities", *Proceedings of the National Academy of Sciences*, USA, April, 1982, Vol. 79, pp2254-2258.

[Jud90] Judd, J.S. *"Neural Network Design and the Complexity of Learning"*, The MIT Press, Cambridge, Massachusetts, 1990.

[Lam88] Lambert, J.L. "Some Consequences of the Decidability of the Reachability Problem for Petri Nets", *Advances in Petri Nets 1988, Lecture Notes in Computer Science*, Springer-Verlag, Vol. 340, pp266-282.

[Lip87] Lippmann, R.P "An Introduction to Computing with Neural Nets", *IEEE Acoustics, Speech, and Signal Processing (ASSP) Magazine*, April, 1987, pp 4-22.

[MBC86] Marsan, M.A.; Balbo, G.; Conte, G. *"Performance Models of Multiprocessor Systems"*, The MIT Press, Cambridge, Massachusetts, 1986.

[MPRV87] McEliece, R.J.; Posner, E.C.; Rodemich, E.R.; Venkatesh, S.S. "The Capacity of The Hopfield Associative Memory". *IEEE Transaction on Information Theory*, 1987, Vol. 33, No. 4.

[Mol85] Molloy, M.K. "Fast Bounds for Stochastic Petri Nets", *Proceedings of the International Workshop on Timed Petri Nets*, Torino, Italy, 1985, pp 244-248.

[Mol87] Molloy, M.K. "Structurally Bounded Stochastic Petri Nets", *Proceedings of the International Workshop on Petri Nets and Performance Models*, Madison, Wisconsin, 1987, pp 156-163.

[Pet81] Peterson, J.L. *"Petri Net Theory and the Modeling of Systems"*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[XT90] Xu, X.; Tsai, W.T. "Constructing Associative Memories Using Neural Networks", *Neural Networks*, 1990, Vol. 3, pp. 301-309.

[ZT85] Zargham, M.R.; Tyman, M. "Neural Petri Nets", *Proceedings of the International Workshop on Timed Petri Nets*, Torino, Italy, 1985, pp 72-77.