ADAPTIVE SELF-ORGANIZING LOGIC NETWORKS

Tony Ramon Martinez

June 1986
CSD-860093

The dissertation of Tony Ramon Martinez is approved.

Tomas Lang

Tomas Lang

Jack W. Carlyle

Jack Carlyle

J Schlag

John Schlag

Segundo

Jose Segundo

Jacques J. Vidal, Committee Chair

University of California, Los Angeles

1986

Table of Contents

List of Figures

List of Tables

VITA

July 15, 1958	Born, Great Falls, Montana
1982	B.S. in Computer Science, Brigham Young University
1982-1983	Teaching Assistant, University of California, Los Angeles
1982-1983	Computer Analyst, NASA Jet Propulsion Laboratories
1983	M.S. in Computer Science University of California, Los Angeles
1983-1985	Teaching Associate, University of California, Los Angeles
1984-1985	Computer Analyst Senior, System Development Corporation
1985-1986	Post Graduate Research Engineer University of California, Los Angeles

ABSTRACT OF THE DISSERTATION

Adaptive Self-Organizing Logic Networks

by

Tony Ramon Martinez

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1986

Professor Jacques J. Vidal, Chair

Along with the development of contemporary computer science the limitations of sequential "von Neumann" machines have become more apparent. It is now becoming clear that to handle projected needs in speed and throughput, massively parallel architectures will be needed.

In this dissertation we propose a special purpose architectural model that satisfies a general class of propositional logic problems in a totally distributed and concurrent fashion. The architectural model is identified as *ASOCS* (Adaptive Self-Organizing Concurrent System).

Problem specification is incremental and takes the form of if-then rules (instances) expressed as Boolean conjunctions. Possible applications include symbolic decision systems, propositional production systems, digital pattern recognition and real-time process control.

The approach is based on an adaptive network composed of many simple computing elements (nodes) which operate in a combinational and asynchronous fashion. Control and processing in the network is distributed amongst the network nodes. Adaptation and data processing form two separate phases of operation. During processing, the network acts as a parallel network of Boolean gates. Inputs and outputs of the network are also Boolean. During adaptation the network structure and the node functions can change to update the overall network function as specified. As new rules are added to the rule base, the network independently reconfigures to a logic circuit that remains both minimal and consistent with the rule base. Thus, there is no explicit *programming*. Desired network response is simply presented to the system, following which the network adjusts itself accordingly. Although the functionality of the network can be observed from the outside, the internal network structure is unknown.

The control of the adaptive process is almost completely distributed and efficiently exploits parallelism. Most communication takes place between neighboring nodes with only minimal need for centralized processing. The network modification is performed with considerable concurrency and the adaptation time grows only linearly with the depth of the network.

Chapter 1

INTRODUCTION and MOTIVATION

In this chapter we introduce the dissertation by first giving a brief overview of the model proposed. Section 2 begins a discussion of the motivation and rationale for this research. This is done by first giving an informal and brief historical background retracing the intellectual steps that led to this work. The overall features of the model are given in section 3. Section 4 discusses the realm of possible applications of the proposed system. The final section lists the remaining chapters of the dissertation and briefly describes their contents.

1.1. Model Overview

In this work we propose an architectural model and three specific algorithms tailored for this architecture which solve a specific class of problems in a concurrent fashion. The basic problem is to design concurrent digital networks able to solve problems defined by propositional calculus. Inputs and outputs of the network are boolean. These types of problems include symbolic decision systems, propositional production systems, digital pattern recognition, and real-time control.

The system is composed of many simple computing elements which operate asynchronously in a combinational fashion. The control of this network of computing elements is distributed amongst the elements themselves. Central control is not necessary. The specific domain of application is that of propositional logic, where problem specification is given by incremental input of boolean if-then rules. As new rules are input to the system, it maintains a rule-base which is both consistent and minimal. The architectural model is identified in a later chapter with the acronym *ASOCS* (Adaptive Self-Organizing Concurrent System). Control and processing are separated into two phases of operation. During processing, the network acts as a parallel network of digital gates. During the control, or adaptation phase, the nodes within the network are responsible to change the overall network structure to fit the new function specified by the incremental input of rules.

In addition to proposing a novel parallel solution of the propositional logic problem, this research shows the possibility of *distributed control*, or *self-organization*, within a massively parallel network of nodes solving a specified function.

1.2. Research History

1.2.1. Concurrency and Brain Function

Traditional sequential computing is not sufficient to solve many types of problems in acceptable time. As we attack more knowledge intensive problems[1] we find glaring inadequacies, in terms of speed, in a purely sequential computing scheme. Physical limitations, including the speed of light, are now becoming the determining factor in how fast a single stream of data can be processed. By contrast, human and animal brains, using relatively slow processing units, are able to process huge amounts of data in a very short time. This suggests that there are approaches to computing

[1] And we are just scratching the surface of problems requiring huge amounts of data processing.

with concurrency and massive parallelism which could compute orders of magnitude faster than the current Von Neumann[2] strategies.

The computing world is fast approaching an impasse, where new techniques of concurrent computing will be necessary to solve large classes of problems in realistic time. To be sure, there are many types of problems which match the Von Neumann methodology. Sequential and numeric computing will always be necessary even if massively parallel machines are available. There will always be a limit on space and hardware that will force decomposition of a large problem into a sequence of smaller ones.

My initial research interest was neuromimetics. We certainly know too little about the brain to be able to use it as a detailed model. However, there does appear to be a *style* of information processing in the brain which can be contrasted with that of Von Neumann machines. Table 1.1 lists some of these.

Von Neumann	Brain
Sequential	Massively Parallel
Single or Few Processors	Many Processors
Addressable Storage Separate from Processor	Active Memory Distributed Through Connections
Instructions and Data Fetched and then Executed at Single Processor	Direct Data Flow Execution
Programmed	Self-Adapting

Table 1.1 - Von Neumann vs. Brain Style

Each of these contrasts is briefly discussed below. Each of these points is subjective

[2] Von Neumann is here used as a descriptor for the current computing architectures and software methods commonly used for data processing.

and meant only to give a historical perspective.

Information processing in Von Neumann machines and brains is neither purely sequential nor purely parallel. They both depend on a mix in order to properly function. However, the basic paradigm of computing in the Von Neumann machine is a sequential execution of instructions. The basic paradigm of the brain appears to be a combinational flow of data between inputs and outputs.

Although some Von Neumann machines use more than one processor, the number is always very limited and in practice, the overall speed of the system actually often decreases once the number of processors grow beyond a relative few. In contrast, the brain uses an enormous number of processing neurons, approximately 10^{11}, to fulfill its functionality.

Memory in the Von Neumann machine is stored in a structured addressable storage where a single cell is dedicated to each bit of data. It would appear that in the brain, memory is distributed throughout by means of the interconnections and functions of its neurons.

The Von Neumann machine functions by fetching data and instructions across a bus between the processor and the memory, and then executing the instruction locally at the processor. The brain apparently operates in a more *data flow* fashion. Rather than fetch instructions, data is implicitly processed as it flows through the neural structure.

Finally, the Von Neumann machine must be programmed "prescriptively" and each detail of a desired function must be foreseen and covered by the program. If the function changes, the programming must be changed. The brain has the ability to adapt to a changing environment with only unstructured or generic "instructions."

The initial goals were then to explore architectures and propose methods[3] exhibiting some of these brain-like features.

1.2.2. Adaptive Networks and Combinational Architectures

The central object of this research is a network of adaptive processing elements assumed to interact with its environment by a set of sensors and effectors. Figure 1.1 gives a picture of what is intuitively meant by a network which functions in a concurrent and data flow fashion. Inputs enter from the environment and flow through the nodes of the network at propagation speed. The flow is asynchronous and combinational. Output values can both change the environment and feedback into the inputs.

Logic elements are primitives of all calculation and a network of logic gates is in theory a universal model of computation. However, almost all work done with logic networks has been limited to the *synthesis* of a structure, given a complete function specification. Rather than look for one-time synthesis strategies, we are more concerned with the notion of adaptation. The basic question is: *Assuming the existence of a network already solving a function, how do we modify the network when the function changes (perhaps slightly) without resynthesizing the network.* When an incremental change to the network function is specified, then a relatively small modification to the network should be accomplished, allowing it to correctly accomplish the new function.

[3] In Von Neumann terminology, methods would refer to programs, operating systems, control schemes, etc.

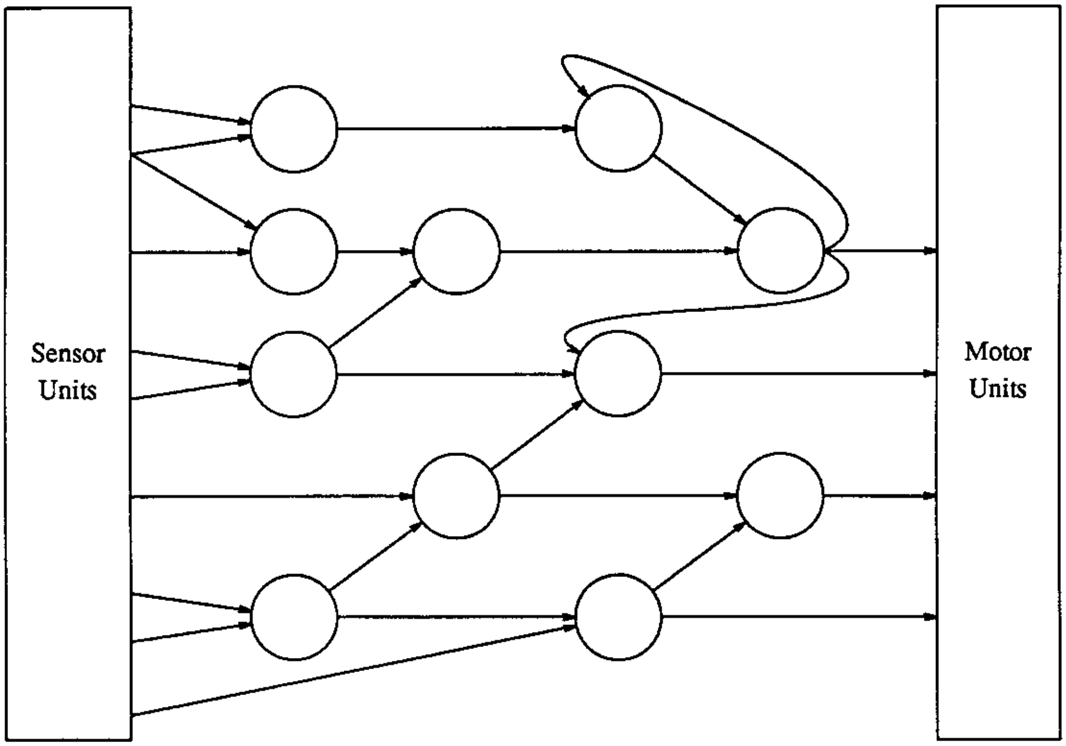


Figure 1.1 - Data Flow Network

This would provide a form of network ***Learning.*** One would not have to start with a fully specified function, rather the function could *grow* and become more precise with time.

Because of the above, we assume two basic modes of operation for an adaptive logic network: ***Execution*** and ***Adaptation.*** In execution mode, the system is processing input to output in the normal data flow fashion of a fixed circuit. When new knowledge is introduced to the system in adaptation mode, the network is modified to handle the added information. After modification it can return to execution mode. Switching between modes can continue indefinitely.

We defined an adaptive network as one which had three basic capabilities:

1. The functions of the network nodes can be modified.
2. The interconnections between nodes can be modified.
3. The number of active nodes in the network can change.

We originally assumed that an *external agent*, probably a Von Neumann machine, would have to work with the logic network in the following ways.

1. The external agent would have complete knowledge of the network structure.
2. The external agent would make all modification decisions and apply them to the passive logic network.

From these initial assumptions developed several approaches They basically reduced to heuristic search techniques which were exponential since the number of states of the logic network grows exponentially with the number of nodes. Thus, the time necessary for the external agent to do adaptation would grow exponentially with the number of nodes in the network. In this initial effort, it was assumed that there could only be a single output variable in the network, and that there would be no feedback.

1.2.3. Self-Organization

The initial results were disappointing and unlikely to be very useful for the above mentioned reasons. The next step was to realize that *concurrent computing power used during the execution mode could be used to aid in the adaptation process*. Using this principle we found that it was possible to devise faster algorithms which enlisted the aid of the logic network for adaptation. This was done by no longer using search, but a deterministic technique which always assured a correctly functioning network after a bounded number of steps. However, there were still many problems arising from the use of one central controller which needed complete knowledge and control of the network. As the network got large, the time necessary for the external agent to be effective would still grow beyond reasonable limits. The need for the agent to be able to address each node within the network would make VLSI implementation difficult. Also, testing and recovery measures become unpractical with one central controller.

In line with neuromimetic thinking, the solution was to let go of the central control, and allow the nodes of the network to accomplish most of the adaptation process. In order to allow the network to *self-adapt*, more functionality and communication capability became needed at each network node. However, control became distributed throughout the network. After discovering algorithms which functioned within these parameters, the two initial assumptions changed to the following.

1. There is no knowledge (by any agent) of the complete network structure.
2. The network is ***Self-Organizing.***

Given these conditions we were able to show that the time necessary for adaptation grows linearly with the depth of the network. *Self-organization* became the key paradigm in the adaptive concurrent architecture. It is probably the key to adaptation in any large complex system. As the amount of information within a system grows, the ability for it to be controlled and made adaptive by an external agent diminishes. We see this daily in systems we are involved with. Even a large software program becomes virtually un-modifiable because of the tremendous amount of information regarding the interaction of the many pieces of the system. In order to have complex adaptive systems, we may have to abandon the pervasive use of complete knowledge and control over each aspect of the system.

Under the self-organization principle the system can be viewed as a black box (figure 1.2).

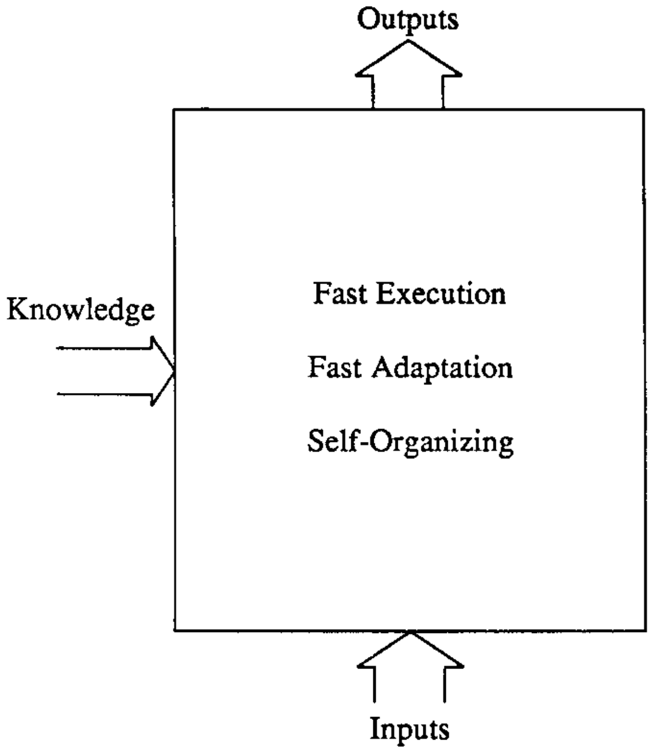


Figure 1.2 - Self-Organizing System

There is no strict prescriptive "programming" of the internals of the system. Rather, one gives *specifications* of the problem regarding what the desired function is, and it is left up to the system itself to accommodate them. By contrast, most of the effort in current programming is spent detailing *how* to solve a problem, rather than *what* to accomplish.

1.3. Features of the Model

In this section we discuss some of the features of the architectural model.

The first is that of *concurrent* processing of data. The data path is a combinational hardware circuit able to compute data at the propagation speeds of the network nodes. During execution, the speed of computation is $O(d)$ where d is the depth of the network.

The next feature is *adaptability*. Using the concurrent logic network, and the self-organizing mechanism, the system is able to reconfigure itself to adjust its function to new requirements. The time necessary for the system to self-adapt is also $O(d)$. This will become apparent during the discussion of the algorithms.

Perfect optimality, in terms of the number of nodes necessary to solve a function, has not been a goal in this research. Rather the desire was to use *redundancy* and *flexibility* in order to achieve the goal of adaptability. However, nodes within the network are able to discover when they are no longer a part of the overall computation of the network function. When this is detected, the node deletes itself from the network. Furthermore, it sends commands to its children and they recursively delete, thus pruning whole sections of the network. Because of this *self-deletion* notion the network maintains a *relatively* optimal number of nodes to compute the current function.

