A Specification-Based Design Tool For Artificial Neural Networks

Wong Wai
Computer Science Department
Chinese University of Hong Kong

Date : 12 June, 1992

This thesis is submitted as a requirement of the M.Phil. degree of the Computer Science Department, the Chinese University of Hong Kong.

ACKNOWLEDGEMENT

I would like to express my greatest gratitude towards my supervisors, Dr. Peter L.M.Liu and Dr. W.K.Kan. This thesis cannot be prepared without their most helpful advice and guidance.

I should also devote my thankfulness to Prof. T.C.Chen and Dr.L.W.Chan, who have given invaluable advice for my research. Their insight has broadened my view of artificial neural networks.

Thanks is also given to all those who have supported me during these years, including my colleagues, my fellowship members and my good friends.

TABLE OF CONTENT

The two components of the proposed design tool will be introduced below. An outline of the whole thesis will then be given.

1.1. SPECIFICATION ENVIRONMENT

The proposed specification environment is to provide the ANN developers with an *interface* so that the ANN systems to be developed can be easily specified. The main advantage of the specification environment is that the *concurrence* of ANNs is introduced automatically.

The interface consists of *forms* for different major *components* of an ANN. Users are requested to describe the *attributes* of these major components by filling in the *entries* of these forms.

Every neuron is assumed to be operating independently of other neurons by the NNPS, except when two or more neurons are specified by the user to be connected and are hence dependent on each other. The NNPS will introduce as much as concurrence without violating the dependencies. Under the proposed specification environment, which adopts the *dataflow specification* methodology and requires no control information from the users, the users do not have to care about "dynamic" property of concurrence. They just have to specify the "static" *connection* among the neurons.

Uniform notations in specification are also important. A set of specification rules and notations are provided for users. The behaviour of the 3 major components of an ANN, namely, the processing units (the **formal neurons**), the connection among the formal neurons (the **configuration**) and the system-wide coordination among the formal neurons (the **control neuron**), are declared uniformly.

1.2. SPECIFICATION ANALYSIS

A specification analyzer performs *specification analysis* to locate errors and pass the information for future use by the NNPS. The two most important analysis methods are *data dependency analysis* and *attribute analysis*.

Data dependency analysis is performed by the construction of *data dependency graphs* on the basis of the specification. *Cyclic dependency analysis* is then performed to test for any *cyclic dependency* among the elements in the graphs.

Cyclic dependency among neurons is a forever-waiting situation and is also known as *deadlock* [3, 19]. When each of the neurons in a group has to wait for signals from other neurons before it can evaluate its internal parameters, it is possible the neurons may involve in a circular waiting state. The result is that within the cycle, every neuron is waiting for other member(s) to give signal(s) in order to generate signal(s) needed by other members to proceed. This is similar to the case of message deadlock in a

message passing parallel system. Every neuron can be compared to a process in the parallel system, and the signals are similar to messages. Hence authors in [3, 19] use the term "deadlock" to address this problem. However, to emphasize the dependency property, and to avoid the general term "deadlock", the term "cyclic dependency" is employed in this text. A more elaborated discussion of the relation between message deadlock and cyclic dependency is given in Appendix III.

Attribute analysis makes use of *syntactic and semantic checkings*, *simple matching* and *simple computations* to automatically identify errors among *entries* of the *specification forms*. Two categories of analysis, namely, *parameter analysis* and *constraint checking*, are performed. Parameter analysis focuses on input, output and usefulness (involved in generating output) of parameters. Constraint checking enforces consistency, such as number of connections defined in different attributes, among the ANN attributes.

1.3. OUTLINE

The thesis begins with a literature survey in Chapter 2. The overview of the design tool is discussed in Chapter 3. An application of the specification environment on Back-Propagation Network (BP-Net) is described in details in Chapter 4 to explain the features of the specification environment.

For the sake of simplicity, *specification analysis* is discussed in 2 separate chapters. Chapter 5 explains the *data dependency analysis* and chapter 6 addresses the *attribute analysis*. A brief conclusion is then given in Chapter 7

This design tool has been experimented with some typical network cases. The BP-Net is used as the chief example throughout chapters 4 to 6. Other networks such as Boltzmann machine, Perceptron and fragmented examples are introduced wherever appropriate.

2. SURVEY

This survey focuses on the two main issues mentioned in the previous chapter: 1) expressing the concurrent nature of an ANN system through specification environment; 2) verifying the specification through specification analyzer. Existing systems are investigated to see how they address these two issues, and the proposed new approach is drawn into comparison whenever appropriate.

A number of researchers realized the difficulty in developing ANN systems [3, 5, 6, 8, 10, 15, 17, 18, 19, 24, 29, 30, 34]. They have proposed a number of different design tools or simulation systems for developing ANNs. The objective of developing these systems varies from getting some hand-on experience [17] to mapping ANNs onto some existing architecture [3, 5, 8, 15, 19]. For simplicity, these design tools or simulation systems will be called Neural Network Programming Systems (NNPSs) as suggested in [2].

With no exception, the first problem the developers of these NNPS have to solve is "how to specify the concurrent nature of an ANN system". After studying their designs, one can conclude that these systems all offer some kind of solution to this problem.

The second issue has drawn much less attention from the researchers. Not a studied reference has mentioned anything about verifying the correctness of a specification. This may be based on the assumption that ANNs are fault-tolerant system, hence small errors on them can be neglected. This remark is valid in an ANN system, but it is not applicable to the *specification* of such a system. A single error at the specification level will be propagated to a number of processing units.

On the other hand, authors in [3] and [19] have addressed the deadlock (i.e., *cyclic waiting* in this text) problem. Barbosa in [3] proposed a mechanism for implementing deadlock-free Hopfield network but this mechanism is transparent to user, and is therefore not related to specification verification. Kraft in [19] just mentioned the deadlock problem without an in-depth discussion.

2.1. CONCURRENCE SPECIFICATION

The difficulty in developing concurrent systems is well recognized. The number of processing units, neurons, in an ANN can be thousands in number. This is much bigger than the number of processes in other concurrent system, and hence adds difficulty in specifying the concurrence. It is impractical to specify the concurrent execution of the neurons' one by one.

There are three common approaches adopted by NNPSs for expressing the massive parallelism of ANNs. The first one is to consider the neurons as elements of an array, and sequentially update the neurons. The

second one is to map the ANNs onto some existing concurrent architecture. The third one introduces the concurrence by the NNPS instead of by the users. The last approach is adopted by the proposed design tool due to the greatest flexibility and simplicity.

2.1.1. SEQUENTIAL APPROACH

This approach is adopted by many NNPSs running on single processor machines [6, 17, 18, 29, 41] and many small testing programs written in conventional programming languages. An ANN system is always being emphasized as a massively parallel network, but the same remark may not be true with their simulation programs. A sequential updating of the neurons can be viewed as a synchronous network with deterministic updating order. If the system can support random order of evaluation, it can be used for asynchronous and non-deterministic updating [41].

Although this approach does not match the parallel computation characteristic of ANNs, it is widely adopted as a quick solution in many cases, as it avoids the difficulties in developing concurrent programs. In addition, in single-processor machines, this approach is more efficient than concurrent programs as there is no communication and context switching overheads.

SPECIFYING CONCURRENCE

The specification for ANNs concerns very little or even none about the concurrence among the neurons. Users are just required to specify 1) the operations of the individual neuron types with any suitable language, 2) the neurons as an array of elements, and 3) the connections among neurons explicitly or by incorporating into the updating rules. The updating order of the neurons is determined by a loop-like construct within which neurons are processed one by one.

A simple example can be found in [6], in which the user specifies the attributes of the neurons and the configuration through library functions *MakeUnit()* and *MakeLink()*. *MakeUnit()* requests the neuron type and the internal functions and *MakeLink()* takes in type and pattern of connection. A fixed number of neurons can be created by the corresponding number of calls to the library function *MakeUnit()*. The overall control is specified in C language. The updating order is determined by the value of the loop control variable within *for* loops.

After taking all the information from these library functions, the NNPSs compile the specification into executable programs to be run on UNIX. Users have no controls over the concurrent aspect of the neurons, except that they can choose between synchronous or asynchronous updating.

Limitations of the Sequential Approach

This approach has the obvious limitation in simulating the concurrent properties of ANNs. The sequential approach is very efficient in many single-processor machines, but it cannot help to reveal the concurrent features. Researchers may insist that the concurrent property is essential for ANNs.

In addition, in a SUN3/260 machine, a network of 2000 units each with 100 links took 83 seconds to perform 100 simulation steps [6]. This speed is impractical for any system emulating the functioning of living creatures. The second approach aims at solving this problem.

2.1.2. Mapping onto Concurrent Architecture

A natural consequence of the limitation in the sequential machines' approach is the idea of using actual parallel machines as the underlying simulation environment. Examples of these systems are found in [3, 5, 8, 15, 19, 34]. These machines typically have more than one processor though with a less degree of parallelism than the ANNs. For example, the underlying system given in [34] is four Transputers processors, while the number of neurons in a network can be thousands or even millions.

Specifying Concurrence

The specification should now include 1) the mapping between neurons and the basic processing elements of the NNPS, and 2) the mapping between the processing elements and the physical processors. In Transputer, for example, the processing elements are *processes*, and a simple mapping is to represent every neuron with a process. These processes can further be mapped onto different physical processors.

The first mapping bridges between an ANN system and the simulation system, and the second mapping concerns the execution efficiency and limitations. It seems that only the first mapping is relevant to our discussion but, in practice, the second mapping also affects the resultant specification. For example, in [19], an Actor is selected to represent a layer of neurons but not an individual one because of the efficiency consideration.

At the neuron operation level, the specification is very similar to the sequential approach case. Users can represent a single, a layer of, or even a network of neuron(s) with the basic processing elements of the NNPS. This decision will influence the efficiency and even feasibility of the resultant simulations in case system constraints should be observed.

The communication, and hence the connection, among the neurons is usually specified with the built-in constructs for communication of the underlying architecture, such as the OCCAM communication calls in [34]. As a result, users should take care of the concurrence of an ANN system at the specification level when they directly control the communication and synchronization.

Limitation of the Mapping Approach

While this approach has the advantage of allowing users to have direct control over the concurrence of the ANNs, it also requires users to know more about parallel processing, especially the philosophy behind the parallel processing of the underlying machine. In other words, when the system leaves too much control to the users, they may face the old problem again, i.e., the difficulty in developing parallel systems. They have to take care of the communications, synchronization and cooperation among neurons. This approach, therefore, is not so satisfactory as one of the objective of these systems is to free the users from concerning these operation details.

2.1.3. Automatic Concurrence Introduction

A third approach for specifying the concurrence of ANNs is to introduce concurrence not by the users but by the NNPSs [10, 24, 30]. Users only focus on the behavior of individual neurons, the connections and the overall control among them. The NNPSs will make use of their knowledge about ANN systems and the underlying simulation machine to generate the resultant system.

Specifying Concurrence

It is not necessary for users to concern the concurrence, because it is introduced by the NNPSs. The NNPSs usually assumes that the neurons are candidates for parallel processing. Every neuron can execute independent of others, except communication among neurons are required. The communications are derived from the connections. When a neuron should communicate with others to get signals, it may wait until the signal is available, synchronization is thus required. Users hence have control over the concurrence by declaring the connections, and hence communications.

The specification process is very similar to that of the sequential approaches. Users just specify the operation of the neuron types, declare the neurons belonging to each type, and the connections among the neurons, with similar notations as the sequential approach. The major difference is that users do not specify the order of updating the neurons. This is determined by the NNPS automatically from the specification.

The underlying processing performed by the NNPSs is different from that of sequential approaches. The system will take the information from the specification, together with information from the underlying simulation machine, to generate the corresponding simulation. This suggests that it is possible to separate the characteristic of the underlying simulation machine from that of the specification environment. In other words, the NNPSs can generate concurrent simulations for multi-processor simulation machine, and sequential simulations for single-processor machine, although the specification environment always assumes concurrent execution among the neurons.

This specification approach has the advantage of being simple but still allowing users to control the concurrence. It does not require users to specify all concurrence as in the concurrent architecture case, and it can retain the concurrence that is not supported by the sequential approach. In addition, letting the NNPS to determine the concurrence facilitates the same network to be tested on different underlying architecture by compiling the same specification to different execution programs to be run on the underlying machine.

2.2. SPECIFICATION ANALYSIS

None of the materials studied have discussed how they verify the information supplied by the users. Most probably, they support the standard compiler checking such as syntactic and semantic verifications. These checkings are far from enough. The distinct computation model deserves not only a unique specification environment but also a special verification mechanism. *Cyclic dependency*, for example, cannot be detected by these standard compiler checkings.

Checking for the correctness of ANNs can be quite complicated and tedious. The specification of massive number of elements often introduces problems in analysis. In addition, the high degree of parallel operation and communications makes it very difficult to trace the simulations. In case errors occur, the debugging process is again complicated by the presence of massive parallel computation and communications.

2.2.1. MOTIVATION

The fault-tolerant property of ANNs is often used to question the need for extensive specification verification. The operation mechanism of ANNs guarantees that whenever a few neurons in an ANN break down, the network will still function with graceful degrade. When the faults are considered at the specification level, however, the situation is different.

For example, whenever the specification for a neuron type is incorrect, the error will propagate to a large number of neurons. As mentioned in the previous section, it is quite difficult for users to separately declare the operation of every individual neuron. It is natural for one to define a relatively small number of neuron types, and use these types repeatedly for comparatively much larger number of neurons. Error in a single neuron type will thus be inherited by many neurons. As a result, specification verification is required.

2.2.2. CYCLIC DEPENDENCY

Among the reviewed materials, [3] and [19] have mentioned two systems in which a particular problem, the *deadlock* (i.e., cyclic dependency) problem, has been addressed.

Barbosa's Attempt on Deadlock Prevention

Barbosa in [3] addresses the *deadlock* problem explicitly. The objective of Barbosa was to develop an OCCAM implementation for a Hopfield Neural Network. The model will not be discussed in detail but, basically, every neuron will wait for a signal from all its neighbors and a central control before it can proceed.

According to Barbosa, one problem of particular importance is the potential occurrence of communication deadlocks as a result of the un-buffered communication among OCCAM processes representing Hopfield neurons. It was proved that, under the particular communication scheme suggested in [3], a buffer of size ≥ 2 will guarantee deadlock free communication among the neurons.

The scheme proposed, however, is specific to Barbosa's implementation of Hopfield Network with OCCAM. Special communication scheme and special architecture have been assumed. Although it is a very efficient and useful scheme for implementing Hopfield Network, it is not suitable for others. The paper supports, however, the observation for the need of *deadlock prevention*. It also supports the observation that there are *synchronized communication* and *overall network control* in ANNs, although these are widely ignored. These observations will be discussed in depth in later chapters.

Kraft's Model

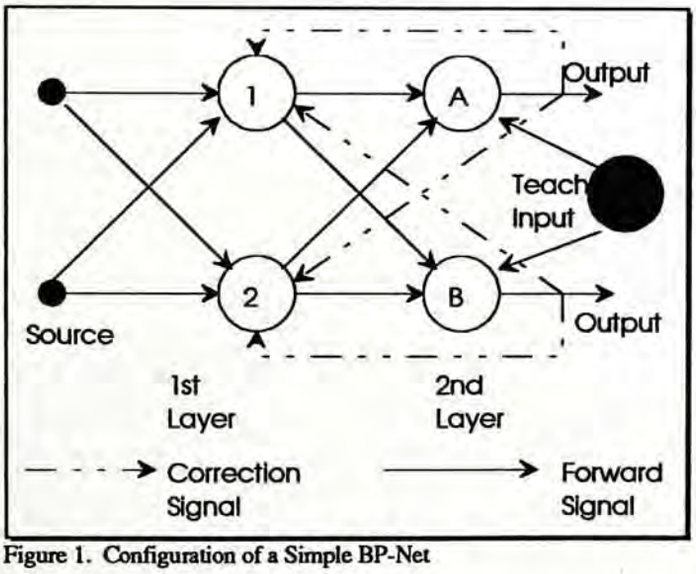


Figure 1. Configuration of a Simple BP-Net

Kraft in [19] has introduced a mechanism to increase the degree of parallelism in a BP-Net running on an Actor model. An interesting result is that the system can prevent cyclic dependency resulting from signal transmission among neurons. A simplified BP-Net is shown in Figure 1. In the network, while neuron *1* sends signal to neuron *A*, it should also wait for message returned from neuron *A* for error correction. In this case, neurons *1* and *A* cannot execute simultaneously. Moreover, it may be possible in some instance that both neurons *1* and *A* are waiting for each other's signal. This is a cyclic dependency case.

In the mechanism developed by Kraft, neurons are declared as actors. When neuron *1* has received its inputs, it will create a duplicated actor *1a* (Figure 2) which will contain the status of neuron *1* of that instance. The new actor, neuron *1a*, can then wait independently for any feedback signal propagated from

neuron *A*. In the meantime, neuron *1* can continue to receive further input signals. In this circumstance, both neurons *1* and *A* can execute in parallel.

An interesting consequence arising from this mechanism is that cyclic dependency is removed from the system as the signal transmission loops no longer exist (Figure 2). Intuitively, none of neurons *1*, *1a* and *A* will wait infinitely as long as there are input from the source. This problem, however, is not the main concern of Kraft as he did not address the issue in details.

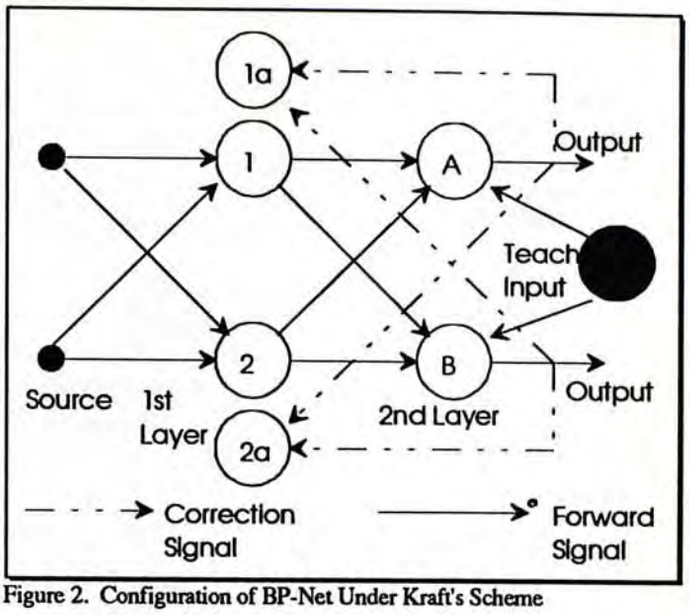


Figure 2. Configuration of BP-Net Under Kraft's Scheme

Besides these two particular systems, other specification environments in the literature do not discuss the cyclic dependency. This will be shown, in later chapters, to be dangerous as the possibility of getting this problem is high and the problem is difficult to locate.

3. THE DESIGN TOOL

The proposed design tool is the front end of an NNPS. The NNPS environment, in addition to the design tool, includes backend compilers to generate software simulations and/or hardware implementation codes. Other constituents, such as libraries for neurons, networks, graphical display and the same may be included [2] . A complete integrated system can thereby be developed for the whole development process. The overall development environment is shown in Figure 3.

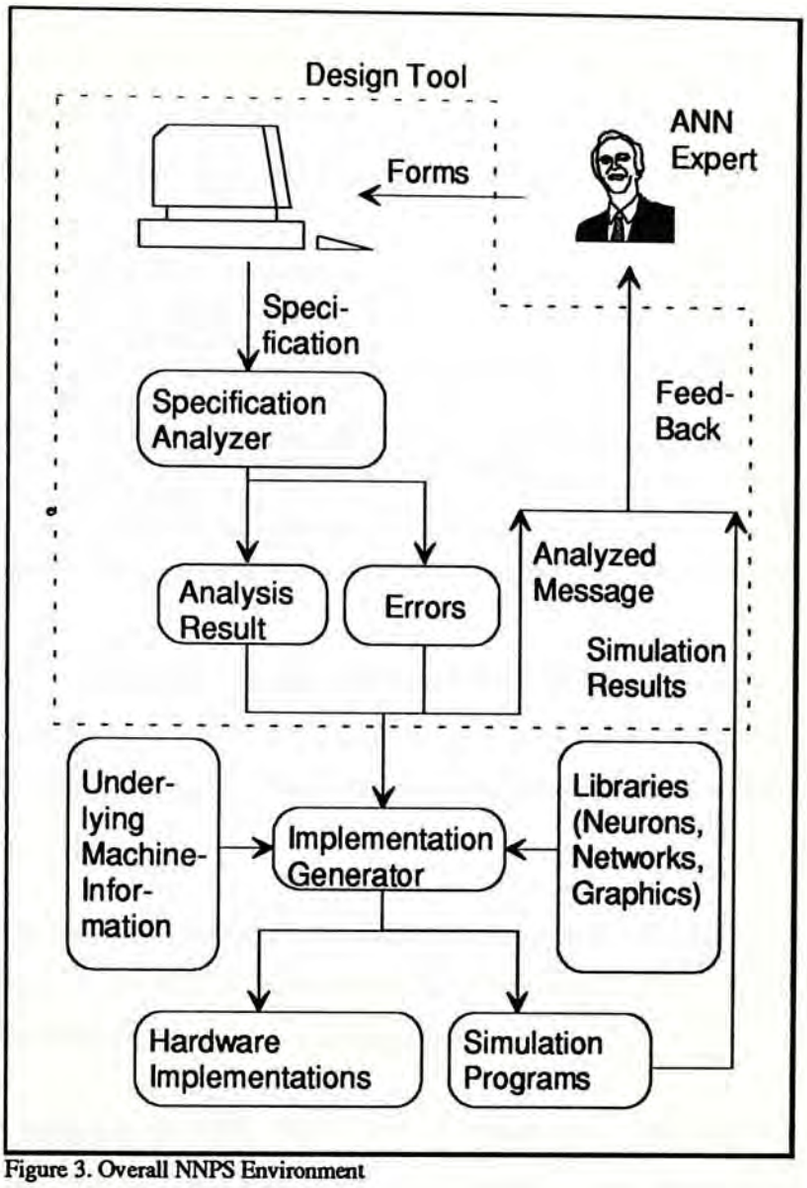


Figure 3. Overall NNPS Environment

Forms are used as the interface between the design tool and users. An ANN specification input through the forms by user is analyzed by a *specification analyzer*. The analyzed results and, if any, errors can be feed-backed to the designer through the computer. The designer can thus modify the design using the information obtained.

The analyzed result from the design tool can be passed to an *implementation generator* that will generate the corresponding software simulation or hardware implementation based on the hardware information of the underlying machine. The libraries contain system-defined neuron types, network configurations and graphics to show the networks in an intuitive manner.

3.1. SPECIFICATION ENVIRONMENT

3.1.1. FRAMEWORK

The framework of an ANN specification has 3 major *components*. The 3 major components are 1) a set of *formal neurons* (FN), 2) the *configuration* (C) of the network, and 3) the *central control* of the network.

The central control is regarded as a high-level neuron and is named *control neuron* (CN) of the system. Each component has a number of *attributes* and every attribute has its *characteristics*.

3.1.1.1. Formal Neurons

A *formal neuron* is a mathematical abstraction of the characteristics of a natural neuron in the brain. Formal neurons are the basic processing elements in an ANN system. A formal neuron can be characterized by the attributes given in Table 1.

Attributes	Description
Neuron Type Name (Nm)	Unique label for different types of neurons.
Input (IN)	Unique labels and the characteristics (range of possible values, discrete or continuous type of values, step size) of the input lines.
Output (Out)	Unique label and characteristic of the output line, similar to IN.
Internal Parameter (IP)	Unique labels for internal parameters and characteristic, similar to IN.
Initial Values (InV)	Initial value for the parameters.
Internal Functions (IF).	State transition of the neurons depending on the input values and the previous neuron state.

Table 1. Attributes of Formal Neuron & Their Description

Every possible combination of the values of the parameters (input, output and internal) is called a *state* of the neuron. The combined states of all the neurons in the network form the resultant state of the network. The **internal functions (IF)** are responsible for updating the output and internal parameter values, hence responsible for state transitions.

The **initial values** of the parameters may influence the subsequent behavior of the neurons and should be carefully controlled. **IN** parameters cannot be initialized, as it comes from other neurons or the environment. **OUT** and **IP** parameters can be initialized to any compatible values.

There should be at least one **internal function** for every **OUT** and **IP** parameters, otherwise the parameters will be kept constant. As there is at least one output from every neuron, there should be at least one **internal function** defined for every neuron.

In common ANNs, the formal neurons will function differently under different *operation phases*. The most common phases are learning and recalling phases. In the *learning* phase, neurons will learn the input patterns and update its parameters. In the *recalling* phase, neurons will just generate output from the input and the internal parameters but will not update these internal parameters. The property of different internal functions under different phases should be included in the specification.

3.1.1.2. Configuration

The **configuration** of an ANN system refers to the connection pattern, hence the communication, of the neurons. A configuration can be specified by the attributes shown in Table 2.

Attributes	Description
Neuron Label (NL)	Unique labels for the neurons.
Neuron character (NC)	Associates the NLs with Nms, & declare the I/O degree.
Connection Pattern (BP)	Connections among the neuron parameters

Table 2. Attributes of configuration & the description

3.1.1.3. Control Neuron

The literature reviewed, except [3], has seldom mentioned the *central control* of an ANN system but it is always essential. For example, in different operation phases, the functioning of the neurons is different and the central control is responsible for the phase transition and notifies the neurons of these transitions.

A central control is also required for *interfacing* the network with the environment. It should direct the input patterns from the environment (e.g., files, camera, scanner) to the first layer and the output from the network to the environment (e.g., files, other networks, other systems). This environment should not be handled by the network itself that is supposed to be independent of the outer environment.

Book-keeping tasks should also be performed by the central control. Usually in the training phase, the input patterns are fed into the input layer and the neurons learn this pattern for a number of iterations. The central control must record the number of iterations for each pattern.

This central control of an ANN is named the **control neuron** of the network. The **control neuron** is considered as a neuron that is connected to every neuron in the network and sends control information to them. This will simplify the specification as the same set of notation used in specifying **formal neurons** can be used. Moreover, considering the central control as a neuron will also simplify the analysis. The attributes of a **control neuron** are listed in Table 3.

Attributes	Description
Global Input (GI)	Input parameters and their characteristics, similar to IN.
Global Output (GO)	Output parameters and their characteristics, similar to IN.
Global Parameter (GP)	Internal parameter and their characteristics, similar to IN.
Input File (InFile)	Input file from the environment
Output File (OutFile)	Output file from environment.
Global Function (GF)	Functioning of control neuron, similar to IF

Table 3. Attributes of control neuron & their descriptions

The **input** and **output files** are responsible for the interface between the system and the environment. The incoming and outgoing messages from the outside world are addressed as *files*, which are not necessarily conventional files in computer storage devices but can be any stream of data. Moreover, these files may be different at different *operation phases*. There are *records* within the files that represent different patterns being input to and output from the network.

The roles of the **global functions** are versatile. In addition to state transition of the **control neuron**, which is similar to that of **internal function** of **formal neuron**, they should also determine the operation

phase transition. Moreover, it should map the **input file** onto the input of the first layer, and output from the last layer to **output file**.

3.1.2. DATAFLOW SPECIFICATION

Dataflow specification approach is adopted because it is more natural in specifying the behavior of the neurons [35]. In addition, this specification approach will facilitate the specification analysis.

3.1.2.1. Absence of Control Information

The first characteristic of *dataflow specification approach* is lack of *control* information. In conventional programming languages, control is effected by a mixture of implicit and explicit structures. The explicit control structures are the programming constructs such as loops and conditional branching. Implicit control means that the textual order of the statements will determine the execution order of the instructions.

In dataflow specification, on the other hand, no control flow information is defined for the execution of a system. All specifications are *data-oriented*, i.e., they just determine how values are assigned to a data under different conditions. Such definitions are called *equations*. Every equation is responsible for specifying the behavior of a datum. At the specification level, the sequence of evaluating the data is of no concern to user. The dependency relations among the data determine the resultant execution sequence automatically. Independent data can be evaluated in parallel.

3.1.2.2. Single-Valued Variables & Explicit Time Indices

The second feature about dataflow specification is the use of *single-valued variables*. In conventional programming languages, variables can assume different values at different times. In dataflow specification approach, variables can only take *one* value forever and hence known as *single-valued variables*. As a result, there is at least and at most one assignment statement for every variable in a specification.

A direct consequence of using single-valued variables is the introduction of *explicit time indices* in the variables. Two variables are considered to be identical if their names and indices (if any) are identical. Hence different elements of the same array are considered to be different. In this way, a variable *V* can be subscript with *explicit time indices* so that it resembles an array in conventional programming language. Every element of this "array" records value of variable *V* in different *instances*, i.e., V[1] records values of V at first instance, V[2] second instance and V[k] the k-th instance. Repeated use of the element in this array avoids the need for introducing new variable each time. In chapter 6 one can see how this explicit time index can simplify the specification analysis.

3.1.2.3. Explicit Notations

The introduction of *explicit time indices* as a consequence of using *single-valued variables* is just one example of explicit notations. Explicit connection pattern, explicit input and output properties of parameters, and explicit type and range of values for parameters are also included.

Explicit connection pattern requires users to explicitly state the connections among the parameters of the neurons. The explicit connection pattern requires users to state the connection explicitly, and hence draw the user's attention. In addition, once the connection is defined, no modifications on other parts of the specification will introduce additional errors on the connection part.

Explicit type and *range of values* for parameters are useful in checking. Connected parameters are compatible only if they have the same type (discrete, continuous) and same range (bounded, unbounded, binary). In conventional programming languages, the type compatibility is not required for the range as the range of values represented by a particular type of data is fixed.

In addition to explicit declaration on types and ranges, *input* and *output* properties of parameters are also stated explicitly. This is solely for sake of checking. "Input parameters should not be updated" is just an example of the many checkings performed. This is different from conventional programming languages, in which the input and output properties of the parameters are determined by the operation on it, not by the explicit declaration of the parameters.

3.1.3. USER INTERFACE

The interface between the user and the design tool are *specification forms*. Each component of the ANN system is specified with one form. Within each form, there are *entries* corresponding to *attributes* of the components, and the filled *values* are the *characteristics* of the attributes.

Form 1 shows the layout of the *specification forms* corresponding to the components. It is not surprising that they resemble Tables 1 to 3, which show the attributes of the components. Each entry in the forms corresponds to exactly one attribute of the component.

*** FORMAL NEURON ***	*** CONFIGURATION ***	*** CONTROL NEURON ***
NEURON TYPE : INPUT : OUTPUT : INTERNAL PARAMETER : INITIAL VALUES : INTERNAL FUNCTION :	NEURON LABEL : NEURON CHARACTER : CONNECTION PATTERN :	GLOBAL INPUT : GLOBAL OUTPUT : GLOBAL PARAMETER : INPUT FILE : OUTPUT FILE : GLOBAL FUNCTION :

Form 1. Basic Entries Headings of the Design Forms

The *specification form* interface has advantages over other approaches. It will remind users about the essential attributes of a network, thus reducing the possibility of missing information. With a suitable editing environment, the system can restrict the inputs of the users. The forms also group relevant information together to facilitate the modification process. Furthermore, developed forms serve as a prototype to be referenced by other users.

3.2. SPECIFICATION ANALYSIS

The *specification analysis* is based on *data dependency analysis* methodology. Data dependency analysis is used for locating *cyclic dependency* and for parameter analysis. Attribute analysis employs *syntactic* and *semantic* checking, *simple matching*, and *simple computations* to locate errors.

3.2.1. DATA DEPENDENCY ANALYSIS

Data dependency analysis focuses on the dependency among data to determine the functioning of an ANN system. "Data" in an ANN system are the **internal parameters** of the **formal neurons** and **global parameters** of the **control neuron**. The analysis can determine such as *order of evaluation* for the parameters in the neurons, the *completed updating* of all parameters except input parameters, and the *usefulness* of parameters.

Data dependency graphs are the core constructs for data dependency analysis. A data dependency graph is a directed graph. *Nodes* in the graph are used to represent data, while a *directed edge* from node x to node y means that datum y depends on datum x, or x determines y. Datum y depends on x means that the value of y is determined by some evaluation involving x (denoted as $x \rightarrow y$, or equivalently, $y \leftarrow x$).

The dependency graph can be used for determining the *order of evaluation* for the data. If y depends on x ($y \leftarrow x$), the evaluation of x should precede the evaluation of y. A cyclic dependency is a problem in which some data are depending on each other. In this case, the data involved do not have a proper order of evaluation (e.g., $x \leftarrow y \leftarrow x$).

The dependency graph can also be used for other checkings. For example, a datum with no incoming edges must be an input datum, otherwise some data are not updated. A datum determining no other data in the graph should be an output datum, otherwise the datum is useless.

3.2.2. ATTRIBUTE ANALYSIS

Syntactic checking is used for enforcing the correctness of the form entry syntax. *Semantic checking* is used for enforcing unique, well-defined labels for names, and compatibility among the parameters in both *type* and *range of values*.

Tailor-made *simple matching* is applied on checking the correct matching among the *characteristic* of the *attributes*. For example, the number of neurons defined in the **neuron label** entry should match with that defined in **connection pattern**, and the number of input and output parameters defined in the **formal neuron** form should agree with that defined in the **neuron character** entry. These checkings are straight forward but useful.

4. BP-NET SPECIFICATION

In this chapter, an example specification on a Back-Propagation Network (BP-Net) [33] is given to illustrate the features of the *specification interface*. It will explain the use of *specification forms*, *form entries*, the use of *notations* and the *characteristics* of dataflow specifications. A formal syntax definition can be found in Appendix I. Additional specifications of other example types of networks can be found in Appendix IV.

The basic paradigm of a BP-Net is introduced first. After that, the specifications on the **formal neuron**, **configuration** and **control neuron** will be discussed separately.

4.1. BP-NET PARADIGM

BP-Nets are characterized by their *error correction signals* propagated back from output layers to input layers. Input layer is the layer (group) of neurons connected to the input source, and output layer is connected to the output target. Layers between these two layers are called hidden layers. The signals propagated from the input to the output layers are called the *forward signal*. The *output* from the output layer is compared with the *teaching* signals. The difference is used to compute the *error correction signals* that are then propagated back from output layer to input layer.

In [33] the characteristic of a BP-Net is described at two levels. The first one is the *individual neuron* level. The behavior of individual neuron is described with the aid of equations such as how to produce outputs and how to update internal parameters with respect to the input values. The equations 4.1 to 4.6 shown below are included for completeness but their exact meanings are unimportant for the discussion of the specification.

The second level of description is the *overall topology* of the whole network. The main concerns are how to connect the neurons together, how the signals are propagated from the input source to the output, and how the neurons should update with respect to others. The paradigm of the sample BP-Net is illustrated below.

4.1.1. NEURONS OF A BP-NET

Every neuron in the *training* phase (or *learning* phase in some literature) involves in two types of signal processing. In the forward propagation direction, one neuron, say, the j-th one, is responsible for computing the activation (A_j) and output (o_{pj}) values that are given by

$$A_j = \sum_i w_{ji}\, o_{pi} + \theta_j \qquad (4.1)$$

$$o_{pj} = \frac{1}{1 + e^{-A_j}} \qquad (4.2)$$

where w_{ji} is the weight of the connection from *i*-th neuron to the *j*-th one, o_{pi} is the output from *i*-th neuron and θj is the threshold of the *j*-th neuron for being activated.

In the backward direction from outputs to inputs, the main responsibility of the neuron is to generate the error correction signal (δ_{pi}) and the change of weights (Δw_{ji}). The error correction is declared as

$$\delta_{pj} = (t_{pj} - o_{pj})\, o_{pj}\, (1 - o_{pj}) \qquad (4.3)$$ neuron in output layer

$$\delta_{pj} = o_{pj}\, (1 - o_{pj}) \sum_{k} \delta_{pk} w_{kj} \qquad (4.4)$$ neuron in hidden layers,

where t_{pi} is the teaching signal and δ_{pk} is the correction signal from the *k*-th neuron in the subsequent layer.

Moreover, to increase the learning rate without leading to oscillation, a momentum term is included in the generalized delta rule to give

$$\Delta w_{ji}\,(T+1) = \eta(\delta_{pj}\, o_{pi}) + \alpha\, \Delta w_{ji}\,(T) \qquad (4.5)$$

$$w_{ji}\,(T+1) = w_{ji}(T) + \Delta w_{ji}\,(T) \qquad (4.6)$$

where Δw_{ji} is the change of w_{ji}, the subscript *T* indexes the presentation number, η is the learning rate and α is the momentum.

On the other hand, the functioning of the neurons in the *recalling* phase is simple. The main function of the neurons is just the computing of the activation (A_j) and output (o_{pj}) with the same equations defined as before in equations (4.1) and (4.2).

4.1.2. CONFIGURATION OF BP-NET

The configuration of a the BP-Net is shown in Figure 4. This is a simplified view of a network, in which only the forward connections, but not the feedback ones, are shown. The network contains 5 layers of neurons. There are 256 neurons at the first layer, and this is also the size of the input pattern. The number of neurons decreases by half in every subsequent layer. Every neuron is connected to *all* the neurons of the preceding and succeeding layers, except the first (i.e., input) and last (i.e., output) layers. The neurons in the first layer are fully connected to the signal source, while the neurons in the last layer send output to the environment. This is just a sample configuration for demonstrating the capability of the specification so the implication of such a configuration will not be discussed.