The method of knowledge input is assumed to be in the form of *if-then* rules, where the variables in the rules are boolean. This type of knowledge specification is *natural* for many types of problems related to propositional logic. The rule base allows for multiple outputs and feedback. Rule input is a natural method to allow incremental change of an overall function.

The fact that the system is built out of many nodes, each with exactly the same structure has many potential advantages. One is efficient VLSI design. Basically, one node must be designed, then copies of that node are placed in a regular array with a consistent connectivity structure. Outputs from one node become inputs to the next. Another important VLSI constraint is that of *pin* requirements. As the number of circuits put on a normal chip increases, the ability to control the circuits from the outside decrease. One reason is that it is not possible to put sufficient pins on the outside of the chip in order to control specific units within. In the proposed model, because of self-organization, the number of pins on a chip can stay constant regardless of how many nodes there are inside of the chip. In fact there is neither need nor means to address nodes within a chip. Only pins for input, output, and rule introduction are necessary.

Since there is no way to control or even know the state of the internal structure of the network, it is essential to provide for *self-test* and *self-recovery*. The proposed systems are indeed able to fulfill both of these functions, and self-repair of faulty circuits will take place without any outside control. Both fabrication faults, a common occurrence in VLSI fabrication, and faults that develop over time are handled by the systems fault tolerance mechanisms.

Another feature of the model is that it is amenable to *decomposition*. Because inputs and outputs are compatible, any number of systems can be combined together with the outputs of one system becoming inputs to another.

1.4. Possible Applications of the Model

In this section we discuss possible practical applications of the model.

The model is best fitted to *Symbolic* computation, rather than *Numeric* computation. This is do in large part to the tendency of symbolic applications to have a relatively *sparse* amount of important outputs in a large input domain. This contrasts with many numerical applications having a very dense input-output mapping. For example, in an integer multiply it is essential that there be a defined output for every permutation of the input variables. This is not the case in many symbolic applications. This follows for two basic reasons. First, there are typically many *don't care* variables in a symbolic function. At any given time only a subset of the input domain is essential to make a decision. The other factor is that of *impossible states*. Given a large domain of input variables, only a small percentage of the possible permutations may actually occur in nature. If-then rule bases are natural for defining outputs in domains including many don't care variables and impossible input states. Many of these symbolic problems are also combinational in that all inputs are presented simultaneously and the output is not bound by sequential constraints.

Another important feature of applications fitted to the current effort is that of *adaptability*. There are two general reasons for it. The first is that we seldom do anything right the first time. Nor do we correctly understand the problem during initial implementation. This equates to the constant debugging and design changes of current systems. The second main reason is that many problems are adaptive by nature, in that their function must change in time in order to pursue changing goals. If the function changes over time, then it is essential that the application be built on a methodology which allows that change.

1.4.1. Examples of Applications

In this section we discuss classes of applications in which the current model could fit. For ease of communication, the model is referred to by its generic name ASOCS.

One class of system which many times has the need of high speed and adaptivity is that of *embedded systems*. These systems are generally doing real-time decision making with environmental sensor data as input. Because of real-time requirements, these applications are candidates for combinational computing. A controller in a satellite is a typical example. If it is desired to change slightly the function of the controller it would not be efficient to bring down the satellite, make physical changes, and then relaunch it. Instead, it would be better to send the specification modifications to the satellite and allow it to reconfigure. With the ASOCS, the same high speed could be maintained after reconfiguration. Many embedded systems must be adaptive, and currently the only method is to carry a Von-Neumann machine aboard. However, this method can be too slow and too costly in terms of space for many applications.

Robotics is an example of embedded system technology where high adaptivity will be required. Current robots do not display the amount of adaptivity which will be required for new applications. If adaptivity is desired, the current solution is to run a robot by a communication link from a Von Neumann machine. Again, this is much too slow for the types of applications now being discovered for robotics.

The field of *production systems*, including *expert systems* and *knowledge based* systems, is one of the most promising applications of artificial intelligence [Wate78, Haye83, Buch84]. However, as the size of the knowledge bases grow, the search techniques used in Von Neumann systems do not allow these systems to function in reasonable time. If concurrency can be used in the knowledge retrieval process then expert systems with large rule bases will become feasible. Expert systems have shown that many applications can be specified as a set of rules. Thus there is a natural link between the ASOCS methodology and many expert system models.

Another point about production systems has to do with execution. Using completely sequential mechanisms, a production system functions by selecting one of many possible rules, sometimes in a random fashion, and then causing that rule to be *fired*. By using a concurrent mechanism all applicable rules could be discovered and fired simultaneously.

Control and decision systems in general are those which cause or output control information depending on the input state. These are very natural for a rule-based and concurrent system. An example of this type of system is a *fault isolation and recovery system*. In this case, a number of sensors in a machine, for example a jet aircraft, are connected to a high speed control unit. When certain states of these sensors arise, signaling a fault condition, then immediate recovery steps must be taken. The control unit, an ASOCS, can both recognize the fault states and map these into recovery states which cause aircraft mechanisms to be utilized in the recovery. This mapping of fault state to recovery state is naturally defined in terms of rules. As new sensors and functionality are added, the control unit is easily modified by the addition of new rules. In this light, the ASOCS functions as a parallel *pattern recognizer* and *classifier*.

The whole class of problems which are currently processed on *Programmable Logic Arrays* are all based on boolean rules (truth tables) and are naturally amenable to the ASOCS structure. One example of this type of application is a fault isolation and recovery system prepared for the space shuttle which is fully described by boolean production rules [Hell84]. In the simulation chapter a detailed example of a similar NASA system used for flight control is discussed.

1.4.2. The Model as an Integrated Tool

In the preceding section, the ASOCS is discussed as a potential methodology for solving a class of applications. Another way in which it could be used is as an ***Integrated Tool*** working together with conventional machines. The ASOCS could be used either as a component in building up other machinery, or as a special purpose processor.

A first obvious use of the ASOCS is for building adaptive logic components. Most basic MSI or LSI devices can be built using an ASOCS, including applications currently accomplished by a PLA. The advantage in using the ASOCS is its flexibility and the fact that it automatically does the minimizing and configuring to the given application.

Let us consider a simple logic device. Assume that we want to build an encoder which maps n inputs into $\log_2(n)$ outputs, for example a 4 to 2 encoder. For each of the four input possibilities 2 rules could specify the output. The system would then function as a normal encoder. However, assume that we now want to add a reset pin. Then a new rule specifying a reset input and its corresponding output is given. If an enable pin, one which only allows the output to change when it is high, is desired, then this function is again described by a set of rules. The advantage of the scheme is that it is only necessary to specify rules, and the ASOCS will take care of making the system *consistent* and of doing all reconfiguration. Thus, a user only need know how to specify the desired function.

Of course, if the function is completely predetermined, then it is not necessary to use an ASOCS. However, one particular need is in *prototyping* and *testing*. If a high speed system is being built which is controlled by logic, then it can be built and tested using real time adaptive components. An ASOCS could be used for this prototyping. Once a final design is reached, the logic could be set into a fixed structure. This is much cheaper than having to build a custom logic device for each phase during system design.

Using feedback, ASOCS can also be used to build asynchronous sequential circuits. By adding a layer of flip-flops at the output of the ASOCS, it can be used to build finite state machines.

As mentioned above, the ASOCS could be used as a combinational special purpose processor together with other machinery. Figure 1.3 shows a possible integration of an ASOCS with a Von Neumann machine.

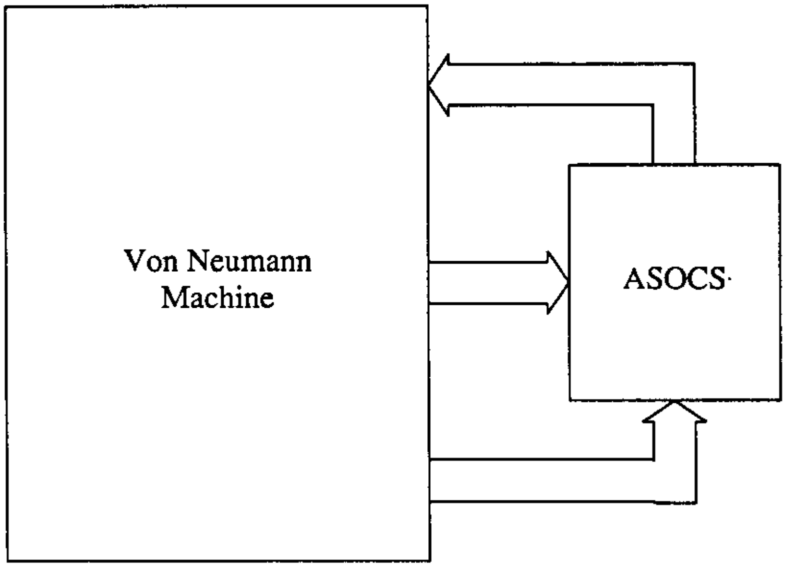


Figure 1.3 - ASOCS as Special Purpose Processor

In this case inputs and outputs come from and return to the Von Neumann machine, although they could be mixed with environmental inputs and outputs. The rules are fed from the Von Neumann machine. Thus, time intensive sections of computations could be computed in a concurrent network in an integrated fashion within sequential software. The ASOCS function could be *specified* by applications programmers who need high speed concurrency to build practical applications. Any number of ASOCS could be allocated to or contained in a Von Neumann system to be used as tools. Thus, in a sense, the programmer would have the ability to actually configure the hardware on which he works to a structure which naturally fits the current problem.

In this environment, one utilization example would be a *truth-maintenance* system for an AI knowledge base system. Since some ASOCS algorithms use the network to keep the knowledge base *consistent* and *minimal* they provide a concurrent method to solve the inherently slow problem of truth-maintenance.

1.5. Layout of Dissertation

In this section we discuss the layout of the complete dissertation, including the contents of each chapter, and how the chapters correlate.

Chapter 2, *Combinational Architectures of Programmable Nodes*, reviews work done with with both fixed and adaptive networks composed of programmable computing elements. Models are discussed which have led to or have similarities to the current research model. Similarities and differences between this research and other efforts are discussed.

Chapter 3, *Knowledge Base*, describes the way in which knowledge is incrementally input to the system by means of boolean rules. The growth and maintenance of the rule base as a *consistent* and *minimal* structure are also discussed.

The goal of chapter 4, *System Architecture*, is to define the basic modules, together with their functionality, which make the foundations of the model. This chapter also defines communication capabilities needed between the defined modules. Some implementation issues are also mentioned.

Chapter 5, *Primitive Mechanisms in ASOCS Systems*, explains the basic mechanisms, or primitives, which are the building blocks out of which ASOCS systems can be built. There are four basic mechanisms discussed in this chapter which are used in different forms in the three different algorithms presented in detail in the following chapters. Fault tolerance is also discussed.

Chapter 6, *Adaptive Algorithm 1 (AA1)*, is the first of three chapters discussing detailed ASOCS algorithms. These three chapters do not cover all possible methods of using the architecture and functionalities defined in the previous chapters. Rather they present three effective methods which could be implemented. There are a number of extensions to each of the algorithms and a few of the most important are mentioned within the chapters. Adaptive algorithm 1 is probably the least efficient as a physical implementation, yet it contains many interesting features not found in the other algorithms.

Chapter 7, *Adaptive Algorithm 2 (AA2)*, details the second adaptive algorithm which is quite different in its approach than AA1. It does not use all of the primitive mechanisms defined in chapter 5. It does, however, efficiently solve the basic problem discovered in AA1, that of memory growth within the network nodes.

Chapter 8 discusses *Adaptive Algorithm 3 (AA3)*, which is similar in many ways to AA2. It is however simpler in its approach than AA1 or AA2, and it solves some of the implementation difficulties inherent in AA2. Adaptive algorithm 3 is perhaps the most implementable of the three algorithms, although all three are realizable.

Chapter 9, *Algorithm Simulations*, describes simulations used to simulate the three adaptive algorithms. Statistics about the structure and efficiency of the network for both the learning and execution phases are given for the different algorithms. Worst case scenarios are shown and discussed. Also, a real world problem is detailed, together with its mapping into boolean rules and the structure of a network to solve it.

Chapter 10, *Future Directions*, discusses extensions and future research directions for the model.

1.6. Acknowledgements

I would like to acknowledge and express my appreciation for the inspiration and technical guidance of my advisor, Professor Jacques J. Vidal. I am also thankful for the many thoughtful comments and ideas of Tomas Lang, Rik Verstraete, and Donald Moore.

Chapter 2

COMBINATIONAL ARCHITECTURES of PROGRAMMABLE NODES

In this chapter we review some of the work on parallel networks composed of programmable nodes, with emphasis on aspects that lead to the current research. The first section discusses networks based on threshold gates. Much of the work on adaptive networks has been done in this arena. The next section discusses digital networks built from Boolean nodes. Very little work has been done on adaptive digital networks, but much effort has been put into methods of synthesis of fixed networks. The third section introduces the type of network element used in this research.

This chapter is by no means meant to be a complete survey of relevant topics. For a broad survey of highly parallel computing see [Hayn82]. One key feature of the networks mentioned in this chapter is that they function in a concurrent fashion.

2.1. Threshold Networks

Many of the studies on adaptive networks have used threshold gates as the basic "processor" [Stra61, Hans63, Hu65, Muro71, Weav75, Bobr78]. Figure 2.1 is a pictorial representation of a threshold gate. A threshold gate is a module which computes a linear weighted sum of its inputs $\{x_i\}$ to some threshold θ, to produce an output $z \ \varepsilon \ \{0,1\}$. Thus

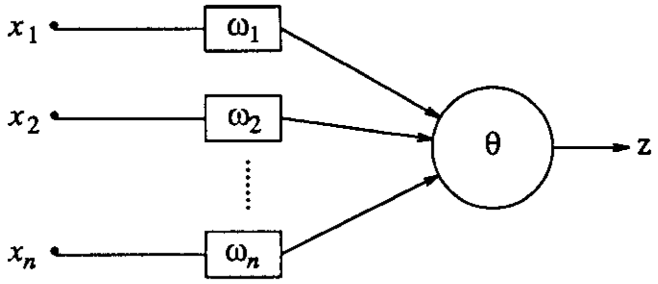


Figure 2.1 - Threshold Gate

$$z = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} \omega_i x_i \geq \theta \\ 0 & \text{if } \sum_{i=1}^{n} \omega_i x_i < \theta \end{cases}$$

where θ is a threshold value and ω_i are weights. Without loss of generality θ and ω_i can be limited to integers and x_i to the binary values $\{0,1\}$. By adjusting the weights and threshold, the threshold gate can compute different boolean functions of its inputs. Threshold gates are linear discriminators. They separate the boolean input vectors into two classes. However, threshold gates are limited to linearly separable classes, the percentage of which are very small for large n and the corresponding 2^{2^n} possible boolean functions. The two input threshold gate cannot compute all 16 of the possible boolean functions of two variables. A single threshold gate cannot be configured to compute the *exclusive-or* or the *equivalence* functions. A network of threshold gates, however, made large enough, can implement any boolean function. Any function which a threshold gate can compute can also be realized by a network of boolean gates.

Rosenblatt's perceptron was one of the earlier attempts to use threshold gates to simulate an adaptive neural network capable of learning boolean classifications [Rose58, Rose62a, Rose62b, Bloc62, Mins69]. In so doing, He laid the foundation for an important line of research to which the effort can be related. The perceptron in its simplest form is shown in figure 2.2.

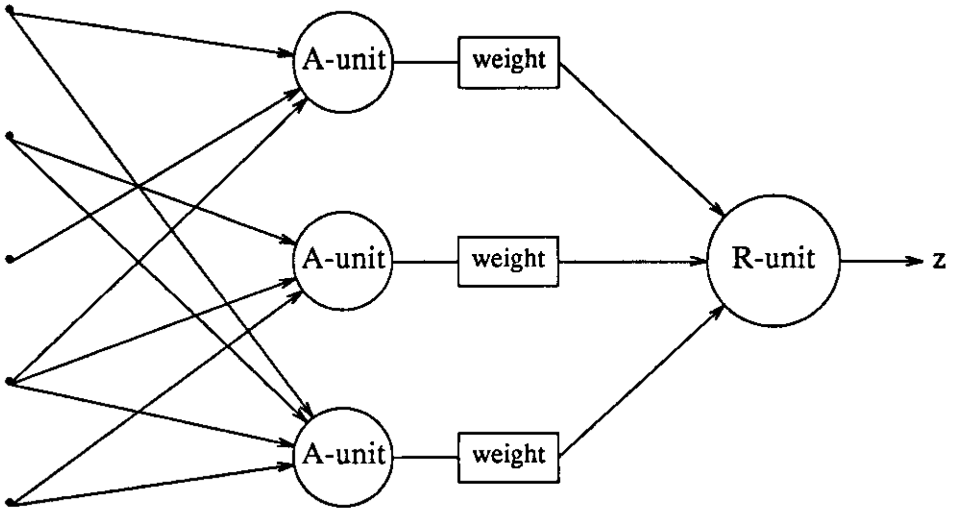


Figure 2.2 - Simple Perceptron

Each A-unit (Association unit) is a threshold gate with fixed weights. The inputs to the A-units come from sensors called S-units (Sensor units). These inputs are connected randomly to the set of A-units. The R-unit (Response unit) together with the weights represent a tunable threshold gate. Example patterns with their correct output are presented to the perceptron and the weights and threshold of the R-unit are modified to make the perceptron correctly classify the given input. Rosenblatt proved that there are learning algorithms which are guaranteed to cause the simple perceptron to classify a finite linearly classifiable set in finite time.

The Perceptron learning algorithms share some similarities with the current effort. Both are incremental in that desired classifications are given one at a time. In the convergence algorithms given by Rosenblatt it is assumed that the input examples are consistent (i.e. the output of a given example is always the same), as opposed to the current research where the desired output of an instance may be changed with time. Also, the modifications made to the weights after each example do not guarantee that the perceptron will function correctly; an example may have to be shown more than once. In the current model on the other hand, the network always solve the correct function immediately after each change. Another important difference is that the perceptron's network structure is usually fixed; only the weights of the inputs change. In this research the network can be dynamically restructured to absorb new information.

Much work on adaptive devices has been performed since the seminal work of Rosenblatt: [Widr64, Feld81, Bart81, Hint84]. Many are analog models in that the inputs and outputs to the system are analog values.

2.2. Boolean Network Synthesis

The major efforts in the area of digital networks has been in the development of methods for *synthesis* of optimal boolean circuits.

Implementation of boolean functions using minimal networks of gates has been studied in depth from the 40's until the present. Methods of minimization and decomposition of boolean functions continue to appear [Curt63, Kamm79, Roze79, Sing80, Cris80, Bran83, Roth83]. The motivation has been very practical. The desire has been to build optimal circuits with such criteria as the total number of computing elements, minimization of the longest path in the network, and minimization of fan-out of individual elements. Though there have been many algorithms presented to minimize the above costs, the majority of them break down as soon as the number of variables in the network becomes large ($n > 10$).

2.2.1. Universal Logic Modules and Multiplexer Trees

Most conventional network synthesis is aimed at networks built with fixed gates. However, around 1960, some researchers investigated the possibility of realizing more uniform structures with ULM's or *Universal Logic Modules* capable of producing all boolean functions of its inputs. By far the most studied building block for universal boolean networks is the multiplexer. A $2^n \times n$ multiplexer can be perceived as a universal boolean gate with n inputs. For example, a 4 by 2 multiplexer is a universal boolean gate of two inputs. This multiplexer can be configured to compute any of the 16 functions of 2 variables by placing appropriate values on the 4 multiplexed lines. The multiplexer in figure 2.3 computes the XOR function of the variables x_1 and x_2, where x_1 and x_2 are viewed as two input variables.

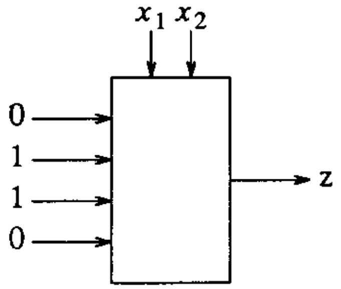


Figure 2.3 - Multiplexer

By choosing the values of the four multiplexed lines appropriately, the multiplexer can be set to any of the 16 boolean functions of two inputs.

The theoretical foundation for using the multiplexer, forthwith called a universal logic module (ULM), as a unit for building universal networks is Shannon's Theorem [Shan49].

$$f\ (x_1,x_2,\ldots,x_n) = x_1 f\ (1,x_2,\ldots,x_n) + \bar{x}_1 f\ (0,x_2,\ldots,x_n)$$

Using the above theorem, one layer of 2 by 1 ULM's can be used to decompose the original function into a function of one less variable. Thus n layers of 2 by 1 ULM's are sufficient to implement any boolean function of n variables. Shannons theorem can be expanded to work for any $2^n \times n$ ULM. In the case of the 4 by 2 ULM the equation is as follows:

$$\begin{aligned} f\ (x_1,x_2,\ldots,x_n) = x_1x_2 f\ (1,1,x_3,\ldots,x_n) + x_1\bar{x}_2 f\ (1,0,x_3,\ldots,x_n) + \\ \bar{x}_1x_2 f\ (0,1,x_3,\ldots,x_n) + \bar{x}_1\bar{x}_2 f\ (0,0,x_3,\ldots,x_n) \end{aligned}$$

The ULM network shown in figure 2.4 solves the boolean function:

$$\begin{aligned} f = \bar{x}_1\bar{x}_2\bar{x}_3x_4 + x_1x_2x_3x_4 + x_1\bar{x}_2\bar{x}_3\bar{x}_4 + x_1\bar{x}_2\bar{x}_3x_4 + \\ x_1x_2\bar{x}_3\bar{x}_4 + x_1x_2\bar{x}_3x_4 + x_1\bar{x}_2x_3x_4 \end{aligned}$$

There have been many algorithms expounded to synthesize ULM networks [Yau68, Ande69, Yau70]. Typically the criterion being optimized is again the number of modules necessary to implement a function. The ULM network shown in figure 2.4 is complete in that there is a stored bit in the network for every minterm of the function. Most boolean functions can be simplified such that not every minterm is needed. For example, the following functions are equivalent:

$$x_1\bar{x}_2 + x_1x_2 = x_1$$

The function computed by the ULM network of figure 2.4 can be simplified to the following:

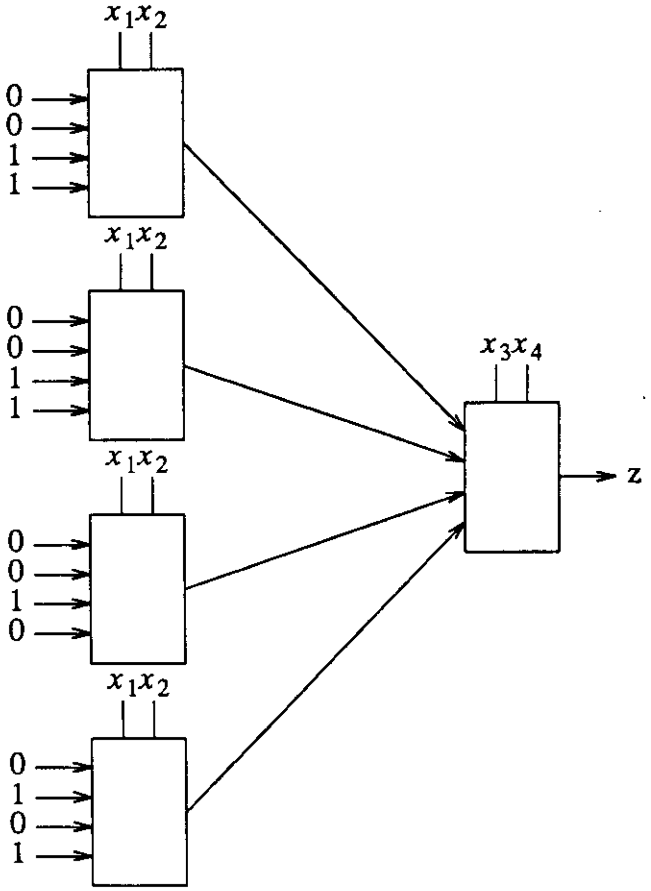


Figure 2.4 - ULM Tree

$$f = \bar{x}_1x_3 + x_1x_3x_4 + \bar{x}_1\bar{x}_2\bar{x}_3x_4$$

The network of figure 2.4 used 5 ULM's, while a more optimal ordering of the function variables produces a network with 3 ULM's as shown in figure 2.5.

2.2.2. Binary Decision Trees

Another form of the same class of structure is the binary decision tree (BDT) [Aker78, Cern79, More82, Quin83]. In a BDT each node of the binary tree tests a single boolean variable. Depending on the outcome of the test control is passed to one child or the other child. The testing starts at the root of the tree and proceeds towards

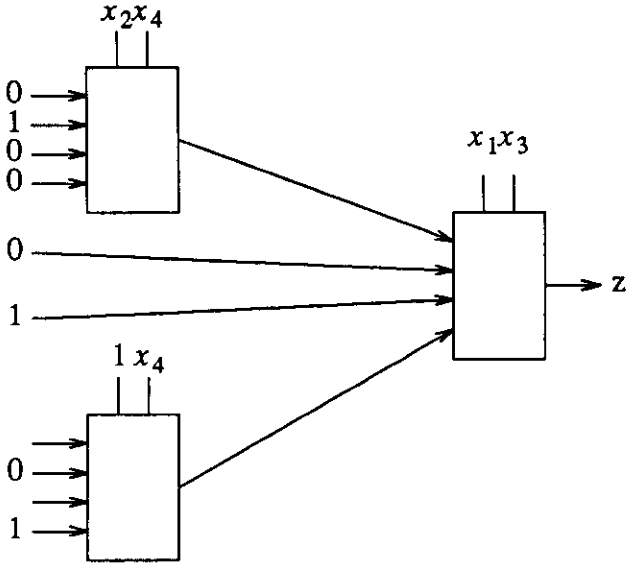


Figure 2.5 - Simplified ULM Network

the leaf nodes. One single leaf node is always selected as control passes through one path of the tree. Usually each layer of the tree tests one boolean variable, but like the multiplexer structure, an optimal ordering of the variables tested can allow for simplification of the tree structure. Thus a complete BDT will have a normal depth of n where n is the number of variables to be tested. The outputs of the function are found at the leaves of the tree.

Logically, a BDT is completely analogous to a ULM network and optimality algorithms for the multiplexer or BDT structures can be used interchangeably. Figure 2.6 gives the BDT representation of the ULM network of figure 2.4. Figure 2.7 shows the BDT form of the simplified ULM network of figure 2.5.

Close inspection of the multiplexer and BDT networks reveals that the structure is a memory addressing scheme where memory cells lie at the leaf nodes, and the input variables are addresses which are decoded to give access to one of the memory

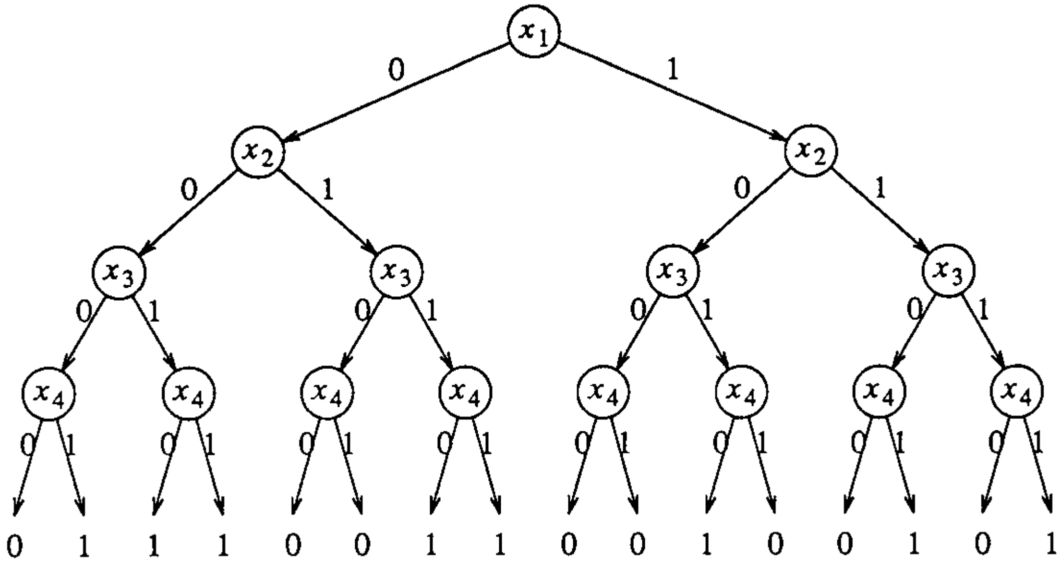


Figure 2.6 - Binary Decision Tree

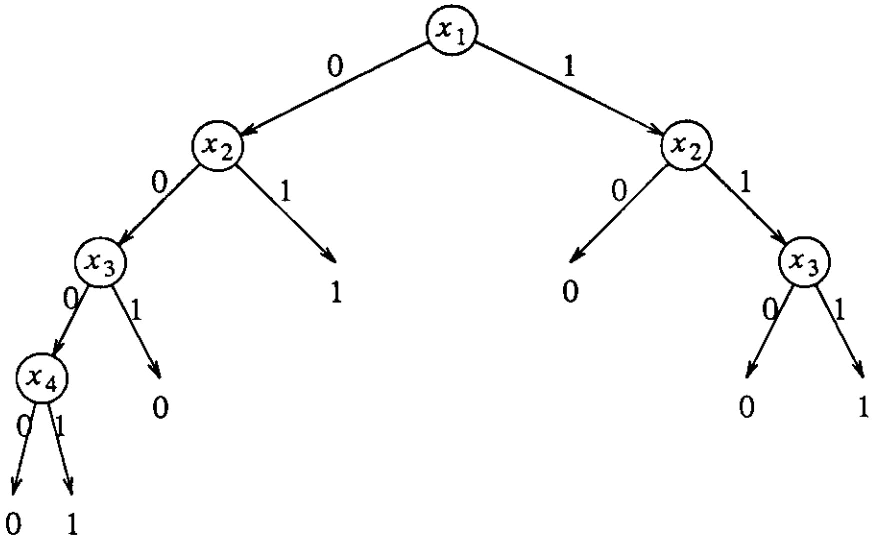


Figure 2.7 - Simplified Binary Decision Tree

cells. Thus, except for some amount of optimization, there must be an actual binary memory cell for every possible output of a given set of function variables.

2.2.3. Adaptive Digital Networks

There have been some attempts to model adaptive digital systems that perform recognition on boolean patterns: [Alex68, Holl78, Arms79]. In all of these systems, in contrast with this research, the control algorithms are centralized. It is also the case for most current adaptive and synthesis algorithms, that the methods can only be used for a specific subset of boolean functions. The current research applies to all classes of boolean functions.

By contrast, one distributed computing model which shows some similarities to the our effort is the connection machine [Hill84]. Although the goals and applications of the connection machine are more general and less defined, they both use *broadcast* and *self-organization* as a paradigm for node interconnection, rather than external control.

2.3. Dynamic Programmable Logic Module

A recent variation of the ULM is called the Dynamic Programmable Logic Module (DPLM). The DPLM is not one specific architecture, but a family of programmable universal logic gates. DPLM's have been under study at UCLA, with a special emphasis on a specific network of these modules known as the general purpose perceptron (GPP) [Moor83, Vida83, Vers83]. In this research, the DPLM is a 2 input gate which can be dynamically set to realize any of the 16 boolean functions of 2 variables.

Previous work done at UCLA involving the DPLM has been based on taking a fixed network of DPLM modules and changing the functions of different modules until the network is able to compute a pre-defined boolean function [Mart83, Vers86].

At the single module level a DPLM can be considered as a ULM rotated 90 (Figure 2.8).

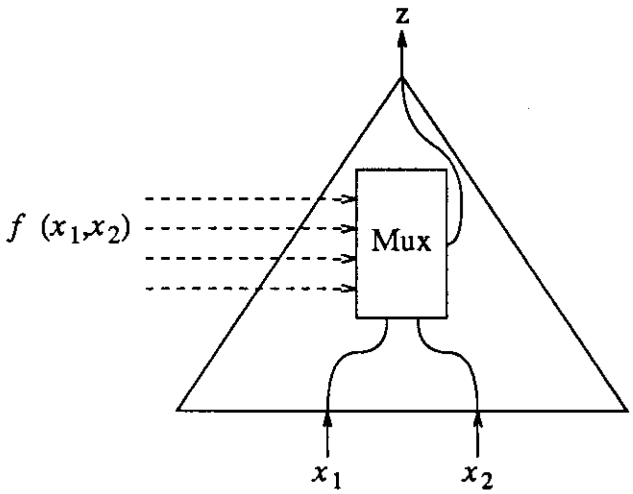


Figure 2.8 - DPLM as a Multiplexer

At the network level, however, the similarity ends. Figure 2.9 shows a picture of a DPLM network which can realize any function of 3 variables. This particular network has been studied in fair detail at UCLA and is known as the GPP-3. All function variables are input at the bottom of the network. There is no longer a natural mapping of layers to function variable, since all layers of the GPP-3 work on all variables available. If we consider the DPLM network as a memory which outputs a unique bit for any given set of variables, we see that the memory is a property of the *functions* and *connections* of the nodes in the entire network. This is different from the ULM networks where there is an explicit location in which each bit of memory can be found. In a DPLM network the memory is distributed amongst all the nodes of the network.