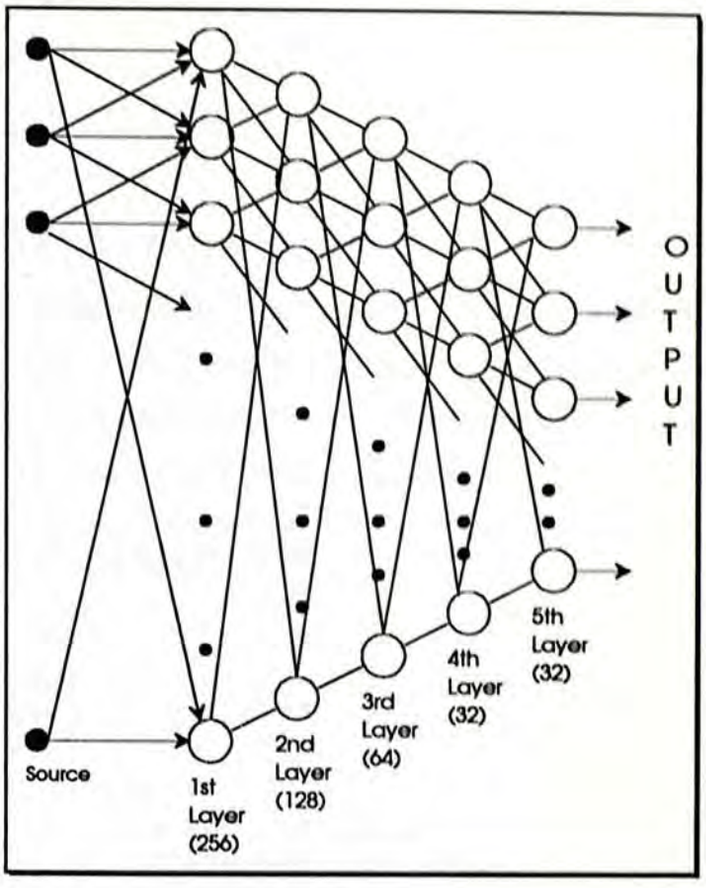


Figure 4. Configuration of Example BP-Net

4.2. CONSTANT DECLARATIONS

Before specifying the major components, it is convenient for one to declare some meaningful alias names for constant values. This is common in many programming languages and is also supported by the design tool in question. To provide for this, a **Constant Declaration** form will allow users to declare constant values by names. In the sample BP-Net, the constants for the network configuration include the number of layers (*NumOfLayer*) which is *5* and size of the input pattern (*PatSize*) which is *256*. Furthermore, the number of maximum iterations (*MaxIter*) for a pattern is defined as *1000*, threshold (*Theta*) is *0.1*, momentum (*Alpha*) is *0.9* and the learning rate (*Neta*) is *0.5*. These are shown in Form 2. These values of the constants are of no concern for the discussion.

```
*** CONSTANT DECLARATION ***

#DECLARE ( NumOfLayer, 5 );
#DECLARE ( PatSize, 256 );
#DECLARE ( MaxIter, 1000 );
#DECLARE ( Theta, 0.1 );
#DECLARE ( Alpha, 0.9 );
#DECLARE ( Neta, 0.5 );
```

Form 2. Constant Declaration Form for BP-Net

4.3. FORMAL NEURON SPECIFICATION

A specification form for the *hidden* unit of the BP-Net is shown in Form 3. In a BP-Net, there are three groups of neurons, namely the *hidden*, *input* and *output* units. These three types of neurons are specified separately. They are named as *BPInput*, *BPOut* and *BPHidden* respectively. Only the specification for *BPHidden* is shown and discussed in details here. The others can be found in Appendix IV.

```
*** FORMAL NEURON ***

NEURON TYPE :
     BPHidden( M, N, Id );
/*   M -- number of neuron in previous layer, equals to number of input
     N -- number of neuron in next layer, equals to number of output
     Id -- position of current neuron in the layer. */

INPUT :
     I [ 1..M ] RANGE (0, 1) INCLUSIVE CONTINUOUS;          /* input from previous layer */
     ( WI [1..N], DeltaI [1..N] ) CONTINUOUS;               /* correction signals from following layer */

OUTPUT :
     Out RANGE (0, 1) INCLUSIVE CONTINUOUS;                 /* output to following layer */
     (W [1..M], Delta) CONTINUOUS;                          /* correction signals to previous layer */

INTERNAL PARAMETER :
     (A, DeltaW[1..M]) CONTINUOUS;                          /* activity and computed correction value */

INITIAL VALUE :
     W[*][0] = 0; Delta[0] = 0; DeltaW[*][0] = 0;           /* all initial values are 0 */

INTERNAL FUNCTION :
/*
 * Internal function common to all phases
 */
     A[T] = I[T] * TRANSPOSE( W[T] ) + Theta;
     Out[T] = 1 / ( 1 + EXP( - A[T] ) );
/*
 * Internal functions for Training phase
 */
     Training {
          Delta[T] = Out[T] * (1 - Out[T]) * (DeltaI[T] * TRANSPOSE( WI[T] ));
          DeltaW[*][T] = Neta * Delta[T] * Out[T] + Alpha * DeltaW[*][T-1];
          W[T+1] = W[T] + DeltaW[T]; };
/*
 * Internal functions for Recalling phase
 */
     Recalling {
          DeltaW[T] = 0;
          Delta[T] = 0;
          W[T+1] = W[T];      };
```

Form 3. A Form for Formal Neurons of a BP-Network

4.3.1. MAPPING THE PARADIGM

The first concern for the specification is how it can capture the properties of a BP-Net formal neuron to ensure that its functioning can be accurately simulated. The functioning of the formal neurons is expressed in terms of symbols and equations, but these equations, in their original mathematical forms, are not suitable for specification purpose as it is quite difficult to type in these equations. Proper ways of mapping these equations into expressions that can be taken by the computer have thus to be devised.

4.3.1.1. Mapping Symbols onto Parameter Names

Limited by the standard alphabets available in the standard keyboard, convenient parameter names with standard alphabets are used to represent the Greek symbols in equations (4.1) to (4.6). These names are shown in Table 4.

Symbol	Name	Use of Parameters
o_{ni}	I	Outputs from preceding neurons
δ_{nk}	DeltaI	Correction signals from following neurons
w_{ki}	WI	Weights of connection to following neurons
t_{ni}	Teach	Teaching signal from the environment
o_{ni}	Out	Output of this neuron
δ_{ni}	Delta	Correction signal of this neuron
w_{ji}	W	Weights of connection into this neuron
A_j	A	Activation value
Δw_{ji}	DeltaW	Weight changes of this neurons
θ	Theta	Threshold value
η	Neta	Learning rate
α	Alpha	Momentum

Table 4. Symbols and Their Names

The output from other neurons o_{pi} is mapped onto the local input *I* of the neuron under investigation. This will clarify the role of the parameter. The teaching signal t_{pj} from the environment, which is mapped to the parameter *Teach*, is not included in the specification for *BPHidden*. This is because the hidden units do not receive teaching signals from the environment. This parameter will therefore appear only in the specification form for *output* units (i.e., *BPHidden*).

4.3.1.2. Mapping Neuron Equations onto Internal Functions

The mapping between the mathematical neuron equations and the **internal function** in the **formal neuron** form is shown in Table 5.

There is no need to use summation symbol in the **internal functions** because matrix operations are employed. For example, the expression

$$\sum_i w_{ji}\, o_{pi}$$

is replaced by the product of two matrices *I* and *W*. The function *Transpose* is used to transpose the 1xN matrix *W* so that the product is a scalar value. The other notations are straight forward, except the explicit time index *T*, which will be discussed in the next section.

Neuron Behavior Equations	Internal Functions
$A_j=\sum_i w_{ji} o_{pi} + \theta_j$	A[T]=I[T] * Transpose(W[T])+Theta
$o_{pj}=\frac{1}{1+e^{-A_j}}$	Out[T]=1 / (1 + Exp(- A[T]))
$\delta_{pj}=o_{pj}(1 - o_{pj})\sum_k \delta_{pk} w_{kj}$	Delta[T]=Out[T] * (1-Out[T]) * (DeltaI[T] * Transpose(WI[T]))
$\delta_{pi}=(t_{pi}-o_{pi}) o_{pi} (1 - o_{pi})$	Delta[T]=(Teach[T] - Out[T]) * Out[T]*(1 - Out[T])
$\Delta w_{ji}(t+1)=\eta(\delta_{pi} o_{pi})+\alpha \Delta w_{ji}(t)$	DeltaW[*][T]=Neta * Delta[T] * Out[T] + Alpha * DeltaW[*][T-1]
$w_{ji}(t+1)=w_{ji}(t)+\Delta w_{ji}(t)$	W[T+1]=W[T] + DeltaW[T]

Table 5. Correspondence Between Neuron Behavior Equations and Internal Functions

4.3.2. FORM ENTRIES

The notations of the specification will be explained in more details, and the possible variations on the options will also be mentioned. This will give an overview on how to use the **formal neuron** form to capture the characteristic of different types of neurons.

4.3.2.1. Neuron Type Entry

```
NEURON TYPE :
    BPHidden( M, N, Id );
/*  M -- number of neuron in previous layer,
    N -- number of neuron in next layer,
    Id -- position of current neuron in the layer. */
```

Form 4. Example Neuron Type Entry

The **neuron type** entry for the *hidden unit* is reproduced in Form 4. The neuron type is augmented with *neuron definition parameters* (NDPs). These input values are used in specifying *configuration* attributes such as the number of input and output signals, number of connection weights and the like. In this form, *M* is the number of neurons in the previous layer, and hence the number of input signals and weights. *N* is the number of neurons in the next layer. *Id* is the position of this neuron in its own layer. These neuron definition parameters should be compilation constants, i.e., they can be evaluated at compile time.

4.3.2.2. Input, Output and Internal Parameter Entries

```
INPUT :
    ❶I [ 1..M ] ❷RANGE (0, 1) INCLUSIVE CONTINUOUS;     /* input from previous layer */
    ( WI [1..N], DeltaI [1..N] ) CONTINUOUS;❸            /* correction signals from following layer */

OUTPUT :
    Out RANGE (0, 1) INCLUSIVE CONTINUOUS;               /* output to following layer */
    (W [1..M], Delta) CONTINUOUS;                        /* correction signals to previous layer */

INTERNAL PARAMETER :
    (A, DeltaW[1..M]) CONTINUOUS;                        /* activity and computed correction value */
```

Form 5. Example Input, Output and Internal Parameter Entries

The parameters have been grouped as *input*, *output* and *internal* ones. The first three (*I*, *WI* and *DeltaI*) are input, the next three (*Out*, *W* and *Delta*) are output, and the last two (*A* and *DeltaW*) are internal parameters. The specifications for the entries are extracted in Form 5.

❶**Parameter Declaration.** The parameters are specified with their name, optional *array* size declarations, and their *properties*. The array declaration is Pascal-like. A *square bracket* immediately after a parameter name indicates that the parameter is an array. The values inside are the *index boundaries*. Hence there are *M* inputs having the same name *I* but with different subscripts.

Some parameters are defined as arrays while some are not. It depends on whether or not the parameter is used for storing a number of different values at the same time. For example, although their values are sent to more than one receiver, the output parameters *Out* and *Delta* are defined with no subscripts, as the same value is broadcast to all neurons. On the other hand, the input parameter *I* is declared as an array because it should receive different values from different sources at the same time.

❷**Parameter Properties.** The first property associated with the parameters is an optional *range boundary* for the values that a parameter can assume, either inclusively or exclusively. The second attribute is the *type* of the parameter, which may be continuous, binary, integral or discrete with any step size. In case of discrete values, the third attribute, the step size, should also be given.

In the given forms, for example, the value parameter *I* can assume any values between 0 and 1 inclusively, while the value of *DeltaI* is unbounded and continuous. These properties are not given in the equations 4.1 to 4.6 but they are required for analysis purpose. A summary of the properties of the parameters is given in Table 6.

Name	Characteristics
I	Continuous, bounded by (0,1)
DeltaI	Continuous, unbounded
WI	Continuous, unbounded
Teach	Continuous, bounded by (0,1)
Out	Continuous, bounded by (0,1)
Delta	Continuous, unbounded
W	Continuous, unbounded
A	Continuous, unbounded
DeltaW	Continuous, unbounded

Table 6. The Properties of the Parameters

❸**Reasons for Multiple Input/Output Parameters.** One may wonder why there is more than one parameter in both the **input** and **output** entries. Most of the standard text on ANNs would classify the output from preceding layers *I* as input to the neuron, but disregard the other two correction signals from succeeding layers as inputs. They assume that the neurons have ways to obtain the values of the correction signals *DeltaI* from the succeeding layers and connection weights *WI* among the neurons.

The proposed design tool, on the other hand, does not allow accessing information of other neurons except through signal passing. There is no global information accessible by neuron except system constants.

Hence the correction signals *Deltal* and the connection weights *WI* must be imported. As a result, there are three input parameters.

The same argument applies for the parameter declaration in the **output** entry. As the neurons should export the correction signals, connection weights and their standard output, there are three output parameters from the hidden units.

4.3.2.3. Initial Value Entry

This is the entry for *initializing* parameters. The expressions in this entry are used to assign constant values into some particular *instances* or *elements* of the parameters. The **initial value** entry is reproduced in Form 6.

```
INITIAL VALUE :
    W[*][0] = 0; Delta[0] = 0; DeltaW[*][0] = 0;          /* all initial values are 0 */
      ❷            ❶
```

Form 6. Example Initial Value Entry

❶Temporal Dimension & Instances of Parameters. The use of *explicit time indices* in chapter 3 defines an *instance* of parameter *P* as an element of the array of parameter in the temporal dimension. This dimension is always the last dimension of a parameter. Hence a constant value *k* in the last dimension of a parameter *P* (i.e., *P[k]*, if *P* is scalar) indicates the *k*-th instance of *P*. The system is assumed to start from instance 1 so a zero or negative value (e.g., *Delta[0]* in the form), addresses values of *Delta* before the system starts. Furthermore, a "*" is a wildcard indicating every instance if used in the last dimension. This is not found in the given form.

❷Elements of Non-temporal Dimensions. It is also possible to initialize the elements of the array in the non-temporal dimensions. This is similar to normal array addressing, in which the index value used must be within the array range. A "*" is also used as the wildcard to indicate every element in that dimension. In the given example, *W[*][0] = 0* says that every element in the first dimension of *W* is initialized to 0.

4.3.2.4. Internal Function Entry

Internal functions are the denotations of equations 4.1 to 4.6 (except equation 4.3 that is for output neuron only) in a form understandable to the design tool. The specification for **internal function** entry is extracted in Form 7.

```
INTERNAL FUNCTION :
/*
* Internal function common to all phases
*/      ❶          ❷
    A[T] = I[T] * TRANSPOSE( W[T] ) + Theta;
    Out[T] = 1 / ( 1 + EXP( - A[T] ) );
/*
* Internal functions for Training phase
*/          ❹
    Training {
        Delta[T] = Out[T] * (1 - Out[T]) * (DeltaI[T] * TRANSPOSE( WI[T] ));
                   ❸
        DeltaW[*][T] = Neta * Delta[T] * Out[T] + Alpha * DeltaW[*][T-1];
        W[T+1] = W[T] + DeltaW[T];  };
/*
* Internal functions for Recalling phase
*/
    Recalling {
        DeltaW[T] = 0; ❺
        Delta[T] = 0;
        W[T+1] = W[T];      };
```

Form 7. Example Internal Function Entry

❶**The Temporal Index.** Temporal index, denoted as *T* in the equations, is a pre-defined parameter. It is always use in the last dimension, the temporal dimension. It is used to denote the "current" instance of a parameter. The exact value of *T* is hence depending on the number of iterations the system has had.

This temporal index *T* is not used to refer parameters among different equations but just to express temporal relation within the same equation. Consider the equation

W[T+1] = W[T] + DeltaW[T].

The equation indicates that the value of *W* is dependent on the values of itself and another parameter, *DeltaW*, which are both *one* instance earlier. The exact value of *T* at any time is immaterial, but just the relative relation is essential. Hence one can rewrite the equation in an equivalent form

W[T] = W[T-1] + DeltaW[T-1].

❷**Matrix Operations.** The notation for matrix multiplication is exactly as those for scalar ones, except that the orientation of the matrices may be adjusted. No special notation is required. The function *Transpose* serves this purpose. All one-dimensional arrays are viewed as 1xN matrices. The system makes use of the characteristic of the parameters to determine the nature (i.e., scalar or matrices) of the operators automatically. The dimensions of the equations will be checked to enforce proper operations.

❸**Wildcard Assignment.** The use of star (*) in a dimension has the same effect as that in **Initial Value** entry. Every element in that dimension is assigned the value resulted from the expression.

❹**Operation Phases.** Two *operation phases*, *Training* and *Recalling*, are defined in the example. Equations restricted under the scope of any phase are used only when the system is in that phase, and equations that are not bounded by the scope of any phase can be applied at all times.

❺**Parameters Kept Constant.** There are parameters that are changing in one phase (*Training*) but are kept constant in the other (*Recalling*), such as *W*, *Delta* and *DeltaW* in the example. *W* is referenced in the *Recalling* phase so its value of the previous instance is propagated to the new instance through direct assignments (*W[T+1] = W[T]*). On the other hand, *Delta* and *DeltaW* are arbitrarily set to *0*. These equations seem to be unnecessary but the system requires the user to explicitly state the computation for all parameters of *every* instance. This ensures that every instance of the parameters can be evaluated.

4.4. CONFIGURATION SPECIFICATION

```
*** CONFIGURATION ***

NEURON LABEL :
    {{  Neu [x] [y];                    /* Number of neurons = 256, 128, 64, 32, 16 in layers 1 to 5 */
        y : 1..PatSize/( 2^(x-1) );                                                    }
            x : 1..NumOfPat;                                                           }

NEURON CHARACTER:
/*
*   Hidden neurons. Input and output degree are depending on the layer it is in (i.e. y). x is the position
*   of the neuron in the layer
*/
    {{{ Neu [y][x] TYPE BPHidden( z*2, z/2 , x )
        INPUT-DEGREE (z*3) OUTPUT-DEGREE (9*z/2);
        x : 1..z;                                                                      }
            z = PatSize / ( 2 ^ (y-1) );                                               }
                y : 2..NumOfLayer-1;                                                   }
/*
*   Input neuron. Input degree PatSize (pattern size) * 2, output PatSize / 2
*/
    {   Neu [1][x] TYPE BPInput( PatSize, PatSize/2 , x )
        INPUT-DEGREE (PatSize*2) OUTPUT-DEGREE (PatSize/2);
        x : 1..PatSize;                                                                }
/*
*   Output neuron. Input degree is num of out from prev. layer, output is 5 * z (num of neu in last layer)
*/
    {{  Neu [NumOfLayer][x] TYPE BPOut( 2*z, x)
        INPUT-DEGREE (2*z+1) OUTPUT-DEGREE (5*z);
        x : 1..z;                                                                      }
            z = PatSize / ( 2 ^ (NumOfLayer - 1) );                                    }

CONNECTION PATTERN :
/*
*   Forward propagation, connect I for every neu in the layer from every Out of every neu in the prev layer
*/
    {{  Neu[x+1][w].I[y] = Neu[x][y].out;
        y : 1..PatSize / ( 2 ^ (x -1) );  w : 1..PatSize / ( 2 ^ x )                   }
            x : 1..NumOfLayer-1                                                        }
/*
*   Backward propagation, connect WI & DeltaI for every neu in the layer from every W & Delta (error
*   correction signals generated by every neuron) in the next layer
*/
    {{  Neu[x][w].WI[y] = Neu[x+1][y].W[w];
        Neu[x][w].DeltaI[y] = Neu[x+1][y].Delta;
        w : 1..PatSize / ( 2 ^ (x-1) ); y : 1..PatSize / ( 2 ^ x) )                    }
            x : 1..NumOfLayer-1                                                        }
/*
*   Connect input I of 1st layer to input source (Input-Pattern) defined in Control Neuron (CN)
*/
    {   Neu[1][x].I = CN.Input-Pattern[x];
        x : 1..PatSize                                                                 }
/*
*   Connect input Teach of last layer to input source (Teach) defined in Control Neuron (CN)
*/
    {   Neu[NumOfLayer][x].Teach = CN.Teach[x];
        x : 1..PatSize / ( 2 ^ ( NumOfLayer-1) )                                       }
/*
*   Connect output Out of last layer to output target (Output-Pattern) defined in Control Neuron (CN)
*/
    {   CN.Output-Pattern[x] = Neu[NumOfLayer][x].Out;
        x : 1..PatSize / ( 2 ^ (NumOfLayer - 1 ) )                                     }
```

Form 8. Example Form for Configuration of BP-Net

4.4.1. FORM ENTRIES

A number of subscripts are used in specifying the configuration, hence may give an impression of being complicated. This is because the network contains different number of neurons in different layers. After the following explanation, one should find the subscripts easy to understand. In addition, in Appendix III, one can find other simple configurations with corresponding very simple specifications.

4.4.1.1. Neuron Label Entry

The **neuron label** entry defines the names of the neurons to be used subsequently in other entries. In the example, the neurons are named as elements in a two dimensional array called *Neu*. The first dimension is the layer number and the second is the number of neurons in that layer. This uniform name for all neurons is useful for subsequent configuration specification but it is not a necessity. In Appendix IV, an example specification form on Boltzmann machine shows how different names are used for different groups of neurons. The corresponding specification for the **neuron label** entry of the example BP-Net is extracted in Form 9.

```
NEURON LABEL :
    {{  Neu [x] [y];  ❶
        y : 1..PatSize/( 2^(x-1) );  ❷
                                        /* Number of neurons = 256, 128, 64, 32, 16 in layers 1 to 5 */
                                                                                    }
            x : 1..NumOfPat;                                                        }
```

Form 9. Example Neuron Label Entry

❶**Difference with Conventional 2-D Arrays.** One may found the specification and the conventional array declaration very similar. This is true except one important difference. The given network is not a complete matrix of neurons, as the number of neurons in every layer is different. This implies that a declaration of the form

Neu [1..M] [1..N]

where *M* and *N* are constants is not possible. The value of *N* should vary according to different values assumed in the preceding index.

As a result, some additional notations are required to indicate the layer number (or current value in the first index). This introduces the use of the *index variable x* that iterates from the lower bound *1* to the upper bound *NumOfLayer*. This value is used in the following dimension, the number of neurons in that layer, to compute the corresponding boundary for *y*. Now *y* varies from *1* to *256* when *x* equals to 1, *128* when *x* equals to 2 and so on.

❷**Format of Index Variables.** The index variables *x* and *y* are declared with a *postfix format*, i.e., the boundary of the variable is declared after its reference. The syntax of an index variable *V* is expressed as

{ <Effective Scope> *V* : <Lower Bound> .. <Upper Bound> }.

The scope of the variable is within the *<Effective Scope>*. Its *<Lower Bound>* and *<Upper Bound>* values may be expressions depending on any other index variable whose scope *V* is in. Thus the pair of braces is very important in identifying the scope of the variable.

4.4.1.2. Neuron Character Entry

The **neuron character** entry describes how the neuron labels defined are associated with the types of the neurons (*BPHidden, BPInput* and *BPOut*) and the *input* and *output* degrees. The Input/Output degrees are the number of physical connections with other neurons for each neuron, and should match with those resulting from the specification on the **connection pattern** entry. Every specification statement starts with a neuron label already defined in the **neuron label** entry. The specification for the BP-Net is shown in Form 10.

```
NEURON CHARACTER:
/*
*   Hidden neurons. Input and output degree are depending on the layer it is in (i.e. y). x is the position
*   of the neuron in the layer
*/                              ❷
    {{{  Neu [y][x] TYPE BPHidden( z*2, z/2 , x )
         ❸INPUT-DEGREE (z*3) OUTPUT-DEGREE (9*z/2);
         x : 1..z;                                                    }
             ❶z = PatSize / ( 2 ^ (y-1) );                            }
                 y : 2..NumOfLayer-1;                                 }
/*
*   Input neuron. Input degree PatSize (pattern size) * 2, output PatSize / 2
*/
    {    Neu [1][x] TYPE BPInput( PatSize, PatSize/2 , x )
         ❸INPUT-DEGREE (PatSize*2) OUTPUT-DEGREE (PatSize/2);
         x : 1..PatSize;                                              }
/*
*   Output neuron. Input degree is num of out from prev. layer, output is 5 * z (num of neu in last layer)
*/
    {{   Neu [NumOfLayer][x] TYPE BPOut( 2*z, x)
         ❸INPUT-DEGREE (2*z+1) OUTPUT-DEGREE (5*z);
         x : 1..z;                                                    }
             z = PatSize / ( 2 ^ (NumOfLayer - 1) );                  }
```

Form 10. Example Entry for Neuron Character

❶**Short-hand Variables.** There is a variable *z* defined in the third line in addition to the two variables *x* and *y* required. This variable is not an index, as it is defined to be equal to an expression *PatSize/(2^(y-1))* (the number of neurons in this layer), not a range of values. This variable is just a short hand representation for the expression. In other words, the first statement can be expressed without using the variable *z* at all. Every occurrence of *z* in the statement can be substituted with the expression, except that the statement will be more complicated and more difficult to be modified.

❷**Type Declarations.** The type declaration identifies the *neuron type* to be used for a particular neuron. The first statement declares that the type to be used is the *BPHidden* type, as it is dealing with hidden units. Three parameters are passed into the *BPHidden* units. These are the actual values of the *neuron definition parameters* (NDPs) defined in the **formal neuron** forms. *2*z* is the number of neurons in the

previous layer, *z/2* is the number of neurons in the next layer and *x*, which is defined as ranging from *1* to *z*, is the position of the neuron under consideration.

❹Input/Output Degrees. The corresponding *input* and *output* degrees of the neurons in the different layers are summarized in Table 7, where *z* is the number of neurons in that layer. In the *y*-th layer, the value of *z* is *PatSize / (2^(y-1))*, which is the number of neurons in that layer. The difference in input and output properties of the neuron types further supports the need for distinct neuron types in the network.

Layer	Neuron Type	Input	Degree	Output	Degree
First	BPInput	I WI DeltaI	PatSize PatSize / 2 PatSize / 2	Out	PatSize / 2
		Total	PatSize * 2	Total	PatSize / 2
Hidden Layers	BPHidden	I WI DeltaW	z * 2 z / 2 z / 2	Out W Delta	z / 2 z * 2 z * 2
		Total	z * 3	Total	z * 9 / 2
Last	BPOut	I Teach	z * 2 1	Out W Delta	z z * 2 z * 2
		Total	z * 2 + 1	Total	z * 5

Table 7. I/O Degree of the Neurons in the Example BP-Net

The *inputs* of the hidden neurons are the corresponding outputs from the previous and next layers. So the *j*-th hidden unit should have three groups of outputs, *Out*, *Delta* and *W*. The input and output degrees for the first layer are, anyway, different from those hidden layers. This layer is directly connected to the input source and the number of input signals *I* is equal to the number of neurons *PatSize* in that layer. There is also no correction signal output from this layer.

4.4.1.3. Connection Pattern Entry

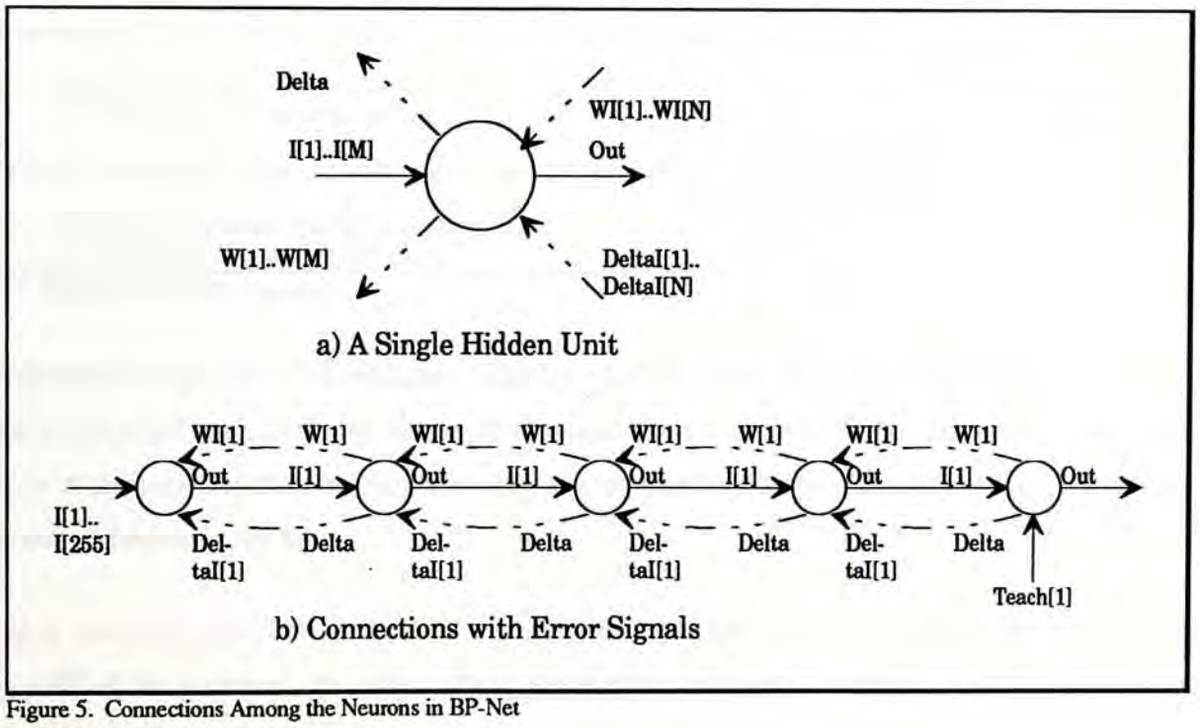


Figure 5. Connections Among the Neurons in BP-Net

The **connection pattern** for the sample BP-Net is different from the connection illustrated in Figure 4. Figure 4 adopts the conventional means of illustrating the network configuration. The actual connections are more complicated as there are also error correction signals propagated from the forward layers to the backward ones. Figure 5a illustrates the input and output parameters of a hidden unit. The resultant connection of the first neurons in the five layers is shown in Figure 5b. One can appreciate the complexity of the connections, and hence the full connection of the network will not be shown.

The communication links should indicate the *content* of the messages (i.e., the parameters) intended to be received and sent by the neurons. There may be more than one type of input and/or output from a single neuron. This is shown in the **connection pattern** entry, in which the connections among the neurons are augmented with the names of the parameters. The sample **connection pattern** entry is shown in Form 11.

```
CONNECTION PATTERN :
/*
*   Forward propagation, connect I for every neu in the layer from every Out of every neu in the prev layer
*/
    {{   ❶Neu[x+1][w].I[y] = Neu[x][y].out;
         y : 1..PatSize / ( 2 ^ (x -1) );  w : 1..PatSize / ( 2 ^ x )                    }
              x : 1..NumOfLayer-1                                                        }
/*
*   Backward propagation, connect WI & DeltaI for every neu in the layer from every W & Delta (error
*   correction signals generated by every neuron) in the next layer
*/
    {{   ❶Neu[x][w].WI[y] = Neu[x+1][y].W[w];
         ❶Neu[x][w].DeltaI[y] = Neu[x+1][y].Delta;
         w : 1..PatSize / ( 2 ^ (x-1) ); y : 1..PatSize / ( 2 ^ x) )                     }
              x : 1..NumOfLayer-1                                                        }
/*
*   Connect input I of 1st layer to input source (Input-Pattern) defined in Control Neuron (CN)
*/                          ❷
    {    Neu[1][x].I = CN.Input-Pattern[x];
         x : 1..PatSize                                                                 }
/*
*   Connect input Teach of last layer to input source (Teach) defined in Control Neuron (CN)
*/
    {    Neu[NumOfLayer][x].Teach = CN.Teach[x];
         x : 1..PatSize / ( 2 ^ ( NumOfLayer-1) )                                       }
/*
*   Connect output Out of last layer to output target (Output-Pattern) defined in Control Neuron (CN)
*/
    {    CN.Output-Pattern[x] = Neu[NumOfLayer][x].Out;
         x : 1..PatSize / ( 2 ^ (NumOfLayer - 1 ) )                                     }
```

Form 11. Example Connection Pattern Entry

❶**Explanation of the First 3 Equations.** The first equation states that the output *Out* from every neuron in layers except the last one should be sent to the input *I* of the neurons in the succeeding layers. The *x* is the layer of the neuron for receiving the output, *w* is the position of the input neuron and *y* is the position of the output neuron in the layer.

The next two equations are about the communication for the error correction signal *DeltaI* and the weights *WI* of the neurons. The first of them states that the internal weight *W* of neurons in succeeding

layer is the signal source of *WI* of the preceding layer. The second equation declares that the error correction signal *Delta* of the succeeding layer neurons is the signal source of *DeltaI* of the preceding layer neurons. The 2 equations are declared together because the index variables used are the same.

❷**Use of Special Name : CN.** The last three equations for connection have a special name *CN* that is not defined elsewhere. This is the initials of the name **control neuron**. The first equation among them refers to the input pattern to the first layer, which is known as *Input-Pattern* in the **control neuron**. The second one state that the teaching input for the output layer is equal to the value of the variable known as *Teach* in *CN*. And the last equation declares that the output from the last layer is accepted by *CN* as *Output-Pattern*.

4.4.2. CHARACTERISTICS OF THE SYNTAX

POSTFIX INDEX FORMAT

The postfix format used in index variable declarations allows a user to focus on the important specification before considering the indices. For example, in **connection pattern** entry, the connection among the parameters of the neurons is specified first before the indices are considered. This clear arrangement helps the designers in locating the connection information, instead of index computations.

COMPILATION CONSTANTS

Index variables are determined at *compilation time*, not in execution time. This is because all the information supplied by the user should be tested before execution. The *configuration* therefore cannot depend on some values that can be determined only in execution period.

DATAFLOW SPECIFICATION APPROACH

The **connection pattern** is also defined with *dataflow specification* methodology. This can support a uniform and simple notation for all levels of description, which will reduce the chance of making errors or getting troubles. Moreover, data dependencies analysis in specification analysis can easily incorporate the information from the *configuration*.

4.5. CONTROL NEURON SPECIFICATION

```
*** CONTROL NEURON ***

GLOBAL INPUT :                          /* Input to CN = output from network */
    Output-Pattern [ 1..PatSize/(2^(NumOfLayer-1)) ]
        RANGE (0, 1) INCLUSIVE CONTINUOUS;

GLOBAL OUTPUT :                         /* Output from CN = input to network */
    Input-Pattern [ 1..PatSize ] BINARY;
    Teach;                              /* Teaching signal to last layer */

GLOBAL PARAMETER :
    (Iteration, Curr-Pattern) INTEGRAL;

GLOBAL INITIAL VALUE :
    Iteration[0] = 1; Curr-Pattern[0] = 1;

INPUT FILE :
    ( Training, Recalling ) {           /* Files are applicable to both phases */
        FILE 'INPAT.DAT', RECORD Inpat [ 1..5 ] BINARY,          /* input pattern */
        Teach [ 1..OutSize ] RANGE (0, 1) INCLUSIVE CONTINUOUS,  /* teaching signal */
        Inpat-Control INTEGRAL; };                               /* control signal */

OUTPUT FILE :
    ( Training, Recalling ) {           /* Files are applicable to both phases */
        FILE 'OUTPAT.DAT', RECORD Outpat [ 1..5 ] BINARY; };     /* output pattern */

GLOBAL FUNCTION :
/*
*   Number of iteration for current pattern, from 1 to MaxIter in Training phase, always 1 in Recalling phase
*/
    Iteration[T+1] =
        IF Phase[T] = Recalling THEN 1
        ELSE IF Iteration[T] = MaxIter THEN 1
        ELSE Iteration[T] + 1;
/*
*   Current pattern to use, depending on whether the MaxIter iterations has passed or not
*/
    Curr-Pattern[T+1] =         /* the current pattern number */
        IF Iteration[T+1] = 1 THEN Curr-Pattern[T] + 1
        ELSE Curr-Pattern[T];
/*
*   Map input pattern from file (Inpat) to input pattern of the network (Input-Pattern)
*/
    Input-Pattern[T] = Inpat[ CURR-PATTERN[T] ];
/*
*   Map pattern from output of network (Output-Pattern) to output file pattern (Output), kept only last one
*   for the same input pattern (i.e. with the Iteration number of times)
*/
    Outpat[ Curr-Pattern[T] ] =
        IF Iteration[T] = 1 THEN Output-Pattern[T];
/*
*   Phase transition control, depending on the control signal from file (Inpat-Control)
*/
    Phase[T] =
        IF Inpat-Control[ Curr-Pattern[T] ] = 2 THEN Terminate
        ELSE IF Inpat-Control[ Curr-Pattern[T] ] = 1 THEN Recalling
        ELSE Training;
```

Form 12. Example Form for Control Neuron of the BP-Net

Although the BP-Net paradigm mentions nothing about the environment, it is reasonable to regard it being in a specific environment. The **control neuron** form is used to specify the environment and *operation controls* for the network. The sample **control neuron** form of the example BP-Net is given in Form 12.

4.5.1. FORM ENTRIES

This form is very similar to that of *formal neuron*. This is not surprising as the *control neuron* is considered as a specific neuron (or a group of neurons) responsible for controlling the global functioning of the network under consideration. The main difference is the presence of *environmental* (**input** and **output file** entries) declarations and the absence of *operation phases* in the **global function** entry.

4.5.1.1. Global Input, Output, Parameter & Initial Value Entries

These four entries use identical notations as their counterparts in **formal neuron** forms. The example entries are shown in Form 13.

```
GLOBAL INPUT :                          /* Input to CN = output from network */
❶Output-Pattern [ 1..PatSize/(2^(NumOfLayer-1)) ]
      RANGE (0, 1) INCLUSIVE CONTINUOUS;

GLOBAL OUTPUT :                         /* Output from CN = input to network */
❶Input-Pattern [ 1..PatSize ] BINARY;
❷Teach;                                 /* Teaching signal to last layer */

GLOBAL PARAMETER :
❸(Iteration, Curr-Pattern) INTEGRAL;

GLOBAL INITIAL VALUE :
    Iteration[0] = 1; Curr-Pattern[0] = 1;
```

Form 13. Example Entry for Global Input, Output, Parameter & Initial Value

❶**Input/Output Parameters.** From the specification, there is one input parameter known as *Output-Pattern*. The name of the parameter may be confusing as this parameter is the *input* to the *control neuron* but the *output* from the network. The *control neuron* receives the output patterns from the last layer and transforms them into the streams of data to the environment. It should therefore matches with the property of the parameter *Out* from the *output* layer.

There are two output signals from the *control neuron*. The first one is *Input-Pattern*, which is the input pattern feeding into the *input* layer. It shares the same property as the *I* of the *input* layer.

❷**Parameter without Properties.** Another output parameter is *Teach*. *Teach* is a parameter *without* associated properties. This is because its property is defined in the **input file** entry. *Teach* is just a *dummy* name indicating the part of the record that is responsible for carrying the teaching input. It is necessary to include it in the output list to explicitly state which part of the file record is used as output

directly. The use of dummy names to represent the input or output records is the only allowable case of names without properties in parameter declaration entries.

❸**Global Counters.** There are two global parameters, *Iteration* and *Curr-Pattern*. Both of them serve as counters. *Curr-Pattern* indicates the current pattern from among the input records, and *Iteration* indicates how many times this pattern has been fed into the system. They are both initialized to 1 before the system starts. Their operations will be explained in the section on **Global Function** entries.

4.5.1.2. Input & Output File Entries

The **input** and **output file** entries are responsible for interfacing the network with the environment. Different files may be used for different operation phases, so the files used should be under the scope of the corresponding operation phases. The sample **input file** and **output file** entries are shown in Form 14.

```
INPUT FILE :
    ( Training, Recalling ) {                    /* Files are applicable to both phases */
        ❶FILE 'INPAT.DAT, RECORD Inpat [ 1..5 ] BINARY,                    /* input pattern */
        Teach [ 1..OutSize ] RANGE (0, 1) INCLUSIVE CONTINUOUS,            /* teaching signal */
        Inpat-Control INTEGRAL;  };                                        /* control signal */

OUTPUT FILE :
    ( Training, Recalling ) {                    /* Files are applicable to both phases */
        FILE 'OUTPAT.DAT, RECORD Outpat [ 1..5 ] BINARY;  };               /* output pattern */
```

Form 14. Example Input and Output File Entries

❶**File & Record Structures.** Every declaration in the **input** and **output file** entries is separated into two parts. The first part starts with the key word **FILE** and declares the name of the file to be used. The second part starts with the key word **RECORD** and specifies the components of every record. Every component is addressed by a name and a property description with the same notations as those of other parameter declarations.

In the example, the **input file** entry declares the same file to be used throughout the two phases. The name of the file is *INPAT.DAT* and the records contain three parts, one is the input pattern *Inpat*, the other is the teacher's input *Teach*, and the final one is *Inpat-Control* for controlling the operation phase of the system. The output file name is *OUTPAT.DAT* and the records *Outpat*. *Outpat* has the same format as *Output-Pattern*.

4.5.1.3. Global Function Entry

The **global function** entry is the main control of the behavior of the *control neuron* and hence the whole system. It is declared in a similar notation as the **internal function** entry in the **formal neuron** form, except in dealing with file records. The example **global function** entry for the sample BP-Net is extracted and shown in Form 15.

```
GLOBAL FUNCTION :
/*
*   Number of iteration for current pattern, from 1 to MaxIter in Training phase, always 1 in Recalling phase
*/
    ❹Iteration[T+1] =
        IF ❶Phase[T] = Recalling THEN 1
        ELSE IF Iteration[T] = MaxIter THEN 1
        ELSE Iteration[T] + 1;
/*
*   Current pattern to use, depending on whether the MaxIter iterations has passed or not
*/
    ❹Curr-Pattern[T+1] =          /* the current pattern number */
        IF Iteration[T+1] = 1 THEN Curr-Pattern[T] + 1
        ELSE Curr-Pattern[T];
/*
*   Map input pattern from file (Inpat) to input pattern of the network (Input-Pattern)
*/
    Input-Pattern[T] = ❷Inpat[ CURR-PATTERN[T] ];❺
/*
*   Map pattern from output of network (Output-Pattern) to output file pattern (Output), kept only last one
*   for the same input pattern (i.e. with the Iteration number of times)
*/
    Outpat❷[ Curr-Pattern[T] ] =
        IF Iteration[T] = 1 THEN Output-Pattern[T];❺
/*
*   Phase transition control, depending on the control signal from file (Inpat-Control)
*/
    Phase[T]❸ =
        IF Inpat-Control[ Curr-Pattern[T] ] = 2 THEN Terminate
        ELSE IF Inpat-Control[ Curr-Pattern[T] ] = 1 THEN Recalling
        ELSE Training;
```

Form 15. Example Global Function Entry for the BP-Net

❶The Pre-Defined Parameter : Phase. The first equation specifies the behavior of a parameter *Phase* that is not defined by the users. This is a system-defined parameter responsible for indicating the current *operation phase*. Assigning different values into *Phase* will change the operation phases.

There are three possible operation phase values in the example. The first one is *Terminate* that is the special phase value signalling the system to stop operations. The other two phase values are *Training* and *Recalling*.

❷Characteristic of File Records. The phase transition depends on *Inpat-Control* which is part of the input record. Note that *Inpat-Control* is declared as a single integral variable but it is used as an array in the equation. This is because all files are assumed to contain a number of records, so every component of the record part is an array of that name.

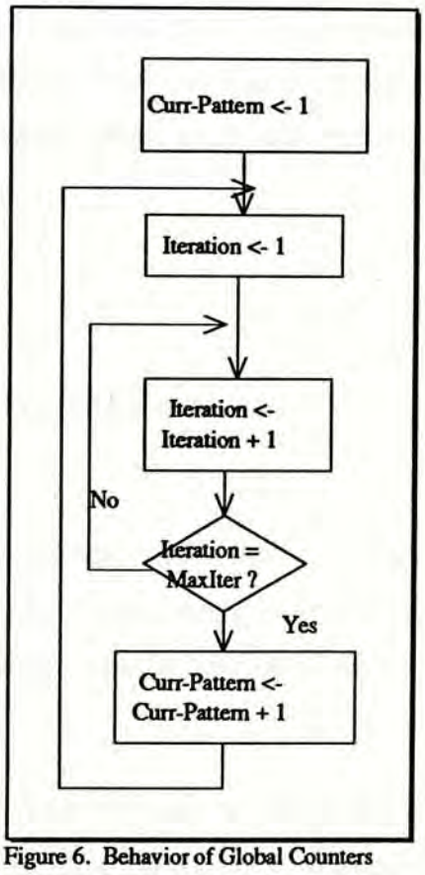


Figure 6. Behavior of Global Counters

In addition, the index for the array is not the temporal index *T* but another parameter *Curr-Pattern*. This means that *Inpat-Control* is not a *time dimension* array and it can be addressed by any other integral parameters, not just the temporal index *T*. The parameter *Curr-Pattern* indicates the current pattern under consideration. The behavior of *Curr-Pattern* makes it possible for the same record to be referenced for more than one time throughout the processes. Hence it is not necessary for users to prepare one record for every iteration.

❸**Phase Transition.** The meaning of the equation for *Phase* is now clarified. It depends on the value of *Inpat-Control* and *Curr-Pattern*. As *Curr-Pattern* changes, it is possible to address any specific record inside the file. The *Inpat-Control* component of the record under consideration determines the operation phase the system is in. This demonstrates how users can control the operation phase through values in input file. In the Boltzmann example in Appendix IV, the transition is solely determined by the global counters.

❹**Behavior of the System Counters.** The equations for the other two parameters (*Curr-Pattern*, *Iteration*) are straight forward. They are used as global counters, and their behavior can be summarized by Figure 6. In imperative language, the *Curr-Pattern* will be a counter of an outer loop, and *Iteration* will be that of an inner loop. The inner loop counter determines its own updating speed and periodically reset itself to some initial value. The outer counter depends on the behavior of the inner counter within its scope to determine the updating. The behavior of the counters in *Training* phase, when expressed in C, will be like:

```
for (Curr_Pattern=1; ; Curr_Pattern++)
    for (Iteration=1; Iteration <= MaxIter; Iteration++)
    {
        .....          /* updating and assignments of the parameters */
    }
```

The expression for *Iteration* says that in the *Training* phase, *Iteration* will start from 1 until the constant value *MaxIter* and then value *1* in the *recalling* phase. This controls that every training pattern will be fed into the network for *MaxIter* times before switching to the next pattern. On the other hand, *Iteration* will be kept 1 forever in the *Recalling* phase.

Curr-Pattern depends on *Iteration*. Every time when *Iteration* is reset to *1*, the value of *Curr-Pattern* is incremented by *1*. In both phases, *Iteration* is reset to *1* when the required number of iterations has passed, and a new pattern should be used for training or recalling. Hence the value of Curr-Pattern will be increased by 1 to address the next record.

❺**Communication with the Environment.** The equation on *Input-Pattern* says that the input to the first layer of the network is the *Inpat* part of the current record. The same input to the first layer is maintained for *MaxIter* iterations in training phase, and *1* iteration in the recalling phase, as *Curr-Pattern* changes only when the required number of iterations has passed.

The *Outpat* field of the current record is equal to the *Output-Pattern* from the last layer for many iterations. This violates the single-valued variable constraint as the value of *Output-Pattern* changes in every iteration but its value is passed to the *Outpat* field of the *SAME* record for a number of iterations.

Outpat is not restricted to be single-valued because it is not reasonable for the output file to keep all the intermediate outputs from the network. Only the result for the final iteration may be of interest to us. It is therefore better to "overwrite" the old values of *Outpat* by the new and subsequently the final value.

In addition, the records of the *output files* are *sinks* in communication (i.e., no other parameter references them). They will not involve in *cyclic dependency*. It is therefore possible to relax the single-valued variable constraint for the output file records.

As a result, the only cases allowed for using parameters as indices to arrays are in the file records. But the values of these indices are determined at execution time and very difficult to be analyzed in a static manner. The designers are hence responsible for making sure that the correct number of records are generated.

5. DATA DEPENDENCY ANALYSIS

Specification analysis involves a) *data dependency analysis* for locating *cyclic dependency* and b) *attribute analysis* for locating specification errors other than the cyclic dependency problem. Parameters are described to be in cyclic dependency if they depend on one another in a circular manner and therefore cannot be properly evaluated. This section is to discuss the mechanism for locating this problem. Figure 7 illustrates an overview of the data dependency analysis. It consists of three steps: (i) *dependency graph construction*, (ii) *dependency cycle detection* and (iii) *dependency cycle analysis*.

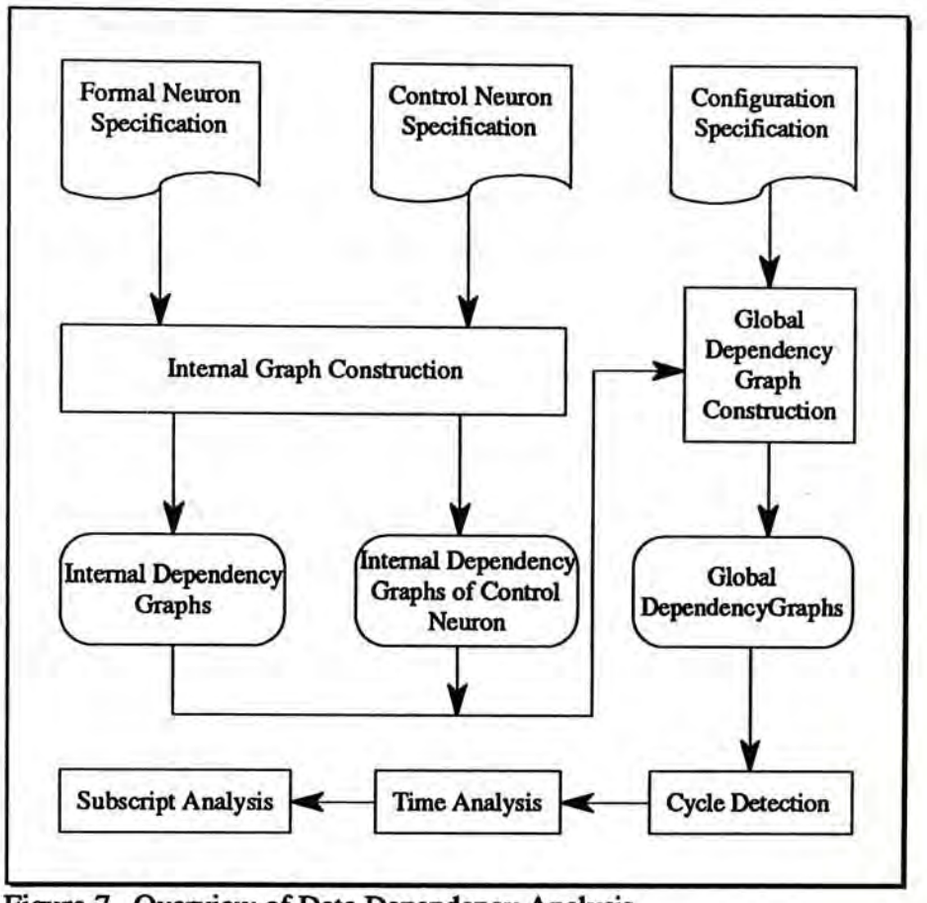


Figure 7. Overview of Data Dependency Analysis

Three kinds of data dependency graphs can be constructed from a given specification. *Internal dependency graphs* (IDGs) of neuron types show the dependency among the parameters *within* one neuron type. *Internal dependency graph* of the control neuron (CnIDG) is similar to the IDGs of the formal neurons as a control neuron is merely a special neuron. *Global dependency graphs* (GDGs) indicate the overall dependency relationship among the input/output parameters of *different* neurons in the network.

An exhaustive graph traverse through the *global dependency graphs* will be performed to extract all *cycles* in the graphs. A cycle in a dependency graph implies that the evaluation of parameters is dependent on each other, which is a necessary condition for cyclic dependency. Whether or not there is a cyclic dependency depends on the relative *instance* of the parameter being involved, and this is checked by the *cycle analysis*.

The presence of a massive number of cycles requires automatic dependency analysis. Dependency cycle analysis determines whether there is a proper order to evaluate the parameters by considering the *temporal relationship* among the parameters in the cycles. The temporal relationship can be obtained by

considering specific instances of the parameters. If these parameters are referenced at the same instance, they are mutually dependent on one another and cannot be properly evaluated. Otherwise, the equations associated with the evaluations of the parameters can be *scheduled* properly for execution.

The above three steps will be discussed in details in the coming sections. The BP-Net is used as the primary example for illustration. A summary of the results for the Boltzmann machine and Perceptron is also given where appropriate.

5.1. Graph Construction

5.1.1. Simplification and Normalization

Before constructing the dependency graphs, the equations in the **internal** and **global function** entries are simplified to high light the dependency relationship. The temporal indices are also normalized to standardize the *temporal offsets*.

5.1.1.1. Removing Non-Essential Information

For each equation in the specification forms, the operators, functions and constants are deleted, as they do not affect the dependency relations. The equal sign "=" is replaced with an arrow "<-". to highlight the dependency among the parameters. For example, the internal function

A[T] = I[T] * TRANSPOSE(W[T]) + Theta

is replaced by

A[T] <- I[T], W[T].

In case of conditional dependency checkings such as

Curr-Pattern[T+1] = IF Phase[T]=Recalling THEN Curr-Pattern[T]+1,

the resultant expression will include not just *Curr-Pattern[T]* at the right hand side but also *Phase[T]* as the evaluation of *Curr-Pattern[T+1]* depends on it. The resultant expression is therefore

Curr-Pattern[T+1] <- Phase[T], Curr-Pattern[T].

The parameter on the left hand side of the arrow is called the *depending parameter* and the parameters on the right hand side are called the *determining parameters*.

5.1.1.2. Removing File Record Parameters

File record parameters will not be involved in dependency cycles and can therefore be ignored in the dependency graphs. File record parameters are those parameters used to address the record fields in input and output files. The input file record parameters cannot be updated and the output one cannot be referenced. Thus intuitively, they will not be involved in a cycle waiting state and removed from the expression to further simplify the expressions.

When a file record parameter is a depending parameter (i.e. an output file), the whole expression can be ignored because no dependency edge (see section 5.1.2.) can be constructed when the depending parameter is removed.

When a file record parameter is a determining parameter (i.e. an input file), the file record parameters may be ignored but the *record reference parameters* cannot. A record reference parameter is the parameter used to select a record within the file. Consider the example

Input-Pattern[T] = Inpat[Curr-Pattern[T]],

which is an equation in the control neuron of the BP-Net. *Input-Pattern* is an internal parameter, *Inpat* is a file record parameter and *Curr-Pattern* is the corresponding record reference parameter, which is also an internal parameter. The evaluation of *Input-Pattern* is depending on the evaluation of *Curr-Pattern*, so the dependency expression should be

Input-Pattern[T] <- Curr-Pattern[T]

by removing the file record parameter.

5.1.1.3. Rearranging Temporal offset

The objective of normalization is to remove the *temporal offset* of the *depending parameters*. The simplified expression for any equation is

Y[T+Offset] <- X1[T1], X2[T2], X3[T3], ..., Xn[Tn].

To normalize the expression means that the analyzer will subtract *Offset* of the depending parameters from all Ti's, the temporal indices of the determining parameters.

Form 16 shows the simplified and normalized expression of the **internal function** entry for our example BPHidden neuron.

```
1.  A[T] <-- I[T], W[T];
2.  Out[T] <-- A[T];
    Training {
3.      Delta[T] <-- Out[T], DeltaI[T], W[T];
4.      DeltaW[T] <-- Delta[T], Out[T], DeltaW[T-1];
5.      W[T] <-- W[T-1], DeltaW[T-1];    };
    Recalling {
6.      Delta[T] <-- 0;
7.      DeltaW[T] <-- 0;
8.      W[T] <-- W[T-1];                          };
```

Form 16. Normalized Expression for Hidden Unit of a BP-Net.

5.1.1.4. Conservation of Temporal Relationship

The normalization process will not change the meaning of the dependency equations as the temporal indices of the determining and depending parameters are just relative references. For example, the index *T+1* in equation 6 of the original form for BPHidden means that the depending parameter *W* is evaluated 1 time interval after the evaluation of the determining parameters. Hence equation 6 of Form 16 gives the same temporal relationship as before.

5.1.1.5. Zero/Negative Offset for Determining Parameters

It is obvious that the resultant temporal offsets for the determining parameters should be zero or negative in value, otherwise the depending parameters are referring to a *future* instance of the determining parameter, which is impossible. Any violation of this restriction can be detected easily during the normalization process.

5.1.2. INTERNAL DEPENDENCY GRAPHS (IDGS)

The construction of internal dependency graphs (IDGs) is based on the information from **formal neuron** specification forms. The IDGs of the neurons are different for every neuron type and operation phase. The number of IDGs is equal to the number of neuron types times the number of operation phases.

Figures 8 (a) to (e) show the IDGs of the sample BP-Net. The parameters are represented by nodes, and the dependency relations by edges. The construction process is to insert an edge pointing from every determining parameter in every equation to each corresponding dependent parameter.

All the edges are labelled with the temporal offset of the determining parameter. The temporal offset of the depending parameter need not be recorded as it is always 0.

If the parameter has indices other than the temporal one, these indices are also attached as labels. The complete algorithm for the construction of the IDGs can be found in Appendix II.

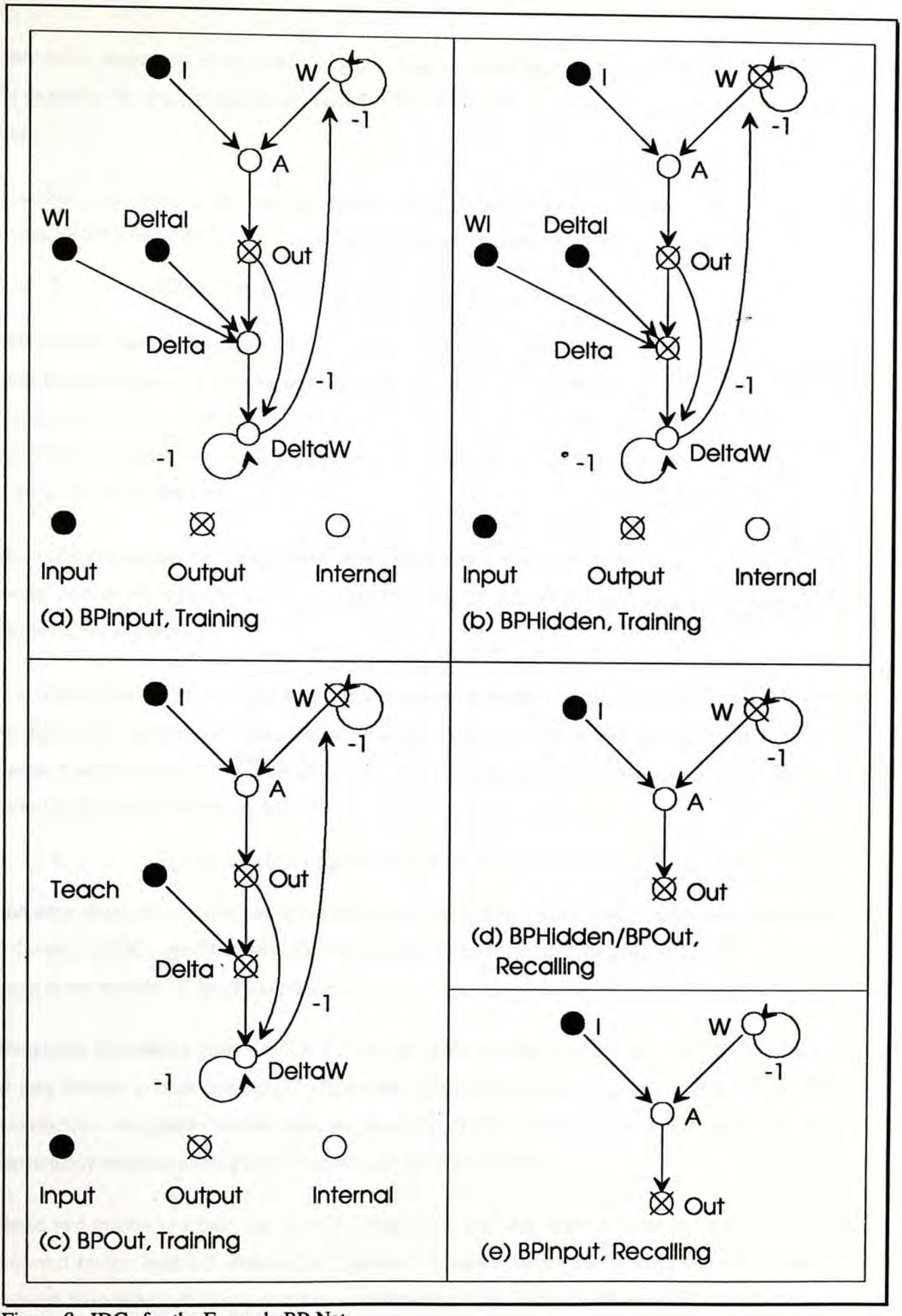


Figure 8. IDGs for the Example BP-Net

Only those parameters being referenced by one or more equations are included. In Figure 8 (d) and (e), for example, the input parameters *Deltal* and *WI* are not shown as they are not referenced in the recalling phase.

In addition, parameters that depend on constant values only are not included as they will not involve in cycles. For example, *Delta* and *DeltaW* are not shown in Figure 8 (d) and (e).

5.1.3. IDG OF CONTROL NEURON (CNIDG)

The internal dependency graph of control neuron (CnIDG) is constructed from the information in **control neuron** specification form. The control neuron does not have *operation phases* and hence it is not necessary to use different graphs for different phases. The number of CnIDG is always 1 for every network.

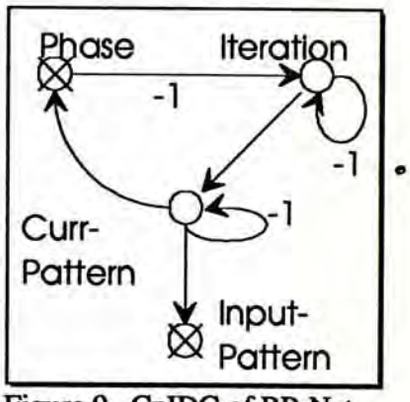


Figure 9. CnIDG of BP-Net.

The same mechanism for building an internal dependency graph (IDG) of formal neuron is applied here. The resultant CnIDG for the sample BP-Net is shown in Figure 9.

The dependency graph in Figure 9 is pretty simple as a number of parameters are removed. First of all, the file record parameters *Inpat*, *Inpat-Control*, *Teach* and *Outpat* are deleted. The input parameter *Output-Pattern* that determines *Outpat* only is also ignored, as it is now related to none of the parameters. Hence just four parameters are left.

5.1.4. GLOBAL DEPENDENCY GRAPHS (GDGS)

The dependency among neurons is extracted from the **configuration** specification form, together with the IDGs and CnIDG. As different IDGs are used for different phases, the number of GDGs for a network is equal to the number of operation phases.

The global dependency graphs for the BP-Net are given in Figures 10 and 11. The GDGs are complicated as they involve a large number of neurons and many parameters. The solid rectangles are the neuron boundaries -- the graphs within them are the IDGs of the specific neurons. The broken arrows are the dependency relations among the parameters of different neurons.

These two graphs just take one neuron from each layer, and other neurons are not required. This is achieved by the approach discussed in section 5.4, which makes use of the symmetric property of the network to include just representative neurons in the graph. Without this approach, the graph constructed will be extremely complicated and cannot be analyzed.

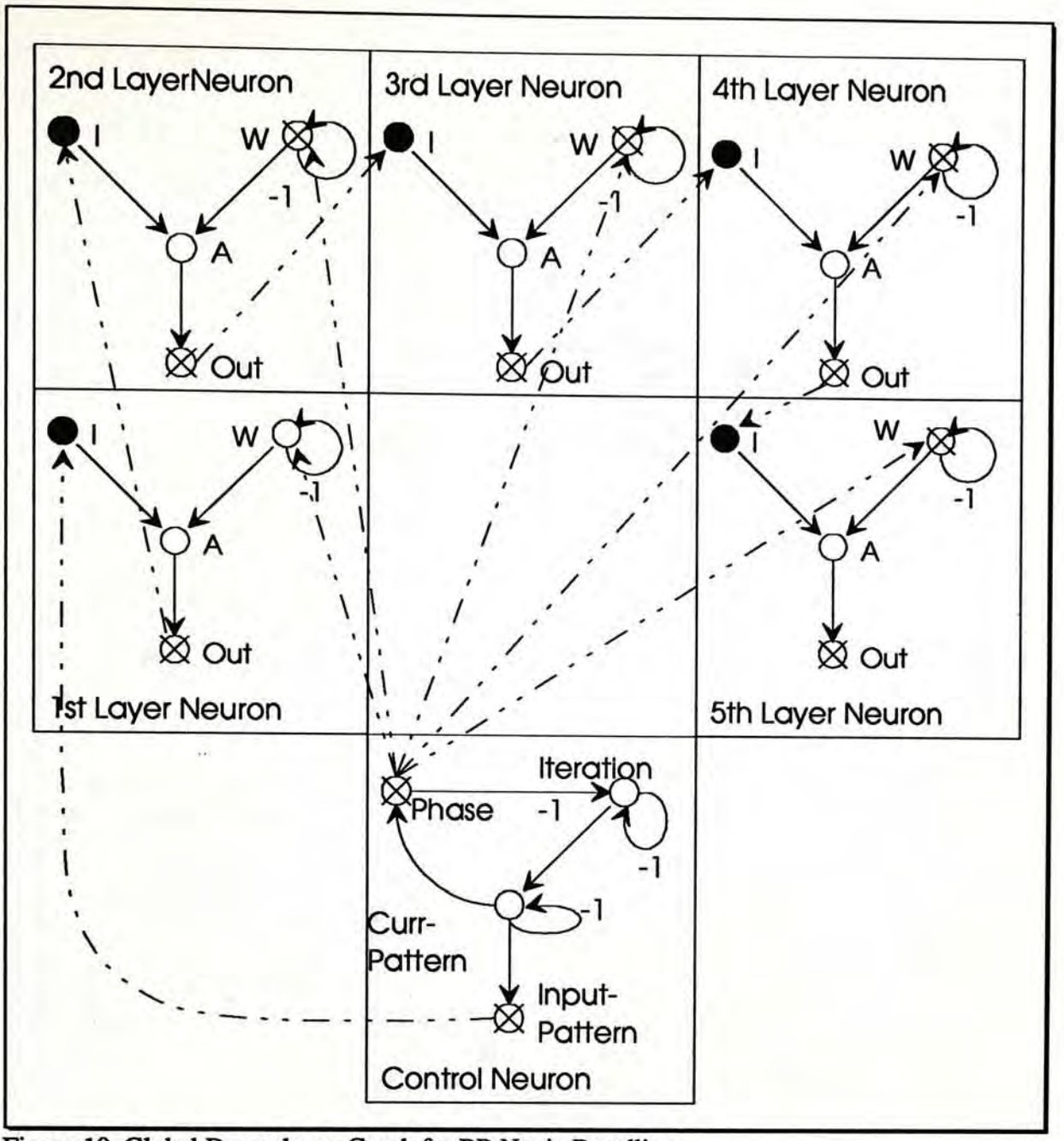


Figure 10. Global Dependency Graph for BP-Net in Recalling

The global parameter *Phase* is heavily connected to other parameters. In the specification forms, the internal parameters of the formal neurons are not dependent on *Phase* of the **control neuron**. In the graphs, however, they are. This dependency is introduced by the system automatically for all parameters which function differently in various phases.

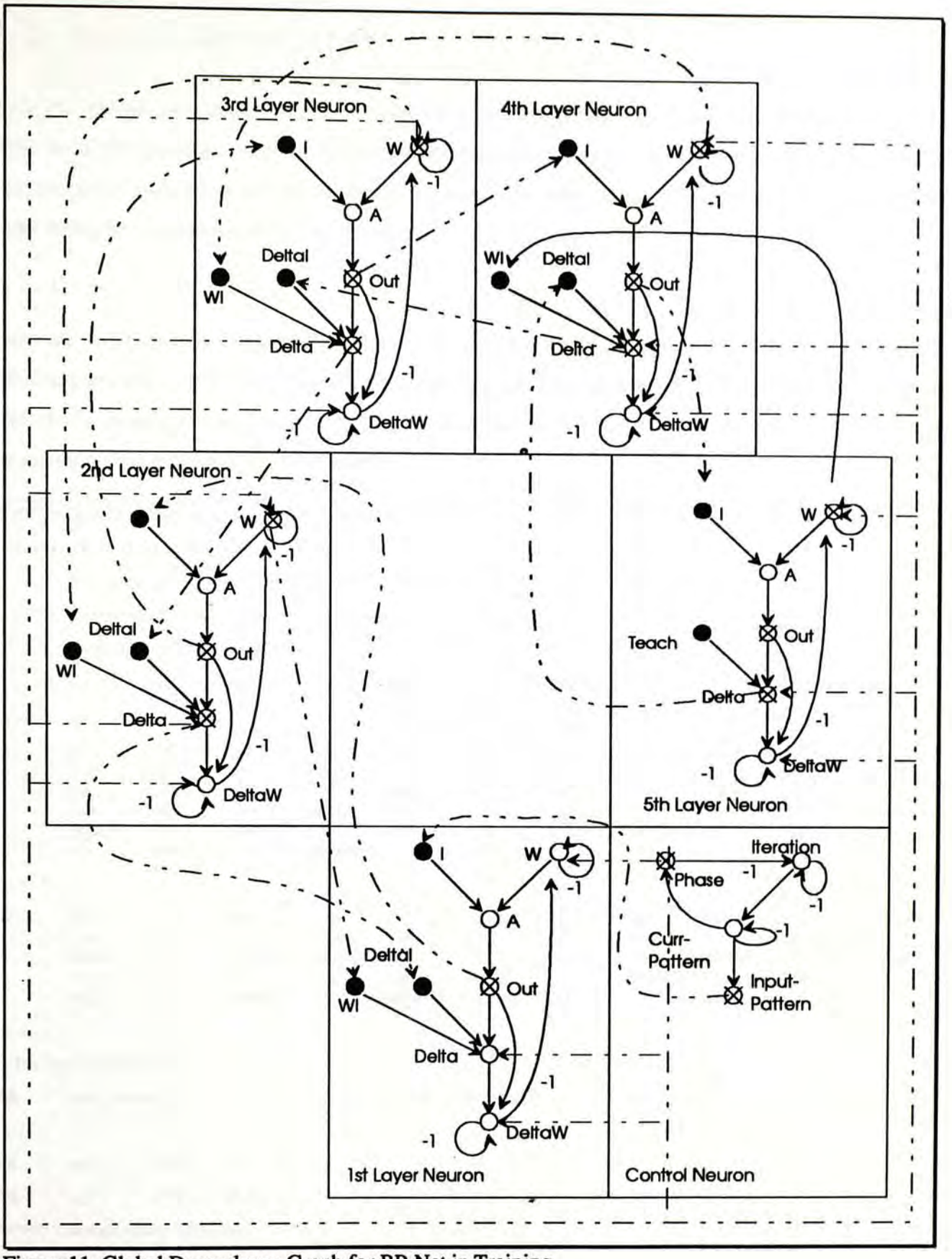


Figure 11. Global Dependency Graph for BP-Net in Training

5.2. CYCLE DETECTION

Once the global dependency graphs (GDGs) are constructed, cycle detection may be performed. In addition to the resultant cycles of the sample BP-Net, the cycles of two other networks, a Boltzmann machine and a Perceptron, are shown for comparison. The number and length of cycles for the BP-Net under different conditions will also be discussed.

5.2.1. BP-NET

There are 101 cycles for *Training* phase, and 8 cycles for *Recalling* phase in the BP-Net example. The difference in number of cycles for the different phases can be predicted by the complexity of the two global dependency graphs in the previous section. Part of the result is shown in Figure 12, in which all parameter names are shown in lower cases.

```
******** Cycles for Back Propagation Network in file bp.dat ********
---- For Operation Phase 1
1.      curr-pattern,0      -->     phase,0     -(-1)->  iteration,0 -->     curr-pattern,0
2.      curr-pattern,0      -(-1)-> curr-pattern,0
3.      iteration,0 -(-1)-> iteration,0
4.      a,1         -->     out,1       -->     delta,1     -->     deltaw,1    -(-1)->  w,1         -->     a,1
.....
40.     a,1         -->     out,1       -->     i,2         -->     a,2         -->      out,2       -->     i,3
        -->         a,3     -->         out,3   -->         i,4     -->         a,4      -->         out,4   -->
        i,5         -->     a,5         -->     out,5       -->     delta,5     -->      deltaw,5    -(-1)-> w,5
        -->         wi,4    -->         delta,4 -->         deltaw,4 -(-1)->    w,4      -->         wi,3    -->
        delta,3     -->     deltai,2    -->     delta,2     -->     deltaw,2    -(-1)->  w,2         -->     wi,1
        -->         delta,1 -->         deltaw,1 -(-1)->    w,1     -->         a,1
.....
71.     a,2         -->     out,2       -->     i,3         -->     a,3         -->      out,3       -->     i,4
        -->         a,4     -->         out,4   -->         i,5     -->         a,5      -->         out,5   -->
        delta,5     -->     deltaw,5    -(-1)-> w,5         -->     wi,4        -->      delta,4     -->     deltaw,4
        -(-1)->     w,4     -->         wi,3    -->         delta,3 -->         deltaw,3 -(-1)->     w,3     -->
        wi,2        -->     delta,2     -->     deltaw,2    -(-1)-> w,2         -->      a,2
.....
---- For Operation Phase 2
102.    curr-pattern,0      -->     phase,0     -(-1)->  iteration,0 -->     curr-pattern,0
.....
108.    w,4         -(-1)->  w,4
109.    w,5         -(-1)->  w,5
******> There are totally 109 cycles
******> The maximum length of the cycle is 33
```

Figure 12. Cycles in Global Dependency Graphs for Training Phase

Each cycle is illustrated in parameter names with numbers, and some arrows with or without labels. The number associated with the parameter indicates the layer the number is in. For example, *a,1* at the

beginning of cycle 4 indicates that it is parameter *A* (i.e., activity) in the neuron of the first layer (the cases are immaterial). A value 0 associated with the parameter indicates that the parameter is in the *control neuron.*

When the arrows are not labeled, there is no temporal offset on the determining side. If the arrows are labeled, like that between *deltaw* and *w* of cycle 4, a temporal offset *-1* is in the dependency relation.

Hence expression *4* in the figure shows that internal parameter *a* is determining output parameter *out* of the same instance, which in turn determines internal parameter *delta* of the same instance, and then *deltaw* also of the same instance. *Deltaw* determines the value of *w* that is one instance after, and which will determine parameter *a* of the same instance. These parameters are all of the neuron in the first layer.

Some of the cycles are very short, i.e. involving very few or just one parameter(s), like the second and the last cycles. On the other hand, some as if the 40-th cycle is long. The longest one involves 33 parameters of neurons in different layers.

5.2.2. OTHER EXAMPLES

The dependency cycles detected from the specifications of the Perceptron and Boltzmann machines are discussed in this section. They are given here to illustrate the great variations in number and length of cycles obtained from different networks. Moreover, these examples can be used to show that the massive number of cycles is not a particular feature of BP-Net but rather a general phenomenon found in different networks.

The full cycle list for Perceptron is shown as the list is short. On the other hand, only part of the cycle list for the Boltzmann machine is shown. The complete specification forms for these two networks can be found in Appendix IV.

5.2.2.1. The Perceptron

```
******** Cycles for Perceptron in file percep.dat ********
---- For Operation Phase 1
1.      curr-pattern,0      -(-1)->   phase,0    -(-1,0)->  curr-pattern,0
2.      curr-pattern,0      -(-1)->   curr-pattern,0
3.      iteration,0 -(-1)->  iteration,0
4.      a,1         -->      out,1     -->       e,1        -(-1)->  w,1      -->      a,1
5.      w,1         -(-1)->  w,1
---- For Operation Phase 2
6.      curr-pattern,0      -(-1)->   phase,0    -(-1,0)->  curr-pattern,0
7.      curr-pattern,0      -(-1)->   curr-pattern,0
8.      iteration,0 -(-1)->  iteration,0
9.      w,1         -(-1)->  w,1
******> There are totally 9 cycles
******> The maximum length of the cycle is 4
```

Figure 13. Cycle List for the Example Perceptron

The example network is a single layer perceptron. The input pattern is 1024 Bi-state values known as *predicates*, and there are also 1024 neurons for learning these values. Every neuron is connected to 1/4 of the inputs, with every neuron accepting input from a continuous pattern one position lower than the previous one in a round-ribbon manner.

CYCLES OBTAINED

The cycle list for the Perceptron is shown in Figure 13. The number of cycles in this perceptron is significantly less than that of BP-Net. There are only 9 cycles, and all of them are internal ones.

The small number of cycles can be predicted from the GDG (in Appendix IV) in which there are only arrows pointing from the **control neuron** but not vice versa. Hence there will not be any cycles involving the **control neuron**. As there are only two types of neurons, **control neuron** and *perceptron* neurons, there will not be any global cycles. With the same reason, the number of parameters involving in the cycles is also very small.

5.2.2.2. The Boltzmann Machine

```
******** Cycles for Boltzmann Machine in file boltz.dat ********
---- For Operation Phase 1
1.      annealing,0          -->     phase,0    -(-1)->  iteration,0 -(-1,0)-> annealing,0
2.      annealing,0          -->     phase,0    -(-1)->  probe,0     -->       iteration,0 -(-1,0)-> annealing,0
.....
---- For Operation Phase 2
24.     annealing,0          -->     phase,0    -(-1)->  iteration,0 -(-1,0)-> annealing,0
.....
38.     deltae,1  -(-1)->  out,1   -->   i,3   -->   deltae,3          -(-1)->  out,3   -->
        i,1       -->      deltae,1
.....
85.     w,3       -(-1)->  w,3
.....
---- For Operation Phase 6
.....
******> There are totally 172 cycles
******> The maximum length of the cycle is 6
```

Figure 14. Cycles List for the Example Boltzmann Machine

This machine is the shifter example on p.299 of *Parallel Distributed Processing*, vol.1 [33]. It follows the example given in the text, in which the neurons are randomly probed for updating. This is a safe approach for ensuring that every neuron can see the most recent states of all the other units. The parallelism is, however, sacrificed. If the system can tolerate time delays, it is just necessary to remove the control signals from the control neuron that is responsible for updating.

The cycles obtained are shown in Figure 14. The number of cycles for Boltzmann is even larger than that of the BP-Net but the length of the cycles for the former is much shorter.

The larger number of cycles in Boltzmann machine, despite the fact that is has only 3 layers, can be accounted by the reason that there are more cycles among the parameters inside the neurons than in the BP-Net (Appendix IV). Hence the vast number of cycles for Boltzmann is due to the number of *internal* cycles, while that for BP-Net is due to the larger number of layers that significantly increases the number of *global* cycles.

On the other hand, the cycles in Boltzmann machine are significantly shorter because the number of layers for the Boltzmann machine is smaller. In addition, the layers for *Visible* and *Shifts* neurons (see Appendix III) are not connected together. Hence the longest cycle can extend only across two layers, excluding the **control neuron**, either from *Visible* to *Hidden* or from *Shifts* to *Hidden* and is therefore limited in length.

5.2.3. NUMBER OF CYCLES

5.2.3.1. Different Number of Layers

The number of *neurons* in a network does *not* affect the number of dependency cycles. The Perceptron network has the largest number of neurons but gets the smallest number of dependency cycles. This is because the networks have no connections within the layers. Thus the length of the cycles will not extend inside the layer, and therefore the number of neurons in the same layer will have no effect on this length.