An important property of DPLM networks is their "enormous" redundancy. Any

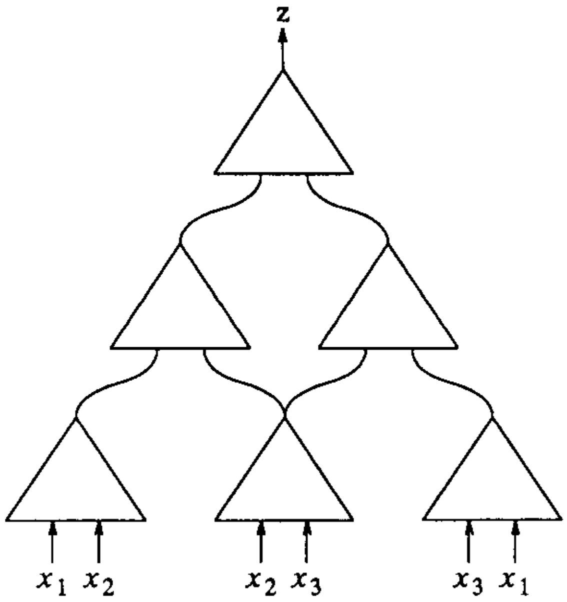


Figure 2.9 - GPP-3

network of DPLM's will have 16^k states, where k is the number of DPLM's in the network. For any set of n boolean variables there are 2^{2^n} possible functions of those variables. Thus, the GPP-3 can compute any of the 256 functions of 3 variables. Yet, the GPP-3 can be in any one of 16^6 or 16 million states, where each state maps to one of the 256 functions. For instance, there are 3,261,376 ways in which the GPP-3 can be configured to compute the simplest of the functions of 3 variables; i.e. *True* and *False*. There are 2496 ways in which it can compute the most difficult functions of the 3 inputs.

2.4. Conclusion

Progress has been slow with computing models based on adaptive networks with learning. This appears to stem from two basic problems.

1. The perceptron algorithms and its late variants deal with a *single*

layer, and cannot handle multi-layer structures.

2. Convergence learning requires a large amount of computing time before the correct function is guaranteed. (i.e. it is unknown whether an applied modification will aid or worsen the function of the network).

As will appear, these problems do not occur in this effort. The multi-layer problem becomes manageable when a self-organizing paradigm is used for control. There is no convergence problem, since the network, while adapting incrementally to each new data, is every time effectively programmed. In learning terminology, each input brings "one-shot" learning to the system.

Chapter 3

KNOWLEDGE BASE

In this chapter we discuss the knowledge base and the method of knowledge input into the system. The knowledge base is basically a rule set which describes what outputs the system should have for certain states of the environment. The first section gives an informal overview of the rule base. Section 2 gives a brief informal example of the incremental growth of the rules. The remaining sections give a more detailed and formal treatment of the different aspects of the knowledge base.

3.1. Informal Overview

The atomic knowledge element of the system is the *instance*, which is a boolean rule defining what the system should output when confronted with a given input. Each instance is an if-then statement where the antecedent is a conjunction of boolean variables, and the consequent is a single boolean variable.

An instance is a type of ***Production Rule,*** which production rules are used in many expert system applications. A production rule, like an instance, has an antecedent which can evaluate to either true or false. It is an implication of the form

$$\text{If } Condition_1, Condition_2, \cdots, Condition_n \text{ Then}$$
$$Action_1, Action_2, \cdots, Action_n$$

An instance is a special type of production rule where the consequent of the rule is a boolean variable. Thus, the instance is a lower level representation of a production rule. However, any production rule can be represented by some number of instances, where the sum of the instance consequents represent the production consequent, or where a number of boolean instance outputs are encoded to represent a multi-valued output. An instance can be thought of as a boolean *horn clause.*

Following are examples of instances.

$$X_1 X_2 \rightarrow Z_1$$

$$X_1 \bar{X}_2 X_3 \rightarrow \bar{Z}_2$$

$$\bar{X}_2 \bar{X}_3 Z_2 \rightarrow Z_3$$

Each variable in the instance is a boolean variable. The $\rightarrow$ is the boolean implication operator that states "if the consequent is true, then the antecedent must be true." Note that an output variable can be used in the antecedent of the instance. Thus, each instance can be viewed as a state transition in a finite state machine. Discussion of instances with output variables used in the antecedents, know as *feedback variables*, is given in a later section of this chapter.

It would also be reasonable to allow the consequent of an instance to be a conjunction of output variables. However, no generality is gained by allowing this, since an instance containing n variables in its consequent is equivalent to n instances containing one variable in each consequent.

Instances are input to the system by an outside agent *incrementally,* and the current totality of instances is called the *Instance Set.* The instance set is the knowledge base of the system and it is maintained *Consistent* and *Minimal.* By consistent it is meant that no two instances in the instance set can contradict each other; i.e. no two instances can specify an opposite output for the same input. Minimality means that the instance set is stored using a minimal representation. These terms are defined in detail later.

3.2. An Intuitive Example

We now go through an example of what the instance set represents, and how incremental addition to the instance set causes its overall representation to change. The purpose of this example is to give an intuitive overview of what the knowledge base represents and how it relates to the system.

Let us assume that we wish to control whether or not to open (use) or close an umbrella. There are initially no variables defined in the system. Variables become defined in the system when a new instance contains the variable. We specify the opening of the umbrella by the boolean output variable Z. The command to close the umbrella is controlled by $\bar{Z}$.

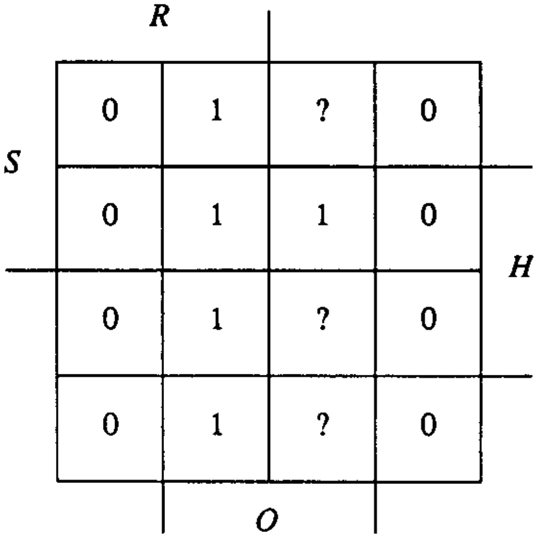
The first instance entered into the rule base is $R \longrightarrow Z$. R is active when it is raining and inactive when it is not raining. Thus, whenever it is raining the umbrella will be opened. However, it is soon seen that it is not necessary to open the umbrella if one is inside. The next instance input is $\bar{O} \longrightarrow \bar{Z}$, where O represents if one is outside. This instance *contradicts* the first instance because it is possible for it to be raining while one is inside. The first instance is modified so that it represents all that it did before, with the exception of the new constraint implied by the new instance. The first instance will be changed to $R\ O \longrightarrow Z$ in order to make the set consistent. The instance set then appears as follows.

$$R\ O \longrightarrow Z$$

$$\bar{O} \longrightarrow \bar{Z}$$

The next instance states that if one is outside and it is *sunny* and *hot* then one should also open the umbrella. This instance is given as $S\ H\ O \longrightarrow Z$ where S and H represent sunny and hot respectively. This new instance does not contradict any instance in the set. Note that a new instance can only contradict an instance which gives the opposite output. This instance is simply added to the set. To aid in the understanding of what the current instance set represents, figure 3.1 gives a Karnaugh map representation of the 16 possible environment states represented by the 4 defined input variables. Z is represented by a "1" and $\bar{Z}$ by a "0". The instances in the set only imply what the output should be for those states of the environment which match the instances. They say nothing about those states which are not matched by any instance, and those are represented by "?", signifying a *don't know* state.

Thus, the instance set represents a *partial boolean function* of the inputs. The instance set represents a complete 3-state function of the input variables. Note that an instance does not describe a current state of the environment, rather it specifies how the system should change the environment when the instance is matched. Thus, the rule set defines a *temporal logic* function.



0	1	?	0
0	1	1	0
0	1	?	0
0	1	?	0

Instance Set

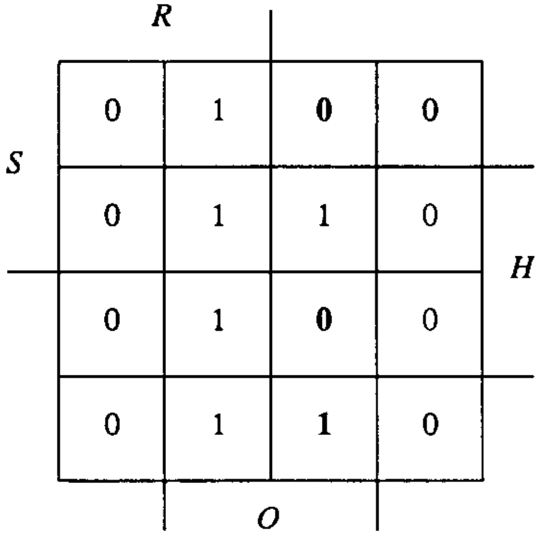
$R\ O \rightarrow Z$
$\overline{O} \rightarrow \overline{Z}$
$S\ H\ O \rightarrow Z$

Figure 3.1 - Instance Set Representation

It is found that due to a skin condition the umbrella's owner should open it any time it is sunny and the location is outside. The new instance is $S\ O \rightarrow Z$. This instance does not contradict any instance, but it does cause minimization. After addition of the new instance, the old instance $S\ H\ O \rightarrow Z$ will be deleted, since its conditioned is covered by the new instance. Minimization can only occur between instances implying the same output. Minimization can never change the consistency of the set, since minimization only causes a change in the *representation* of the same knowledge.

3.2.1. Rule Base and Network Function

A logic network fulfills the current instance set when for any set of inputs which match an instance in the instance set, the network outputs the variable as defined by that instance. When for a given input, no instance in the instance set is matched, the logic network may output either 1 or 0. Although the instance set represents a 3-state knowledge space, a boolean logic network always computes a complete 2-state function. Figure 3.2 shows a possible complete function computed by a logic network which would fulfill the instance set as it was represented in figure 3.1.



0	1	**0**	0
0	1	1	0
0	1	**0**	0
0	1	**1**	0

Instance Set

$R\ O \rightarrow Z$

$\bar{O} \rightarrow \bar{Z}$

$S\ H\ O \rightarrow Z$

Figure 3.2 - Possible 2-State Implementation

The cells which used to contain don't knows, now contain either a 1 or a 0. There are any number of ways to configure a logic network to fulfill a given instance set. The main effort of this research is to show how to modify a given logic network, after the instance set is modified.

3.3. Instances

For ease of explanation we make two assumptions which will hold for the following sections. First, during this initial discussion, we assume that output variables do not occur in the antecedent of an instance. A section at the end of this chapter discusses this concept in detail, and shows that the same concepts as here explained carry over for the case of output variables occurring in the antecedents of instances. Second, we assume that there is only one output variable being referenced by the example instances. We do not lose generality under this assumption because instances implying different variables are treated separately anyway. Consistency and minimization occur only between instances which imply the same variable. It is equivalent to have n instance sets, where n is the number of output variables, or just have one instance set containing instances implying n different output variables.

The relationship between the number of input variables and the number of output variables is dependent upon the application. There can be any number of output variables corresponding to input variables, although there can only be 2^{2^n} unique output variables for n input variables.

It is assumed that there is an ***Environment*** made up of some number of boolean features. These features are represented as boolean variables arbitrarily labeled by letters A, B, etc. These variables are called ***Input Variables.*** There are also some number of boolean ***Output Variables*** in the environment. In this section we limit the environment to one output variable Z. In a general system, the number of input and output variables in the environment can change with time. Any time a new instance is introduced which contains variables heretofore not defined, these variables become part of the environment. In fact, the only way in which input or output variables are defined as part of the environment is when they first appear in an instance given to the system.

Knowledge, or the rule base, is input to the system in the form of ***Instances.*** An instance is a logical implication in which the antecedent is a conjunction of input variables and the consequent is a single output variable. The variables in the antecedent of an instance are a subset of the variables in the environment. At any given time, each of the variables in the environment has the value "1" or "0". The values of all variables at a given time is called the ***State of the Environment.***

A variable in the antecedent is ***Active*** for that particular instance if the value of the variable in the current state of the environment is 1 for a non-negated variable in the instance, or 0 for a negated variable.

A ***Positive Instance*** is an instance with a consequent of Z, and a ***Negative Instance*** is an instance with a consequent of $\bar{Z}$. Recall that we are assuming that only the single output variable Z is defined for this environment.

An instance is ***Matched*** by the environment state when all variables in the antecedent are active for the current state of the environment. Any variable not explicitly specified in an instance is considered as a ***Don't Know*** variable for that instance, and does not effect whether the antecedent of the instance is matched.

Instances, as defined, are a *logically complete* method of function description, in that any boolean function can be represented by a set of instances. However, the natural use of instances is in the representation of partial functions, rather than complete boolean functions. Instances are rules and imperative by nature which make them a very natural knowledge element for this type of application. The instance $A \longrightarrow B$ represents the same knowledge as the boolean construct $\bar{A} + B$, where + is the *Or* function. The boolean construct, however is not intuitive and is only understood when expanded into a complete truth table for all possible permutations of the input. The instance is naturally incomplete, in that one need only consider the single state when the antecedent is active.

3.4. Instance Set

In this section we discuss the relationships between instances. These relationships only occur between instances defining the same output variable. The instances $A \longrightarrow Z_1$ and $B \longrightarrow Z_2$ are in no way related.

For any two instances, a variable which occurs in one of the instances, but not in the other, is a ***Disjoint Variable*** in terms of those two instances. A variable occurring in both instances is a ***Shared Variable*** in terms of the two instances.

For any two instances, a shared variable can either be a ***Concordant Variable*** or a ***Discordant Variable.*** A concordant variable is one which is not negated in either instance, or negated in both instances. A discordant variable is one which is negated in one instance and not negated in the other instance. A single instance cannot contain both the negated and non-negated form of a variable. Following is an example of this.

$A\ B\ \bar{B} \rightarrow Z$

The above instance could never be matched in an environment, and is not allowed.

Any two instances are ***Concordant Instances*** if they are both positive instances, or both negative instances. Any two instances are ***Discordant Instances*** if one is a positive instance and one is a negative instance.

Any two instances which are both matched for some state of the environment are called ***Overlapping Instances.***

An ***Instance Set,*** as here defined, is a set of instances which fulfills the assumptions of ***Consistency*** and ***Minimization.***

For all proofs regarding instances we note that only the conjunctions of variables in the antecedents of instances are compared. The proofs derived are all about the relationships of the antecedents of the instances within an instance set, and boolean theorems and identities will be used in proving these relationships.

3.4.1. Instance Set Consistency and Minimization

A consistent and minimal instance set is represented as Ω which is initially the empty set ϕ. Ω is divided into two disjoint sets Ω_p and Ω_n. Ω_p contains all the positive instances of Ω and Ω_n contains all the negative instances. Each of these sets can be represented as Ω_t, where t is the ***Polarity*** of the set and t can be replaced by either p or n. The set $\Omega_{\bar{t}}$ is the ***Discordant Set*** to Ω_t. Any set representation with a polarity subscript contains only concordant instances. It thus follows that the members of the polarized sets need only be products of boolean variables, without necessity to include the implication operator and the consequent value.

A consistent but possibly non-minimal set is represented as Λ_t.

Each instance in a set is represented as β_i, where i ranges from 1 to the number of instances in the set. Each β_i takes on the type of the set which it is a member.

A new instance which is to be added to the IS is represented as α_p or α_n depending on the type of the new instance.

When a new instance (NI) is added to the IS a new IS is created by application of the consistency operator μ, and the minimization operator γ. The operators μ and γ are meant as proof of theory operators and do not necessarily reflect the manner in which these methods are implemented.