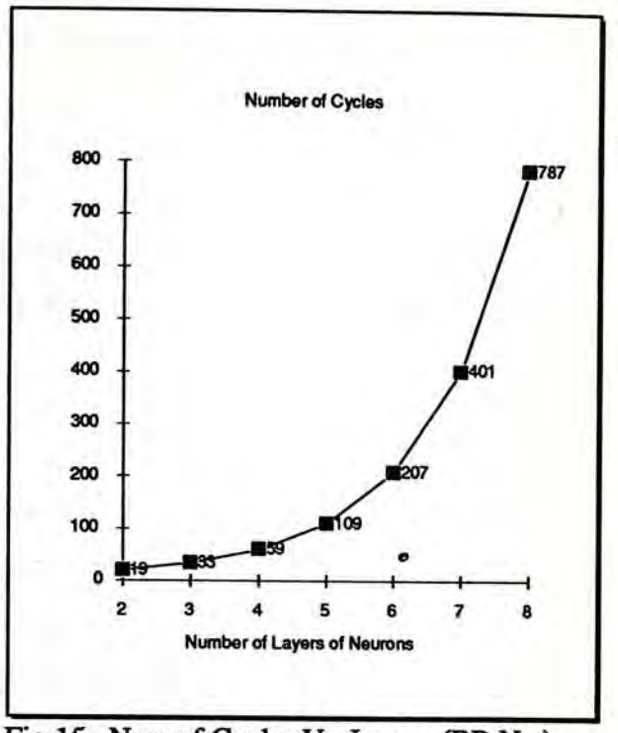


Fig 15. Num of Cycles Vs. Layers (BP-Net)

The **number of layers** is, however, significant in determining the number of dependency cycles. Among the three examples given, only the BP-Net can be freely configured to include more layers. The chart in Figure 15 shows the number of cycles for the BP-Net with different number of layers. The first layer is formed by the *BPInput* neuron type and the last layer is formed by the *BPOut* type. The other layers are all of *BPHidden* type.

It can be observed that the number of cycles for the BP-Net roughly doubles for each additional layer. When the difference between succeeding number of cycles for different number of layers is compared, the relation is concluded as

$$N_k = N_{k-1} * 2 - (2k - 1)$$

where k is the number of layers, and N_k is the number of cycles in a network with k layers.

5.2.3.2. Different Network Types

When BP-Net is reduced to 3 layers, the number of cycles between BP-Net and Boltzmann machine can be compared. Table 8 compares the number of cycles for Boltzmann and BP-Net. The comparison clearly shows that Boltzmann machine has significantly more cycles than BP-Net.

	No. of cycles
BP-Net	33
Boltzmann	172

Table 8. Comparing Boltzmann and BP-Net

The difference is due to the vast number of internal cycles found in the **formal** and **control** neurons of Boltzmann machine. The greater number of parameters in Boltzmann machines also accounts for the greater number of cycles.

5.2.4. CYCLE LENGTH

The length of a cycle is again not affected by the number of neurons. The Perceptron has the shortest cycle length among the networks yet it has the largest number of neurons.

5.2.4.1. Different Number of Layers

Figure 16 shows the length of the cycles against the number of layers in the BP-Net. This result is a straight line because of the symmetric properties of the network. For every additional layer, the number of parameters is increased by a value that is the equal to maximum number of parameters of the neuron type possibly to be included in a cycle.

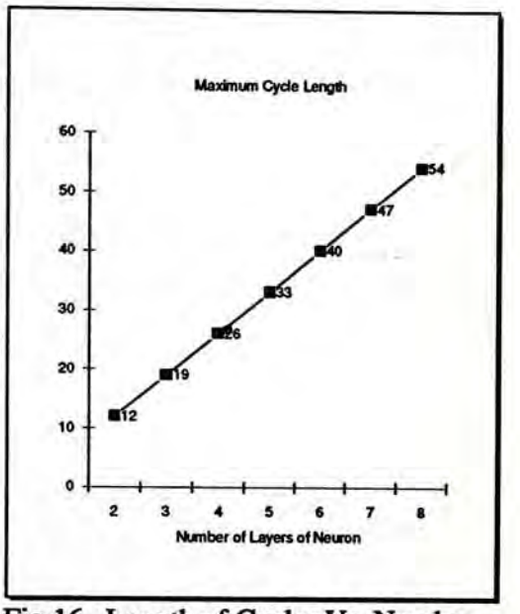


Fig 16. Length of Cycles Vs. Number of Layers (BP-Net)

5.2.4.2. Comparison Among Different Networks

When comparing the BP-Net with Boltzmann machine (both in 3-layers configuration), the BP-Net is found to have longer cycle lengths (see Table 9). This is because of shorter path from the input to the output parameters in the Boltzmann machine. The shorter path results in smaller number of parameters being involved in the cycles.

	Cycle Length
BP-Net	19
Boltzmann	6

Table 9. Comparing Boltzmann and BP-Net

5.2.5. DIFFICULTIES IN ANALYSIS

The massive number and great length of the dependency cycles found in the sample networks strongly support the need for an automatic mechanism for locating these cycles. This is particularly important for networks with a large number of layers. Taking the BP-Net as an example, the number of cycles increases exponentially with the number of layers, and the length of the cycles increases by a constant factor. This results in an exponential increment in the effort for locating all dependency cycles. Locating the cycles manually is impractical, if not impossible.

Even with an automatic tool and a limited number of layers, the task is still time consuming. The algorithm, discussed in next section, for exploring all dependency cycles is an NP-completed one. This,

together with the exponential increment in number of cycles and the linear increment in length, will consume a lot of computation resources.

Nevertheless, the resource required in locating these dependency cycles is still less than that needed for the trial runs of the ANN under investigation. A practical approach in developing an ANN system is that the designer should prepare a network with the minimum number of layers, such as one for each neuron type, at first. This network can be used for testing the correctness of the specification. After all checkings are performed, the designer can expand to the actual number of layers required. This greatly reduces the time for data dependency analysis.

At the end of next section, a more efficient algorithm for exploring the dependency cycles will be presented. That algorithm will combine the cycle detection and initial condition analysis to be described in the next section.

5.3. Dependency Cycle Analysis

A cycle in a dependency graph indicates the parameters are dependent on each other. This may give the problem of *cyclic dependency*. If this problem does not occur, the specification analyzer can determine a proper order of evaluation for the parameters. A more elaborated discussion of the relationship between message deadlock and cyclic dependency is shown in Appendix III.

5.3.1. Temporal Index Analysis

In the discussion of normalization of temporal indices, it was shown that all temporal offsets on the depending parameters are changed to zero, and the temporal offsets of the determining parameters should be zero or negative. The initial conditions in evaluating parameters concern whether it is possible to evaluate the *first* instance of these parameters. If the first instance of all parameters can be evaluated, the subsequent evaluations are possible as the second instance of the parameters can make use of the result for first instance and so on.

The first instance of a parameter can be evaluated provided that all the referenced instances of the determining parameters are already evaluated or given as initial values. In case of a dependency cycle, it is impossible to evaluate the first instance of any parameter in the cycle unless 1) there is at least one negative temporal offset of magnitude $k > 0$, so that at least one of the parameter instance being referenced may be found in the given initial values; and 2) there are k initial values from instance -1 to -k for the corresponding parameter whose previous instance(s) are being referenced.

The design tool can walk through the dependency cycles and check if it can find any negative offset. If one is found, the evaluation may start from the depending parameter associated with the negative offset. An example can be found in the fourth cycle in Figure 12, which is

a,1 ---> out,1 ---> delta,1 ---> deltaw,1 --(1)--> w,1 ---> a,1.

Parameter *W* is depending on the previous instance of parameter *DeltaW*. Evaluating the first instance of *w* requires that *deltaw* should have one initial value, *deltaw[0]*. It is therefore necessary to check that all determining parameters *P* with negative offsets of magnitude *k* has the corresponding *k* initial values. In the example given above, *DeltaW* is assigned an initial value 0 for instance 0 in the **Initial Values** entry of the **Formal Neuron** form.

5.3.2. NON-TEMPORAL INDEX ANALYSIS

When negative temporal offsets are not found, the parameters inside the cycle can still be properly evaluated if their indices are present and their ranges are consistent with each other. This will be illustrated by another simple example (as this case does not occur in the BP-Net example), and the analysis mechanism will be discussed for single and multiple parameter cases.

5.3.2.1. A Simple Example

***** FORMAL NEURON *****

NEURON TYPE :
Neu;
INPUT :
OUTPUT :
Out INTEGRAL;
INTERNAL PARAMETER :
(W[1..5], A[1..5], B[1..5], C[1..5]) INTEGRAL;
INITIAL VALUE :
W[1][*] = 1; A[1][*] = 1;
INTERNAL FUNCTION :
W[x:2..5][T] = W[x-1][T]+1;
Out[T] = W[T] * Transpose(W[T]);
B[T] = A[T]+1;
A[x : 2..5][T] = B[y : 1..4][T];

Form 17. Formal Neuron of a Imagined Network

As none of the common examples given in Appendix III requires the use of non-temporal indices in addressing individual elements of the parameters, an imagined **formal neuron** form shown in Form 17 is used for discussion.

This neuron is a very simple one. It has no input, one output *Out* and three internal parameters *W*, *A* and *B*. It always generates the same set of output, namely, the multiplication product of the matrix parameter *W*=(1, 2, 3, 4, 5) with its own transpose, giving an output of value 55 repeatedly. The other two internal parameters *A* and *B* do not involve in generating output.

Consider the equation for parameter *W* in Form 17. The simplified expression is

W[x:2..5][T] <- W[x-1][T] (5.1).

It is obvious that after the cycle analysis, the parameter *W* will be involved in at least one cycle of the form

w,1 ---> w,1.

This cycle does not have negative temporal offset associated and is therefore rejected by the *temporal index analysis* discussed in previous section.

W, however, can be evaluated properly. When the non-temporal index x iterates from 2 to 5, every T-th instance of the x-th element will depend on the already computed T-th instance of the (x-1)-th element. As the first element for every instance of W is defined in the **initial value** entry, it is possible to evaluate W[2][T] as it depends on W[1][T]. After that, W[3][T] can be evaluated as W[2][T] is computed. This process can be repeated until W[5][T] is computed.

5.3.2.2. Single Parameter

TWO INDEXES

Expression 5.1 is just a special case of the expression

P[x : u..v] <- P[y : w..z] (5.2),

where *P* is any parameter, *x* and *y* are index variables, and *u*, *v*, *w* and *z* are integer constants.

Checking the index boundaries can identify the errors in the specification. A summary of the errors is shown in Table 10. This table is constructed under the assumption that the number of values from *u* to *v* is already verified to be the same as that from *w* to *z*.

The first class of errors is *multiple assignment*. This indicates that some particular element will be assigned a value twice, or an element will be referenced before being assigned a value. Consider the following problematic case that is an example of the first row in Table 10,

Value of x	Value of y	Multiple Assignment	Mutual Dependency
Increase	Increase	$u \leq w$ AND $v \geq w$	$u = w$
Decrease	Decrease	$u \geq w$ AND $v \leq w$	$u = w$
Increase	Decrease	(1) $u \leq z$ AND $v \geq z$ (2) $u \leq w$ AND $v \geq w$	Multiple Assignment AND (w-u) is even
Decrease	Increase	(1) $u \geq z$ AND $v \leq z$ (2) $u \geq w$ AND $v \leq w$	Multiple Assignment AND (u-w) is even

Table 10. Checkings For Expressions Involving Single Parameters Only

W[x : 2..5] [T] <- W[y : 3..6] [T] (5.3).

W[2] is assigned by *W[3]*, and is *W[3]* assigned by *W[4]* and so on. Either one of two problems may occur. The first one is that *W[3]* is not evaluated when it is being referenced by *W[2]*. The second problem is that if *W[3]* has been computed, possibly through some other expressions, it will be assigned another value from *W[4]* soon. This violates the requirement of *single-valued variables* in our specification approach.

The second class of errors is mutual dependency. The first two conditions for mutual dependency are straight forward. A parameter element cannot assign value to itself. The last two conditions deserve further discussion. Take the last row as an example. The value u-w is even means that there is an odd number of elements between u and w. In every iteration, the assignment will "consume" two elements, one from the head and one from the tail. At last, the middle element between u and w will assign to itself, and this is not allowed.

MULTIPLE INDEXES

Consider the expression of the form

$P[s_1 : l_1..u_1] <- P[s_2 : l_2..u_2], P[s_3 : l_3..u_3], \ldots P[s_n : l_n..u_n]$ (5.4)

where *P* is the name of the parameter, s_i's are index variables, l_i's are the lower bounds, u_i's are the upper bounds. The test of boundaries for $l_1..u_1$ with every pair $l_k..u_k$, $k = 2, 3, \ldots n$ can be applied. If all the tests give valid results, the expression is valid and else otherwise.

5.3.2.3. Multiple Parameters

TWO PARAMETERS

The next case is one in which there are two parameters in a cycle. The example specification in Form 17 gives the following two expressions

B[T] <- A[T] (5.5)

A[x : 2..5][T] <- B[y : 1..4][T] (5.6)

and *A[1]* is defined for every instance as *1*.

One may suggest evaluating the parameters by the order *A[1], B[1], A[2], B[2], ... A[5], B[5]*, which is proper and well-defined. This involves, however, the alternative evaluation of the expressions *5.5* and *5.6*. As every equation is considered as a *single unit* to be evaluated, this is not allowed in the system. In other words, the depending parameter of an equation should be evaluated once for all by applying the equation. Hence expression *5.6* will be rejected.

MULTIPLE PARAMETERS

The two parameter case given above can be generalized to multiple parameter cases. For a cycle involving more than one parameter, the index range of the incoming edges of every parameter should not overlap with the index range of the outgoing edges. Otherwise, there will be a mutual dependency relationship for the overlapped elements of the parameter.

To get around the problem just mentioned, it is possible for the users to rearrange the equations or break down the equations for every individual element. For the given equations for parameters *A* and *B*, here is a better substitute :-

$$B[T] = A[T]+1 \qquad (5.7)$$

$$A[x : 2..5][T] = A[y : 1..4][T] + 1 \qquad (5.8).$$

5.3.3. COMBINED METHOD

The dependency cycle detection and cycle analysis can be combined to reduce the time required for computation. During the normalization process, all the offsets associated with the determining parameters are already checked to be negative. In addition, the corresponding number of initial values are also guaranteed. Under these assumptions, the only problematic case is that there is no offset at all throughout the whole cycle. The cycle is problem free whenever an offset is found.

Graphs are therefore traversed as long as no offset is found. Whenever an offset is found, the search will stop. This combination of cycle sum test with cycle detection reduces the number of different paths to be explored for every parameter and hence significantly reduces the computation. In the BP-Net example, the computation time is reduced by about 90%.

5.3.4. SCHEDULING

When all cycles are accepted by the cycle analysis, the equations defining the parameters can be scheduled for a proper order of evaluation. The scheduling is automatically performed by the analysis tool and free users from concerning this task. The analysis is based on the internal dependency graphs of the neuron types (IDGs) and the control neuron (CnIDG). It seems unnecessary for one to care about this problem, as

the equations are very simple and small in number. This is not always true, as one may refer to the example on control neuron of the Boltzmann machine given in Appendix IV.

5.3.4.1. Algorithm

With respect to every IDG, the system simulates the execution of the *first* iteration. It initializes the parameter instances defined in the specification. After that, the system tries to evaluate the instances of the parameters for the first time interval. It selects parameters randomly from the graph, locate its predecessors, and check whether they are available or not. If all the predecessors are available, the parameter can be evaluated and the corresponding expression can be inserted. If not, the system will try to evaluate the predecessors, which will involve evaluating the determining parameters of the predecessors. The input parameters are always assume to be available for all instances. As the cyclic dependencies have already been eliminated, the evaluation attempt will ultimately find a parameter with all-available predecessors.

5.3.4.2. Schedule for the BP-Net

```
W[T]          <- W[T-1], DeltaW[T-1];                          /* Scheduling of BPHidden, Training Phase */
A[T]          <- I[T], W[T];
Out[T]        <- A[T];
Delta[T]      <- Out[T], DeltaI[T], WI[T];
DeltaW[T] <- DeltaW[T-1], Delta[T], Out[T];

W[T]          <- W[T-1];                                       /* Scheduling of BPHidden, Recalling Phase */
A[T]          <- I[T], W[T];
Out[T]        <- A[T];

W[T]          <- W[T-1], DeltaW[T-1];                          /* Scheduling of BPInput, Training Phase */
A[T]          <- I[T], W[T];
Out[T]        <- A[T];
Delta[T]      <- Out[T], DeltaI[T], WI[T];
DeltaW[T] <- DeltaW[T-1], Delta[T], Out[T];

W[T]          <- W[T-1];                                       /* Scheduling of BPInput, Recalling Phase */
A[T]          <- I[T], W[T];
Out[T]        <- A[T];

W[T]          <- W[T-1], DeltaW[T-1];                          /* Scheduling of BPOut, Training Phase */
A[T]          <- I[T], W[T];
Out[T]        <- A[T];
Delta[T]      <- Teach[T], Out[T];
DeltaW[T] <- DeltaW[T-1], Delta[T], Out[T];

W[T]          <- W[T-1];                                       /* Scheduling of BPOut, Recalling Phase */
A[T]          <- I[T], W[T];
Out[T]        <- A[T];

Iteration[T]      <- Phase[T-1], Iteration[T-1];               /* Scheduling for Control Neuron */
Curr-Pattern[T]   <- Iteration[T], Curr-Pattern[T-1];
Phase[T]          <- Curr-Pattern[T]
Input-Pattern[T]  <- Curr-Pattern[T], Inpat;
Outpat[T]         <- Curr-Pattern[T], Iteration[T], Output-Pattern[T];
```

Figure 17. Expression Evaluation Schedule for the Sample BP-Net

The scheduling of the evaluation of the parameters for different neuron types of different phases for the sample BP-Net is given in Figure 17. The construction of this evaluation sequence is an additional support for the correctness of the analysis.

5.4. SYMMETRY IN GRAPH CONSTRUCTION

The number of processing units in a neural network can be very large. The limitation in memory and computation time makes it impossible to generate a complete graph to represent the ANN system under consideration. A complete graph will require that all the neurons and the connections among them are represented by nodes and edges respectively. But such a graph for representing a network with, say, ten-thousand nodes and a million connections is impossible to be constructed, stored and analyzed. Even if such a graph can be constructed, the computation time required to analyze such a graph is not acceptable.

It is therefore necessary to simplify the graphs or extract the most representative information from the graphs. A mechanism taking advantage of the *symmetric* properties of a configuration is employed, and simpler graphs can be generated.

5.4.1. BASIC APPROACH

The basic approach is to include as *few* neurons in the GDG as possible. The algorithm for building GDG is to start with a minimal set of neurons with incomplete connections, and then try to connect all parameters not yet connected. In the process of making these connections, additional neurons may be included into the selected group, and hence there will be new demand for connections. The algorithm will stop when all parameters of all neurons in the selected group are connected to parameters of other neurons in the group. The neurons are the selected group are those used in building the GDG.

To minimize the number of neurons to be included in the selected group, when the system wants to connect, say a parameter P of neuron A to parameter Q of another group of neurons B_i, $i = 1..k$, it will first check if any B_j, j in $1..k$, can be found in the already selected group of neurons. If such a B_j is found, the system will use B_j instead of introducing new neurons to the group. Only if such a B_j cannot be found will a new B_i be included in the group.

Moreover, whenever an array of elements is required for connection, just one (this is arbitrarily set to the one with the smallest index) is selected for connection. Others are ignored. Similarly, when a parameter, like *Out* in the sample BP-Net, is connected to a number of neurons, only one (also with the smallest index) is selected. For example, to make the connection for *Out* in neuron *1*, layer *1* (*Neu[1][1]*), only neuron *1* of layer *2* (*Neu[2][1]*) is selected. Other neurons *Neu[2][x]*, $x <> 1$, are ignored.

The algorithm can be viewed as a *demand-driven* algorithm. Every neuron has a need for connections. In making these connections, the system will only introduce a new neuron when there is a need. This introduction of new neurons will in turn introduce other connection needs that should be satisfied. The algorithm will be repeated until all the connection needs are satisfied.

The algorithm functions properly with *symmetric intra-layer connection* network, i.e. whenever a parameter P of neuron A_i is connected to parameter Q neuron A_j within the same layer, parameter P of neuron A_j should also be connected to parameter Q neuron A_i. The case without intra-layer connection is also a special case of symmetric intra-layer connection.

In addition, every parameter is used for one purpose only, i.e., every element of the parameter is connected to the same parameter of other neurons. If a parameter P is connected to a parameter Q, P should not be connected to any other parameter R where $R <> Q$. This is called *homogeneous connection.*

Taking advantage of the symmetric intra-layer and homogeneous connection properties, the GDGs can be constructed with just one or two neurons from each layer.

5.4.2. CONSTRUCTION OF THE BP-NET GDG

The algorithm can be demonstrated with the BP-Net example for the *training* phase. For the sake of clarity, the global dependency graph in Figure 11 is simplified to that in Figure 18, in which the internal dependency graphs and the dependency on *Phase* are omitted.

The example BP-Net has symmetric connection in that 1) there is no intra-layer connection among the neurons, and 2) the parameters of every neuron type is used for one purpose only. The use of single-purpose parameters implies that it is possible to investigate the network without considering all connections of all elements of a single parameter. Instead of considering the whole array of parameters like *I*, just a representative, say, I[1] is extracted and connected. The absence of intra-layer connection implies that only one neuron from every layer will be introduced, as the connections required for any neuron will not demand the introduction of a neuron in the same layer.

Initially, there are only the control neuron *CN* in *UList* (the Un-connected List), the list of neurons with incomplete connections, and in *G*, the list of identified neurons. The CnIDG indicates that it has two output parameters, *Input-Pattern* and the phase parameter, *Phase*. Assume that the system tries first to connect *Phase* to other neurons. It will discover that it cannot find an explicit connection for it. This is because *Phase* is implicitly connected to all the neurons with parameters functioning differently in various phases. This special parameter will be put aside and handled later.

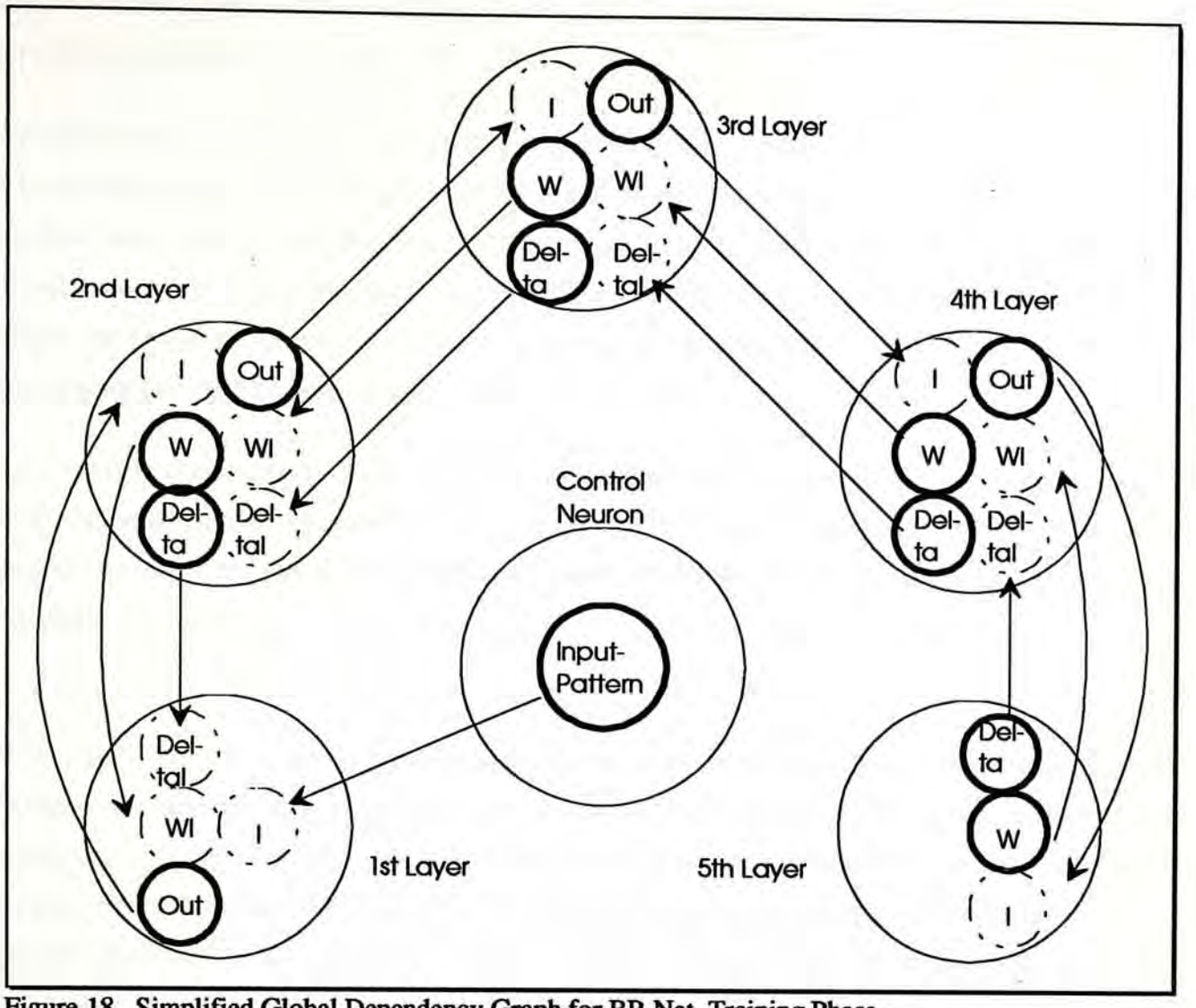


Figure 18. Simplified Global Dependency Graph for BP-Net, Training Phase

The system will then connect *Input-Pattern* to other neurons. Through the information in the **connection pattern** entry, the system discovers that *Input-Pattern* should be connected to parameter *I* of *Neu[1][x]*. As there is no neuron in the first layer in *G*, the system will try to introduce the smallest one among the possibilities, which is *Neu[1][1]*. *CN* is now fully connected and removed from *UList*. *Neu[1][1]* in *UList* will trigger another cycle. The IDG identified is of *BPInput* type and has three input parameters *I*, *WI* and *DeltaI*, and one output parameter *Out*. The system tries connecting these parameters.

Parameter *I* is already connected. Now parameter *WI[y]* is tried. It is connected to *W[w]* of *Neu[2][y]*. Not any neuron with index 2 in the first dimension is found in *G* so a new one is generated. As the system has choices, it will connect *WI[1]* to *W[1]* of *Neu[2][1]*. *Neu[2][1]* is then added to *G* and *UList*.

Parameter *Out* is now being tried to connect to *I[1]* of *Neu[2][w]*. When the system search for *G*, it finds that *Neu[2][1]* can satisfy the index value for *Neu[2][w]* by instantiating w to be 1. It will then connect *Out* to this neuron. Hence the system is stopped from using a new neuron by looking at the available one

first. The same will occur to parameter *Delta*. All the parameters of *Neu[1][1]* are connected and removed from *UList*.

Neu[2][1] will be left in *UList* and trigger another cycle. The whole process is now repeated. Afterwards, another set of neurons will be left and so on. According to the connection pattern and internal dependency graphs, *Neu[2][1]* will introduce *Neu[3][1]* into *UList*. *Neu[3][1]* will in turn introduce *Neu[4][1]* and *Neu[4][1]* will introduce *Neu[5][1]*. The connections between these two neurons will make all parameters marked as connected and no other neurons are left in UList. The global connections are now the same as those shown in Figure 18.

After that, the dependency on *Phase* will be handled again. From the specification on the **formal neuron** form, the parameters *W*, *Delta* and *DeltaW* of all neuron types function differently in various phases. So they all depend on the global phase parameter, *Phase*. Finally the dependency graph in Figure 11 can be obtained.

5.4.3. LIMITATION

It is quite obvious that the algorithm cannot handle *asymmetric* connections within the layers. For example, if every neuron is connected to the neuron just below it and not vice versa, the demand for connecting, say, the first neuron in a layer will introduce the second neuron in the same layer. This second neuron will in turn introduce the third, which introduces the fourth and so on. All the neurons in the same layer will thus be generated. It seems that there is no good solution for handling this problem but, luckily, this configuration is rare in practical systems.

Connecting just one element of an input/output parameter array can also be dangerous if *homogeneous connection* is not followed. For example, if the elements of *DeltaW* are used for different purposes and different connections, only one of them will be selected. The subsequent analysis depending on the graphs constructed will thus ignore the possible errors occurring in other connection cases. Homogeneous connection can be enforced by allowing only one connection expression for each input/output parameter.

6. ATTRIBUTE ANALYSIS

Attribute analysis is used to locate specification errors other than the cyclic dependency problem. The first group of tests, known as *parameter analysis* in this context, is still based on data dependency analysis. The second group of tests, known as *constraint checking*, is based on some other *simple checkings* including syntactic and semantic checkings, and simple matching among information from different entries. In this chapter, supplementary simple examples other than the BP-Net are used to illustrate the basic idea. The complete algorithms are shown in Appendix II.

6.1. PARAMETER ANALYSIS

Parameter analysis is to ensure that the parameters of the *formal neurons* and *control neuron* are well-defined. Well-defined means that the associations among the attributes of the parameters are consistent. For example, an internal function for an input parameter is meaningless and should be removed. These types of checkings are straight forward but useful.

***** FORMAL NEURON *****

```
NEURON TYPE :
    Redundant;
INPUT :
    A BINARY;
OUTPUT :
    (B, C, D) BINARY;
INTERNAL PARAMETER :
    (E, F, G) BINARY;
INITIAL VALUE :
INTERNAL FUNCTION :
    C[T] = A[T];
    B[T] = D[T];
    A[T] = B[T];
    Phase1 {
        F[T] = C[T] XOR D[T];
        E[T] = F[T]; }
    Phase2 {
        F[T] = E[T-1];
        E[T] = 0;    }
```

Form 18. A Form for Formal Neuron with Redundant Parameters

As there is no attribute problem in the sample BP-Net, a supplementary example specification in Form 18 is used to discuss the checking mechanism. It is not easy to determine whether there is any error associated with this form at this moment, but it will be shown later that some parameters can be removed.

6.1.1. INTERNAL DEPENDENCY GRAPHS (IDGs)

The *internal dependency graphs* for formal neurons (IDGs) and the control neuron (CnIDG) mentioned before should be used for checking the properties of the parameters. These tests should be performed as soon as the IDGs are generated because any subsequent analysis assumes the correctness of the dependency graphs.

6.1.1.1. Correct Properties of Parameters in IDGs

Rule 1. *All parameters in the graphs should be defined parameters.* An error will be reported if violated. Users are requested to check the names used in the equations and parameter definitions.

Rule 2. *All input parameters in the graphs should not have incoming edges in the graphs,* which is obvious as input parameters cannot be updated. An error will be reported if violated. Users are requested to modify the input property or change the associated equation.

Rule 3. *All parameters in the graphs, except the input ones, should have at least one incoming edge,* as every parameter should be updated continuously. An error is reported if violated. Users are requested to change this parameter into a constant, add an equation for updating this parameter or delete this parameter.

Rule 4. If a parameter has more than one incoming edge, *the indices of them should not be overlapped.* Possessing more than one incoming edge for the parameters does not contradict with the single-valued variable requirement, as different equations may be used for updating different parts of the same array. This will introduce more than one incoming edge to the same parameter but in such cases, the indices of the same parameter should be tested to guarantee that they are all different to avoid multiple assignments to the same element of an array. An error is reported is violated. Users are requested to check the indices of the parameters in the equations.

6.1.1.2. Example

The above possible errors can be demonstrated with Form 18. The internal dependence graphs for both *phase1* and *phase2* of Form 18 are shown in Figure 19. The parameters are marked according to the convention in Figure 8. The following errors are identified in the graphs.

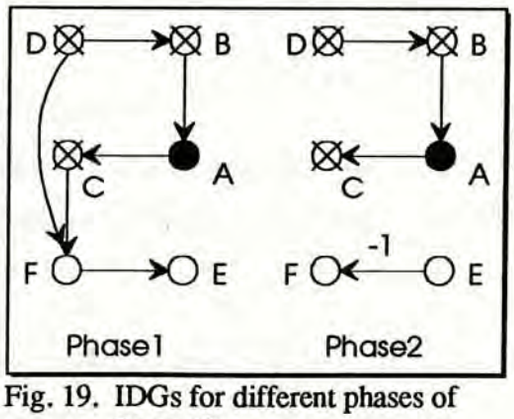


Fig. 19. IDGs for different phases of Form 18

There are no violations of rules 1 and 4, but 1) Input parameter *A* has incoming edge in the graph (rule 2); 2) Internal

parameter *D* has no incoming edge in both phases (rule 3); 3) internal parameter *E* has no incoming edge in *Phase2* (rule 3).

After receiving error messages from the system, the user determines that 1) it is that *B* is dependent on *A* but not vice versa; 2) *D* is dependent on *B* in both phases; 3) *E* is also dependent on *F* in Phase2. The resultant IDGs after these modifications are shown in Figure 20 and they can pass the tests.

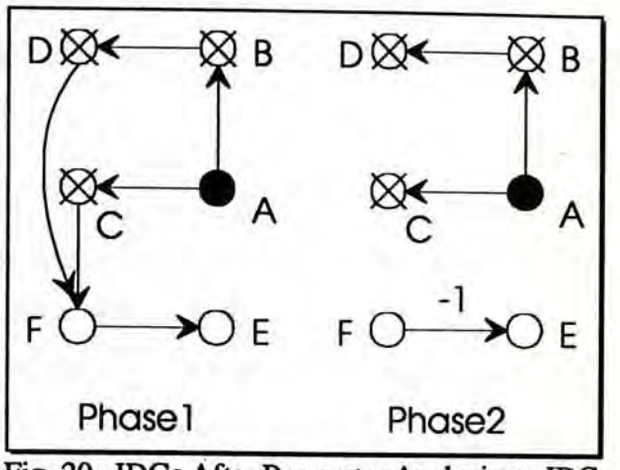


Fig. 20. IDGs After Parameter Analysis on IDG

The tests on the internal dependency graph of the **control neuron** are the same as those on other internal dependency graphs, except that it is not necessary to have different phases in the former case.

6.1.2. COMBINED INTERNAL DEPENDENCY GRAPHS (CIDG)

A type of graph not mentioned before is the combined internal dependency graphs (CIDGs), which are constructed by merging the IDGs of different phases together. This type of graph is for parameter analysis only, and the analysis is applied immediately after the IDGs are constructed. Every neuron type will have its corresponding CIDG. The CIDG for Form 18 is shown in Figure 21.

The tests on CIDG are also applicable to *internal dependency graph* of a control neuron (CnIDG). This is because there is no difference between the *internal dependency graphs* and *combined internal dependency graph* for a control neuron, and they are both equal to CnIDG. Checkings unique to CnIDG are that *the parameter Phase[T] should be defined in the equations and a pre-defined value Terminate should be assigned to Phase[T] under some conditions.*

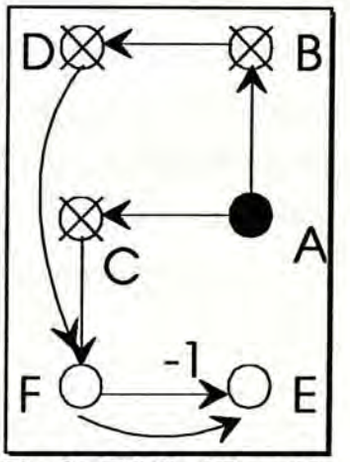


Fig. 21. CIDG of Form 18

6.1.2.1. Tests on Parameters of CIDG

Rule 5. *All parameters should appear in the CIDG.* An error is reported if violated. Users are requested to replace the parameter with a constant (if the parameter is a constant in all phases and hence not appearing in all IDGs), to remove the parameter (if the parameter really does not specify in all phases), or to modify the equations.

Rule 6. *All parameters, except output ones, must have at least one outgoing edges in CIDG.* The values of the internal parameters should be used by other parameters within the same neuron, otherwise the

parameters are useless. The output parameters send values to other neurons so their values may not be used internally and hence no outgoing edges in CIDG. An error is reported if violated. Users are requested to delete the parameter or to modify the equations. This rule cannot be checked using individual IDGs, as parameters not being referenced in one phase may be storing some value to be used in other phases.

Rule 7. The number of output parameters in the CIDG should be counted to ensure that *there should be at least one output parameter* for every neuron. A neuron without input parameter is acceptable because it may be a neuron acting as a source for generating input for subsequent layers. On the other hand, a neuron without output is not acceptable as the neuron will be useless in the network. An error is reported if violated. Users are requested to assign output properties to some parameters and modify the equations.

Rule 8. *Any parameters, except output parameters, should be directly or indirectly updating some output parameters*, i.e. all parameters should involve in updating functions that will eventually modify the output from the neuron. As the main purpose of the neuron is to generate output values, parameters that do not affect the output are useless. To locate these parameters, the system starts from the output parameters and traverses back through the edges of the CIDG, marking all nodes being visited. Those parameter nodes that are not marked will be reported as an error. Users are requested to remove the parameters with their associated updating equations, modify the equations or change some internal parameters to output ones.

6.1.2.2. Example

There is no violation of rules 6 and 7 in this example. Parameter *G*, however, is not found in the CIDG shown in Figure 21(rule 5) so the system will request the user to check its usefulness. In addition, parameters *E* and *F* are not involved in updating the output parameters (rule 8). Hence although they are updating and being updated by each other, they are useless in the processing unit. Assuming that all these parameters are decided to be removed, the checked and modified neuron is that in next section.

6.1.3. FINALIZED NEURON OBTAINED

The resultant graph after the user has responded to all the warnings and removed the problematic parameters is shown in Figure 22. All the redundant and useless parameters are identified and removed by the user. The resultant graph is much simplified and in this case all the internal parameters have been removed. The resultant form for the revised neuron type is shown in Form 19. One can appreciate the significant differences between this form and the original one, and the difficulties in identifying these removals.

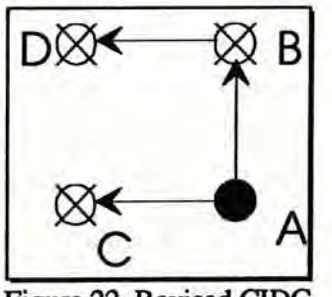


Figure 22. Revised CIDG

*** FORMAL NEURON ***

```
NEURON TYPE :
    Redundant;
INPUT :
    A BINARY;
OUTPUT :
    (B, C, D) BINARY;
INTERNAL PARAMETER :
INITIAL VALUE :
INTERNAL FUNCTION :
    C[T] = A[T];
    D[T] = B[T];
    B[T] = A[T];
```

Form 19. Resultant Specification For Formal Neuron After Removing Redundant and Useless Parameters

6.1.4. CIDG OF THE BP-NET

The CIDGs for the example BP-Net are shown in Figure 23. The CIDGs are actually the same as IDGs of the training phases as the IDGs for the recalling phases are just subgraphs of them. The CIDGs have passed all the tests mentioned in this section.