3.4.2. Consistency Operator

The consistency operation is

$$\mu(\alpha_t, \Omega_{\bar{t}}) = \Lambda_{\bar{t}}.$$

A new instance is only applied to the discordant set of instances. An instance cannot contradict a concordant instance.

μ is broken down into the operations of expansion ε and deletion ψ.

$$\varepsilon(\alpha_t, \Omega_{\bar{t}}) = \Lambda_{\bar{t}}'$$

$$\psi(\alpha_t, \Lambda_{\bar{t}}') = \Lambda_{\bar{t}}$$

Both operations are parallel in that the α_t can be simultaneously applied to each β in the discordant set.

The expansion operation is applied first and it is appled to each β in the set as follows.

$$\varepsilon(\alpha_t, \beta_{\overline{ti}}) \leftarrow \begin{cases} \textit{If discordant variable} & \beta_{\overline{ti}} \\ \textit{If no discordant variable} & \varepsilon(\alpha_t, \beta_{\overline{ti}} \,||\, \bar{d}), \varepsilon(\alpha_t, \beta_{\overline{ti}} \,||\, d) \end{cases}$$

where d is a single variable which occurs in α and not in β_i. Recall that a discordant variable is a shared variable which is negated in one of the instances. ← signifies that the β will be replaced by the right hand side of the equation. The *if-part* decides whether there is a discordant variable between α_t and $\beta_{\overline{ti}}$. The | | operator causes the concatenation of of a variable to the given β. ε is a recursive function.

The expansion operator is followed by the deletion operator which is applied to every instance in the new set Λ_t' as follows.

$$\psi(\alpha_t, \beta_{\overline{ti}}) \leftarrow \begin{cases} \textit{If discordant variable} & \beta_{\overline{ti}} \\ \textit{If no discordant variable} & \phi \end{cases}$$

Thus, the new instance set is consistent since:

1. There are no discordant instances which contradict the NI, since all instances without a discriminant variable are deleted by the ψ operator.
2. Ω_t was already consistent with all $\Omega_{\bar{T}}$ before application of μ.

A ***Consistent*** set of instances is one in which no two instances ***Contradict*** each other. Two instances contradict each other if and only if for some state of the environment, they are both matched, and they are discordant instances. Thus, no overlapping discordant instances can exist in an instance set. Following are three inconsistent sets of instances. Each is followed be an example environment state for which the instances contradict each other. (Note that they are not instance sets by definition, since they are inconsistent.)

$A\ B \longrightarrow Z$ (set 1)

$A\ B \longrightarrow \bar{Z}$

$A = 1, B = 0, C = 1$ Environment State

$A\ \bar{B} \longrightarrow \bar{Z}$ (set 2)

$\bar{B}\ C \longrightarrow Z$

$A = 1, B = 1$ Environment State

$A \longrightarrow Z$ (set 3)

$B \longrightarrow \bar{Z}$

$A = 1, B = 1$ Environment State

Theorem: Contradictions occur when, for two discordant instances, no discordant variables occur.

For the proof of the above theorem, we need only show that two antecedents of instances with no discordant variables, can for some value of the variables both be matched.

We prove this by contradiction. Assume two instances, $X \longrightarrow Z$ and $Y \longrightarrow \bar{Z}$ where X and Y represent any two legal antecedents which contain no discordant variables. Since any variables not occurring in the first instance are don't know variables for that instance, the instance $X\ Y \longrightarrow Z$ follows from the first instance because all variables in Y are don't know variables or concordant variables with X. But by the same argument, the instance $Y\ X \longrightarrow \bar{Z}$ follows from the second instance. These new formed instances contradict each other for all states of the environment for which they are matched.

There are in fact, 2^{n-u} expansions of the two instances with no discordant variables which overlap, where u is the number of concordant variables in the two instances and n is the number of variables in the environment.

A corollary to the above theorem is that all instances of one polarity contain at least one discordant variable for each discordant instance.

Another name for a discordant variable is a ***Discriminant Variable.*** All other variables contained in the two instances are called ***Non-Discriminant Variables.*** There can be any number of discordant variables between two instances, but one discordant variable is sufficient to ***Discriminate*** between the two instances. Any two instances which are discriminated can never contradict each other for any state of the environment.

3.4.3. Minimization Operator

Before the minimization operation γ is applied, α_t is added to Ω_t. γ is applied to both Ω_t and $\Omega_{\bar{t}}$. γ only needs to be applied to the set discordant to the NI if that set was modified during the μ operation.

The minimization operation is defined as

$$\gamma(\alpha_t, \Lambda_t) = \Omega_t .$$

Minimization can only occur between concordant instances. Minimization can never cause new inconsistencies.

Minimization of boolean *sum-of-products* forms is well established in the literature and the proofs will not be reproduced here. γ is based on repeated application of the following 3 identities between all combinations of β in the set:

1. $x + xy = x$
2. $xy + x\bar{y} = x$
3. $x + \bar{x}y = x + y$

The resultant minimal set always has the least possible number of instances, and no instance in the set is reducible to an instance with less variables. However, the minimal set is not unique, and there may be many equivalent sets having the same number of instances with differing variable combinations.

The instance $A\ B \longrightarrow Z$ is really a simplified representation of many possible instances. If the environment consists of the variables A, B, C, and D, then the one instance $A\ B \longrightarrow Z$ is a representation for the following set of instances.

$$A\ B\ C\ D \longrightarrow Z$$

$$A\ B\ C\ \bar{D} \longrightarrow Z$$

$$A\ B\ \bar{C}\ D \longrightarrow Z$$

$$A\ B\ \bar{C}\ \bar{D} \longrightarrow Z$$

The exact same information is given by the single instance as is given by the set of four ***Fully Expanded*** instances. The total instances represented by a single instance is the ***Representation Set*** of the instance.

A set of instances is ***Minimal*** if no two concordant instances can be equivalently represented by one instance, or by two instances with fewer variables.

For a state of the environment, it is possible to have more than one instance matched, as long as all the matched instances are concordant instances. For example, $A \longrightarrow Z$ and $B \longrightarrow Z$ are minimal with respect to each other, and both would be matched in an environment state where A and B were both equal to 1.

3.5. Addition to the Instance Set

In this section we define new terms describing ways in which the addition of a new instance causes modification to the instance set. Although a formal discussion of this concept has already been given, it is necessary to decompose this process into specific categories which will be used in later chapters which describe the algorithms used on the ASOCS system.

Knowledge is continually given to the system by the ***Incremental*** addition of new instances. Instances are input one at a time. When an instance is input to the system it is known as the ***New Instance (NI).***

The addition of new instances to the instance set allows the overall system to be ***Adaptive,*** in that its overall function changes in time.

When a NI is added to the instance set (IS), it is necessary to insure that the IS is both consistent and minimal. Order of instance input can then influence the outcome. When there is a contradiction, one of the instances in the contradiction must be deleted from the instance set. A ***Precedence*** must be given to the instances if a deterministic methodology is desired. It is decided that highest precedence is given to the newest instances. Once the IS is consistent, there is no need to remember chronological precedence amongst the instances in the set. This follows from the assumption of consistency. When the set is consistent, each instance is independent of each other, and none has precedence.

The method of presentation here given is geared towards making explanations in future chapters simpler. In order to do this we give a summary table of the types of changes which will occur in an instance set, when a certain type of new instance is input. There are three basic categories of IS change occurring between instances. They are contradiction, *normal* minimization, and *one-difference* minimization. Normal minimization occurs between instances containing no discriminant variables (i.e. $x + xy = x$). One difference minization refers to the two identities:

$$x + \bar{x}y = x + y$$

$$xy + \bar{x}y = y$$

In terms of these types of changes, the relationship between the NI and a single OI can be categorized as:

1. Subset - The NI is a subset to the OI.
2. Equal - The NI and OI share all of the same variables.
3. Superset - The NI is a superset to the OI.
4. Overlap - The NI and the OI overlap.
5. Discriminated - There is at least one discriminant variable in the NI and OI.

Discriminated instances will never participate in any change when a NI is added, and thus this category is not included in the table below. Table 3.1 gives a brief summary of the IS modification necessary for the different permutations of the above features.

	Subset	Equal	Superset	Overlap
Contradiction	Delete Old Instance	Delete Old Instance	Do *Overlap Modification*	Do *Overlap Modification*
Normal Minimization	Delete Old Instance	Delete New Instance	Delete New Instance	No Change
One Difference Minimization	Minimize the Old Instance	Replace the Old and New Instances with a Minimized Instance	Minimize the New Instance	No Change

Table 3.1 - Instance Set Modification in terms of the NI Type

In the table it is assumed that contradiction is considered only between discordant instances, and minimization between concordant instances. A little liberty is taken with the category terms. An equal one-difference instance really contains one discriminant variable. Also, subset equal, and superset instances are subsets of overlap instances. Overlap modification is defined below.

The following subsections detail each type of change which can be made to an IS for all classes of NI to IS interaction.

3.5.1. Simple New Instance Addition

If the NI does not contradict any ***Old Instance (OI),*** and it cannot be minimized with any OI, then it is simply added to the IS with no further change. The network may already fulfill the NI, or it may have to be modified.

3.5.2. New Instance Consistency

When a NI contradicts an OI in the IS, there is always a change made to the IS. The NI, in this case, is added without modification since it has precedence. A NI may contradict any number of OI's. Once the IS has been made consistent it may be necessary to minimize the set. The NI in this case cannot not be minimized. There are only two unique ways in which contradiction can occur upon input of a NI These are explained in the following two subsections.

3.5.2.1. Subset Contradiction

When the NI is a subset or equal instance to an OI, the OI is deleted from the IS. This type of contradiction is called ***Subset Contradiction***. An example of subset contradiction follows.

$$A\ B\ C \longrightarrow \bar{Z} \qquad \text{(OI)}$$

$$A\ B \longrightarrow Z \qquad \text{(NI)}$$

In this case the OI is removed and the NI added. This is equivalent to the deletion operator ψ.

3.5.2.2. Overlap Contradiction

When an OI overlaps with the NI and is not a subset of the NI, ***Overlap Contradiction*** takes place. In this case the OI is removed, but one or more modifications of the OI are added to the IS. The change to the IS is such that the IS includes the intersection of the representation sets of the OI and the NI. Thus, the added instances are always of the form of the OI concatenated with one discriminant variable of the NI. Following is an example.

$A\ B \longrightarrow Z$ (OI)

$A\ C\ D \longrightarrow \bar{Z}$ (NI)

$A\ C\ D \longrightarrow \bar{Z}$ Consistent and Minimal Set

$A\ B\ \bar{C} \longrightarrow Z$

$A\ B\ \bar{D} \longrightarrow \bar{Z}$

The modified OI instances added to the set are called ***Overlap Modifications.***

By adding one instance for each discriminant variable in the NI the set is made immediately minimal. Thus n overlap modifications are added for each overlap instance, where n is the number of discriminant variables which occur between the NI and the OI. Note that the key point is that there must be at least one discriminant variable between all discordant instances.

As shown in the formal description, it is also be correct to add 2^n-1 overlap modifications to the IS, where n is the number of discordant variables between the OI and the NI. This is equivalent to the expansion operator ε. Following is an example.

$A \longrightarrow Z$ (OI)

$A\ B\ C \longrightarrow \bar{Z}$ (NI)

$A\ B\ C \longrightarrow \bar{Z}$ Consistent Set

$A\ B\ \bar{C} \longrightarrow Z$

$A\ \bar{B}\ C \longrightarrow Z$

$A\ \bar{B}\ \bar{C} \longrightarrow Z$

This set would then be minimized such that there would be two overlap modification instances each with one discriminating variable. The final set is the same regardless of the method used.

Note that a NI which is a superset to a discordant OI is also a type of overlap contradiction.

3.5.2.3. Contradiction and Network Function

An important point about contradiction modifications is that only the NI can cause any change to be made to the current network. Any instances deleted are trivially fulfilled and all overlap modifications are already fulfilled by the network. This is because by definition the network already fulfilled any OI's which were contradicted by the NI. The representation sets of the overlap modification instances are subsets of the representation sets of the OI's which were removed. Thus, the overlap modifications must still be fulfilled by the current network.

3.5.3. New Instance Minimization

If a modification was made to either Ω_t or Ω_T, by the μ operation, or by addition of the NI, then it is possible that minimization must take place within the respective sets. As specified earlier, precedence and order do not effect minimization. If two instances are not minimal with respect to each other, the change to the IS will be the same regardless of which instance is the NI. Again, this is because upon addition of the NI to the IS, all information is consistent, only the representation, not the content, of that information is to be changed. However in terms of the algorithms used to modify the logic network, it is necessary to distinguish which instance is the NI.

For use in the later chapters on algorithms, five categories of minimization are defined. The first two are types of *normal* minimization, and the last three are types of *one-difference* minimization.

3.5.3.1. New Instance Superset Minimization

If the NI is a superset or equal instance to an OI, then the NI is removed from the IS. ***Superset Minimization,*** (and ***Subset Minimization,*** which is explained in the next section), is based on the boolean identity

$$x + xy = x.$$

The NI may be a superset instance to more than one OI, but it is a sufficient condition that there be just one, for removal of the NI from the IS. When a NI is a superset instance, there is never a need to change the network, since the IS is not changed.

Following is an example of this kind of minimization. In the examples which follow, an initial consistent and minimal IS is assumed. The OI's are numbered and the NI is labeled as such.

$$A \longrightarrow Z \qquad (1)$$

$$B \longrightarrow Z \qquad (2)$$

$$A\ B \longrightarrow Z \qquad (\text{NI})$$

Either OI 1 or 2 is sufficient to cause removal of the NI.

3.5.3.2. New Instance Subset Minimization

If the NI is a subset instance to an OI then the OI is removed from the IS. A NI may be a subset to any number of OI's, and all of those OI's are removed from the IS.

$$A\ B\ \bar{C} \longrightarrow \bar{Z} \tag{1}$$

$$B\ \bar{D} \longrightarrow \bar{Z} \tag{2}$$

$$A\ E \longrightarrow \bar{Z} \tag{3}$$

$$A \longrightarrow \bar{Z} \tag{NI}$$

In the above example, OI's 1 and 3 would be deleted from the IS. OI 2 would not be deleted because for subset minimization there must be a non-null subset. The NI would be added to the IS. For this type of minimization, the network may or may not have to be changed.

3.5.3.3. New Instance One-Difference Equal Minimization

One-Difference Minimization refers to the case when two concordant instances contain one discordant variable, and one instance is a subset of the other. When the instances are proper subsets, the minimization is called ***One-Difference Equal Minimization,*** and it is based on the boolean identity

$$xy + x\bar{y} = x$$

In this case, the two instances are replaced by one instance without the discriminant variable.

The NI may match up with a maximum of n OI's at once in this manner, where n is the number of variables in the NI. This follows from the fact that in one-difference minimization only one discriminant variable can occur in any OI, together with the assumption that the IS is minimal before the NI addition. Below is an example.

$$A\ \bar{B}\ C \longrightarrow Z \qquad (1)$$
$$A\ B\ \bar{C} \longrightarrow Z \qquad (2)$$
$$A\ B\ C \longrightarrow Z \qquad (\mathrm{NI})$$

In the above example both 1 and 2 can be minimized with the NI. The NI and 1 reduce to $A\ C \longrightarrow Z$, and the NI and 2 reduce to $A\ B \longrightarrow Z$. The new IS would be made up of these two instances.

3.5.3.4. New Instance One-Difference Subset Minimization

A NI, fulfilling one-difference requirements, which is a subset instance, may be minimized with any number of OI's simultaneously. ***One-Difference Subset Minimization,*** (and ***One-Difference Superset Minimization,*** which is explained in the next section), is based on the identity

$$x + \bar{x}y = x + y$$

Following is an example.

$$A\ \bar{B}\ C\ D \longrightarrow \bar{Z} \qquad (1)$$
$$\bar{A}\ B\ D \longrightarrow \bar{Z} \qquad (2)$$
$$\bar{A}\ \bar{B}\ C \longrightarrow \bar{Z} \qquad (3)$$
$$A\ B \longrightarrow \bar{Z} \qquad (\mathrm{NI})$$

The NI is added as is to the IS. Instance 1 is reduced to $A\ C\ D \longrightarrow \bar{Z}$, and 2 is reduced to $B\ D \longrightarrow \bar{Z}$. 3 cannot be reduced since there is more than one discordant variable between it and the NI.