6.2. CONSTRAINT CHECKING

Besides the data dependency analysis mentioned in the previous section, there are also some other simple tests that can be used to check the correctness of a network system. They should be applied before the use of *data dependency analysis* to locate any obvious mistakes.

6.2.1. SYNTACTIC, SEMANTIC AND SIMPLE CHECKINGS

6.2.1.1. The Syntactic & Semantic Techniques

The *syntactic and semantic* checkings are well-developed techniques in conventional programming language compilers. The *syntactic checking* is used in enforcing the correctness of the form entry syntax. It can be used for checking that all the essential attributes of the entries are present and the optional attributes, if any, are correct.

The *semantic checking* is mainly used for checking unique labels, well-defined labels, and matching among connected parameters. The matching among parameters involves compatible range of values, matched type (continuous, binary, integral, or discrete), and, if any, matched step size. This is an extension of the common type checking in conventional programming language.

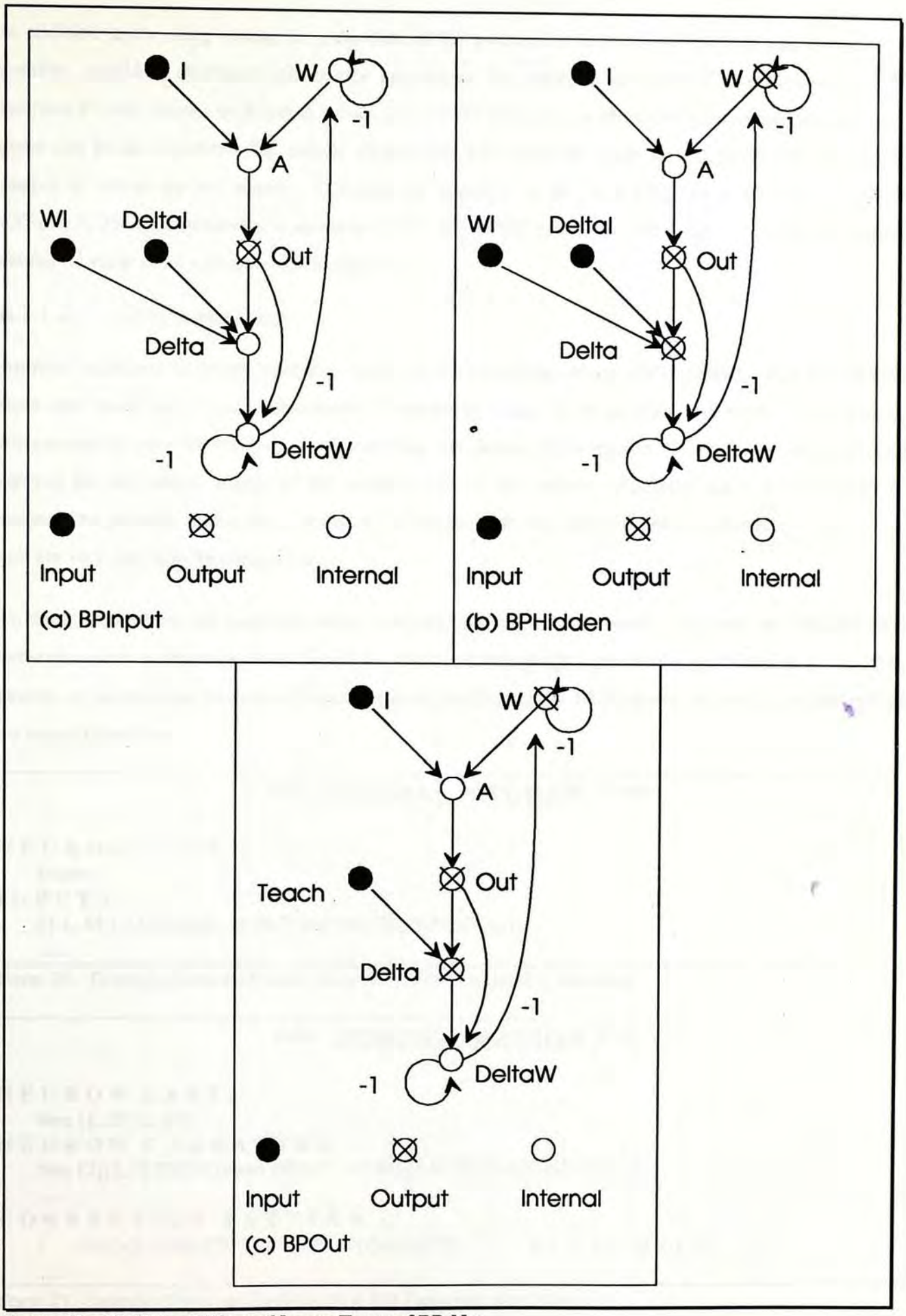


Figure 23. CIDG for Different Neuron Types of BP-Net

In addition to checking among the properties of the parameters, all built-in functions and operators, if possible, should be attributed with similar properties. For example, the range of values returned by the function e^x with respect to different values of x can be defined as a particular type. After that, the computer can locate equations that cannot produce the full range of values for the parameter, or that the ranges of values do not match. Consider an equation of the form *OUT[T] = EXP(A[T])* where *EXP(A[T])* is the exponent of a parameter *A[T]*. If *OUT[T]* is defined with range (-1, 0), the assignment cannot be valid as no exponent of *e* is negative.

6.2.1.2. Simple Matching

Another technique is *simple matching* based on the knowledge about ANN systems. One can observe from the forms that much information is repeatedly stated in more than one entry. This dummy information is used for verification by matching the values of the entries. For example, there are two sources for the output degree of the neurons, one in the **neuron character** entry and the other in **connection pattern**. These two values should match with each other. These checkings are tailor-made and are very useful in locating errors..

To demonstrate how this matching work, a simple case can be considered. A partial specification on a network system is shown in Form 20 and 21. There are two design errors in the specification: 1) the 10-th neuron of second layer has missed its character definition; 2) the 10-th neuron in every layer has missed an input connection.

***** FORMAL NEURON *****

```
NEURON TYPE :
    Demo;
INPUTS :
    I [ 1..M ] RANGE (0, 1) INCLUSIVE CONTINUOUS;
    ......
```

Form 20. Example Form on Formal Neuron For Demonstrating Matching

***** CONFIGURATION *****

```
NEURON LABEL :
    Neu [1..2] [1..10];
NEURON CHARACTER :
    Neu [2][1..9] TYPE Demo INPUT-DEGREE 10 OUTPUT-DEGREE 9;
    ......
CONNECTION PATTERN :
    {   Neu[x][y].In[z][T+1] = Neu[x+1].Out[y][T];        x:1..2; y:1..10; z:1..9      };
    ......
```

Form 21. Example Form on Configuration For Demonstrating Matching

Table 11 shows part of the attributes generated for neuron Neu[2][10]. The first error, the 10-th neuron not being defined in **neuron character**, is found when *Neuron_type* is *undefined*. This indicates that the system cannot determine the neuron type. The system also cannot determine the input and output degrees of this neuron from the definition in **neuron character**.

The second error is detected as the *In_label_num* does not match with *Connected_in*. This is because the value of *Connected_in*, as determined by the connection pattern, is only 9, while the number of input labels, *In_label_num*, inherited from the neuron type is 10. Hence the second error is detected.

```
Record
    Neuron_type      : ?                          /* type of neuron */
    In_label_num     : 10                         /* inherited from type */
    Input_num        : ?                          /* from neuron character*/
    Output_num       : ?                          /* from neuron character */
    layer_pos        : 10                         /* position in layer */
    Out_list         : <output connection list>
    Connected_in     : 9                          /* connected input labels */
    ......     ......
```

Table 11. Internal Representation of Neu[2][10]

6.2.2. CONSTRAINTS

Different constraints have to be applied on different specification forms and tests mentioned in previous sections can be employed to enforce these restrictions. The aim is to ensure that the attributes defined in different entries do not contradict with each other.

The discussion will focus on the application of the mechanism just mentioned. It will identify the constraints associated with the entries in every component separately.

6.2.2.1. Constraints on Formal Neuron

Entries	Requirement	Checkings
Neuron Type	every type name should be unique	Matching
Parameter Declaration	initialization value is within boundary	Matching
Functions	type and range compatible identical phases for all neuron types	Semantic Matching

Table 12. Constraints and Checkings on Formal Neuron Attributes

The constraints on the attributes of the *formal neuron* are shown in Table 12. The first requirement is similar to the case for unique identifier name in conventional programming languages. In fact, all labels/names in the system must be unique. Parameters with the same name but within different neurons

are considered as different labels. To enforce this uniqueness, it is just necessary to compare all the labels throughout the three forms.

When an internal parameter is declared with *boundary* and *initialization* values, the initial value must be within the boundary. This can be enforced by comparing the initialization value with the upper and lower bounds.

Every *function* and *operators* in the system, whenever possible, is associated with types and range of values it will return. This helps to verify that the functions being applied are compatible with the attributes of the parameters. In addition, the phase labels used under the **internal function** entries of different neuron types should be identical. A simple comparison among them will be sufficient for checking this.

6.2.2.2. Constraints on Configuration

The constraint on **neuron label** entry is the same as that for any labels. In **neuron character** entry, labels used should already be defined. This can be checked by simply comparing the labels with the defined one. The number of *neuron definition parameters* (NDPs) should match with that in formal neuron declarations. Moreover, these parameters should be able to be evaluated at compile time. In other words, the parameters should be constants or iterating variables of definite loops.

Entries	Requirement	Checkings
Neuron Label	unique labels	Matching
Neuron Character	neuron type used is defined	Matching
	correct number of NDPs	Matching
	parameters are compilation constants	Semantic
Connection Pattern	correct I/O degree	Matching
	match in range, type and step size	Semantic
	matching I/O parameters	Matching

Table 13. Constraints and Checkings on Configuration Attributes

There are three constraints for the **connection pattern** entry. Firstly the input and output degrees should match with those defined in other sources. Apart from the connection degree, the connected input and output should match in their characteristics. These characteristics are those defined by the input declaration of the neuron type. This is to guarantee that the signals transmitted can be understood by the receiver. Obviously, the equations should define transmitting signals from an output parameter to an input one.

6.2.2.3. Constraints on Control Neuron

Entries	Requirement	Checkings
Global Parameter	initialization value is within boundary	Matching
Input & Output File	compatible with I/O layers	Semantic
Global Function	type and range compatible well-defined phase transitions	Semantic Matching

Table 14. Constraint and Checkings on Control neuron

The constraints on **global parameter** entry are the same as those of **internal parameter**. For the **input** and **output file** entries, the input pattern generated should be compatible with the neuron characteristic of the input layer, i.e., range boundary, type and step size of the neuron inputs which can be verified by semantic checkings. These checkings are also applicable to the output case.

The **global function** entry has requirements and checking mechanism similar to those enforcing the type, range and step size compatibility of the **internal functions.** It is also responsible for determining phase transitions. All and only those phase variables declared inside the neurons should be included in phase transitions. Moreover, there should be condition(s) for the system to end properly by assigning the pre-defined value *Terminate* to *Phase*.

6.3. COMPLETE CHECKING PROCEDURE

After all the analyses for the specifications are explained, the complete procedure for the design analysis process can be illustrated. The specification forms are first passed through the simple test that identifies any trivial problems. Afterwards, the **formal neuron** and **control neuron** forms are passed to construct the *internal dependency graphs*, and *combined internal dependency graphs*. These graphs are tested by the *parameter analysis* procedure.

After that, the internal dependency graphs of control neuron and ordinary neurons will combine with the information from the specification on **configuration** form to produce the *global dependency graphs*. After that, the *deadlock analysis* can be applied. If the analysis result is positive, the *schedule analysis* can be applied to the *internal dependency graphs* of formal neuron and control neuron after parameter analysis.

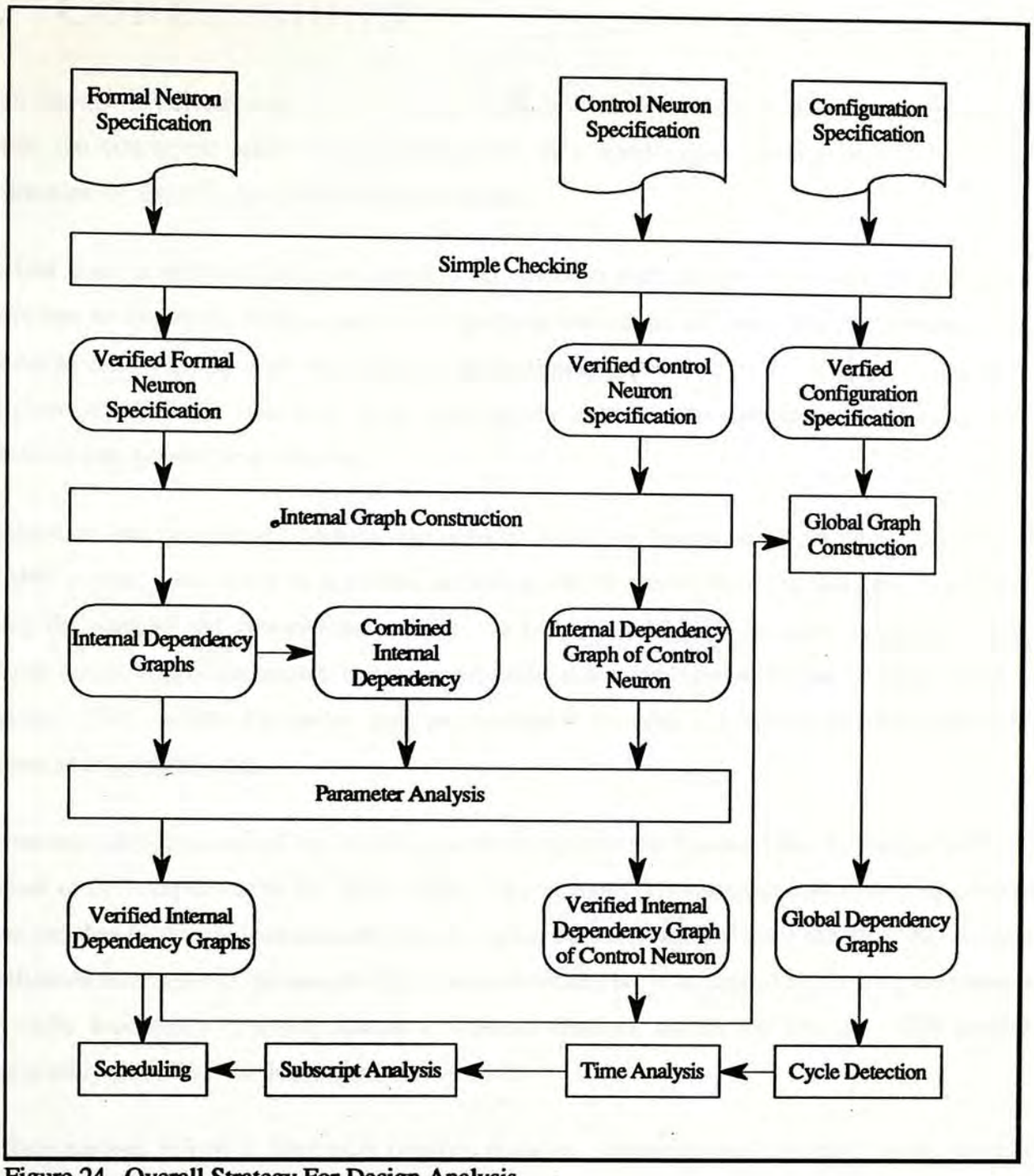


Figure 24. Overall Strategy For Design Analysis

7. CONCLUSIONS

There are two important issues in developing Artificial Neural Network (ANN) systems, namely 1) to express the concurrent nature of an ANN system in a specification environment; 2) to verify the specification by a specification analyzer automatically.

The first issue is addressed by using dataflow specification methodology. This specification approach allows user to specify the static connection property instead of the dynamic concurrence property of the network to control the parallelism. All other parallelism is introduced by the design tool automatically. Designers of ANNs are thus freed from handling the detailed communication and timing problems associated with parallel programming.

The dataflow specification methodology also provides a uniform framework to descfibe various aspects of an ANN system. All levels of activities, including simple neuron internal functions, configurations among the neurons and system-wide controls, can be specified by the *dataflow equations*. Both the dynamic functioning of the neurons and the static configuration are captured by the equations of the specifications. This uniform framework lays the foundation for both a user-friendly environment and a uniform analysis mechanism.

The second issue is addressed by building extensive analysis mechanisms into the design tool. Every attribute of the components in the ANN system is given some characteristics and associated constraints. Errors can then be detected automatically by the system and much manual labor can be saved. Among the specification errors, one of the most difficult one to be located is cyclic dependency among the parameters. The cyclic dependency is closely related to message deadlock among the neurons. This problem is automatically located by the data dependency analysis.

Attribute analysis is used to filter other common mistakes. Parameter analysis relies on the dependency graphs to remove useless parameters, enforce input/output properties of parameters and the same. Constraint checking routines make use of the knowledge of each neuron type to enforce consistency among the attributes of the components.

In brief, the design tool helps a user to analyze an artificial neural network under development. The specification approach will make the dynamic behavior of the target system to be stated more explicitly, so that static analysis (i.e. at compilation time) can be performed. This saves a lot of computation resource in testing. In addition, with a suitable implementation generator, the design tool may also help the development of both hardware implementations and software simulations for the ANN system under consideration.

7.1. LIMITATIONS

The current design tool is not perfect. As the design tool is based on dataflow specification approach, in which the *dependency* relation among parameters and neurons should be well-defined, it is impossible at this moment for the design tool to handle models with dynamic configurations and dependency relations. Some other limitations will be investigated in more details.

7.1.1. EXCLUSIVE CONDITIONAL DEPENDENCY CYCLES

The *exclusive conditional dependency* cycles are formed by equations that are dependent on one another under different conditions. Consider the equations

A[T] = IF C[T] > 0 THEN 1 ELSE B[T]*5 and

B[T] = IF C[T] ≤ 0 THEN -1 ELSE A[T]*2.

The two variables *A* and *B* are dependent on each other of the same instance. It is therefore not possible to determine which parameter is depending on the other *in all cases*. In normal conditions, this will imply a cyclic dependency. The given two equations, however, are in mutually exclusive conditions that will guarantee that the two parameters will not be dependent on the other at the same time. If *C[T]* > 0, *A[T]* can be evaluated first but if *C[T]* ≤ 0, *B[T]* can be evaluated first. In both cases, the value of the other parameter can then be evaluated and hence the computation can proceed. This is an example case in which the dependency among the parameters is *dynamic*.

In the dependency graph shown in Figure 25, nevertheless, the system will find that there is a cycle for parameters *A* and *B* and a problem is reported although there is actually no problem. To avoid rejecting this valid case wrongly means that the system should be able to recognize the meanings of the conditions. This is not easy as the condition(s) can be very complicated and the number of conditions can be very large. Incorporating this capability into the design tool is very costly and, more importantly, not worthwhile. The expression can in fact be transformed into

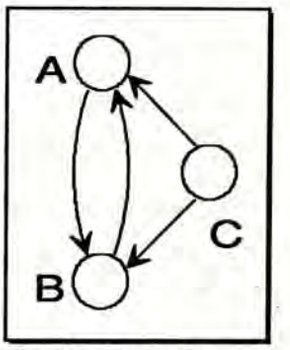


Fig. 25. Exclusive Cycles

A[T] = IF C[T] > 0 THEN 1 ELSE -5 and

B[T] = IF C[T] ≤ 0 THEN -1 ELSE 2.

In other words, one can investigate the expression for, say, *A* and check when should *B* be executed before it. In that case, *B* should be dependent on some other parameters or constants. This parameter or constant can be put directly into the expression for *A* and hence the dependency on *B* is removed. The same investigation is also applied to *B* and the cyclic dependency can be removed.

7.1.2. MAXIMUM PARALLELISM

The specification given by the users may not be optimal in the degree of parallel execution. An algorithm, given in Appendix IV, has been developed to extend the degree of parallelism of some simple cases. Three dependency subgraphs *1*, *2* and *3* are shown in Figure 26 as examples. Subgraphs *1* is the case in which the descendants of a parameter form independent branches. In subgraph *2*, the descendent branches after some levels has common depending parameters. These two kinds of subgraphs can be analyzed by the system to increase parallelism. The broken lines in the figure are the grouping of the parameters that will be evaluated in the same processing sub-neuron.

The case for subgraph *3* is more complicated. The degree of parallelism can also be increased by the indicated allocation, as *V* and *X* can be computed in parallel, although they may be of different instances. The system will not suggest this parallel execution as *X* is also a dependent of *V*. A dependent is supposed not to execute in parallel with its predecessors, as usually this will not improve the performance. The resultant execution is that the four parameters are evaluated in a single processing unit that can compute them in sequence. Some more sophisticated algorithms may help but it is beyond the scope of the specification analyzer.

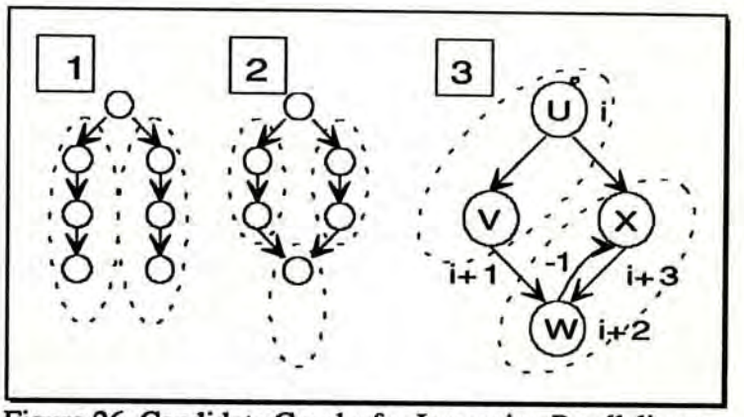


Figure 26. Candidate Graphs for Increasing Parallelisms

A more sophisticated analysis should also consider the time difference between the evaluation of different equations. For example, the equations involving matrix computations will not take the same computation time as those involving single variable manipulations. In addition, the communication cost between parallel neurons should be included. An algorithm taking into consideration these factors will give a more efficient system.

REFERENCE

1. Akingbehin, K. "A Hybrid Architecture for Programmable Computing and Evolutionary Learning" *Journal of Parallel and Distributed Computing* Vol. 6, pp.243-263(1989).

2. Azema, M., Lee, M.K.O. and Treleaven, P.C. "Neural Network Programming Systems." London, 1989 (received from the author).

3. Barbosa, V.C., Lima, P.M.V, "On the Distributed Parallel Simulation of Hopfield's Neural Networks." *Software - Practice and Experience*, Vol. 20(10), 967-983 (October 1990).

4. Cruz, C.A., Hanson, W.A., Tam, J.Y., "Neural Network Emulation Hardware Design Considerations." *IEEE Proceedings of First International Conference on Neural Networks*. Vol.111, San Diego, 1987, pp.427-434.

5. Deprit, E., "Implementing Recurrent Back-Propagation on the Connection Machine." *Neural Networks*, Vol. 2, pp.295-314, 1989.

6. Feldman, J.A., Fanty, M.A. and Goddard, N.H., "Computing with Structured Neural Networks." *IEEE Computer*, 3 [1988],91-103.

7. Ghosh, J. and Kwang, K. "Mapping Neural Networks onto Message-Passing Multicomputers." *Journal of Parallel and Distributed Computing*, vol 6, pp.291-330(1989).

8. Hanson, W.A., Cruz, C.A., Tam, J.Y. "CONE - Computational Network Environment." *IEEE Proceedings of First International Conference on Neural Networks*. Vol.111, San Diego, 1987, pp.531-538.

9. Habib, M., Newcomb R.W. "Neuron Type Processor Modeling Using a Timed Petri Net", *IEEE Transactions on Neural Networks*, Vol.1, No.4, December 1990.

10. Hood, G. "A Graphical Design System for Neural Networks." *IEEE Proceedings of First International Conference on Neural Networks*. Vol.111, San Diego, 1987, pp.453-460.

11. Hwang, J.N., Vlontzos, J.A. and Kung, S.Y. "A Systolic Neural Network Architecture for Hidden Markov Models." *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol.37, No.12, December 1989.

12. Kan, W.K. and Aleksander, I., "A Probabilistic Logic Neuron Network for Associative Learning." *IEEE Proceedings of International Conference on Neural Networks*, San Diego, June 1987, Vol.II, pp.541-548.

13. Kan, W.K., "A Logic Connectionist Approach to Self-Organized Associative Memory." *Proceedings of IEEE International Conference on System, Man and Cybernetics*, pp.690-693, Peking, August, 1988.

14. Kemke, C. "Modelling Neural Networks by Means of Networks of Finite Automata." *IEEE Proceedings of First International Conference on Neural Networks*. Vol. III, pp.23-29, San Diego, 1987.

15. Koikkalainen, P., Oja, E. "Specification and Implementation Environment for Neural Networks Using Communicating Sequential Processes." *IEEE Proceedings of First International Conference on Neural Networks*, Vol. 1, pp 533-540, 1988.

16. Kohonen, Teuvo, "State of the Art in Neural Computing", *IEEE Proceedings of First International Conference on Neural Networks*. Vol.1, pp.79-90, San Diego, 1987.

17. Korb, T. and Zell, A. "A Declarative Neural Network Description Language. " *North-Holland Microprocessing Microprogramming* 27, [1989], 181-188.

18. Korn G. A., "A New Environment for Interactive Neural Network Experiments." *Neural Networks*, Vol. 2, pp.229-237, 1989.

19. Kraft, T.T., Frostrom, S.A., MacRitchie, B. and Rogers, A.E. "The Specification of a Concurrent Backpropagation Network Architecture using Actors." Science Applications International Corporation, San Diego (received from the author).

20. Kung, S.Y. and Hwang, J.N. "A Unified Systolic Architecture for Artificial Neural Networks." *Journal of Parallel and Distributed Computing* Vol.6, pp.558-587(1989).

21. Kung, S.Y. and Hwang, J.N. "Neural Network Architectures for Robotic Applications." *IEEE Transactions on Robotics and Automation*, Vol. 5, No.5, October 1989.

22. Lee, I., Prywes, N. and Szymanski, B., "Partitioning of Massive/Real-Time Programs for Parallel Processing." *Advances in Computers*, Vol.25, 1986.

23. Lee, M.K.O. "PYGMALION -- A European Neural Computing Research Project." *Proceedings of Meeting on Neural Computing*, London, 1989.

24. Lester, B.P., "A Neural Network Simulation Language Based on Multi-Pascal." *IEEE Proceedings of First International Conference on Neural Networks*. Vol.111, San Diego, 1987, pp.347-354.

25. Liu, L.M., and Prywes, N.S. "Using Executable Specification Languages for Interface Checking of Large, Real-Time/Distributed Systems." *Proceedings of International Conference of Distributed Computing Systems* 1989.

26. Liu, L.M. and Prywes, N.S. "SPCHECK : A Specification-based Tool for Interface Checking of Large, Real-time/Distributed Systems." *Information Processing 89*, North-Holland, 1989.

27. Liu, P.L., Kan, W.K., Wong, W. "A Specification Approach to Artificial Neural Network Design." *Proceedings of IEEE TENCON'90*, Hong Kong, Sept. 1990.

28. Maekawa, M., Oldehoeft, A.E., Oldehoeft R.R., *Operating System, Advanced Concepts*, Benjamin Publishing Company, 1987.

29. Nijhuis, J., Spaanenburg, L. and Warkowski, F. "Structure and Application of NNSIM: A general-purpose Neural Network SIMulator." *North-Holland Microprocessing and Microprogramming* 27, [1989], pp.189-194.

30. Paik, E., Gungner, D. and Skrzypek, H., "UCLA SFINX - A Neural Network Simulation Environment." *IEEE Proceedings of First International Conference on Neural Networks* Vol.111, San Diego, 1987, pp.367-375.

31. Parhi, K.K., Messerschmitt, D.G., "Static Rate-Optimal Scheduling of Iterative Data-flow Programs via Optimum Unfolding", *IEEE Transactions on Computers*, Vol.40, No.2, February 1991.

32. Prywes, N., Shi, Y., Szymanski, B., and Tseng, J., "Supersystem Programming with Model." *IEEE Computer*, pp.50-60, 2[1986].

33. Rumelhart, D.E., McClelland, J.L. and the PDP Research Group, *Parallel Distributed Processing : Explorations in the Microstructure of Cogni-tion, Vol1: Foundations*, 1986.

34. Smith, A.B., "A Parallel PDP Network Simulator." *IEEE Proceedings of First International Conference on Neural Networks*. Vol.III, San Diego, 1987, pp.377-384.

35. Smith, M. and Downs, T. "On the Development of a Neural Net Coprocessor Based on Dataflow Principles." pp.43-44, 1989.

36. Treleaven, P., "Neurocomputers." University College London (received from the author).

37. Wadge, W.W. "An Extensional Treatment of Dataflow Deadlock." *Lecture Notes in Computer Science*, Vol. 70, International Symposium on Semantics of Concurrent Computation, 1979.

38. Wohl, P., Christopher, T.W. "A Fine Grained Neural Net Simulator Encoded in Coarse Grain OOMDC", *Proceeding of the 4th Annual Parallel Processing Symposium*, April 4-6, 1990, edited by K.H.Canter, IEEE Computer Science Press.

39. Wolfe, M. "Multiprocessor Synchronization for Concurrent Loops." *IEEE Software*, 1988.

40. Wolfe, M. and Banerjee, U. "Data Dependence and Its Application to Parallel Processing." *International Journal of Parallel Programming*, Vol.16, No.2, 1987.

41. Zipser, D., Rabin, D.E. P3: A Parallel Network Simulating System. Parallel Distributed Processing, Vol. I: Foundations, Rumelhart D.E., McClelland J.L., MIT Press, [1986].

APPENDIX

I. FORM SYNTAX

A. SYNTAX CONVENTIONS

<> Items enclosed in angle brackets should be substituted with the further specific value.
[] Items enclosed in square brackets are optional.
[]* Items enclosed in square brackets with a star can be repeated from zero to infinite times.
[]+ Items enclosed in square brackets with a cross can be repeated from one to infinite times
{} Vertical items in braces are exclusive, only one of them can be used.
BOLD Items in capital bold are keywords.

B. FORM DEFINITION

The form definition start with the definition for <form>.

1. Form Structure

<form> :==
<Constant Declaration> <Formal Neuron>
<Configuration> <Control Neuron>

2. Constant Declaration

<Constant Declaration> :==
[**#DECLARE** (<constant name>, <constant expression>);]*

3. Formal Neuron Declaration

<Formal Neuron> :==
[<Neuron Type> <Input> <Output>
<Internal Parameter><Initial Value><Internal Function>]+

<Neuron Type> :==
NEURON TYPE : <Neuron Type Name>
[<Neuron Definition Parameter List>] ;

<Input> :==
INPUT : [<Input Label>[<Range>]<Type>;]+

<Output> :==
OUTPUT : [<Output Label>[<Range>]<Type>;]+

<Internal Parameter> :==
INTERNAL PARAMETER : [<IP Label>[Range>]<Type>;]*

<Initial Value> :==
INITIAL VALUE : [$\left\{ \begin{matrix} \text{<IP label>} \\ \text{<Output label>} \end{matrix} \right\}$[<Time>] = <Constant expression>;]*

<Internal Function> :==

INTERNAL FUNCTION :[{ <IP Label> / <Output Label> } [<Time>] = <expression>;]*

[<Phase label> { [{ <IP Label> / <Output Label> } [Time>] = <expression> ;]* }]*

4. Configuration Declaration

<Configuration> :==

<Neuron Label> <Neuron Character><Connection Pattern>

<Neuron Label> :==

NEURON LABEL :

[<Indexed Neuron Label>]+

<Neuron Character> :=

NEURON CHARACTER :

[<Indexed Neuron Character>]+

<Connection Pattern> :==

CONNECTION PATTERN :

[<Indexed Connection Pattern>]+

5. Control Neuron

<Control> :== <Global Input><Global Output><Global Parameter>
<Global Initial Value><Input File><Output File><Global Function>

<Global Input> :==

GLOBAL INPUT : [<GInput Label>[<Range>]<Type>;]*[<ORecord label> ;]*

<Global Output> :==

GLOBAL OUTPUT : [<GOutput Label>[<Range>]<Type>;]*
[<IRecord label> ;]*

<Global Parameter> :==

GLOBAL PARAMETER : [<GIP Label>[<Range>]<Type>;]*

<Global Initial Value> :==

GLOBAL INITIAL VALUE :

[{ <GIP label> / <GOutput label> } [<Time>] = <Constant expression>;]*

<Input File> :==

INPUT FILE : [<Phase label> { **FILE** <File name>, **RECORD** <IRecord label> [<Range>]
<Type> [, <IRecord label> [<Range>] <Type>]* ;]+

<Output File > :==

OUTPUT FILE : [<Phase label> { **FILE** <File name>,

RECORD <ORecord label> [<Range>] <Type>
[, <ORecord label> [<Range>] <Type>]* ;]+

<Global Function> :==

GLOBAL FUNCTION :[{ <GIP Label> / <GOutput Label> }
[<Time>] = <expression> ;]* **PHASE**[T] = <Transition Condition>;

6. Supplementary Definition

<Input Label>, <Output Label>, <IP Label>, <Phase Label>, <N Label>, <GInput Label>, <GOutput Label>, <GIP Label>, <IRecord Label>,
<ORecord label> :==
{ <name> / (<name> [, <name>]*) }

<name> :==
<id> [<Index Definition> [, <Index Definition>]*]

<Range> :==
RANGE (<Constant Expression>, <Constant Expression>) { **INCLUSIVE** / **EXCLUSIVE** }

<Type> :==
{ **INTEGRAL** / **BINARY** / **CONTINUOUS** / **DISCRETE STEPSIZE** <Constant expression> / **VALUE** (<Constant Expression>[, <Constant Expression>]+) }

<Expression> :==
{ <Conditional expression> / <Simple expression> }

<Condition Expression> :==
IF <Condition> **THEN** <Expression> [**ELSE** <Expression>]

<Simple Expression> :==
{ (<Simple Expression> [<operator><Simple Expression>]*) / <Constant expression> / <name> }

<Indexed Neuron Label> :==
{ { <Indexed Neuron Label> [<Index Definition> ;]+ } / <Simple Neuron Label> }

<Index Definition> :==
{ <Index> : <lower bound >..<upper bound> / <Index> = <Constant Expression> }

<Simple Neuron Label> :==

<Neuron Label> [<Range Declaration> [, <Range Declaration>]*]

<Range Declaration> :==

{ <lower bound> .. <upper bound> | <Constant Expression> | <Index> }

<Indexed Neuron Character> :==

{ <Simple Neuron Character> | { <Indexed Neuron Character> [<Index Definition> ;]+ } }

<Simple Neuron Character> :==

[<Simple Neuron Label> **TYPE** <Neuron Type Name>
INPUT-DEGREE <Constant expression>
OUTPUT-DEGREE <Constant expression>]*

<Indexed Connection Pattern> :==

{ { <Indexed Connection Pattern> [<Index Definition> ;]+ } | <Simple Connection Pattern> }