3.5.3.5. New Instance One-Difference Superset Minimization

When a NI, fulfilling one-difference requirements, is a superset to an OI, it may simultaneously be minizable with less than 2^n instances in that category, where n is the number of variables in the NI. In this case only the NI is modified. This is a case where the minimization can be non-unique. In the example below the NI is a one-difference superset to both of the OI's. Although it could be minimized with either one, once it has been minimized with one, it cannot be minimized with the other. The NI can be modified to $B\ C\ D \longrightarrow Z$ with 1, and to $\bar{A}\ C\ D \longrightarrow Z$ with 2. Either representation is minimal.

$$A\ B \longrightarrow Z \tag{1}$$
$$\bar{A}\ \bar{B}\ D \longrightarrow Z \tag{2}$$
$$\bar{A}\ B\ C\ D \longrightarrow Z \tag{NI}$$

The above example demonstrates that the minimal IS in not a unique IS, although it always represents the same knowledge. The non-uniqueness of minimal boolean representations is a well established fact in boolean algebra.

3.5.3.6. Algorithmic Issues of Instance Set Minimization

Following are a couple of points about NI minimization which can be useful in terms of algorithms used to minimize the IS when an NI is presented. Two basic methods of accomplishing minimization are here discussed.

The first way is to compare the NI against all OI's simultaneously. This is the parallel method of minimization. It is also equivalent to the algebraic method, where all of the instances, including the NI, are put into one large boolean equation and minimized using methods of boolean algebra. Two of the defined algorithms do a parallel maintenance of the instance set in the network.

The other method is the sequential method which compares the NI individually against each OI and minimizes the NI only against one given OI at a time, by repeated applications of the three boolean identities shown in an earlier section. This is the natural method for programming on a Von Neumann machine, and this method is used in the first defined algorithm. When using a sequential algorithm, one which tests the NI individually against each OI, to minimize the IS, the following considerations occur.

3.5.3.6.1. Mutual Exclusivity of Minimization Categories

When an NI is initially added to the IS, it can be minimizable with single OI's under only one of the NI minimization categories, normal or one-difference minimization. For example, if the NI is a superset instance to an OI, then it cannot be minimized with any other OI, except as a superset instance. This follows from the assumption that the IS is already minimal. This property holds for all types of NI minimization.

3.5.3.6.2. Minimization Ripple

Although a NI can only initially be minimized under one NI minimization category with any OI, after that minimization has taken place, it is possible that the modified instance(s) can be minimized with other OI's under a different NI minimization category. This notion is referred to as ***Minimization Ripple.*** Following is an example of minimization ripple.

$$A\ B\ \longrightarrow Z \qquad (1)$$

$$A\ C\ D\ \longrightarrow Z \qquad (2)$$

$$A\ \bar{B}\ C\ \longrightarrow Z \qquad (NI)$$

Initially the NI can only be minimized with 1 as a one-difference minimization. After the NI has been minimized to $A\ C\ \longrightarrow Z$, it can then be minimized with 2 using subset minimization. In this case, OI 2 is deleted from the set.

The notion of ripple does not occur when making the IS consistent. One comparison, and possible modification, of the NI to each discordant OI, in parallel, is sufficient to make the IS consistent.

3.5.3.7. Why Keep the Instance Set Minimal

The assumption of minimality is not necessary for the correct functioning of the system. The assumption is desirable though, for two basic reasons.

The first has to do with efficiency of hardware usage. The computer storage of fully expanded instances can be orders of magnitude greater than that necessary for the minimal set of instances. Other memory and communication necessities are also greatly reduced by using a minimal set.

The other reason is heuristic. If one desires to find a minimal hardware circuit to compute a boolean function, one usually minimizes the function in order to find which groupings of variables can best be utilized to synthesize the minimal circuit. An instance which is minimal, a ***Prime Implicant,*** gives information about which groupings of variables tend to best discriminate the desired output. Thus, a minimal IS is a resource for aiding in the formation of optimal networks to fulfill the IS. When formulating algorithms to cause adaptation of the network, the minimal instances in the IS will allow for higher efficiency and optimization.

For neither of the above cases is it necessary to have perfect minimization; i.e. the smallest theoretical set of instances which can represent the information. Having ***Relative Optimality,*** defined as being within a "reasonable" distance of perfect optimality is a notion which we hold to throughout this research. Perfect optimality is rarely worth the effort it takes to find it and usually requires global knowledge.

3.6. Feedback Variables

Once an output variable is defined, by having been used in an instance in the instance set, it can be used in the antecedent of instances later input to the system. When an output variable is thus used, it is called a ***Feedback Variable.***

There are two basic ways in which feedback variables can be used. The first is to accomplish ***Rule Chaining.*** Rule chaining is a commonly used method for simplification and efficiency for both storage and modification requirements. Assume the following instance set.

$$X_1 X_2 \rightarrow Z_1$$

$$\bar{X}_1 X_3 \rightarrow Z_1$$

Now we wish to add an output variable, Z_2, which should always be 0 when Z_1 is 1. We can reproduce all instances which cause Z_1 to become 1, and give them the antecedent $\bar{Z}_2$. This method is not only wasteful of storage, but is incorrect, since Z_1 could be 1 for states of the environment which do not match any instances. The obvious answer is to allow the new instance to be defined directly.

$$Z_1 \rightarrow \bar{Z}_2$$

Rule chaining is also a useful method to do instance decomposition. Rather than have instances with very long antecedents, rule chaining can be used to create higher abstractions of variables which in turn can be used in a more parsimonious and adaptive manner (i.e. It is much easier to change the definition of a single variable, than to change the definition of all instances which define that variable).

The other use for feedback variables is for actual ***Feedback.*** The rule "if X_1 is 1 and Z_1 is currently 1, then cause Z_1 to become 0" is represented as:

$$X_1 Z_1 \rightarrow \bar{Z}_1$$

This example demonstrates that the knowledge base defines a finite state machine which maps partial state to partial state. Note that through the use of feedback variables it is possible to create continuous oscillations between output variables which can be used for both sequencing and timing.

It might initially appear that allowing feedback variables in instances would make it very difficult to keep the instance set consistent and minimal. Indeed this is the very cause of inefficient truth-maintenance systems in many Artificial Intelligence applications. However, there is a simple solution. We state that for consistency and minimization maintenance, a feedback variable is treated as if it were any other input variable.

Assume the instance set is made up of the following two instances.

$$X_1 \longrightarrow Z_1$$

$$X_1 \longrightarrow Z_2$$

Now we add the instance $Z_1 \longrightarrow \bar{Z}_2$. This NI contradicts the second instance. In this case, since Z_1 is treated like any other input variable, the second instance becomes $X_1 \bar{Z}_1 \longrightarrow Z_2$. This new instance is always transient at best, since the first instance will never allow this condition to remain. Perhaps this very transience between states will be a key in allowing sequential solutions to problems in this new paradigm. The overall result, though, is that the new instance is fulfilled; i.e When Z_1 is 1 then Z_2 becomes 0.

At first glance of the above problem, one might be tempted to say that for consistency it is necessary to substitute all occurrences of feedback variables with the antecedents that define them. However, this initially intuitive notion can not succeed because of the partiality of the instance set function. The fact that we replace a feedback variable with the antecedents that define it, does not account for the times when the variable is active even though the environment state does not match any instance in which the variable is defined. For example, assume that we have the case as just defined above. One might suggest that we replace $Z_1 \longrightarrow \bar{Z}_2$ with $X_1 \longrightarrow \bar{Z}_2$, since $X_1 \longrightarrow Z_1$ is the only instance that defined Z_1 as an output variable. However, that definition of Z_1 is only a partial definition. There are cases in which Z_1 could equal 1 while X_1 would be 0, in which case Z_2 would not be set to 0 if the substitution were allowed. Thus, we see that substitution is not a correct method since it can change the meaning of the instance set.

3.7. 3 State Inputs

In the earlier section explaining how the logic network fulfills the instance set, we assumed that the logic network is a boolean network. This is the way in which it will probably be used in terms of fulfilling the instance set. However, the logic network to be proposed is sometimes a ternary network. The 3-state capabilities are needed for some algorithms to make the network amenable to adaptation. Thus, if it were desired, one could allow 3-state inputs from the environment and 3-state outputs from the logic network. But, even though this is possible, it is not allowed to define instances in terms of 3-state variables. For example, one could not input an instance which states that when a certain variable is "don't know" then the output should be at some 3-state level. These kind of instances could be made acceptable if it proved to be a desirable functionality, but the current definition of instances allows only boolean variables in an instance.

If 3-state inputs from the environment were used within the current definition of instances, the overall change to the network function is minor. When only boolean environment variables are allowed, the cells in the state space which are "don't knows" become either a 1 or a 0. When 3-state variables are input from the environment, the "don't know" cells become 1, 0, *or* "don't know".

Throughout the rest of this paper, we assume that boolean variables are used in the environment, having mentioned that a ternary network could be used with 3-state variables.

Chapter 4

SYSTEM ARCHITECTURE

The overall ***System Architecture*** is discussed in this chapter. Details of physical implementation, such as memory sizes, bus widths, and detailed interconnection schemes and protocols are not discussed in this chapter. ***Physical Implementation*** is referred to throughout the dissertation, but is beyond the scope of this document. Thus, for example, we can simply say that there is a communication path between two elements allowing passage of certain types of messages, without having to state details of how the message would be packaged and communicated. Although VLSI and wafer-scale implementation efforts are ongoing, these are not covered in this dissertation.

Another reason to delay implementation discussion is that implementation depends on the particular ***Algorithm*** which is used. The basic architecture presented is amenable to a number of different algorithms. For example, each node of the network can be designed as a simple finite state machine. The exact capabilities of this simple unit would be different for each algorithm. However, there are some basic characteristics shared by nodes in all algorithms, such as RAM.

Explanation as to why each method was chosen and ways in which the architecture can be used are discussed in the next chapter on primitive mechanisms. First the overall architectural structure is discussed. Each part of the system is then discussed in more detail in the following sections. Communication between subsystems are mentioned in the next section. The final sections discuss methods of ASOCS usage which allow for more efficient use of the architecture. Discussion of the integrated functions carried out by the modules of the architecture is found in the next chapter on primitive mechanisms.

4.1. Overall Structure

The generic name given to the system is ***Adaptive Self-Organizing Concurrent System (ASOCS).*** If the system were thought of as a black box, it would appear as in figure 4.1.

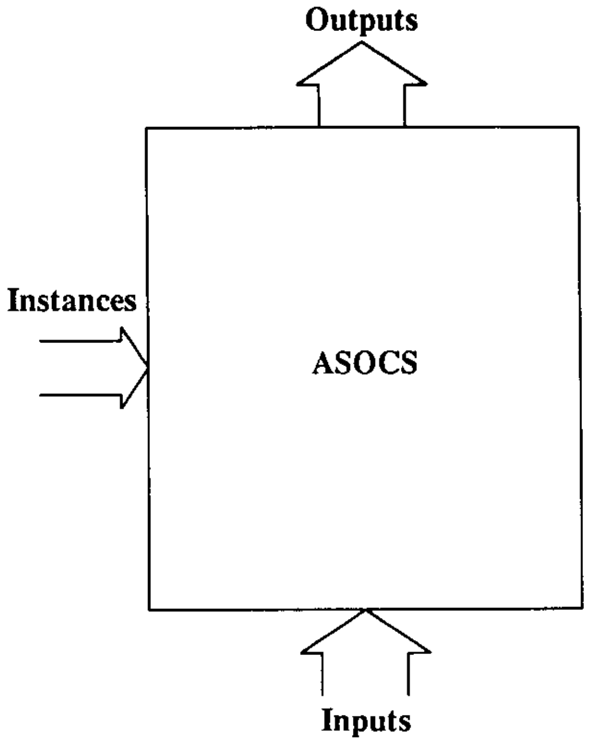


Figure 4.1 - ASOCS System

The system has two modes of operation: ***Execution*** and ***Adaptation.*** During execution, the system receives boolean inputs and maps them into boolean outputs. This mapping is done in a combinatorial and data flow fashion, thus enabling execution at very high speeds. When in the adaptation mode, new instances can be added to the system. During adaptation mode, inputs from the environment are ignored, while the system reconfigures itself to accommodate the change to its function. These are the only modes of operation, and there are no other necessary inputs or outputs to the system other than those shown in figure 4.1. The Adaptation mode could also be called the *Learning Mode.*

4.1.1. Input/Output Relationships

The number of inputs and outputs are limited by the number of nodes within the network. Figure 4.2 shows the representation of the system, where x_i are the input variables and z_i are the output variables.

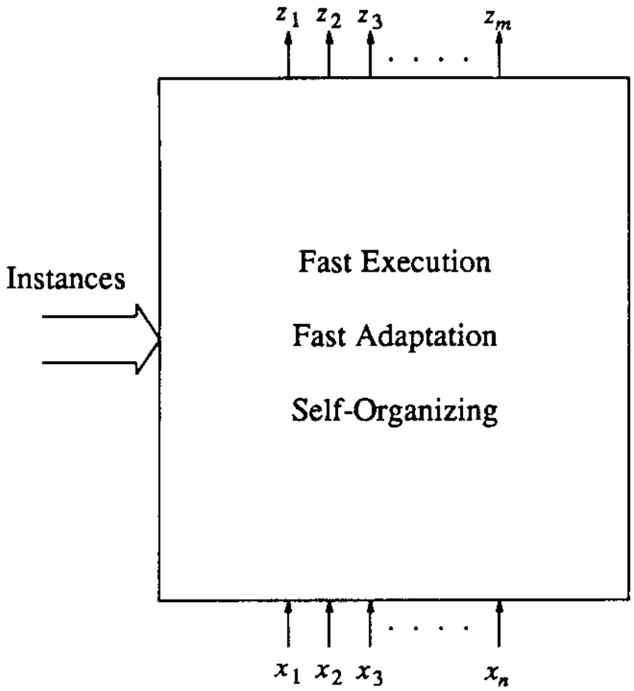

z_1 z_2 z_3 . . . . z_m
Instances
Fast Execution
Fast Adaptation
Self-Organizing
x_1 x_2 x_3 . . . . x_n

Figure 4.2 - Input/Output Variable Relations

Logically, there is no limit on either the number of inputs or the outputs or the relationship between them. For n inputs there can be any number of outputs defined, although there can only be 2^{2^n} unique outputs having differing functions.

Thus, the overall topography of an ASOCS network can be very diverse. The network could have a converging topography like a triangle where a number of inputs converge to a few outputs for decision and classification. It could have a diverging topography, like an upside-down triangle for coordinated control of multiple motor units.

4.1.2. ASOCS Structure Overview

This section gives a brief summary of the modules contained in the ASOCS system. Figure 4.3 shows the structure of the system. The two main entities are the ***Adaption Unit (AU)*** and the ***Logic Network.*** During execution mode, only the logic network is active. Data comes in at the inputs and flows asynchronously through the network with only propagation delays. Figure 4.4 shows that part of the total ASOCS system which is active during execution mode.

During adaptation mode, the other parts of the system become active. The AU and the logic network are connected by a ***Broadcast Bus***. The AU can ***broadcast*** a message to the entire logic network by placing the message on the broadcast bus, but it cannot address a specific node within the network. A node within the network can also place a message on the broadcast bus, which can be read in turn by the AU or any other node within the network.

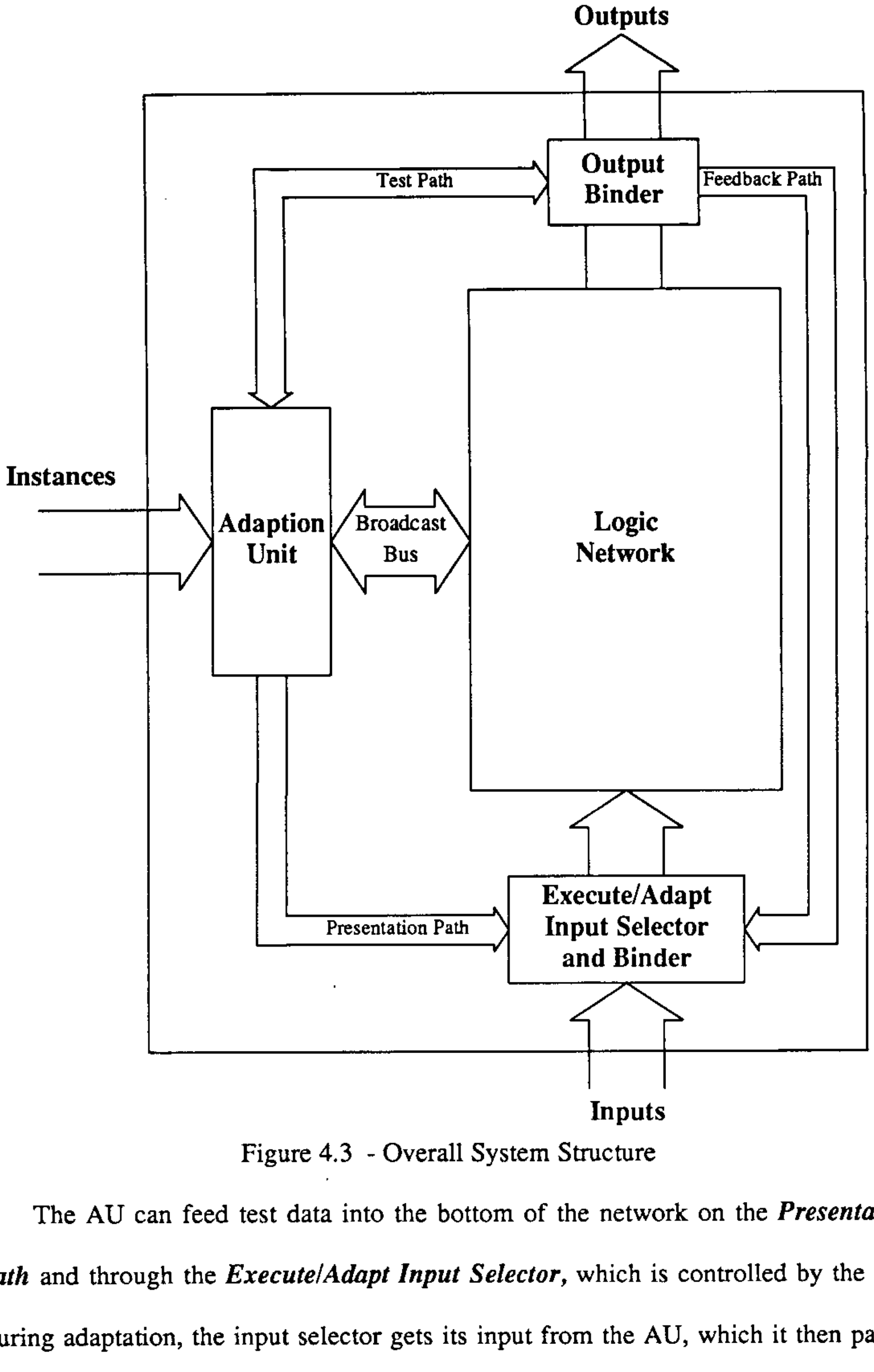


Figure 4.3 - Overall System Structure

The AU can feed test data into the bottom of the network on the ***Presentation Path*** and through the ***Execute/Adapt Input Selector,*** which is controlled by the AU. During adaptation, the input selector gets its input from the AU, which it then passes

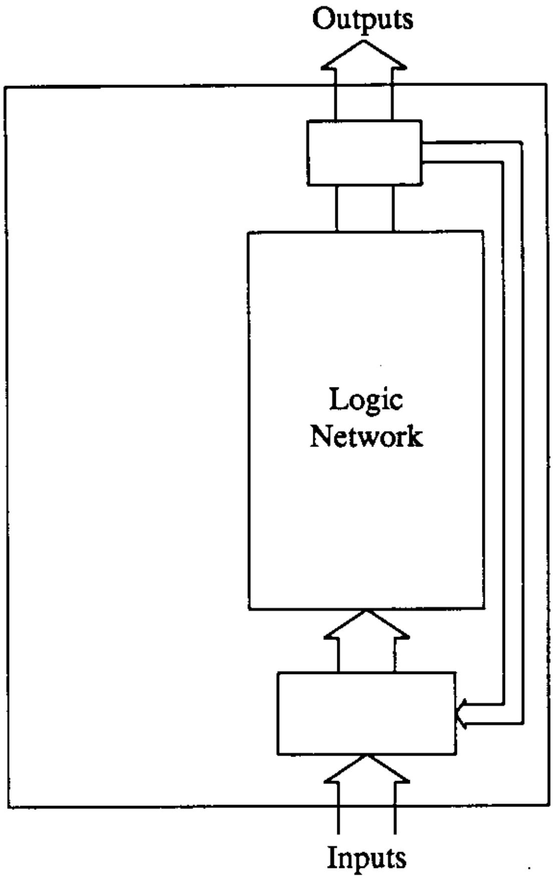


Figure 4.4 - ASOCS during Execution Mode

to the network. During execution, the input selector passes the boolean inputs from the outside through to the network.

It is also necessary to bind variables, in a flexible manner, to actual hardware lines. This is done for both input and output variables by the ***Input Binder*** and ***Output Binder***. Thus when new (not previously used) input or output variables are used in a new instance, an input or output line will be allocated for the new variable. The bindings take place under direction from the AU. The ***Feedback Path*** allows output variables bound in the output binder to be fed back to the input binder.

The AU also has the ability to monitor the outputs of the logic network by use of the ***Test Path.***

As earlier stated, only the logic network is active during execution mode. Boolean variables enter the system from the environment and flow through the logic network in a combinatorial fashion. Output variables can also be fed back to the network in an asynchronous fashion. The AU and binders become active in the adaptation mode. This mode is triggered by the input of new instances to the system.

Each part of the system architecture is now discussed in more detail.

4.2. Network Node

A single node within the logic network is represented in figure 4.5. A node in the network is made up of two basic parts; The ***Control Unit*** and a dyadic ***Dynamic Programmable Logic Module (DPLM).***

During execution mode the DPLM is the only active element, as shown in figure 4.6. (It would of course be possible to allow the control unit to perform unrelated activities, such as self-testing, while the system is in execution mode.)

During adaptation the control unit has the ability to change the function of the DPLM. It can also send and receive messages to or from neighbor nodes, and can change the interconnections between itself and neighbor nodes. The control unit has the ability to read from or write onto the broadcast bus.

4.2.1. Dynamic Programmable Logic Module

The DPLM is a 2 input single output logic gate. The lines going in and out of the DPLM are called the ***data lines.*** The DPLM can be a 3-state, or ternary, device

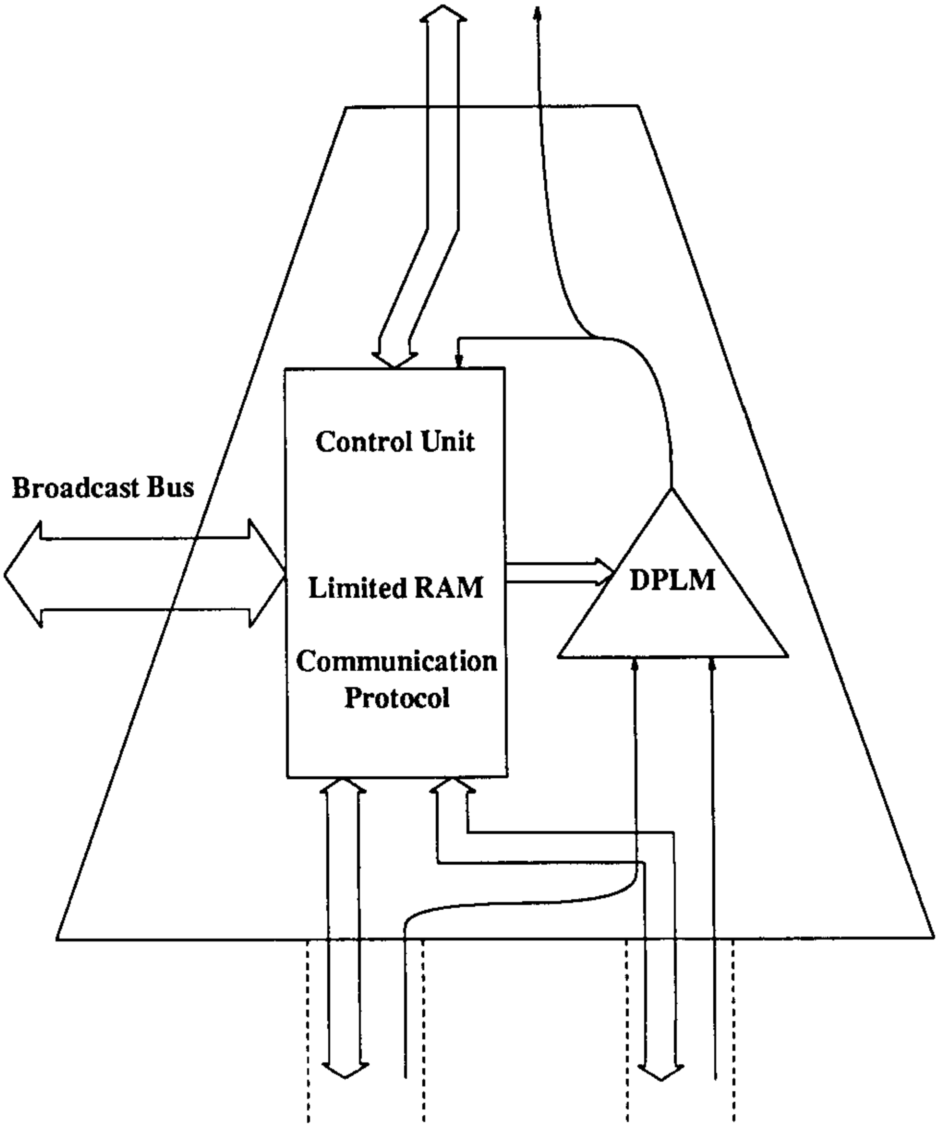


Figure 4.5 - Single Network Node

rather that the typical 2-state boolean gate. Thus, an input or output variable can have any one of three values which are called ***positive, negative,*** and ***don't know.*** These three values are represented as before by "1", "0", and "?".

When an input or output has a value of positive or negative, it is considered as ***asserted.*** The positive and negative levels could be thought of as being equivalent to

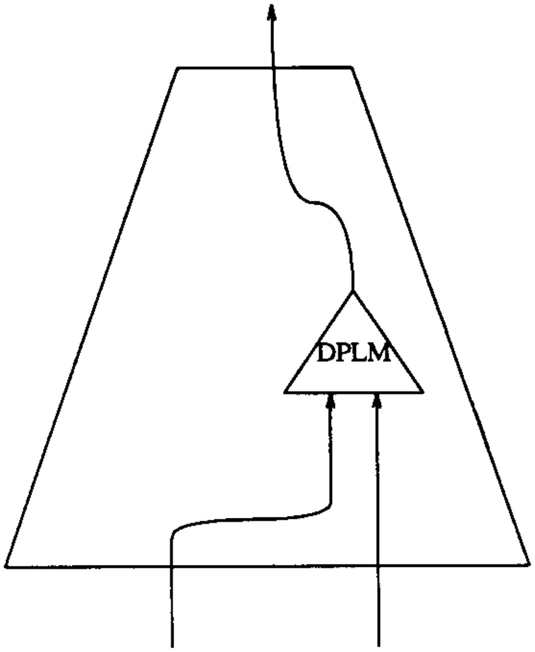


Figure 4.6 - Single Node during Execution Mode

the two states in the boolean world, while a level of don't know means that the line is bound but has not been asserted.

The DPLM must be a ternary device in order to accomplish certain functions during the adaptation mode of the adaptive algorithm 1. The DPLM need be only a 2-state gate for the adaptive algorithms 2 and 3. Thus, the discussion of the ternary DPLM is not important for all algorithms.

During execution mode, boolean values appear on the data lines, and the DPLM is used as an ordinary boolean gate. It is however possible to extend the system and use 3 state inputs from the environment, as discussed in chapter 3.

Although the DPLM can be a 3-state device, it can only be set to any one of the 16 boolean functions of 2 inputs, which are shown in Table 4.1. Figure 4.7 shows the

x_1x_2	x_1x_2	x_1x_2	x_1x_2	
00	01	10	11	Function
0	0	0	0	*Neg*
0	0	0	1	$x_1 \cdot x_2$
0	0	1	0	$x_1 \cdot \bar{x}_2$
0	0	1	1	x_1
0	1	0	0	$\bar{x}_1 \cdot x_2$
0	1	0	1	x_2
0	1	1	0	*XOR*
0	1	1	1	x_1+x_2
1	0	0	0	$\bar{x}_1 \cdot \bar{x}_2$
1	0	0	1	*Equiv*
1	0	1	0	$\bar{x}_2$
1	0	1	1	$x_1+\bar{x}_2$
1	1	0	0	$\bar{x}_1$
1	1	0	1	$\bar{x}_1+x_2$
1	1	1	0	$\bar{x}_1+\bar{x}_2$
1	1	1	1	*Pos*

Table 4.1 - Boolean Functions

symbolic representations that will be used in future examples to represent these 16 configurations.

The DPLM is a universal 2 state device, since it can compute any 2 state function. It is not, however, universal for 3 state functions. The allowable 3 state functions are defined in terms of the 16 boolean functions which the DPLM can perform. For instance, table 4.2 shows the truth table for the 3-state DPLM for the *And*, *Or*, and *Negation* functions. The truth table for any other function can be extrapolated from these three.

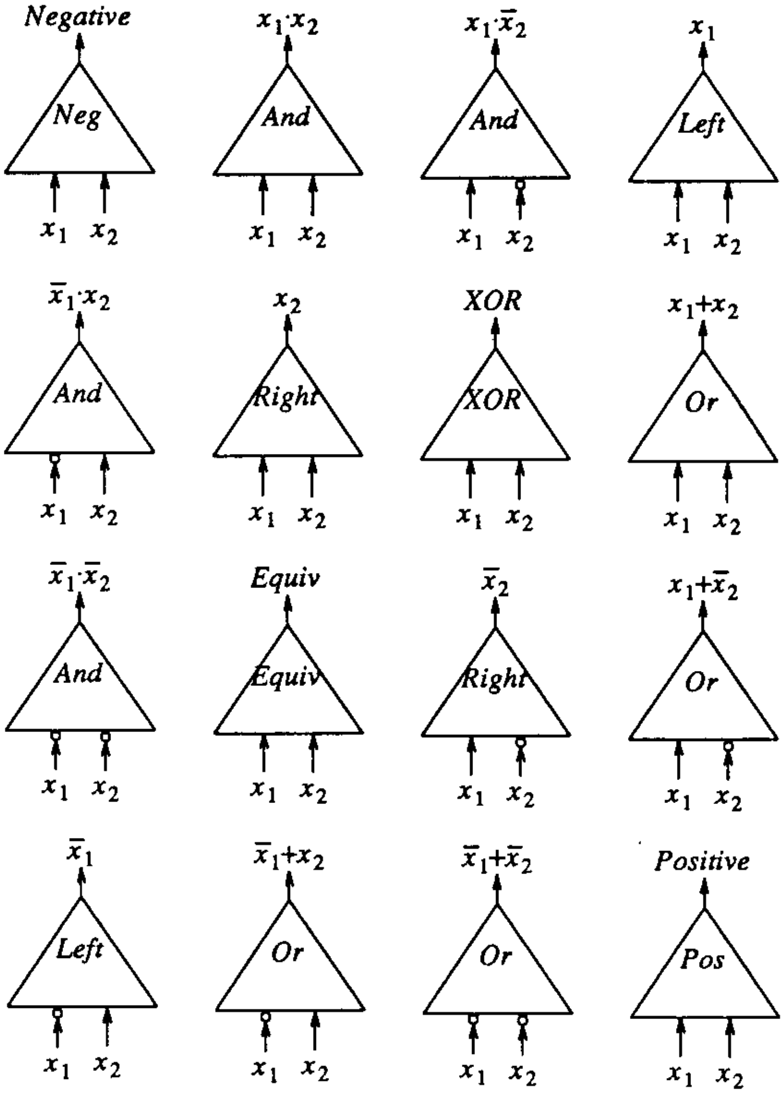


Figure 4.7 - DPLM Representations

4.2.2. Control Unit

The control unit is a small special purpose finite state machine, with some random access memory, and the ability to communicate with the nodes to which it is directly connected. The amount of memory needed and the function accomplished by the control unit differ depending on which algorithm is being used to run the system. Each network node is identical, except for the contents of its memory, and the function

x_1	x_2	*And*	*Or*	$\bar{x}_2$
0	0	0	0	1
0	1	0	1	0
0	?	0	?	?
1	0	0	1	1
1	1	1	1	0
1	?	?	1	?
?	0	0	?	1
?	1	?	1	0
?	?	?	?	?

Table 4.2 - 3 State Truth Tables

of its DPLM.