<Simple Connection Pattern> :==

<Simple Neuron Label>.<Input Label> =
<Simple Neuron Label>.<Output Label>

<Constant Expression> :==

{ <Index> | <Constant> } [<operator> { <Index> | <Constant> }]*

II. ALGORITHMS

These are the algorithms for the dataflow analysis. Some in-line comments, as well as brief description of every procedure, are also given. The algorithms are shown in some pseudo code which is Pascal-like.

```
(*  This is the main procedure for dataflow analysis. It will call upon a number of other procedures for
    performing the functions. It will accept the three different forms and process them in the order:
    formal neurons, control neuron and then configuration. For the first two forms, it will perform
    checkings on their internal data dependency. If no errors are found, the obtained graphs will be used
    with the information from the configuration form to obtain the global dependency graphs. Three
    main objects NeuList, Cf and CN are the list for neurons, configuration and control neuron. They
    contain all the relevant information about these objects, including the different graphs. *)
Procedure DataflowAnalysis( FnForm, CfForm, CnForm : Form );
    Begin
    Normalize( FnForm );                                    (* normalize time indices *)
    Normalize( CfForm );
    Normalize( CnForm );
    NeuList := ConstructDG( FnForm );                       (* add IDGs and CIDG *)
    OK := false;
```

```
For every Neu in NeuList
    If OK then
        If CIDGAnalysis( Neu ) <> T then OK := false;        (*preliminary analysis, CIDG*)
    If OK then
        IF IDGAnalysis( Neu ) <> T then OK := false;         (*preliminary analysis, IDG *)
If OK then
    For every Neu in NeuList do
        CList := CycleDetection( Neu.IDGList );              (* cycle list of IDG *)
        If CycleAnalysis( CList ) <> T then OK := false;     (* pass the cycle tests *)
        If OK then
            TNeu := NParallelAnalysis( Neu );                (*increase parallelism*)
            NSchedule( TNeu );                               (* schedule the equations *)

If OK then
    CN := ConstructCNG( CnForm );                            (* add IDG to CN *)
    CnAnalysis( CN );                                        (* preliminary analysis *)
    CList := CycleDetection( CN.IDG );
    OK := CycleAnalysis( CList );
    If OK then
        TCn := CnParallelAnalysis( CN );
        CnSchedule( TCn );
If OK then
    GDGList := ConstructGDG( CfForm, NeuList, CN );          (* global dependency graph *)
    For every graph GDG in GDGList do
        CList := CycleDetection( GDG );                      (* cycle list of GDG *)
        OK := CycleAnalysis( CList );
End;
```

```
(*  Normalize the time subscripts of the equations.  It can be applied at all three kinds of forms. *)
Procedure Normalize( F : Form);
    Begin
    For every equation (Eq) in F
        Let L & Ri's be the depending and determining parameters of Eq;
        Let T+C be the time index of the innermost expression;
        For all Ri in Eq
            Let S be the time index of the innermost expression;
            Replace S with S-C;
    End;
```

```
(*  This procedure will construct one dependency graph.  It accepts a list of equations and use them to
    construct the corresponding dependency graph.  It uses two sub-procedures, one for the determining
    parameters (ConstructFromEdge) and one for the depending parameter (ConstructToEdge).  Both are
    recursive procedures so that sub-linear time indices can also be handled.  We should note that for
    parameters depending on constants only, no dependency edge is inserted.  This is to simplify the
    following analysis and facilitate the scheduling process.  Moreover, the procedure will also label the
    equations such that when scheduling, it is possible to distinguish all equations.  Note that the
    common equations for different phases will be duplicated and marked with different labels.  *)

Function OneGraph( EqList: EquationList ) : Graph;

    Procedure ConstructFromEdge( PE : ParameterExpression; Eq : Equation, G : Graph );
    Begin
```

```
        Let P be the outmost parameter name of PE;                (* determining parameters *)
        Construct one edge (E) pointing from P to Eq;
        If P has sub-linear time index then                       (* for records only *)
            Set time index value of E to infinity
        Else
            Let T+C be the time index of P in Eq;                 (* time index value *)
            Set C to be the time index value of E;
        associate E with other subscript boundaries of P in Eq;   (* other subscripts *)
        If E not in G then Append E to G;
        For every sub-linear expression (S) within P              (* explore inner parameters *)
            ConstructEdge( S, Eq, G );
    End;

    Procedure ConstructToEdge( LE : ParameterExpression; Eq : Equation, G : Graph );
    Begin
        Let L be the outmost parameter of LE;                     (* depending parameters *)
        Construct one edge (E) pointing from Eq to L;
        If L has sub-linear time index then                       (* for records only *)
            Set time index value of E to infinity
        Associate E with other subscript boundaries of L in Eq;   (* other subscripts *)
        If E not in G then Append E to G;
        For every sub-linear expression (S) within LE             (* explore inner parameters *)
            ConstructToEdge( S, Eq, G );
    End;

Begin
    G := nil;
    For every equation (Eq) in EqList
        Give a unique label to Eq;                                (* for scheduling *)
        If right hand side is not all constants then
            For every parameter expression (PE) at right hand side of Eq
                ConstructFromEdge( PE, Eq, G );
            Let LE be the left-hand-side parameter expression of Eq;
            ConstructToEdge( LE, Eq, G );
    Return( G );
End;
```

```
(*  This function is used for constructing the dependency graphs (IDGs, CIDG) of the neurons.  It will
    prepare an object Neu for every neuron type, keep the associated information such as the list of
    parameters, the list of equations which are common and unique to all phases, and the list of phases.
    It will also keep the IDG list and CIDG.  It will call OneGraph to construct individual graphs*)

Function ConstructDG( F : Form ) : NeuTypeList;
Begin
    NList := nil;
    For every neuron type (NType)                                 (* for every neurons types *)
        Append NType to NList;
        Add all parameters of NType in F to NType.PList;          (* parameter list *)
        Add all common equations in F to NType.CommonEqList;      (* common equation list *)
        Add all phases in F to NType.PhaseList;                   (* phases list *)
        For every phase (Ph) in NType.PhaseList                   (* construct IDGs *)
            NType.PhaseEqList := NType.CommonEqlist;              (* copy common equations. *)
            For every equation (Eq) of phase Ph of NType in F
```

```
                Append Eq to NType.PhaseEqList;
            Append OneGraph( NType.PhaseEqList ) to NType.IDGList;
        EqList := nil;                                          (* construct CIDG *)
        For every Phase (Ph) in NType.PhaseList                 (* combine eqn's from all phases *)
            Add NType.PhaseEqList to EqList;
        Add OneGraph( EqList ) to NType.CIDG;
    Return( NList );
End;
```

```
(* A small function to backtrack the dependent graphs.  Called by CIDGAnalysis and CnAnalysis. *)
Procedure BackTrack( P : Parameter; G : Graph );
Begin
    mark P;
    For every predecessor (D) of P do BackTrack(D, G);
End;
```

```
(*  This function use the CIDG to enforce some properties of the parameters, such as the number of input
    and output parameters, parameters should be used in generating outputs and the same.  It uses a
    simple recursive procedure, BackTrack, which backtracks from the output parameters to mark all
    other ones which determine them.  This is to eliminate useless parameters. Note that the deadlock
    analysis is not performed in this function.  *)

Function CIDGAnalysis( Neu : NeuType ) : Boolean;
Begin
    OK := true;                                                 (* set to false by
    ShowError *)
    If number of input parameter < 1 then
        ShowError( "No input into this neuron" );
    If number of output parameter < 1 then
        ShowError( "No output from this neuron" );
    For every parameter (P) in G.Type.PList
        If P not found in Neu.CIDG then
            ShowError( "Parameter P is not used in the equations" );
        If P has no out edge in Neu.CIDG & not an output parameter then
            ShowWarning( "Value of parameter P is not used by others" );
        Mark the edges pointing at P;
    If there are unmarked edges then
        For every parameter (P) at the head of the edges in Neu.CIDG
            ShowError( "Undefined parameter P" );
    Un-mark all parameters
    For every output parameter P do
        BackTrack( P, Neu.CIDG );                               (* are the parameters useful ?*)
    For every unmarked parameter U do
        ShowError( "Parameter P is useless for generating signals" );
    Return( OK );
End;
```

```
(*  This function is similar to CIDGAnalysis, except that the target is the IDGs.  *)
Function IDGAnalysis( Neu : NeuType ) : Boolean;
Begin
    OK := true;                                                 (* set to false by
    ShowError *)
```

```
    For every Graph G in Neu.IDGList do
        For every parameter P in G do
            If P has incoming edge & P is an input parameter then
                ShowError( "Input parameter P cannot be updated" );
            If P does not have in edge & P is not an input parameter then
                ShowError( "Parameter P is not updated" );
            If P has more than one incoming edge then
                If P is not subscripted then
                    ShowError( "Multiple assignment into P" )
                Else if subscript boundaries overlaps then
                    ShowError( "Part of P has multiple assignment" )
                Else if subscript boundaries and initialization values do not cover all elements of P then
                    ShowError( "Some of the elements of P are not defined" );
    Return( OK );
End;
```

```
(*  This function is used for locating cycles in the graphs. It uses depth first search technique. A recursive procedure known as CheckCycle is called upon. It uses exhaustive search to detect the cycles. The parameters are given an arbitrary order. The system tries to construct paths as long as possible. In these paths, every parameter in the path should be later in order than its predecessor in the graph. When it is not possible to extend the path further, the system tests whether that a cycle is found connecting the first and last parameters. This process is repeated for every parameter in order. Hence after all the parameters are processed, all the cycles are found. *)

Procedure CheckGlobalCycle( GList : GraphList );

    Procedure CheckCycle( PList : ParameterList, P );
    Begin
        Append P to PList;                                  (* update PList *)
        for every parameter Q with order >= P do
            if there is a edge from P to Q then
                CheckCycle( PList, Q );                     (* recursive call *)
        if P is connected to first element in PList then    (* a cycle is found *)
            show cycle in PList;
        Remove P from PList;
    End;

Begin
    PList := nil;                                           (* parameters in the path *)
    assign arbitrary numbers for parameters in the global dependency graph;
    for every phase do
        for every parameter P in GList do
            PList := nil;
            CheckCycle( PList, P );
End;
```

```
(*  A small procedure to determine the depth of the parameters from input ones. Called by NParallelAnalysis and CnParallelAnalysis. *)
Procedure Depth( P : parameter; d : integer; G : Graph );
Begin
    If P is not marked as traversed then                    (* prevent cycles *)
        if P has depth < d then Label P with d;             (* greater depth *)
```

```
        Mark P as traversed;
        For every depending parameter (Q) of P in G do          (* depth first search *)
            Depth( Q, d+1, G );
        Un-mark P;
End;
```

```
(*  This procedure is used for analyzing cycles. It can determine whether or not the cycles are involving
    in deadlocks and multiple assignments. It first perform the cycle sum test which aims at analyzing
    the time subscripts. When the test reports problem, it will further investigate the subscripts to see if
    there are really problems. *)

Function CycleAnalysis( CList : CycleList ) : Boolean;
Begin
    OK := true;                                                  (* set to false by show
    error *)
    For every cycle (C) in CList
        For every parameter (P) in C                             (* future reference *)
            IF time index from P > 0
                ShowError( "Future dependency for P" );
        S := sum of time indices of C;                           (* cycle sum test *)
        If S < 0 then                                                    (* normal case *)
            For every parameter (P) in C
                Let -D be the time index from P;                 (* check initialization *)
                If number of initialization values for P < D then
                    ShowError( "Not enough initialization values for P" );
        Else If S = 0                                            (* mutual dependency ? *)
            If there is no subscript then                        (* no subscript, sure in deadlock *)
                ShowError( "Deadlock for cycle C" )
            Else For every subscript dimension (S) of P do
                If there is only 1 parameter in C then
                    If subscripts both increase or decrease and overlap incorrectly then
                        ShowError( "Future reference in C" )
                    Else one subscript increases and one subscript decreases and overlap then
                        ShowError( "Multiple assignment into some elements" )
                Else if subscript overlap in both edges of any parameter
                    ShowError( "Deadlock in cycle C");
    Return( OK );
End;
```

```
(* This procedure will determine how group the neurons and split the graphs.  It is called by NParallel-
    Analysis and CnParallelAnalysis.  *)
Procedure Split( G : Graph; GrList : GroupList; GList : GraphList );
Begin
    For very parameter Q in G with depth = 1 do
        Prepare a new group Gr and a new graph GNew, add Gr to GrList, GNew to GList;
        For every determining parameter R do                     (* they are input parameters *)
            Duplicate R, add R to Gr, edge R->Q to GNew;
            Duplicate the connection for R from other neurons;   (* refer to information in CP *)
    For d := 1 to maximum depth value  do
        For every parameter S in G with depth value = d          (* parallel candidates *)
            Let Gr be the group S in and GNew be the corresponding graph;
```

```
            If S is determining only 1 another parameter T
                with depth = d+1 then
                If T is depending on S only
                    add T to Gr and edge S->T to GNew          (* in the same group *)
                Else
                    If T is not allocated to any group then    (* separate a new group *)
                        Prepare new group Gr1 and new graph GNew1 for T;
                        Add Gr1 to GrList, GNew to GList;
                    Add edge S->T between Gr and the group T belongs to;
            Else
                For every depending parameter T of S, with T<> S do    (* different groups *)
                    If T is not allocated to any group then
                        Prepare new group and new graph for T;
                        Add group to GrList, graph to GList;
                    Add S->T as edge between Gr and the group T belongs to;
End;
```

```
(*  This function analyze the CIDG of a neuron type and suggest allocations of parameter evaluation
    which can increase the degree of parallelism. The allocation is based on a depth from the input
    parameters. Parameter with the same depth can be evaluated in parallel. The depth is computed by
    calling the procedure Depth. The resultant allocations are kept in GrList. It then return the modified
    neuron type for scheduling. *)
Function NParallelAnalysis( Neu : NeuType ) : NeuType;
Begin
    N := Neu;
    For every input parameter P of N do Depth( P, 0, N.CIDG );    (* depth from input parameters *)
    Set GrList and GList to Nil;                                  (* every group is a sub-neuron *)
    Split( N.CIDG, GrList, GList );
    Break every element of N.IDGList according to the grouping in GrList;    (* update IDGs *)
    Return( N );
End;
```

```
(*  This procedure accepts an equation and try to evaluate it. If any determining parameter is not yet
    available, it will try to evaluate that parameter first by calling itself recursively. Any equation whose
    parameter is all available is inserted into the schedule. It will be called by NSchedule and
    CnSchedule. AvailList is a list maintaining the already available parameters. *)

Procedure EvaluateEquation( Eq : Equation; G : Graph; T : Time;
                            AvaliList : ParameterList; Sch : Schedule );
Begin
    Mark Eq with T;
    Let D be the depending parameter of Eq;
    For all P predecessor P of Eq in G do                         (* can they be evaluated ?*)
        S := T + weight of edge from P to Eq;                     (* the corresponding instance *)
        If not P[S] in AvailList then                             (* not available, try to evaluate *)
            if name of P = name of D then                         (* self-referenced arrays *)
                Let LbP, UbP, LbD, UbD be lower & upper bounds of P and D;
                If both increasing subscripts then
                    For every equation Eq1 determining any element in D[LbP..LbD-1]
                        & not initialized do                      (* the part to be evaluated first *)
                        EvaluateEquation( Eq1, T, AvailList, Sch )
                Else if both decreasing subscripts then
```

```
                    For every equation Eq1 which determines any element
                        in D[UbP..LbD-1] & not initialized do      (* the part to be evaluated first *)
                        EvaluateEquation( Eq1, T, AvailList, Sch )
                Else
                    For every equation Eq1 which determines any element
                        in P[LbP..UbP]  & not initialized do       (* the part to be evaluated first *)
                        EvaluateEquation( Eq1, T, AvailList, Sch )
            Else
                Let EqP be the equation for evaluating P in G;
                EvaluateEquation( EqP, G, T, AvailList, Sch );
    Add D[T] into AvailList;
    Write the equation associated with Eq into Sch;
End;
```

```
(*  This procedure is responsible for scheduling the evaluation of the parameters. The scheduling is
    based on the idea that a parameter can be evaluated only if its determining parameters are available.
    A list AvaliList records the parameters available so far. The initially available parameters are those
    initialized ones. This procedure will call another procedure known as EvaluateEquation which
    accepts an equation and tries to evaluate it. *)

Procedure NSchedule( Neu : NeuType );
Begin
    For every graph G in Neu.IDGList do
        Add initialized instances of parameters and input ones to AvailList;
        Repeat until all equations in corresponding phase
            of Neu.PhaseEqList are marked
            Select one unmarked equation with the following criterion,
                (i) If an equation depends only on input parameters, select it
                (ii) If an eqn depends partly on input parameters, select it
                (iii) any one equation
                Name this equation as Eq;
            EvaluateEquation( Eq, G, T, AvailList, Sch );
End;
```

```
(* This is the algorithm for constructing the internal dependency graph of control neuron. As this neuron
    does not have operation phases, there is only one IDG for it.        *)

Function ConstructCNG( F : Form ) : CNType;
Begin
    new( CN );                                                    (* new object *)
    Add all global parameters into CN.PList;
    Add all global functions into CN.EqList;
    CN.IDG := OneGraph( CN.EqList );
    Return( CN );
End;
```

```
(*  Function for analyzing the parameter properties of the control neuron. As it has only one IDG, the
    test is in fact a combination of IDGAnalysis and CIDGAnalysis.   *)

Function CnAnalysis( CN : CnType ) : Boolean;
Begin
    OK := true;   (* set to false by ShowError *)
```

```
    If number of input parameter < 1 then
        ShowWarning( "No input into control neuron" );
    If number of output parameter < 1 then
        ShowWarning( "No output from control neuron" );
    Add Phase as output parameter;
    For every parameter (P) in CN.PList
        If P not found in CN.IDG then
            ShowError( "Parameter P is not used in the equations" );
        If P have no out edge in CN.IDG & not an output parameter then
            ShowWarning( "Value of parameter P is not used by others" );
        If P is an output record parameter and P has outgoing edge then
            ShowError( "File record parameter P cannot be referenced");
        If P has incoming edge & P is an input parameter then
            ShowError( "Input parameter P cannot be updated" );
        If P have no incoming edge & P is not an input parameter then
            ShowError( "Parameter P is not updated" );
        If P has more than one incoming edge then
            If P is not subscripted then
                ShowError( "Multiple assignment into P" )
            Else if subscript boundaries overlaps then
                ShowError( "Part of P has multiple assignment" )
            Else if union of subscript boundaries and initialization values
                do not cover all elements of P then
                ShowError( "Some of the elements of P are not defined" );
        Mark the edges pointing at P;
    If none of the value of Phase is "terminate" then
        ShowError( "No terminating condition" );
    If there are unmarked edges then
        For every parameter (P) at the head of the edges in CN.IDG
            ShowError( "Undefined parameter P" );
    un-mark all parameters;
    For every output parameter P do
        BackTrack( P, CN.IDG );
    For every unmarked parameter U do
        ShowError( "Parameter P is useless in control neuron" );
    Return( OK );
End;
```

```
(*  This function analyze the IDG of the control neuron to increase parallelism.  It is similar to the func-
    tion NParallelAnalysis.    *)
Function CnParallelAnalysis( InCN : CnType ) : CnType;
Begin
    CN := InCN;
    For every input parameter P of CN do
        Depth( P, 0, CN.IDG );                          (* depth from input parameters *)
    Set GrList and GList to Nil;                        (* every member is a sub-neuron *)
    Split( CN.IDG, GrList, GList );
    Return( CN );
End;
```

```
(*  This procedure is responsible for scheduling the evaluation of the parameters for the control neuron.
    It is similar to NSchedule. *)
Procedure CnSchedule( CN : CnType );
Begin
    Add all initialized instances of parameters of CN.PList and all input parameters to AvailList;
    Repeat until all equations in CN.EqList are marked
        Select one unmarked equation with the following criterion,
            (i) If an equation depends only on input parameters, select it
            (ii) If an equation depends partly on input parameters, select it
            (iii) any one equation
            Name this equation as Eq;
        EvaluateEquation( Eq, CN.IDG, T, AvailList, Sch, );
End;
```

```
(*  Function for constructing the global dependency graph GDG of the whole network through informa-
    tion in the connection pattern entry. It has two important assumptions. 1) To eliminate the number
    of neurons to be generated, the connection patterns should be symmetric. In case an equation is
    defining a number of neurons, it always take the one with the smallest value in any free index
    dimension. Moreover, if the parameters are repeating for a number of times, only the first one is
    selected. For networks which has not dependency within layers or which the dependency within layer
    is symmetry, this procedure will generate just the number of layers of neurons.
    2) To correctly detect deadlocks, distinct parameters should be used for distinct purpose. As the
    algorithm only select one element from among an array, it will overlook some types of connections if
    other elements are connected for a different purpose. *)

Function ConstructGDG( CfForm : Form; NL : NeuList; CN : CNType ) : GraphList;
Begin
    G := Nil;
    Let N be the first in LHS of first connection pattern equation;
    Add N to AvailNeu and IncompleteG;        (* available neurons and neurons not fully connected *)
    Mark every I/O parameter of N as not connected;
    Repeat until IncompleteG is empty
        Select any M from IncompleteG;
        For every not connected I/O parameter P of M do          (* connect every I/O parameter *)
            Locate the connection pattern equation CP which
                defines/reference P of M;
            Fix the subscripts of CP with that of P and M;
            If there is a K in AvailNeu satisfying requirement of CP then
                Fix the subscripts
            Else
                Use the smallest neuron that satisfies the requirement of CP;
                Add K to NeuG;
                Add K to IncompleteG;                            (* wait for connection *)
            Let Q be the I/O parameter of K in CP;
            If P is an input parameter then add edge K.Q -> M.P to G
            Else add edge M.P -> K.Q to G;
            Add corresponding time and other subscripts;
            mark M.P and K.Q connected;
            If all I/O parameter of K is connected then
                remove K from IncompleteG;
        Remove K from IncompleteG;
    Let N be any neuron type in AvailList;
    Let PhL be N.PhaseList;
```

```
    GList := nil;                                          (* add IDGs *)
    For every Ph in PhL do
        TempG := G;
        For every N in NL do
            Let IDG be the element of N.IDGList of phase Ph;
            Add IDG to TempG to connect the parameters within the neurons;
            Add CN.IDG to TempG to connect the parameters within CN;
            Add TempG to GList;
    Return( GList );
End;
```

III. Deadlock & Dependency Cycles

Deadlock is a problem resulted from concurrent allocation and request of resources. ANNs are usually implemented as message passing system [1, 3, 7, 11, 15, 20, 21]. In this way, the resources are the *messages* (or *communication signals*) among the neurons and the *buffers* for accommodating the messages. A request for messages occurs when a neuron tries to get some input values from other neurons, and an allocation occurs when a buffer is used to store an incoming message.

Deadlocks in any system can be analyzed by a *Resource Allocation Graph*. In an ANN systems, however, the resource allocation graphs can be simplified into *Data Dependency Graphs*. The deadlocked cycles in a resource allocation graph are captured by *mutual dependency cycles* in the data dependency cycles.

A. Deadlock Prevention

Three Approaches for Handling deadlock

The first one is deadlock *detection* and *recovery*. No action is taken to keep deadlock from occurring. Rather, system events may trigger the execution of a detection algorithm. When the group of deadlocked processes is identified, some of them must be terminated (or rolled back to an earlier state if checkpoint information is available) in order to break the deadlock.

The second approach is deadlock *avoidance*. This relies on some knowledge of future process behavior to constraint the pattern of resource allocation.

The last approach, which is also employed by the system, is deadlock *prevention*. The system design prevents entry into a state from which future deadlock is inevitable. To accomplish this, one should deny at least one of the necessary conditions for deadlocks.

1. Necessary Conditions for Deadlock

To prevent the occurrence of deadlocks, it is required to consider the *necessary* conditions for deadlocks. The four most generally identified necessary conditions are listed below [28].

Mutual Exclusion : Processes hold resources exclusively, making them unavailable to other processes.

Nonpreemption : Resources are not taken away from a process holding them; only processes can release resources they hold.

Resource Waiting : Processes that request unavailable units of resources block until they become available.

Partial Allocation : Processes may hold some resources when they request additional units of the same or other resources.

Unfortunately, these conditions are natural consequence of the policy for resource allocations. In other words, most practical system cannot deny the occurrence of these conditions.

On the other hand, when one apply some specific tools in deadlock analysis, such as *resource allocation graphs*, one can derive some additional necessary conditions for the deadlocks. It is possible for us to prevent them from being occurred.

2. Resource Allocation Graphs

A very useful model for deadlock analysis introduced in [28] is *resource allocation graphs*. There are two classes of resources with very different properties. *Reusable* resources are usually fixed in total inventory. Additional units are neither created nor destroyed. Units are requested and acquired by processes from a pool of available units and, after use, are returned to the pool for use by other processes. Examples of reusable resources are processors, I/O channels, memories, devices, busses and files.

On the other hand, *consumable* resources are not fixed in total number. Units may be created (produced or released) or acquired (consumed) by processes. An unblocked producer of the resource may release any number of units. These units then become immediately available to consumers of the resource. In addition, an acquired unit ceases to exist. Examples are interrupts and signals, messages and information in I/O buffers.

3. Cycles and Blocked Requests

Two additional necessary conditions, *cycles* and *blocked requests*, for deadlock can be deduced using the resource allocation graphs.

Cycles

The presence of a *cycle* in a resource allocation graph as the necessary condition for deadlock is proved in [28]. Intuitively, the absence of a cycle implies the existence of a linear ordering of process nodes following arcs from processes to resources to processes. The last process may thus proceed and then release the resources it allocated. This will in turn unblock the other process(Es). Hence at least one cycle should be formed in the allocation graphs for a deadlock to occur.

Blocked Requests

Blocked requests means that

> if there is a deadlock D occurring in a cycle C, every processes P_i involved must be *blocked* by requests to the resources *inside* C.

The blocking is straight forward as if at least one process is not blocked by the request, it will proceed and the cycle is therefore not involving in a deadlock (though it may involve in another one later).

In addition, if a process P_j in C is *not* blocked by any request to resources *inside* C, C is *not* a cycle. This is proved by observing that in a cycle, every node should have both incoming and outgoing edges. For process nodes, the outgoing edges are request edges. When there is no request edges from P_j to any resource in C, P_j has no outgoing edges and hence C is not a cycle.

Note that the case that the processes may *also* be blocked by other resources *outside* C is not excluded, but they should be blocked by some requests *inside* it.

The Implication

The two deduced necessary conditions suggest an approach for deadlock prevention.

> If for *every* resource allocation cycles, at any times, at least one of the request can always be granted, there will not be deadlocks.

The prevention mechanism focuses only on cycles because they are necessary for deadlocks. For processes not involving in any cycles, they are always in a safe state. On the other hand, all the cycles will *not* be involved in deadlocks, as the *blocked request* condition cannot be satisfied. Hence every process in the cycles will also not be deadlocked. Thus the system will always be in a safe state.

The Potential Cycles

Potential cycles are not necessarily resource allocation cycles at some time but they are candidates for being cycles. Potential cycles are used instead of the actual allocation cycles because cycles in resource allocation graphs are formed according to the dynamic system state and not suitable for static analysis. In the next section, the nature of these potential cycles in an ANN system are investigated.

B. Deadlock in ANN Systems

To analyze the deadlocks in an ANN system, it is necessary to identify the "processes" and "shared resources" in an ANN system. The processes in the resource allocation graphs resemble the basic processing unit of an ANN system, the neurons.

1. Shared resources

Two types of "shared" resources in an ANN system can be identified, the *messages* and the *buffers*. The term *share* does not implies that the resources can be used by more than one neurons but that more than one neurons are related to the resource. The other resources, such as processor and memory, are local to the neurons and hence will not be involved in deadlocks.

The *messages* are *consumable resources*. These messages are the communication signals among the neurons which carry the values of the input and output parameters. Whenever a neuron evaluates its output parameters, a set of messages are generated. It is thus the producer of the messages. Whenever a neuron references its input parameters, a set of messages are consumed.

2. Presence of the Necessary Conditions for Deadlocks

Mutual exclusion, *no preemption* and *partial allocation* are natural in ANNs. *Mutual exclusion* occurs as both the buffers and messages cannot be used by more than one neurons at the same time. No other neuron can preempt the resource from the holding neuron (*no preemption*). A neuron may request additional buffers for processing (*partial allocation*) even after it has allocated some.

The need for *synchronization* in ANNs introduces the *resource waiting* requirement. The synchronization requirement is seldom addressed by classical texts but they are essential for proper operations of the network systems. To get the most updated information, the neurons should wait for messages from other neurons if they are not available.

Cycles are common in an ANN system. The presence of *feedback message loops* (e.g. in BP-Net) and *central control* (in nearly every type of network) give arise communication cycles. These communication cycles are called *potential cycles* as they are candidates in which *resource allocation cycles* will occur. Synchronization communication with message cycles raise the problem of *blocked requests*.

3. Operation Constraint for Communication

Two constraints on the communication among the neurons are imposed to prevent deadlocks. They are 1) in every logical iteration, every neuron produces one set of output messages, and consumes one set of input messages; and 2) at least one empty buffer exists between every connected neuron pair.

Role of the Constraints

Intuitively, the first constraint prevents using up the buffers among connected neuron pairs. An initial buffer of size one is enough. The second constraint, on the other hand, guarantees that at least one message can be generated initially. Without the initial empty buffer, every neuron may be blocked by an initial request to write.

Justification of the Constraints

The two constraints seem to be restrictive but they are natural. The first constraint guarantees that the neurons will make use of the most updated information from other neurons for generating their own output, and they will always give most updated information to others. The second constraint is straight forward.

Introducing These Constraints

These two constraints are beyond the control of the users at the specification level. They are automatically introduced by the implementation generator. It is therefore possible to guarantee the existence of these two constraints.

4. Checkings Required

Under the two given constraint mentioned, the only deadlocked case is that all neurons in a cycle request messages from each other before any one will produce a message. This is obvious as if there is no outstanding messages initially, all the neurons will be blocked.

It is therefore just necessary to check that within these cycles, at least one of the neurons can accomplish its task to generate messages used by others. For any neuron to be initially unblocked, it should depend on initial messages. The problem is then reduced to guarantee that sufficient number of initial messages are available for the required neurons.

C. Data Dependency Graphs

The resource allocation graphs are constructed during execution. They are dynamic in nature and changes with respect to the system state. A static analysis, anyway, is preferred to a dynamic one as it will save more resources.

In addition, it is inconvenient for us to use a resource allocation graph in testing whether or not a neuron in a cycle will depend on initial values. This is because the resource allocation graph focuses on the neuron level but the dependency relationship are at the *parameter* level.

The resultant proposal is to simplify the resource allocation graphs into *data dependency graphs*. Data dependency graph is a frozen, worst case representation of a resource allocation graph. The basic argument is that the message and buffer resources can be eliminated from the resource allocation graphs. The neurons are then "expanded" to explore the behaviour at the parameter level. Afterwards, the requesting edge are "frozen" in a reverse directions.

1. Simplifying Resource Allocation Graphs

After the previous discussion, the buffer resources will not involve in deadlocks under the given constraint on the relative number of messages generated and consumed. As the buffers will not be involved in deadlocks, the resource allocation graphs can be simplified by eliminating the *buffer resources* from the graphs.

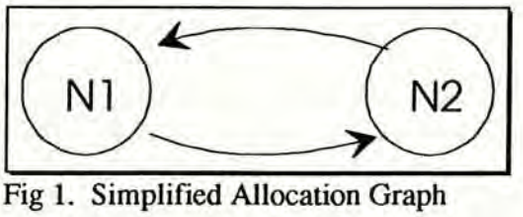


Fig 1. Simplified Allocation Graph

Now there are only one type of resource, the *messages*, in the resource allocation graph. This suggests that the *message resources* among the neurons can also be omitted, as they are understood. They can be replaced by a simple arrow pointing from the receiving side to the sending side as no other neurons may request the resources. Figure 1 shows a pair of neurons without the resources.

2. Expanding into Parameter Level

The graphs in Figure 1 is not suitable for determining whether or not the neurons will send before they reads. This is because the functioning of the neurons are determined at the parameter level. Every neuron has a number of *input*, *output* and *internal* parameters. A send operation is equivalent to generating the value for an output parameter, and a receive operation a reference to an input one. It is hence the behaviour of the parameters which determines whether there are deadlocks.

Consequently, it is necessary for us to "expand" the neurons and include the request among *all* the parameters. The communication among the input/output parameters should be constructed. These parameters, however, will request values from other parameters within or outside the neurons. This can be viewed as a form of "local" request. A neuron can only generate before consume messages if its output parameter, say P, is ready. On the other hand, this output parameter P is ready only when all parameters P is depending on is ready to "send" their value to P. Hence the request among all the parameters should be included to gain a complete resource allocation graph.

Figure 2 shows an instance of cyclic request among the neuron parameters for two sample neurons. The P_i's and Q_j's are the parameters of two different neurons and they form a request cycle. If this cycle is blocked, the parameters, and hence these two neurons, are deadlocked. The behaviour of these parameters determines the result.

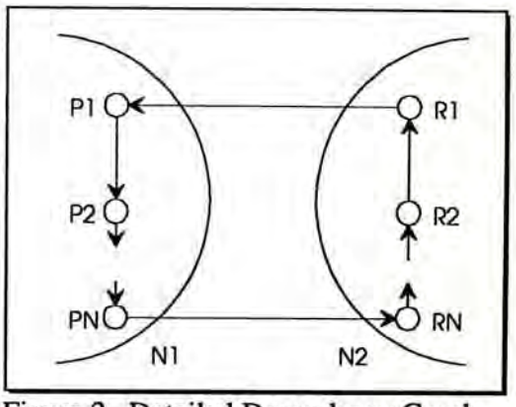


Figure 2. Detailed Dependency Graph

3. Freezing the Request Edges

The resource allocation graphs are dynamic in nature and not suitable for static analysis. It is necessary to introduce a static representation for the resource allocation graph and yet retaining the essential property.

Deadlocks occurs when there is a blocked cyclic request for *messages*. At the parameter level, blocked by cyclic request for messages is equivalent to blocked by cyclic request for *values of parameters*. So the requesting edges among the neurons is replaced by those among the parameters.

The requesting edges among parameters may appear and disappear occasionally, so cycles may not always form. To convert the dynamic resource allocation graph into a static one, one can consider the *worst case* in which the edges *always* exists. This "freezes" the dynamic edges into a static one and they can be construct before execution. Thus a static graph is obtained. If for this worst case graphs, *no* cyclic request will be totally blocked, at least one of the requesting edges in the resource allocation graph will disappear. This proves the absence of deadlock. Figure 2 is an example of this type of graph.

4. Reversing the Edge Directions

This worst case graph is nearly identical to *data dependency graphs*, except that the edges are in reverse directions. An edge pointing from P to Q in the worst case graph implies that P requests values from Q, while an edge from Q to P implies that Q determines the value of P. They are equivalent in meaning when P is assumed to "always" request values from Q (i.e. frozen in the worst case). As the direction of

the edges will not change the result of the previous proof, if they are kept consistently reversed, data dependency graphs instead can be dealt with.

5. Mutual Dependency Cycles

The deadlocked cycles in a resource allocation graph are mapped into cycles in the data dependency graphs. Not every cycle in data dependency graph is deadlocked. The cycles are deadlocked provided that the parameters are mutually depending on each other. In this way, it is not possible to evaluate them and hence they are deadlocked.

On the other hand, if there is always a proper order of evaluation for the parameters, the cycles will be deadlocked free. This is possible only if at least one of the parameters is depending on some initial values. In this way, that particular parameter can be evaluated first. In the resource allocation view, this is equivalent to guarantee that at least one of the producer can generate the initial message by referring to some outstanding messages.

It can be safely conclude now that for every cycles in a data dependency graph of a ANN, if at least one of the parameter can be evaluated before referring to other parameters, there will not be deadlock for this system.

IV. CASE STUDIES

We are now going to illustrate the application of the whole analysis process with the example networks. The full results are given for every analysis procedure if they are not given before. The first one is the BP-Net, with most result given in the main content. Two other examples are *Perceptrons* and *Boltzmann machine*.

A. BP-NET

This is a sample specification on back-propagation network. The network contains 5 layers, with 256 bit input pattern size. Moreover, the number of neurons in the first layer is equal to input pattern size, with the number decrease by one half in every succeeding layer.

1. Specification Forms

There are totally 4 forms, one for constant declarations, one for **formal neuron**, one **configuration** and one **control neuron**. Among the forms for **formal neuron**, only those for *BPInput* and *BPOut* are shown, which represent the input and output neuron types respectively.

a) Constant Declaration

Discussed before and is not shown here.

b) Formal Neuron

***** FORMAL NEURON *****

```
NEURON TYPE :
    BPInput( M, N, Id );
/*  M -- number of neuron in previous layer,
    N -- number of neuron in next layer,
    Id -- position of current neuron in the layer. */
INPUT :
    I [ 1..M ] BINARY;
    ( WI [ 1..N ], DeltaI [ 1..N] ) CONTINUOUS;
```

```
OUTPUT :
    Out RANGE (0, 1) INCLUSIVE CONTINUOUS;
INTERNAL PARAMETER :
    (A, W[1..M] ) CONTINUOUS;
    (DeltaW[1..M], Delta) CONTINUOUS;
INITIAL VALUE :
    A[0] = RANDOM; W[*][0] = RANDOM; DelatW[*][0] = 0;
INTERNAL FUNCTION :
    A[T] = I[T] * TRANSPOSE( W[T] ) + Theta;
    Out[T] = 1 / ( 1 + EXP( - A[T] ) );
    Training {
        Delta[T] = Out[T] * (1 - Out[T]) * (DeltaI[T] * TRANSPOSE( WI[T] ));
        DeltaW[*][T] = Neta * Delta[T] * Out[T] + Alpha * DeltaW[*][T-1];
        W[T+1] = W[T] + DeltaW[T];                                          };
    Recalling {
        DeltaW[T] = 0;
        Delta[T] = 0;
        W[T+1] = W[T];                                                      };


NEURON TYPE :
    BPOut( M, Id );
/*  M -- number of neuron in previous layer,
    Id -- position of current neuron in the layer. */
INPUT :
    ( I [ 1..M ], Teach ) RANGE (0, 1) INCLUSIVE CONTINUOUS;
OUTPUT :
    Out RANGE (0, 1) INCLUSIVE CONTINUOUS;
    (W [ 1..M ], Delta) CONTINUOUS;
INTERNAL PARAMETER :
    (A, DeltaW[1..M]) CONTINUOUS;
INITIAL VALUE :
    W[*][0] = RANDOM; DeltaW[*][T] = 0;
INTERNAL FUNCTION :
    A[T] = I[T] * TRANSPOSE( W[T] ) + Theta;
    Out[T] = 1 / ( 1 + EXP( - A[T] ) );
    Training {
        Delta[T] = (Teach[T] - Out[T]) * Out[T] * (1 - Out[T]);
        DeltaW[T] = Neta * Delta[T] * Out[T] + Alpha * DeltaW[T-1];
        W[T+1] = W[T] + DeltaW[T];                                          };
    Recalling {
        DeltaW[T] = 0;
        Delta[T] = 0;
        W[T+1] = W[T];                                                      };
```

The specification of *BPInput* is very similar to that of *BPHidden*. The **Internal Function** part is in fact identical, i.e. they are functioning in the same way. They are different only in the different way of classifying the parameters *W* and *Delta*. In *BPHidden*, they are classified as output parameters but not in *BPInput*.

On the other hand, the output and internal parameters of *BPOut* is identical to that of *BPHidden*. They have, however, different input parameters and functions. *BPOut* uses a teaching input *teach* from the en-

vironment to determines its error correction signal *Delta* while *BPHidden* computes this value by corrections signals *Deltal* from the following layers.

c) Configuration

Discussed before and is not shown here.

d) Control Neuron

Discussed before and is not shown here.

2. Results After Simple Checkings

No special error is reported from simple checkings.

3. Internal Dependency Graphs Construction

Discussed before and is not shown here.

a) Internal Dependency Graphs

Discussed before and is not shown here.

b) Control Neuron Internal Dependency Graph

Discussed before and is not shown here.

c) Combined Internal Dependency Graph

Discussed before and is not shown here.

4. Results From Parameter Analysis

Nothing special reported from parameter analysis.

5. Global Dependency Graphs Construction

Discussed before and is not shown here.

6. Cycles Detection

Discussed before and is not shown here.

7. Time Subscript Analysis

All cycles can pass the time subscript analysis. In practice, the time subscript analysis is applied simultaneously with the cycle detection process. Cycles with negative time offsets are not explored at all. This will reduce the time and memory requirement of the system. For our example, the list of cycles will not be generated at all and the system will only report that nothing goes wrong.

8. Subscript Analysis

No need to perform the subscript analysis as all the parameters can pass the time subscript analysis.

9. Scheduling

Discussed before and is not shown here.

B. PERCEPTRON

This is a simple single layer perceptron. The input pattern is 1024 Bi-state values known as *predicates*, and there are also 1024 neurons for learning these values. Every neuron is connected to 1/4 of the inputs, with every neuron accepting input from a continuous pattern one position lower than the previous one in a round robin manner.

1. Specification Forms

The four forms for the perceptron are quite simple. This is mainly because there is only one type of neuron defined, the *Perceptron*.

a) Constant Declaration

```
*** CONSTANT DECLARATION ***

#DECLARE ( NumOfNeu, 1024 );
#DECLARE ( InSize, NumOfNeu / 4);
#DECLARE ( MaxIter, 1000 );
#DECLARE ( Neta, 0.1 );
```

There are only four constants defined for the perceptron specification. Their meanings are defined in the following table.

NumOfNeu	Number of neuron in the layer
InSize	Input size for every neuron
MaxIter	Maximum number of iteration for each pattern
Neta	Learning rate (η), a system constant

b) Formal Neuron

```
*** FORMAL NEURON ***

NEURON TYPE :
    Perceptron;
INPUT :
    I [1..InSize] VALUE (-1, 1);
OUTPUT :
    Out VALUE (-1, 1);
INTERNAL PARAMETER :
    (W[1..InSize], A, E) RANGE (-1, 1) INCLUSIVE CONTINUOUS;
INITIAL VALUE :
    W[*][0] = RANDOM; E[*][0] = RANDOM;
INTERNAL FUNCTION :
    A[T] = I[T] * TRANSPOSE( W[T] );
    Out[T] = IF A[T] > 0 THEN 1 ELSE -1;
    E[T] = I[1][T] - OUT[T];
    Training { W[T] = W[T-1] + Neta * E[T-1] * I[T-1]; };
    Recalling { W[T+1] = W[T]; };
```

The input and output parameters are defined as accepting values -1 and 1 only. This is a Bi-state value known as *predicate*. This definition is accomplished by using the **VALUE** clause.

The supervised learning is accomplished by the equation for the error value, *E[T]*. The value of *E[T]* is depending on the difference between the first input value and the output value. By the specification in the configuration, the first input into the i-th neuron is the value of the input pattern at the i-th position. This means that *I[1]* for every neuron is the value of the input at the corresponding position of that neuron. This value is also the expected output of that neuron. Hence the value of *E[T]* is computed in this manner.

c) Configuration

```
*** CONFIGURATION ***

NEURON LABEL :
    N[1..NumOfNeu];
NEURON CHARACTER :
    N [1..NumOfNeu] TYPE Perceptron
        INPUT-DEGREE InSize OUTPUT-DEGREE 1;
CONNECTION PATTERN :
    {   N[y].I[x] = CN.Input-Pattern[( (y+x-2) mod NumOfNeu ) + 1];
        x : 1..InSize; y : 1..NumOfNeu;     }
    {   CN.Output-Pattern[x] = N[x].Out;
        x : 1..NumOfNeu;                    }
```

The configuration is again very simple. The only tricky part is on feeding the input *Input-Pattern* from **control neuron** to the neurons. The specification defines that position
((y + x - 2) mod NumOfNeu) + 1
of the input pattern is feed into the *x-th* input of *y-th* neuron. This specifies a continuous range of input for every neuron. If the neuron is in the i-th position within the layer, the input for the neuron will start at the i-th position, stretching for *InSize* positions. This serves as an example for specifying partial connection.

The expression will always return the value of y when x is equal to 1, as $y \leq NumOfNeu$. The result is that the first input to every neuron is exactly the input at the same position of that neuron.

d) Control Neuron

```
*** CONTROL NEURON ***

GLOBAL INPUT:
    Output-Pattern[1..NumOfNeu] CONTINUOUS;
GLOBAL OUTPUT :
    Input-Pattern[1..NumOfNeu] CONTINUOUS;
GLOBAL PARAMETER :
    (Iteration, Curr-Pattern) INTEGRAL;
GLOBAL INITIAL VALUE :
    Iteration[0] = 1; Curr-Pattern[0] = 1; Phase[0] = Training;
INPUT FILE:
    (Training, Recalling) {
        FILE 'INPAT.DAT', RECORD Inpat [1..5] VALUE (-1, 1) ;          };
OUTPUT FILE:
    (Training, Recalling) {
        FILE 'OUTPAT.DAT', RECORD Outpat [1..5] VALUE (-1, 1);         };
```

```
GLOBAL FUNCTION :
    Iteration[T+1] =                          /* iteration counter*/
        IF Iteration[T] = MaxIter THEN 1 ELSE Iteration[T] + 1;
    Curr-Pattern[T+1] =              /* current pattern number */
        IF Phase[T+1] = Training THEN
            IF Iteration[T] = MaxIter THEN Curr-Pattern[T] + 1
            ELSE Curr-Pattern[T];
        ELSE                                                  /* Recalling */
            IF Phase[T] = Training THEN 1
            ELSE Curr-Pattern[T] + 1
    Input-Pattern[T] = Inpat[ Curr-Pattern[T] ];
    Outpat[ Curr-Pattern[T] ] = Output-Pattern[T];
    Phase[T] =                                        /* phase transition */
        IF EOF( 'OUTPAT.DAT' ) THEN
            IF Phase[T-1] = Training && Iteration[T-1] = MaxIter
                THEN Recalling
            ELSE IF Phase[T-1] = Recalling THEN Terminate
        ELSE Phase[T-1];
```

There are two files for input from and output to the environment respectively. Both of the files use the Bi-state predicate values -1 and 1. The input file is used for both *training* and *recalling* phases.

The counter *Iteration* is to count the number of iterations for the learning of the patterns. In this specification, a pattern will be learned for *MaxIter* number of iterations. After that, the next pattern will be presented to the perceptron. The value of *MaxIter* is not referenced in the *recalling* phase.

The transition among patterns is controlled by *Curr-Pattern*. In the *training* phase, the value of *Curr-Pattern* is depending on the number of iterations the pattern has been presented, i.e. *Iteration*. Every time when *Iteration* has achieved its maximal, the value of *Curr-Pattern* is increased by one. In *recalling*, the value of *Curr-Pattern* is first reset to 1 and the same file is used for recalling values. *Curr-Pattern* then increases in every iteration as it is only necessary to present once for every pattern in recalling.

Hence the perceptron will be given the records of the input file in sequence for twice, once for *training* and once for *recalling*. The same is, however, also true for the output file. This may sound confusing as it is not possible to write two sets of value into the same record. This in fact means that the output file will contain the second set of value, i.e., the result obtained from *recalling* phase only. The first set of value will be generated first but subsequently be overwritten. This is reasonable as we are only interested in the final results.

The phase transition variable *Phase* determines its transition by referring to the input file. If all the records in the input file are read, there are two possible actions. The first case is that the system is in the *training* phase, in which the value of *Phase* should change to *recalling* when the maximum number of iterations has passed. The second case is that the system is already in the *recalling* phase, in which case the system should terminate. In all other cases, the value of *Phase* should not be changed.

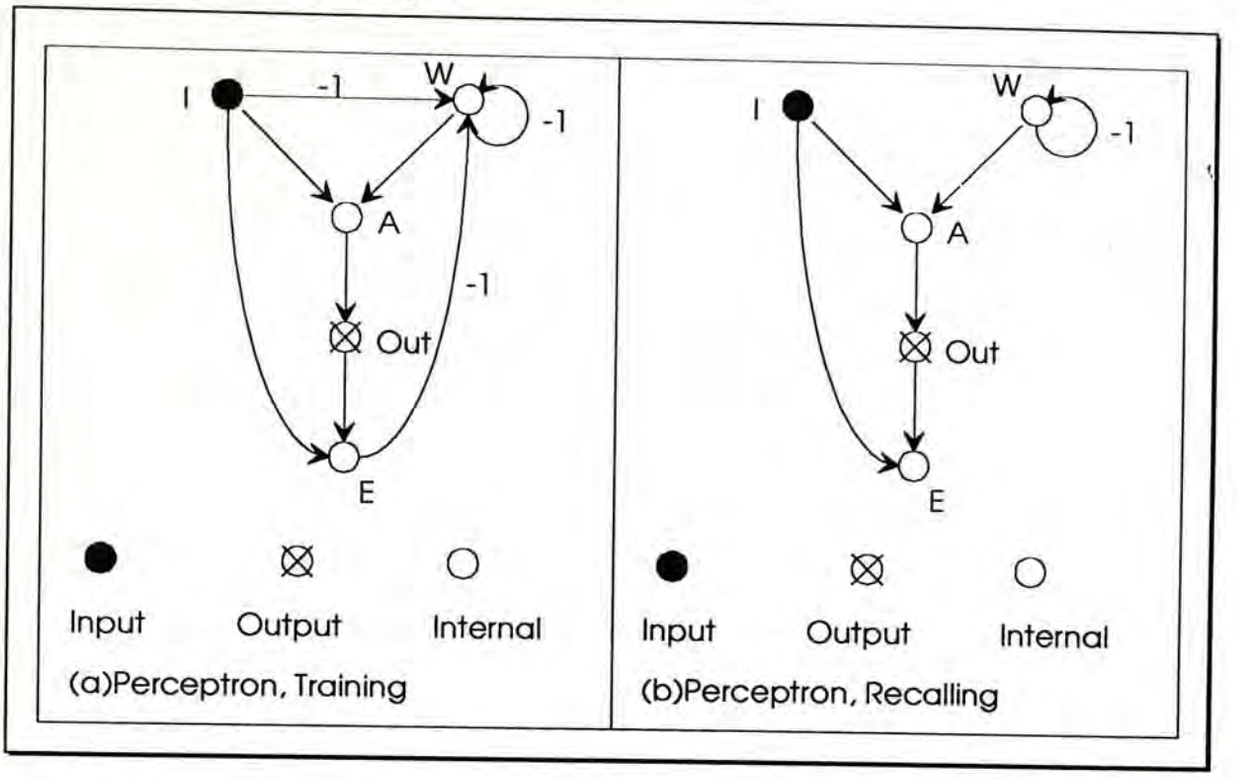
2. Results After Simple Checkings

No special result is reported from simple checkings.

3. Internal Dependency Graphs Construction

The internal dependency graphs returned by the graph construction process is simpler than that of the BP-Net case. These graphs involves less number of parameters, simpler dependency relations and less number of types of neurons.

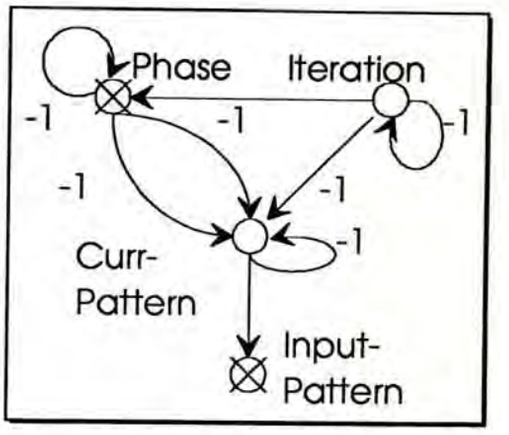
a) Internal Dependency Graphs



There are only two *internal dependency graphs* (IDGs) associated with perceptron, one for *training* and the other for *recalling*. The IDG for *recalling* is in fact a sub-graph of that for *training*.

b) Control Neuron Internal Dependency Graph

The internal dependency graph for control neuron is roughly the same as that of the BP-Net. It is because they are using the similar parameters, but their functioning are different. One can observe that there may be more than one dependency relation between parameters, such as that between *Phase* and *Curr-Pattern*. The parameter *Curr-Pattern* is depending on the same and the previous instance of *Phase*. This is quite common in our specification but the number of connections among the parameters is thus unlimited. This may significantly slow down the efficiency of the cycle detection algorithm.



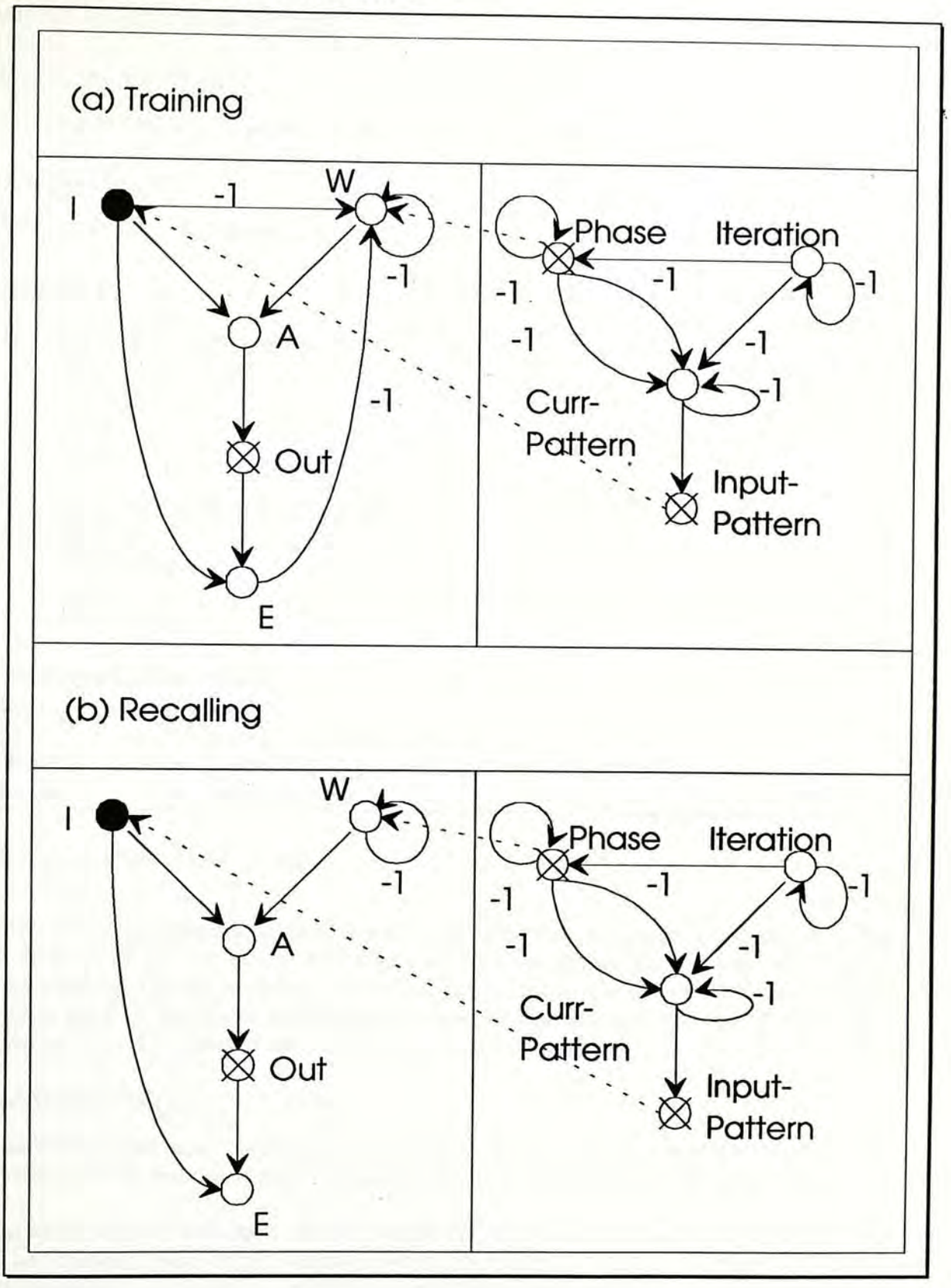
c) Combined Internal Dependency Graph

This graph is the same as the IDG on the *training* phase. This is because the IDG on *recalling* phase is a sub-graph of IDG on *training*. As a result, the combined graph is not different as that of the *training* phase.

4. Results From Parameter Analysis

No special results are reported from parameter analysis. They are all well-defined, well-matching and useful.

5. Global Dependency Graph Construction



The global dependency graph for the specified perceptron illustrates that it is necessary to combined one IDG with the CnIDG to get the resultant graph. A single IDG is sufficient because there is just one layer in the network. The solid arrows are internal dependencies while the broken arrows are global ones.

6. Cycles Detection

Discussed before and is not shown here.

7. Time Subscript Analysis

All the cycles pass the time subscript analysis.

8. Subscript Analysis

The subscript analysis is not applied.

9. Scheduling

```
(*  Scheduling of Perceptron, Training phase                    *)
W[T]    <-  W[T-1], I[T-1], E[T-1];
A[T]    <-  W[T], I[T];
Out[T]  <-  A[T];
E[T]    <-  Out[T], I[T];

(*  Scheduling of Perceptron, Recalling phase       *)
W[T]    <-  W[T-1];
A[T]    <-  W[T], I[T];
Out[T]  <-  A[T];
E[T]    <-  Out[T], I[T];

(*  Scheduling of Control Neuron                    *)
Iteration[T]      <-  Iteration[T-1];
Phase[T]          <-  Iteration[T-1], Phase[T-1];
Curr-Pattern[T]   <-  Curr-Pattern[T-1], Iteration[T-1], Phase[T-1], Phase[T];
Input-Pattern[T]  <-  Curr-Pattern[T];
```

C. BOLTZMANN MACHINE

This is the shifter example on p.299 of *Parallel Distributed Processing*, vol. 1 [34]. It follows the example given in the text, in which the neurons are randomly probed for updating. This is a safe approach for ensuring that every neuron gets to see the most recent states of all the other units. The parallelism is however, sacrificed. If the system can tolerate time delays, it is just necessary to remove the control signals from the control neuron which is responsible for updating.

1. Specification Forms

This specification is the most complicated one we given. This is due to the complicated control performed by the system over the neurons, and the comparatively vast number of book-keeping functions.

The input to the network is through clamped vectors but not through inputs into the neurons. When the neurons are clamped, their output are fixed while other neurons are free to change. Clamped with different vectors means training the neurons to learn different patterns.

There are totally six different operation phases for the neurons. In the *simulated annealing* process introduced by the text, the visible layers of the network is first clamped and the hidden neurons are then allowed to generate the output, one at a time, with a random function depending on a decreasing parameter called *temperature*. This phase is known as *phase*$^+$ (*Phase-plus* in our specification). After

that, every neuron is allowed to run for 10 iterations during which time the frequency with which each pair of connected units were both on was measured. This phase is called *After-Plus* by our specification. This process is repeated for 20 times with different clamped vectors and the co-occurrence statistics were averaged over all 20 runs to yield an estimate known as p_{ij}^{+} (which is called *p-plus*) in our specification.

After that, the system enters *phase*$^{-}$ (Phase-minus in our specification), in which the visible neurons are no longer clamped and all neurons are free to generate their outputs based on the same random function with decreasing temperature. After that, the neurons will record their own output with 10 free iterations. This phase is called *After-minus* in our specification. This process is again repeated for 20 runs with different clamped vectors and the statistics were recorded in p_{ij}^{-} (*p-minus* in our specification).

These four phases resulting in a set of 40 annealings is collectively known as a *sweep*. After every sweep, the neurons can update their own weights by $5(p_{ij}^{+} - p_{ij}^{-})$. In addition, every weight had its absolute magnitude decreased by 0.0005 times its absolute magnitude. This weight decay prevented the weights from becoming too large and it also helped to resuscitate hidden units which had predominantly negative or predominantly positive weights. This phase is called *update* phase. The whole process is then repeated again for 9000 times.

After that, the network will be in the *recalling* state. This is the state in which the system can be used for retrieving some information. The *Visible* units are clamped with input vectors and the *Shifts* units are used to indicate what type of shifts it is. Hence there are totally 6 states and the behaviour for different states are different. In addition, the transition among the states is rather complicated and it produces a rather clumsy specification on **control neuron**.

a) Constant Declaration

*****CONSTANT DECLARATION*****

```
#DECLARE ( PatSize, 8 );
#DECLARE ( Shifts, 3 );
#DECLARE ( VNum, PatSize * 2);
#DECLARE ( HNum, PatSize * Shifts );
#DECLARE ( Theta, 0.1 );
#DECLARE ( Neta, 5 );
#DECLARE ( Dec, 0.9995 );
#DECLARE ( NumOfPat, 20 );
#DECLARE ( StatIter, 10 );
#DECLARE ( Anneal, 16 );
#DECLARE ( MaxSweep, 9000 );
#DECLARE ( Temperature[1..Anneal], [40, 40, 35, 35, 30, 30, 25, 25, 20, 20, 15, 15, 12, 12, 10, 10] );
```

The meaning of the constants are given in the following table.

PatSize	Size of input pattern
Shifts	Ways of shifting the signals (right, left, no)
VNum	Number of visible neurons
HNum	Number of hidden neurons
Theta	Threshold value (θ), a system constant
Neta	Learning rate (η), a system constant
Dec	Weight decay for reducing weight magnitude

NumOfPat	Number of patterns presented
StatIter	Number of iterations for recording statistics
Anneal	Number of simulated annealing temperatures
Temperature	Temperature values for the output functions

One of the most special constant is the constant array *Temperature*. It is used to stored the temperatures for simulated annealing. It is possible for us to use some control to generate the pattern-wise temperatures but it will further complicate the system. Using such a constant array will be the most convenient method.

b) Formal Neuron

```
*** FORMAL NEURON ***

NEURON TYPE :
    Visible;
INPUT :
    (I[1..HNum], Clamp, Update) BINARY;
    Temp INTEGRAL;
OUTPUT :
    Out BINARY;
INTERNAL PARAMETER :
    (W[1..HNum], DeltaE) CONTINUOUS;
    (P-Plus[1..HNum], P-Minus[1..HNum]) INTEGRAL;
INITIAL VALUE :
    W[*][0] = RANDOM;  DeltaE[0] = RANDOM; Out[0] = RANDOM;
        P-Plus[*][0] = 0; P-Minus[*][0] = 0;
INTERNAL FUNCTION :
    ( Phase-Plus, Recalling ) {
        DeltaE[T] = DeltaE[T-1];        /* clamped, no updating */
        W[T+1] = W[T];
        Out[T] = Clamp[T];
        P-Plus[*][T] = 0;
        P-Minus[*][T] = 0;  };
    Phase-Minus {
        DeltaE[T] =
            IF Update[T] = 1 THEN  I[T] * Transpose( W[T] ) - Theta
            ELSE DeltaE[T-1];
        Out[T] =
            IF Update[T] = 1 THEN
                IF RANDOM(0,1) <=
                    1 / ( 1 + EXP( - DeltaE[T-1] / Temp[T-1] ) ) THEN 1
                ELSE 0
            ELSE Out[T-1];
        W[T+1] = W[T];
        P-Plus[T] = P-Plus[T-1];
        P-Minus[T] = P-Minus[T-1];  };
    After-Plus {
        DeltaE[T] =
            IF Update[T] = 1 THEN  I[T] * Transpose( W[T] ) - Theta
            ELSE DeltaE[T-1];
        Out[T] =
            IF Update[T] = 1 THEN
                IF RANDOM(0,1) <=
```

```
                    1 / ( 1 + EXP( - DeltaE[T-1] / Temp[T-1] ) ) THEN 1
                ELSE 0
            ELSE Out[T-1];
        W[T+1] = W[T];
        P-Plus[x : 1..HNum][T] =
            IF Update[T] = 1 && Out[T] = 1 && I[x][T] = 1 THEN
                P-Plus[T-1] + 1
            ELSE P-Plus[T-1];
        P-Minus[T] = P-Minus[T-1];  };
    After-Minus {
        DeltaE[T] =
            IF Update[T] = 1 THEN  I[T] * Transpose( W[T] ) - Theta
            ELSE DeltaE[T-1];
        Out[T] =
            IF Update[T] = 1 THEN
                IF RANDOM(0,1) <=
                    1 / ( 1 + EXP( - DeltaE[T-1] / Temp[T-1] ) ) THEN 1
                ELSE 0
            ELSE Out[T-1];
        W[T+1] = W[T];
        P-Minus[x : 1..HNum][T] =
            IF Update[T] = 1 && Out[T] = 1 && I[x][T] = 1 THEN
                P-Minus[x][T-1] + 1
            ELSE P-Minus[x][T-1];
        P-Plus[T] = P-Plus[T-1];  };
    Updating {
        Out[T] = Out[T-1];
        W[T+1] = Dec * W[T] + Neta * (P-Plus[T] - P-Minus[T]) / NumOfPat;
        DeltaE[T] = DeltaE[T-1];
        P-Plus[T] = P-Plus[T-1];
        P-Minus[T] = P-Minus[T-1];  };    };


/* Shifting Indicators */
NEURON TYPE :
    Shifts;
INPUT :
    (I[1..HNum], Clamp, Update) BINARY;
    Temp INTEGRAL;
OUTPUT :
    Out BINARY;
INTERNAL PARAMETER :
    (W[1..HNum], DeltaE) CONTINUOUS;
    (P-Plus[1..HNum], P-Minus[1..HNum]) INTEGRAL;
INITIAL VALUE :
    W[*][0] = RANDOM;  DeltaE[0] = RANDOM; Out[0] = RANDOM;
        P-Plus[*][0] = 0; P-Minus[*][0] = 0;
INTERNAL FUNCTION :
    Phase-Plus {
        DeltaE[T] = DeltaE[T-1];        /* clamped, no updating */
        W[T+1] = W[T];
        Out[T] = Clamp[T];
        P-Plus[*][T] = 0;
```

```
        P-Minus[*][T] = 0;  };
    (Phase-Minus, Recalling) {
        DeltaE[T] =
            IF Update[T] = 1 THEN  I[T] * Transpose( W[T] ) - Theta
            ELSE DeltaE[T-1];
        Out[T] =
            IF Update[T] = 1 THEN
                IF RANDOM(0,1) <=
                    1 / ( 1 + EXP( - DeltaE[T-1] / Temp[T-1] ) ) THEN 1
                ELSE 0
            ELSE Out[T-1];
        W[T+1] = W[T];
        P-Plus[T] = P-Plus[T-1];
        P-Minus[T] = P-Minus[T-1];  };
    After-Plus {
        DeltaE[T] =
            IF Update[T] = 1 THEN  I[T] * Transpose( W[T] ) - Theta
            ELSE DeltaE[T-1];
        Out[T] =
            IF Update[T] = 1 THEN
                IF RANDOM(0,1) <=
                    1 / ( 1 + EXP( - DeltaE[T-1] / Temp[T-1] ) ) THEN 1
                ELSE 0
            ELSE Out[T-1];
        W[T+1] = W[T];
        P-Plus[x : 1..HNum][T] =
            IF Update[T] = 1 && Out[T] = 1 && I[x][T] = 1 THEN
                P-Plus[T-1] + 1
            ELSE P-Plus[T-1];
        P-Minus[T] = P-Minus[T-1];  };
    After-Minus {
        DeltaE[T] =
            IF Update[T] = 1 THEN  I[T] * Transpose( W[T] ) - Theta
            ELSE DeltaE[T-1];
        Out[T] =
            IF Update[T] = 1 THEN
                IF RANDOM(0,1) <=
                    1 / ( 1 + EXP( - DeltaE[T-1] / Temp[T-1] ) ) THEN 1
                ELSE 0
            ELSE Out[T-1];
        W[T+1] = W[T];
        P-Minus[x : 1..HNum][T] =
            IF Update[T] = 1 && Out[T] = 1 && I[x][T] = 1 THEN
                P-Minus[x][T-1] + 1
            ELSE P-Minus[x][T-1];
        P-Plus[T] = P-Plus[T-1];  };
    Updating {
        Out[T] = Out[T-1];
        W[T+1] = Dec * W[T] + Neta * (P-Plus[T] - P-Minus[T]) / NumOfPat;
        DeltaE[T] = DeltaE[T-1];
        P-Plus[T] = P-Plus[T-1];
        P-Minus[T] = P-Minus[T-1];  };     };
```

```
/* Hidden Units */
NEURON TYPE :
    Hidden;
INPUT :
    (I[1..VNum], Update) BINARY;
    Temp INTEGRAL;
OUTPUT :
    Out BINARY;
INTERNAL PARAMETER :
    (W[1..HNum], DeltaE) CONTINUOUS;
    (P-Plus[1..HNum], P-Minus[1..HNum]) INTEGRAL;
INITIAL VALUE :
    W[*][0] = RANDOM;  DeltaE[0] = RANDOM; Out[0] = RANDOM;
        P-Plus[*][0] = 0; P-Minus[*][0] = 0;
INTERNAL FUNCTION :
    (Phase-Minus, Phase-Plus, Recalling) {
        DeltaE[T] =
            IF Update[T] = 1 THEN  I[T] * Transpose( W[T] ) - Theta
            ELSE DeltaE[T-1];
        Out[T] =
            IF Update[T] = 1 THEN
                IF RANDOM(0,1) <=
                    1 / ( 1 + EXP( - DeltaE[T-1] / Temp[T-1] ) ) THEN 1
                ELSE 0
            ELSE Out[T-1];
        W[T+1] = W[T];
        P-Plus[T] = P-Plus[T-1];
        P-Minus[T] = P-Minus[T-1];  };
    After-Plus {
        DeltaE[T] =
            IF Update[T] = 1 THEN  I[T] * Transpose( W[T] ) - Theta
            ELSE DeltaE[T-1];
        Out[T] =
            IF Update[T] = 1 THEN
                IF RANDOM(0,1) <=
                    1 / ( 1 + EXP( - DeltaE[T-1] / Temp[T-1] ) ) THEN 1
                ELSE 0
            ELSE Out[T-1];
        W[T+1] = W[T];
        P-Plus[x : 1..HNum][T] =
            IF Update[T] = 1 && Out[T] = 1 && I[x][T] = 1 THEN
                P-Plus[T-1] + 1
            ELSE P-Plus[T-1];
        P-Minus[T] = P-Minus[T-1];  };
    After-Minus {
        DeltaE[T] =
            IF Update[T] = 1 THEN  I[T] * Transpose( W[T] ) - Theta
            ELSE DeltaE[T-1];
        Out[T] =
            IF Update[T] = 1 THEN
                IF RANDOM(0,1) <=
                    1 / ( 1 + EXP( - DeltaE[T-1] / Temp[T-1] ) ) THEN 1
```

```
                ELSE 0
            ELSE Out[T-1];
        W[T+1] = W[T];
        P-Minus[x : 1..HNum][T] =
            IF Update[T] = 1 && Out[T] = 1 && I[x][T] = 1 THEN
                P-Minus[x][T-1] + 1
            ELSE P-Minus[x][T-1];
        P-Plus[T] = P-Plus[T-1];  };
    Updating {
        Out[T] = Out[T-1];
        W[T+1] = Dec * W[T] + Neta * (P-Plus[T] - P-Minus[T]) / NumOfPat;
        DeltaE[T] = DeltaE[T-1];
        P-Plus[T] = P-Plus[T-1];
        P-Minus[T] = P-Minus[T-1];  };     };
```

There are totally three types of neurons, the *Visible* type which is those neurons representing the input pattern and its shifted partners, the *Hidden* type which is those hidden neurons responsible for determining the shifts, and the *Shift* type which is responsible for indicating the shift.

At first glance, it seems that these three types of neurons are quite different. The resultant specification, however, reveals that they are quite similar. They differs from one another only in their number of input signals and the use of equations in different phases. The difference in input signals is obvious as they are connected to different groups.

On the other hand, the use of different equations in different phases is more interesting. It seems that the equations for the neurons are quite different but there are in fact only 5 groups of equations. These 5 groups can cater for the 6 phases of these 3 types of neurons. Hence the difference among the neurons are on the different grouping of equations and phases.

There are quite a number of parameters for the neurons. These parameters and their uses are given in the following table. With these explanations, it is very easy to map the equations in the specification to the standard equations in the text.

I	Input	Signals from other groups of neurons
Clamp		Signal for clamping output
Update		Signal for asynchronous updating
Out	Output	Output to other groups of neurons
W	Internal	Weights of input
DeltaE		Energy change, ΔE
P-Plus		Recording the number of 1's in After-Plus
P-Minus		Recording the number of 1's in After-Minus

c) Configuration

```
*** CONFIGURATION ***

NEURON LABEL :
    V[1..VNum], H[1..HNum], S[1..Shifts];
NEURON CHARACTER :
    V[1..VNum] TYPE Visible
        INPUT-DEGREE HNum+3
        OUTPUT-DEGREE HNum;
```

```
    S[1..Shifts] TYPE Shift
        INPUT-DEGREE HNum+4
        OUTPUT-DEGREE HNum;
    H[1..HNum] TYPE Hidden
        INPUT-DEGREE VNum+3
        OUTPUT-DEGREE VNum;
CONNECTION PATTERN :
    {   H[x].I[y] = V[y].Out;              /* input and outputs */
        H[x].I[z] = S[z-VNum].Out;
        V[y].I[x] = H[x].Out;
        S[w].I[x] = H[x].Out;
        CN.Outpat[w] = S[w].Out;                  /* output to control neuron */
        H[x].Update = CN.Update[x];       /* update signal */
        S[w].Update = CN.Update[HNum+w];
        V[y].Update = CN.Update[HNum+Shifts+y];
        H[x].Temp = CN.Temp;              /* temperature */
        V[y].Temp = CN.Temp;
        S[w].Temp = CN.Temp;
        V[y].Clamp = CN.Clamp[y];         /* clamped signal */
        S[w].Clamp = CN.Clamp[VNum+w];
        x : 1..HNum; y : 1..VNum; z : VNum+1..VNum+Shifts; w : 1..Shifts }
```

The configuration of the network is also rather complicated. This is owing to complexity in connection. We use different **neuron labels** for different type of neurons so all the three types of neurons should have their own connection declarations. This is different from the case of BP-Net, in which the same label is used for different types of neurons. The different labelling approach is adopted in this specification because it gives clearer connections among the different type of neurons. If the same label is used for different types of neurons, the connection may be ambiguous. Similar types of connections are grouped together in the specification.

d) Control Neuron

```
*** CONTROL NEURON ***

GLOBAL INPUT :
    Outpat[1..Shifts] BINARY;
GLOBAL OUTPUT :
    Temp INTEGRAL; Clamp[1..VNum+Shifts] BINARY;
    (Update[1..VNum+HNum+Shifts]) BINARY;
GLOBAL PARAMETER :
    (Iteration, Sweep, Probe, Annealing, Curr-Out, Curr-In) INTEGRAL;
GLOBAL INITIAL VALUE :
    Iteration[0] = 0;  Sweep[0] = 0;        Annealing[0] = 1;
    Curr-Out [0] = 0; Curr-In[0] = 0;       Probe[0] = 0;
    Phase[0] = Phase-Plus;
INPUT FILE :
    FILE 'INPAT.DAT', RECORD Inpat [1..19] RANGE BINARY;
OUTPUT CONTROL :
    FILE 'OUTPAT.DAT', RECORD Opat [ FROM 0 FOR 3 ] RANGE BINARY;
GLOBAL FUNCTION :
    Curr-Out[T] =
        IF Phase[T] = Recalling && Annealing[T-1] = NumOfPat * 2 THEN
```

```
            Curr-Out[T-1]+1
        ELSE Curr-Out[T-1];
    Opat[ Curr-Out[T] ] = Outpat[T];
    Curr-In[T] =
        IF Annealing[T] > 20 THEN
            Annealing[T] - 20
        ELES Annealing[T];
    Clamp[T] = Inpat[ Curr-In[T] ];
    Temp[T] =
        IF Phase[T] = Phase-Plus || Phase[T] = Phase-Minus ||
            Phase[T] = Recalling THEN Temperature[ Iteration[T] ]
        ELSE Temperature[Anneal];
    Update[T] =
        IF Phase[T] = Phase-Plus THEN RANDOM(1,HNum)
        ELSE IF Phase[T] = After-Plus THEN RANDOM(1, HNum)
        ELSE IF Phase[T] = Phase-Minus THEN
            RANDOM(1, HNum+VNum+Shifts)
        ELSE IF Phase[T] = After-Minus THEN
            RANDOM(1, VNum+HNum+Shifts)
        ELSE IF Phase[T] = Updating THEN 0
        ELSE IF Phase[T] = Recalling THEN RANDOM (1, Shifts+HNum);
    Probe[T+1] =
        IF Phase[T] = Phase-Plus && Probe[T] = HNum THEN 1
        ELSE IF Phase[T] = After-Plus && Probe[T] = HNum THEN 1
        ELSE IF Phase[T] = Phase-Minus && Probe[T] =
            HNum+VNum+Shifts THEN 1
        ELSE IF Phase[T] = After-Minus && Probe[T] =
            HNum+VNum+Shifts THEN 1
        ELSE IF Phase[T] = Updating && Probe[T] =
            HNum+VNum+Shifts THEN 1
        ELSE IF Phase[T] = Recalling && Probe[T] = HNum+Shifts THEN 1
        ELSE Probe[T] + 1;
    Iteration[T+1] =
        IF Probe[T+1] = 1 THEN
            IF Phase[T] = Phase-Plus && Iteration[T]=Anneal-1 THEN 1
            ELSE IF Phase[T]=After-Plus && Iteration[T]=StatIter-1 THEN 1
            ELSE IF Phase[T]=Phase-Minus && Iteration[T]=Anneal-1 THEN 1
            ELSE IF Phase[T]=After-Minus && Iteration[T]=StatIter-1 THEN 1
            ELSE IF Phase[T]=Updating THEN 0
            ELSE Iteration[T] + 1
        ELSE Iteration[T];
    Annealing[T+1] =
        IF Iteration[T+1] = 1 && Iteration[T] <> 1 THEN
            IF Phase[T] = Phase-Plus || Phase[T] = Phase-Minus
                THEN Annealing[T]
            ELSE Annealing[T] + 1
        ELSE IF Iteration[T+1] = 0 && Iteration[T] <> 0 THEN 1
        ELSE Annealing[T];
    Sweep[T+1] =
        IF Annealing[T+1] = 1 && Annealing[T] <> 1 THEN Sweep[T] + 1
        ELSE Sweep[T];
    Phase[T] =
        IF Iteration[T] = 1 && Iteration[T-1] <> 1 THEN
            IF Phase[T-1] = Phase-Plus THEN After-Plus
```

```
        ELSE IF Phase[T-1] = After-Plus THEN
            IF Annealing[T] = NumOfPat THEN Phase-Minus
            ELSE Phase-Plus
        ELSE IF Phase[T-1] = Phase-Minus THEN After-Minus
        ELSE IF Phase[T-1] = After-Minus THEN
            IF Annealing[T] = NumOfPat*2 THEN Updating
            ELSE Phase-Minus
        ELSE IF Phase[T-1] = Recalling && Sweep[T] = MaxSweep
            THEN Terminate
        ELSE Phase[T-1]
    ELSE IF Iteration[T] = 0 && Iteration[T-1] <> 0 THEN Recalling
    ELSE Phase[T-1];
```

The specification on the **control neuron** is very complex owing to the complicated control required for the operation of the network. There are a lot of counters to be maintained, and the dependency among them introduces a lot of equations for updating these counters. In conventional imperative programming language, we can use loops to control the operation of the neurons but in this type of dataflow language, we should explicitly identify the condition for the updating of the counters. This is not easy for this complicated example.

There are a number of parameters used in the **control neuron** specification form. Their meanings are given in the following table.

Probe	Number of probing for neurons
Iteration	Counter for every phase
Annealing	Number of annealing sets
Sweep	Number of sweeps
Temp	System temperature
Update	Index of the neuron to be updated
Curr-In	Current input pattern
Curr-Out	Current output pattern
Phase	Phase transition variabie

The first 4 parameters are counters for various purposes. The parameter *Probe* is used to record the number of probings for the neurons. In every probing, a neuron is selected randomly from those unclamped ones. Let the number of unclamped neurons be N. After N probings, which means that on average every neuron is updated once, the *Iteration* is increased by 1. The number of iterations for every phase is different. For *Phase-Plus* and *Phase-Minus*, the number of iterations is equal to the number different temperatures used for simulated annealing. For *After-Plus* and *After-Minus*, the number is equal to 10. Each combination of *Phase-Plus* and its corresponding *After-Plus* forms an *Annealing*. So the $Phase^+$ and $Phase^-$ together have 40 annealings. This is called a *Sweep*. There are totally 9000 sweeps.

Temp is the parameter for indicating the temperature of the system at a particular moment. *Update* is a random parameter for determining which neuron is to be updated. *Curr-in* and *Curr-Out* are record addressing parameters. They are used to determine which input and output record are to be accessed. *Phase* is the phase transition parameter for determining the current phase of the system.

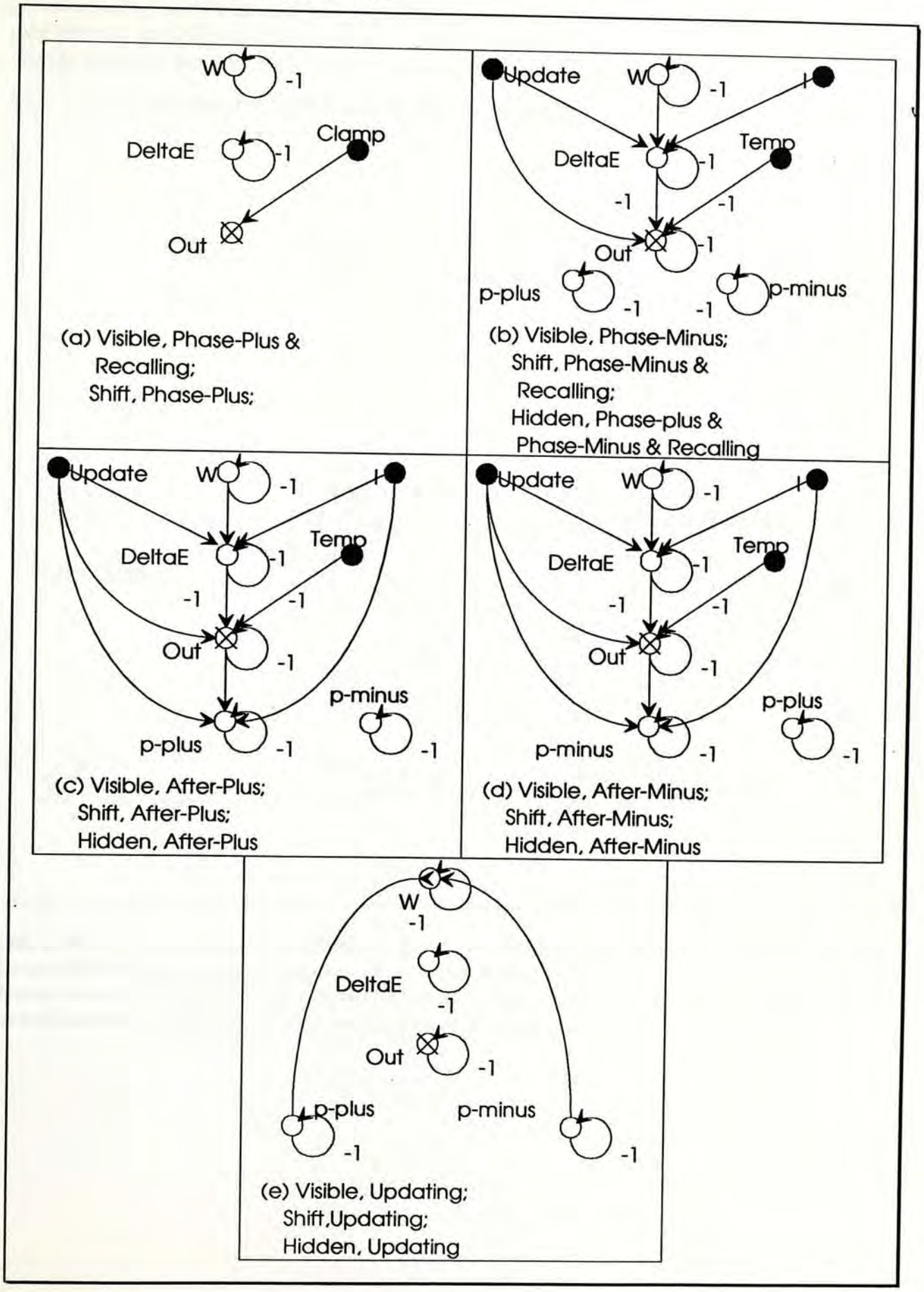
2. Results After Simple Checkings

No special result is reported from simple checkings.

3. Graphs Construction

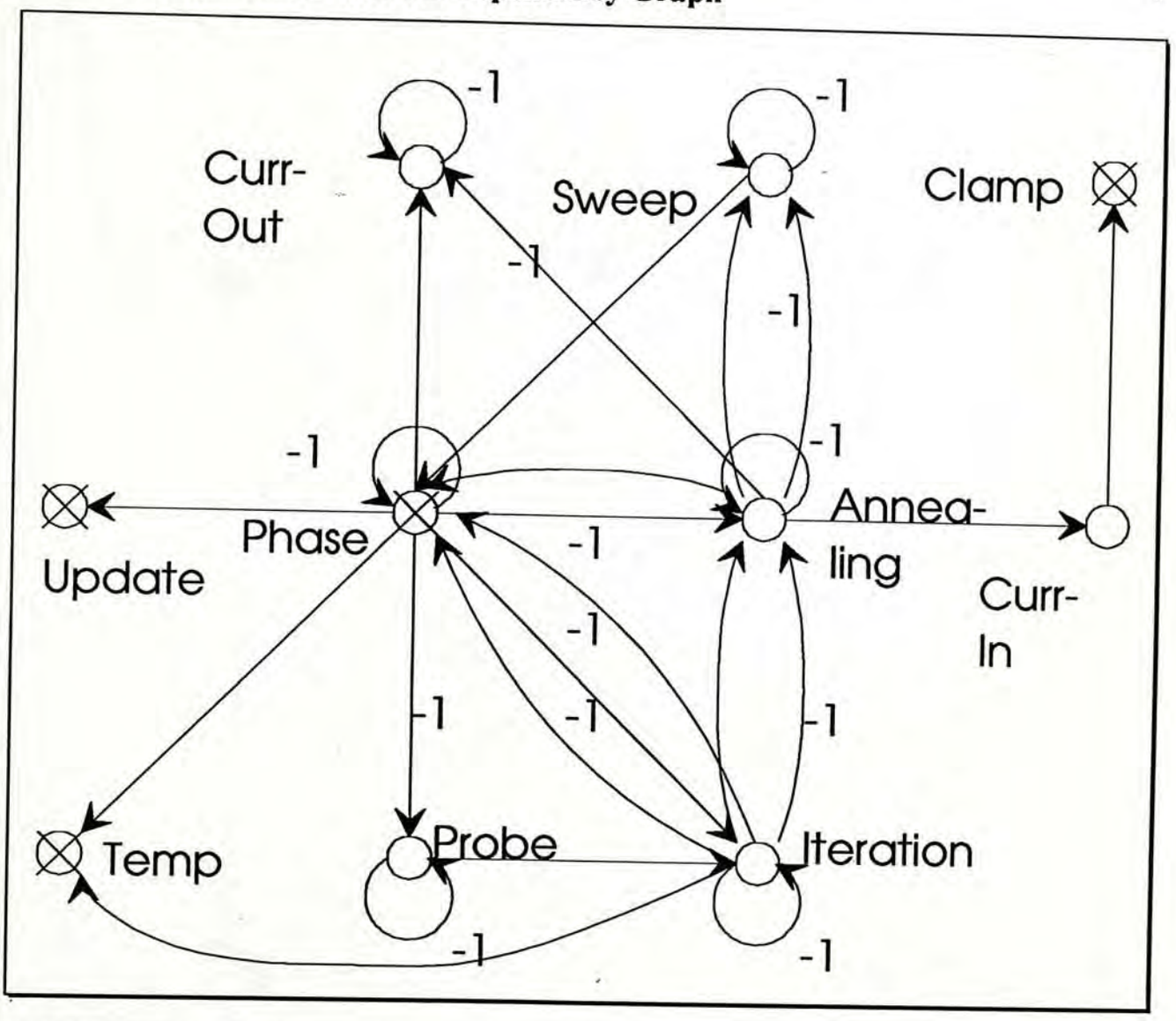
The graphs for our example Boltzmann is again the most complicated. It has the greatest number of *internal dependency graphs* for neurons (IDGs) and control neuron (CnIDG), the most complicated *combined internal dependency graph* (CnIDG) and, surely, the greatest number of *global dependency graphs* (GDG). This means that manual analysis on the example specification will be extremely difficult.

a) Internal Dependency Graph



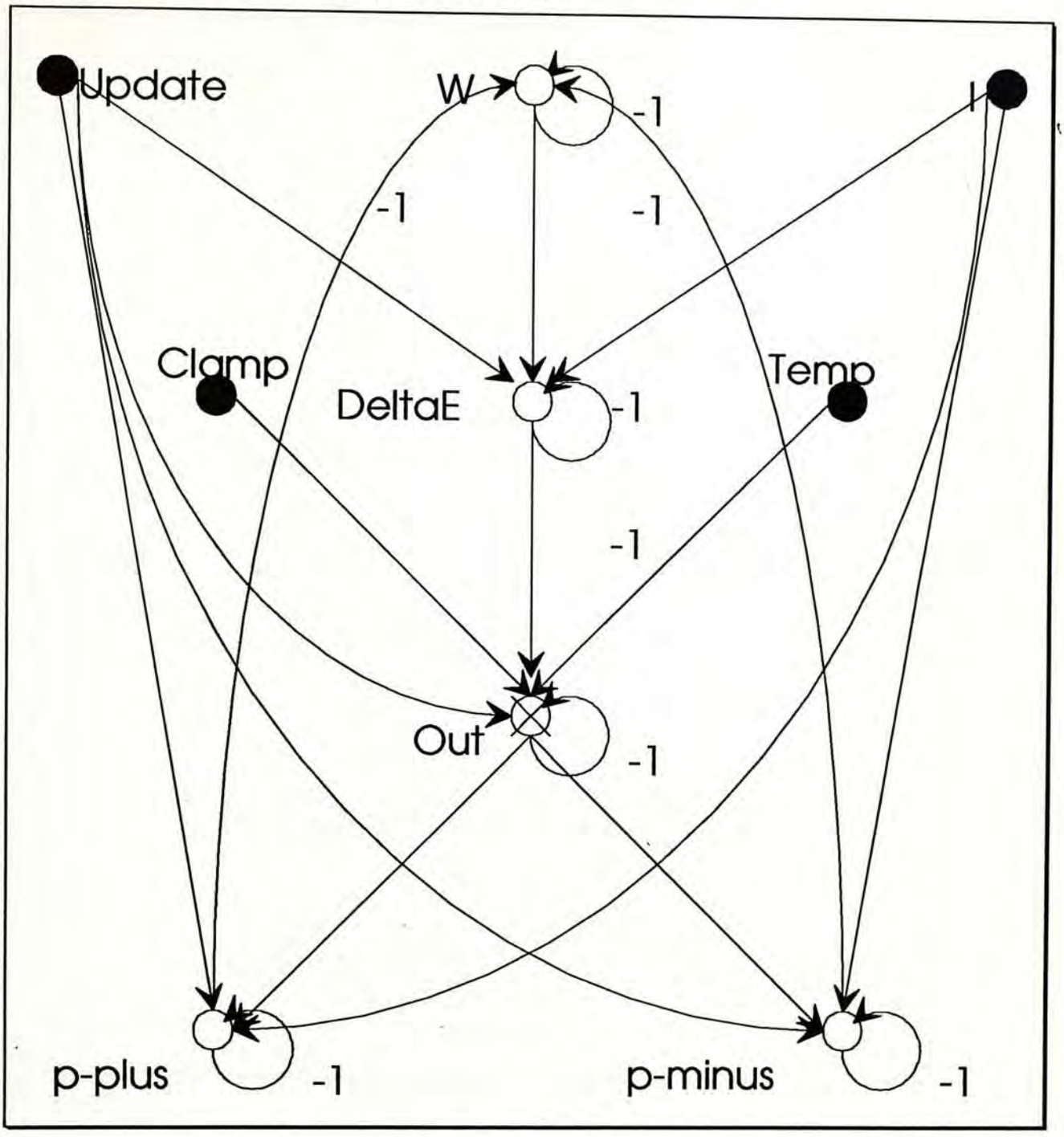
There are totally five *internal dependency graphs* (IDGs) for our example. These are the five groups of equations being used for the neurons in different phases. From the graphs, we can observe that the neurons function similarly in different combination of phases. Hence although there are totally 3 types of neurons each with 6 phases, there are just 5 types of internal dependency graphs.

b) Control Neuron Internal Dependency Graph



This is the most complicated *control neuron internal dependency graph* (CnIDG) we have. There are 10 parameters interacting with one another. This is due to the control scheme of the system, in which iteration counters for different cases has to be handled. One can refer to the previous discussion on the specification on **control neuron** for an explanation of the counters.

c) Combined Internal Dependency Graph

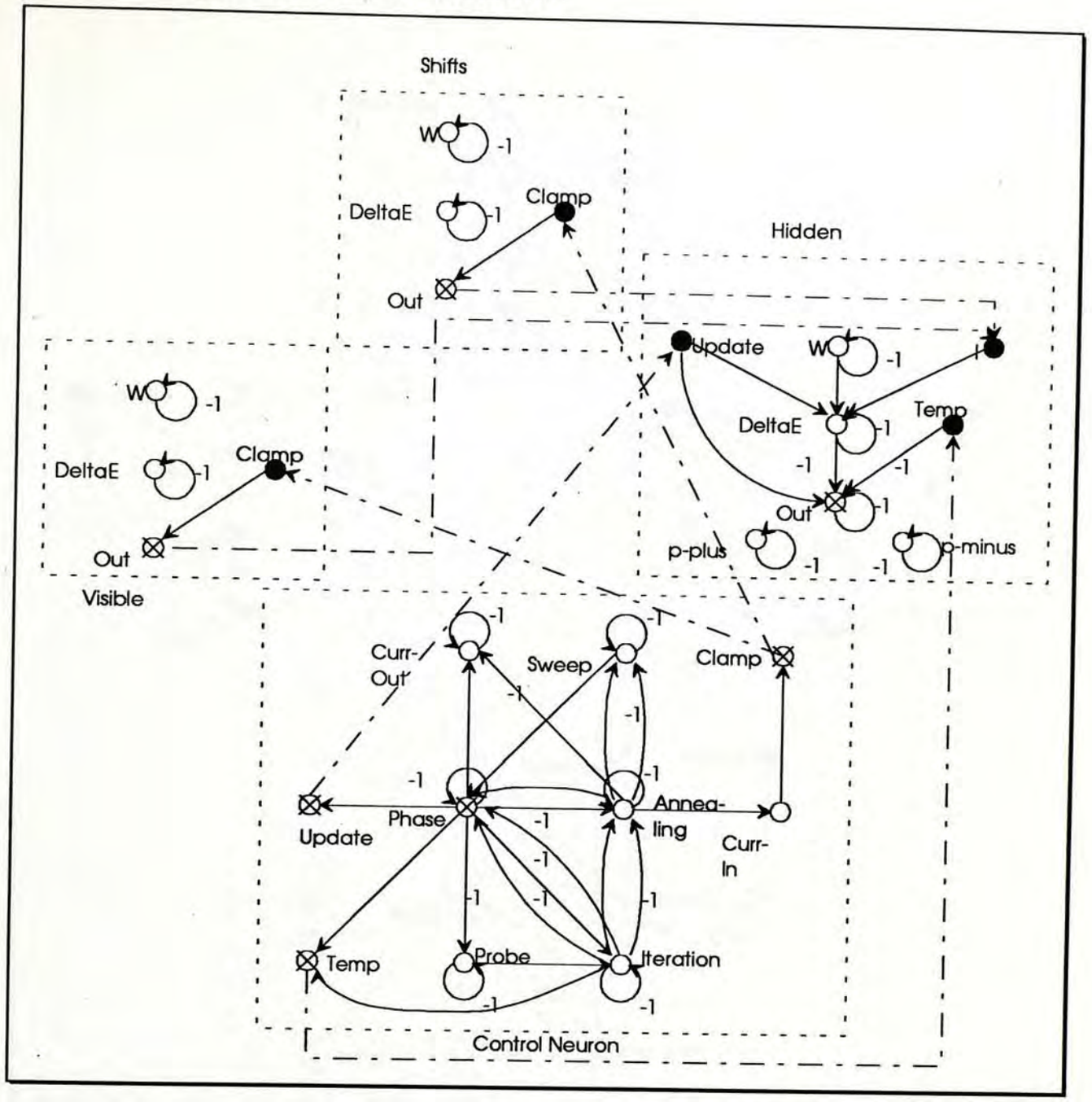


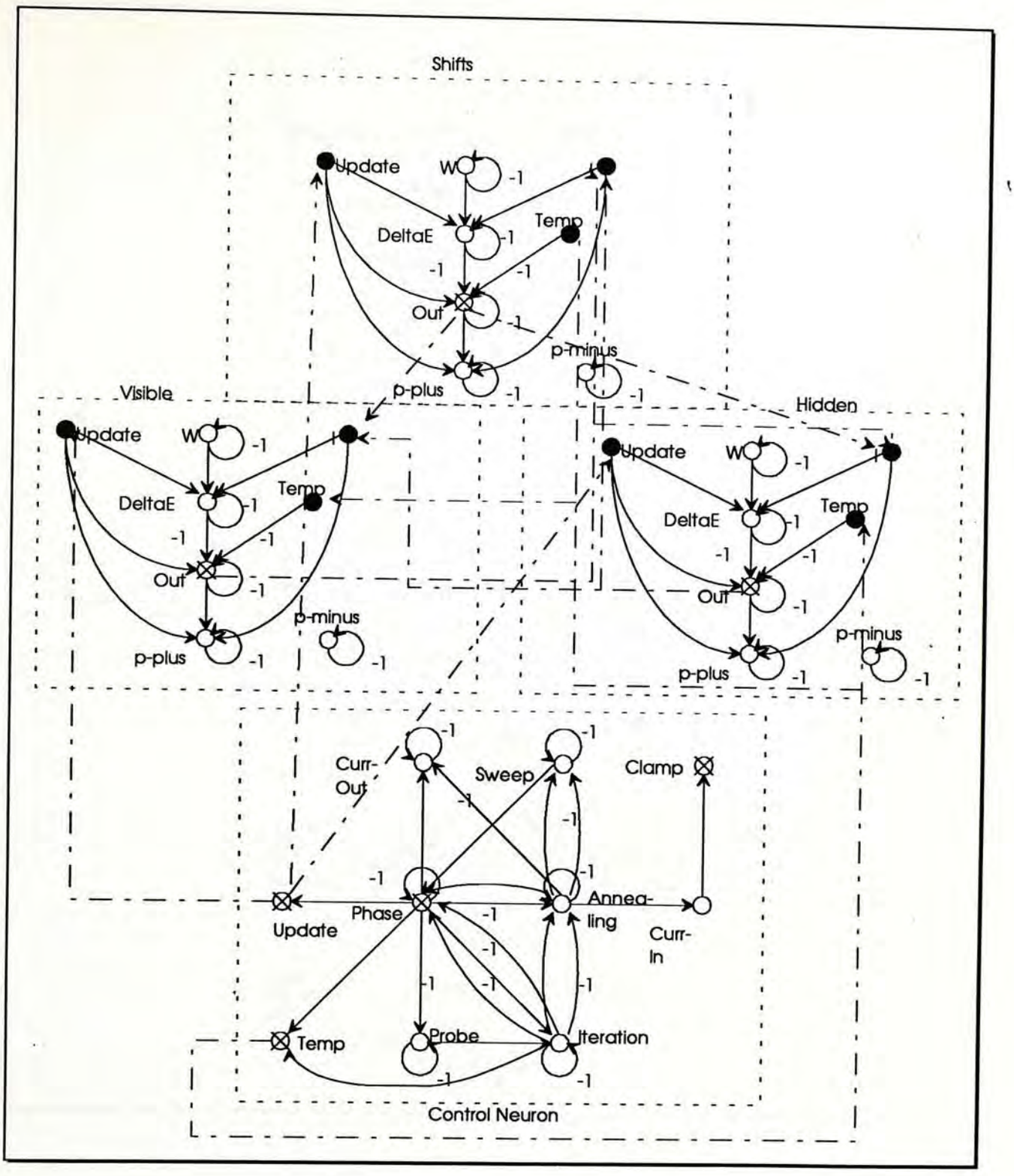
The *combined internal dependency graph* (CnIDG) shows how the nine parameters are linked together when all phases are considered together. Although there is only one output parameter *Out*, all other parameters are directly or indirectly determining it. All parameters are determining other parameters and input parameters are not determined by others.

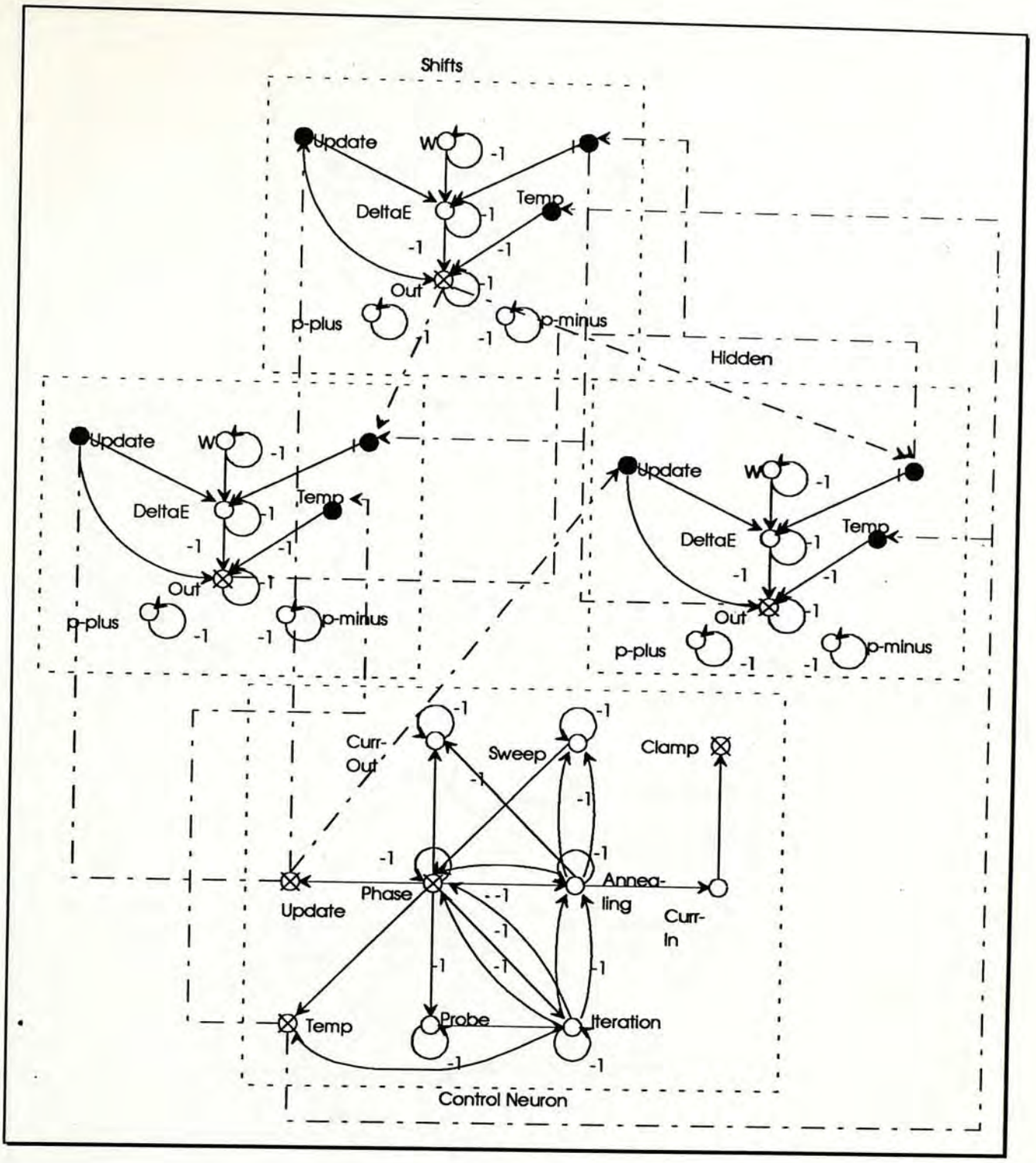
4. Results From Parameter Analysis

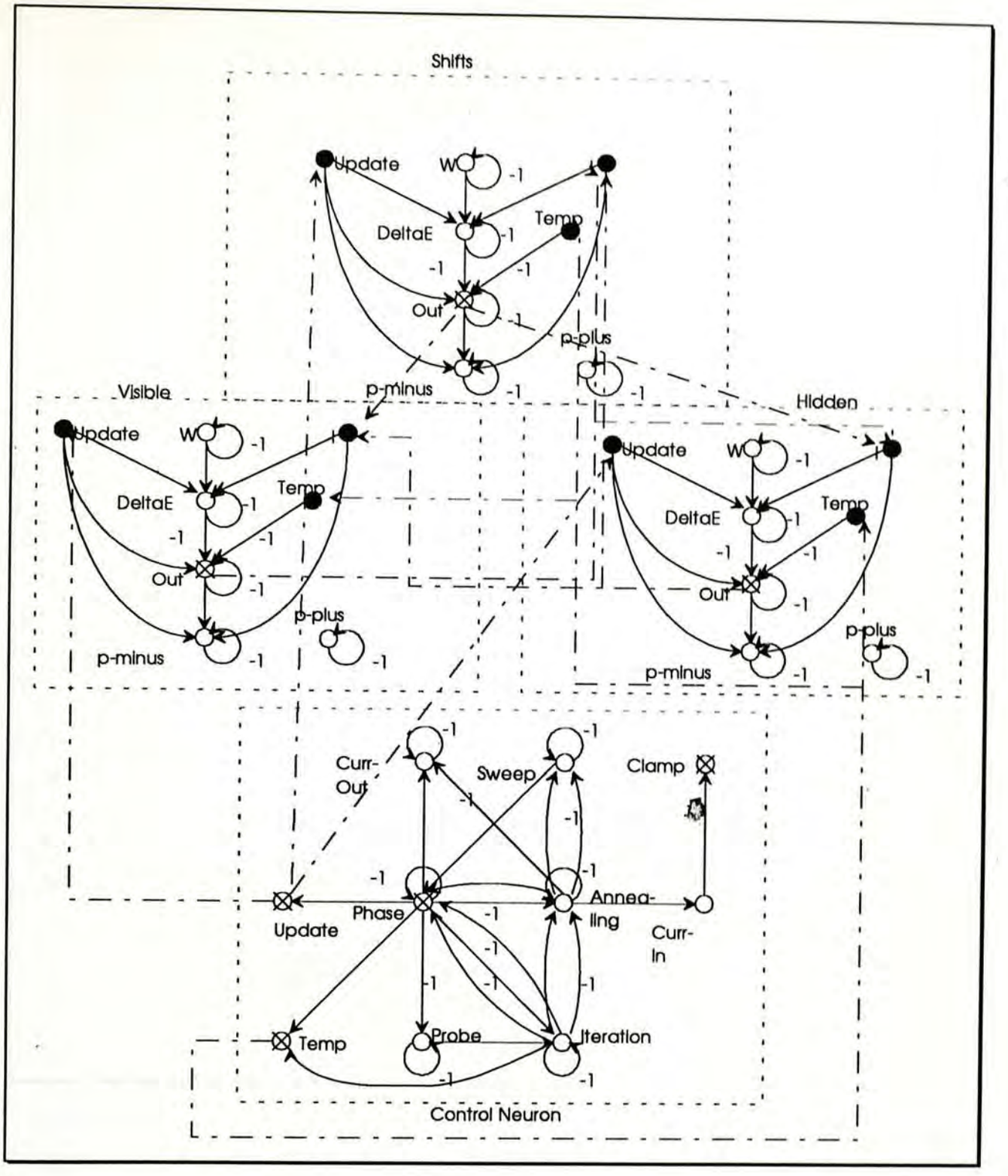
No special result is reported from parameter analysis. All parameters are useful and well defined.

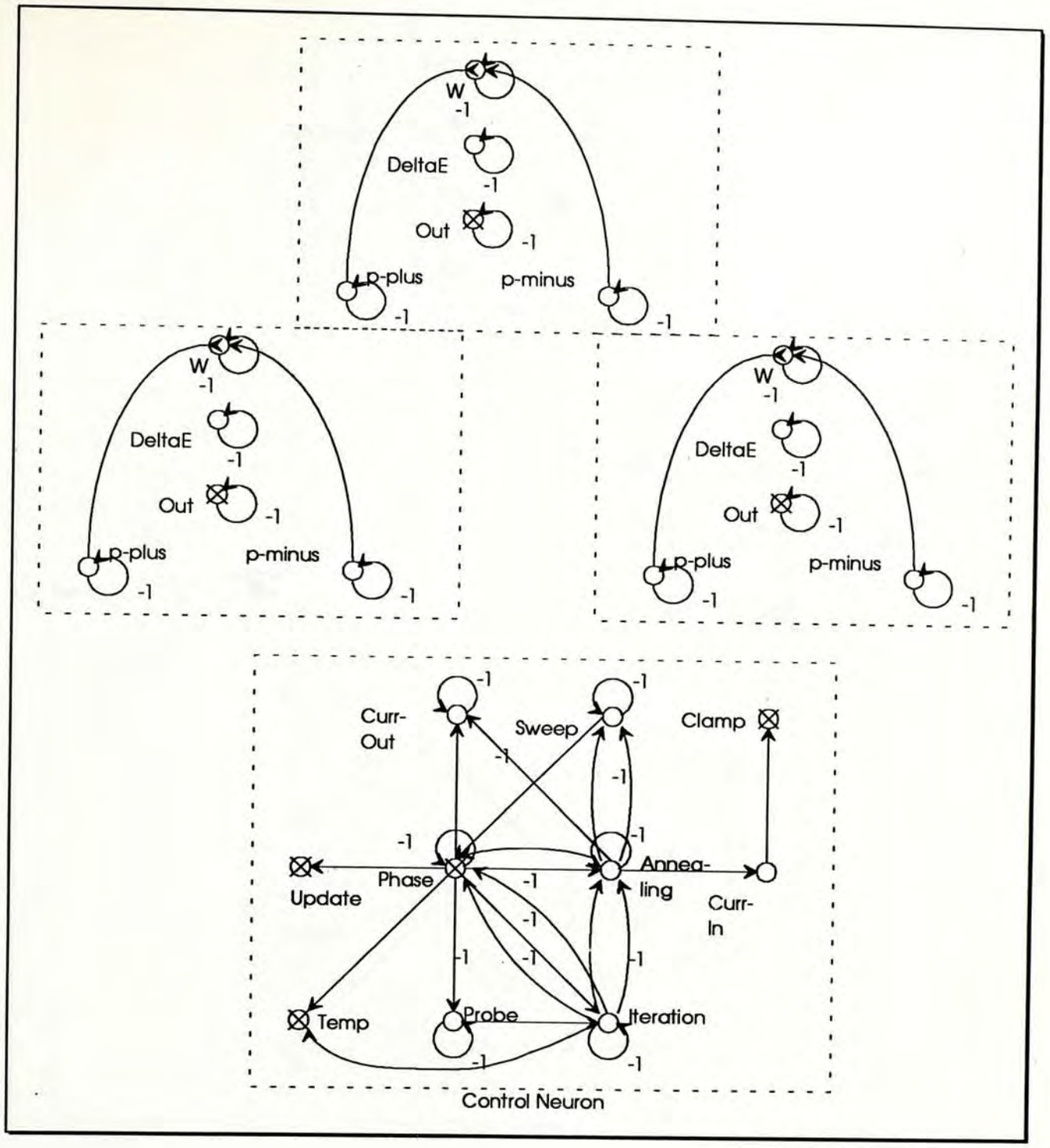
5. Global Dependency Graphs Construction

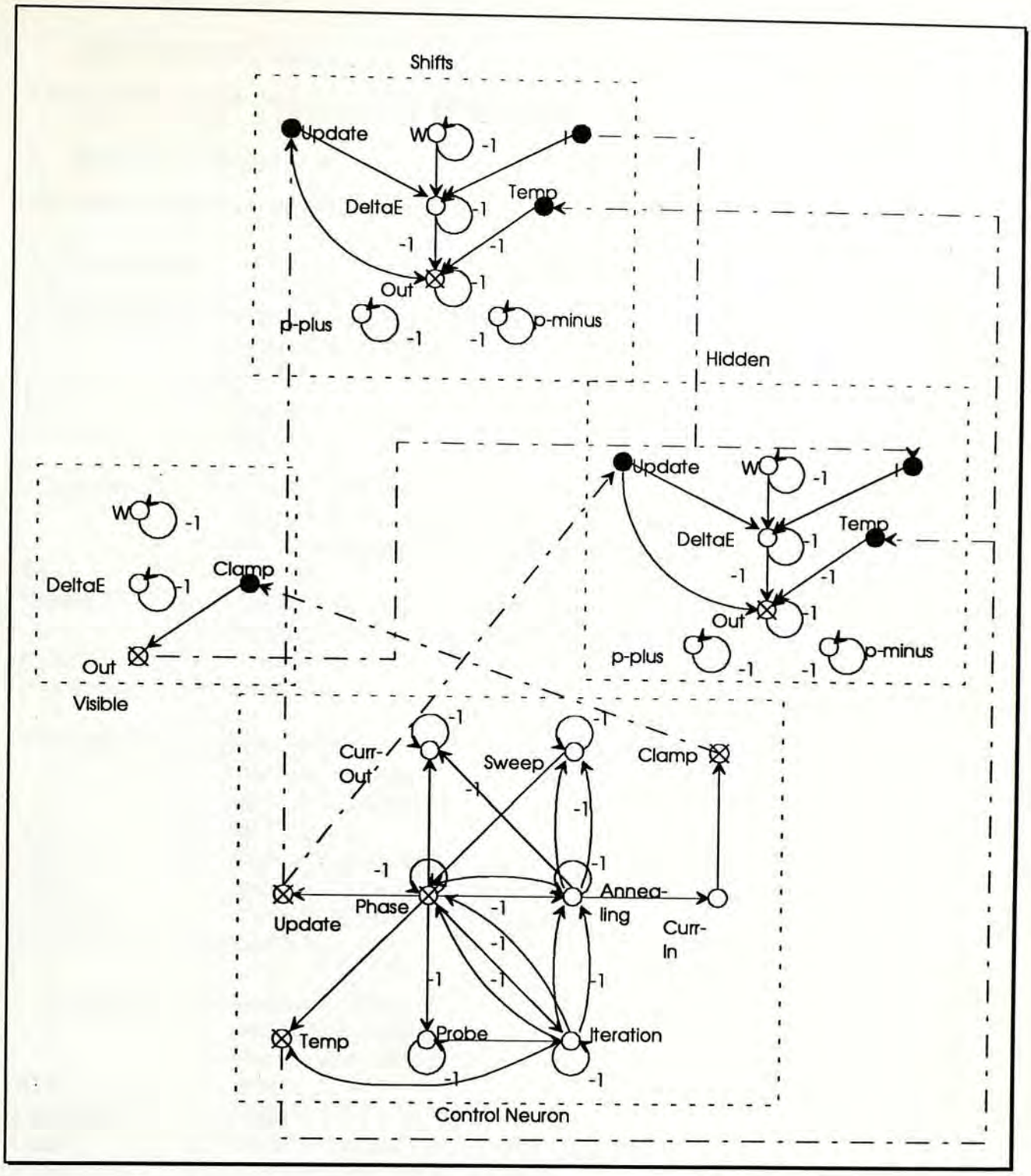



Shifts
Update
W
-1
DeltaE
Temp
Out
p-minus
p-plus
Visible
Hidden
Curr-
Out
Sweep
Clamp
Phase
Annea-
ling
Curr-
In
Update
Temp
Probe
Iteration
Control Neuron


Shifts
Update
W
DeltaE
Temp
Out
p-plus
p-minus
Hidden
Curr-
Out
Sweep
Clamp
Phase
Annea-
ling
Curr-
In
Update
Temp
Probe
Iteration
Control Neuron
-1


Shifts
Update
W
-1
DeltaE
Temp
Out
p-plus
p-minus
Visible
Hidden
Curr-
Out
Sweep
Clamp
Phase
Annea-
ling
Curr-
In
Update
Temp
Probe
Iteration
Control Neuron


W
-1
DeltaE
-1
Out
-1
p-plus
p-minus
-1
-1
W
-1
DeltaE
-1
Out
-1
p-plus
p-minus
-1
-1
W
-1
DeltaE
-1
Out
-1
p-plus
p-minus
-1
-1
Curr-
Out
Sweep
Clamp
Phase
Update
Annea-
ling
Curr-
In
Temp
Probe
Iteration
Control Neuron



There are six *global dependency graphs* for the Boltzmann machine, one for each phase. The graphs are shown in the order *Phase-Plus*, *After-Plus*, *Phase-Minus*, *After-Minus*, *Updating* and *Recalling*. We do not show the dependency of the parameters on the phase transition parameter *Phase* because every parameter is depending on *Phase*. Including the dependency edges in the graphs will make them too complicated to be read. The graph for phase *Updating* has no global dependency edges, as the updating will depend on local information only. This can also be observed from the cycles of this phase, in which they are all local cycles but not global ones.

6. Cycle Detection

Discussed before and is not shown here.

7. Time Subscript Analysis

All the cycles can pass through the time subscript analysis.

8. Subscript Analysis

The subscript analysis is no need to be applied.

9. Scheduling

```
(* Schedule for   phase-plus & recalling of Visible,
                      phase-plus of Shift *)
DeltaE[T]    <-  DeltaE[T-1];
W[T]         <-  W[T-1];
Out[T]       <-  Clamp[T];

(* Schedule for   phase-minus of Visible,
                      phase-minus & recalling of Shift,
                      phase-plus, phase-minus, recalling of Hidden *)
W[T]             <-  W[T-1];
DeltaE[T]        <-  Update[T], I[T], W[T], DeltaE[T-1];
Out[T]           <-  DeltaE[T-1], Out[T-1], Temp[T-1], Update[T];
P-plus[T]        <-  P-plus[T-1];
P-minus[T]   <-  P-minus[T-1];

(* Schedule for   after-plus of Visible,
                      after-plus of Shift,
                      after-plus of Hidden *)
W[T]             <-  W[T-1];
DeltaE[T]        <-  Update[T], I[T], W[T], DeltaE[T-1];
Out[T]           <-  DeltaE[T-1], Out[T-1], Temp[T-1], Update[T];
P-plus[T]        <-  P-plus[T-1], Out[T], I[T], Update[T];
P-minus[T]   <-  P-minus[T-1];

(* Schedule for   after-minus of Visible,
                      after-minus of Shift,
                      after-minus of Hidden *)
W[T]             <-  W[T-1];
DeltaE[T]        <-  Update[T], I[T], W[T], DeltaE[T-1];
Out[T]           <-  DeltaE[T-1], Out[T-1], Temp[T-1], Update[T];
P-minus[T]   <-  P-minus[T-1], Out[T], I[T], Update[T];
P-plus[T]        <-  P-plus[T-1];

(* Schedule for   updating of Visible,
                      updating of Shift,
                      updating of Hidden *)
Out[T]           <-  Out[T-1];
W[T]             <-  W[T-1], P-plus[t-1], P-minus[T-1];
P-plus[T]        <-  P-plus[T-1];
P-minus[T]   <-  P-minus[T-1];
DeltaE[T]        <-  DeltaE[T-1];

(* Schedule for Control Neuron *)
```

```
Probe[T]        <- Phase[T-1], Probe[T-1];
Iteration[T] <- Probe[T], Phase[T-1], Iteration[T-1];
Annealing[T]    <- Iteration[T], Iteration[T-1], Phase[T-1], Annealing[T-1];
Sweep[T]        <- Annealing[T], Annealing[T-1], Sweep[T-1];
Phase[T]        <- Iteration[T], Iteration[T-1], Phase[T-1], Annealing[T],
                   Sweep[T];
Curr-Out[T] <- Phase[T], Annealing[T-1], Curr-Out[T-1];
Curr-In[T]  <- Annealing[T];
Clamp[T]        <- Curr-In[T];
Temp[T]         <- Phase[T], Iteration[T];
Update[T]       <- Phase[T];
```