There is a 4 bit unidirectional line from the control unit to the DPLM. This line can be dynamically set so that the DPLM can operate using any one of its 16 possible functions. The number of boolean functions which a node must be capable of computing is dependent on the specific algorithm.

The control unit can both read and write the broadcast bus The types of messages passed on this bus will be defined in terms of each algorithm.

The control unit can also send messages on the bidirectional path, called the ***Control Path,*** which connects it to its immediate neighbors. The two nodes which give input to a node are known as the ***Child Nodes*** to a given node, while those nodes to which the node sends its output are ***Parent Nodes.*** There may be more than one parent node for a given node, but there cannot be more than two children, although there can be less than two children if one or more inputs are not connected. (i.e. If a node is only doing a function of one of its inputs). The two nodes which input to a parent node are called ***Sibling Nodes*** with respect to each other and their parent node. Figure 4.8 shows a representation of interconnected network nodes.

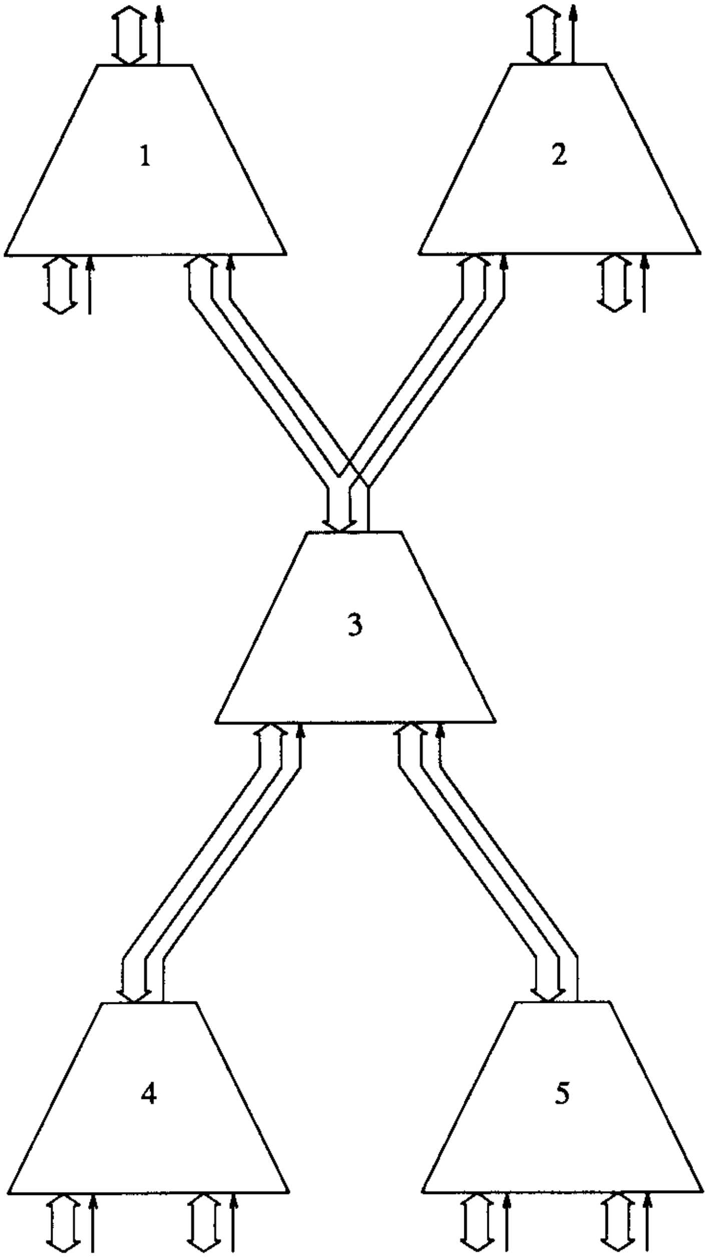


Figure 4.8 - Network Node Interconnections

Each node is numbered for identification. In this case node 3 is the parent to nodes 4 and 5. Node 4 is the ***Left Child*** and node 5 is the ***Right Child*** to node 3. Nodes 4 and 5 are sibling nodes. Nodes 1 and 2 are both parent nodes to node 3.

4.2.3. Physical Implementation Consideration

It is deemed useful to here make a brief note regarding the difference between the basic architecture and the physical implementation of the logic network. In the representational view, we assume that we can just connect the output of a node to any other node. We also assume that we can remove old nodes and allocate new nodes as needed. It is assumed that the interconnections between nodes can be changed at will. In reality, there must at some level be hardwired paths between nodes, which cannot change in time.

One possible model for a physical system would be a large array of interconnected nodes with some form of ***Interconnection Switches*** between layers of nodes. The flexibility of interconnections would be handled through these switches. Figure 4.9 is an example of how one might represent a simple network of 3 nodes at an abstract level. Figure 4.10 is a possible implementation of the same 3 nodes within a physical structure, where the labels A, B, and C represent the same nodes. If a change of connection is desired, the routing would be changed within the switches which sit between groups of nodes. Control of the switches can be handled locally by the nodes themselves, without need of central control from an external agent. In a physical implementation the size and number of switches in between layers would depend on the algorithm and applications.

Interconnection switches between layers of nodes are not necessary for all

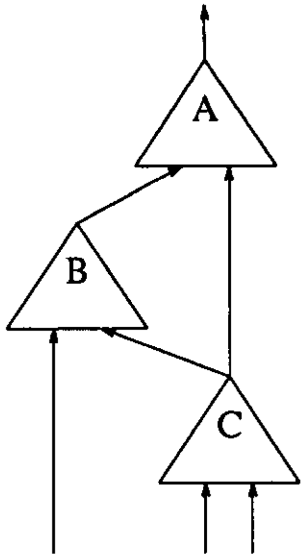


Figure 4.9 - A Network Representation

ASOCS systems. As will be seen, adaptive algorithm 3 requires no interconnection flexibility between nodes, and the node interconnections can be static.

4.3. Adaption Unit

The *Adaption Unit's (AU)* main function is to guide the logic network through adaptation. It does this by receiving new instances, directing variable binding, and broadcasting commands to the logic network.

4.3.1. Knowledge Base Maintenance

In adaptive algorithm 1 (AA1) the AU is also responsible for storage and maintenance of the instance set. In the other algorithms the instance set is both stored and maintained consistent and minimal implicitly in the network.

In AA1, the AU keeps in memory the complete instance set (IS), which is the knowledge base for the system. When a new instance (NI) is entered, the AU must

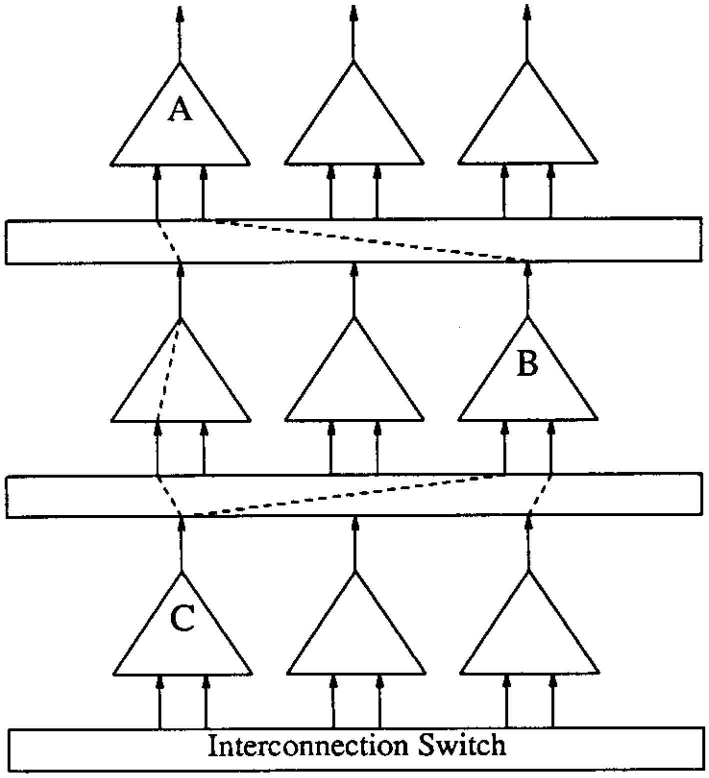


Figure 4.10 - Possible Network Implementation

modify the IS to insure both consistency and minimality. The method used to keep the IS minimal and consistent is independent of the rest of the ASOCS functions, and methods were alluded to in chapter 3.

For all algorithms, the AU has an interface to the outside environment. Using this interface, *new instances* are entered into the system. It is not necessary for this interface to be bidirectional, but it may be useful to have it be so in a system implementation. Information concerning variable *Binding* could also pass through this interface.

4.3.2. Variable Binding

When an instance is entered into the system, it must be stated to which output variable this instance applies. When the output variable is a new[1] variable, then it must be bound to an output line. When the AU receives instances for a new output variable, it sends this fact to the output binder. Both the input and output binders are connected to a large number of boolean ***ports***. Each port has a physical address. Upon receipt of knowledge of a new output variable, the output binder could allocate an unused port and send the address back to the AU, which in turn would inform the user so that a physical connection could be made for the new output. Another method would be for the AU to keep a list of allocated and unallocated ports, thus enabling the binders to be passive, and the AU would send the new address to the binder. In this case it would also be possible for the user to specify a port address together with any new variable input to the system. The same methodology used for binding output variables could be used for binding input variables.

Note that by adding a layer of flip-flops at the output binder, the system can be used as a sequential finite state machine.

4.3.3. Adaptation Guidance

For the logic network to be able to properly adapt, it requires some global guidance. This is supplied by the AU by use of the broadcast bus. Upon receipt of a new instance the AU can communicate information about the new instance to the network. The AU can broadcast a command to the nodes in the network, which can then

[1] One which has not previously been used in the network.

simultaneously execute procedures triggered by the command. The AU can also send information to the network through the input selector, and it can monitor the output of the network.

4.4. System Communication

This section discusses the basic communication paths within the ASOCS system. The data lines and control paths which reside within the logic network were discussed in an earlier section. Also the interface to the outside where instances are input to the AU was mentioned in the section on the adaption unit.

The path which connects the AU and the input selector is called the ***Presentation Path.*** It is a unidirectional point to point path from the AU to the input selector and binder. It has two basic functions. The first is for *presentation* of instances to the network. This is explained in the next chapter. It allows the AU to send data to the bottom layers of the network which can then flow upwards through the network. The other use of this path is in directing the input binder when a new variable is introduced in a new instance.

The path connecting the AU and the output binder is called the ***Test Path.*** It consists of two unidirectional point to point paths. One comes from the output binder to the AU and contains the current settings of all the output variables. Thus, the AU is able to monitor the output of the logic network. The other path initiates at the AU and can send messages to the output binder concerning the binding of new output variables.

The bus directly connecting the AU and the logic network is the ***Broadcast Bus.*** It is a general bidirectional bus which all nodes in the network can both read and write. It is the only true *bus* in the system, since many different modules are connected to it. Only one module (AU or a node) can write on this bus at a time, but all modules can simultaneously read the bus. It is not essential that the broadcast bus be a unique physical entity, since it would be possible to use either the presentation path or the test path to fulfill the broadcast function. In this case either the test or the presentation path would send the broadcast message to the top or bottom layers of the network, and the message could then flow concurrently down or up through the network.

The path between the output binder and the input binder is called the ***Feedback Path.*** The feedback path is a unidirectional point to point link initiating at the output binder. The feedback path allows for bound output variables to be fed back to the input binder where they can enter the logic network. This mechanism allows output variables to be used in the antecedents of instances (as input variables), thus allowing feedback within the network. It also makes it easy to use intermediate variables in instances, which allows for chaining of rules and rule decomposition.

4.5. Hierarchical Composition and Decomposition

As discussed so far, the ASOCS system is made up of one entity containing a single adaption unit together with a logic network. When problems become large, it is many times advantageous to compose or decompose a problem solving system from or into many subsystems. Not only does one gain the advantages of modular and smaller subsystems, but indépendent subsystems can be created, allowing easier independent adaptation. This and the following sections discuss two ways in which the ASOCS system is easily amenable to decomposition.

The main reason that ASOCS is easily decomposed is the regularity of its overall structure, and the equivalence of inputs and outputs. Since outputs of the system have the same form (i.e boolean variables) as inputs, the outputs from one ASOCS system can become inputs to another. Any number of ASOCS systems can be combined in this manner. Figure 4.11 is an example of this type of decomposition.

This type of decomposition is called ***Hierarchical Decomposition.*** This follows because outputs of systems at lower levels are fed into systems at higher levels. For example, ASOCS systems at low levels could receive low level sensor inputs, in turn combine these into more abstract outputs, which become the inputs at a higher level. The regularity of the ASOCS structure allows any number of these systems to be combined. Note that this can be equally considered as a *composition* technique to build up an ASOCS system using a bottom-up approach.

Instances are input independently to the different ASOCS modules, and each module keeps its own instance set. This method also allows for the use of rule chaining and feedback. Variables output from an ASOCS system can become input to a system at a lower level either directly or after having passed through any number of other systems.

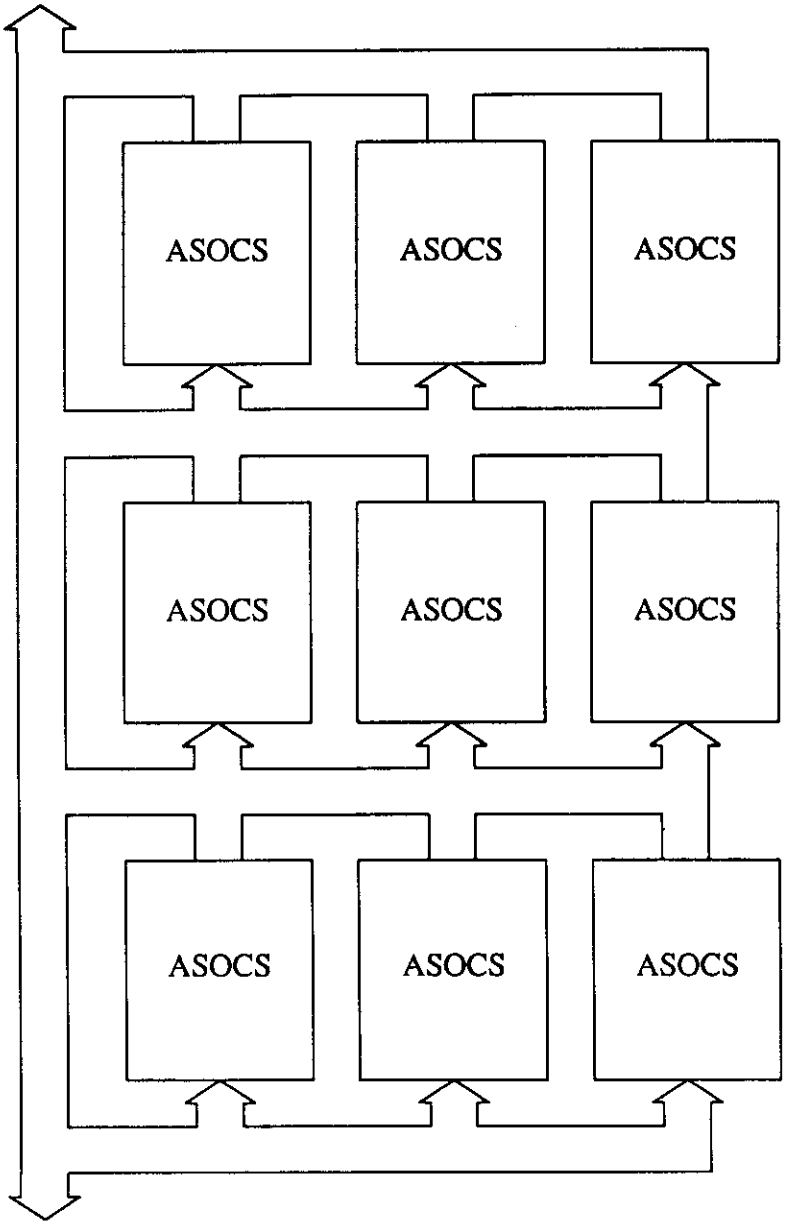


Figure 4.11 - Hierarchical Decomposition

4.6. Layered Architecture

A concept to be used in actual implementation of the ASOCS structures is that of a ***Layered Architecture.*** It was earlier stated that there could be any number of output variables in relation to the input variables. In theory this is true. However, for an

ASOCS system, structured as shown in the previous sections, there is typically a growth in the amount of interconnection connectivity needed between nodes as the number of output variables increase. In order to allow the ASOCS system to function within the set interconnection bandwidths of a physical implementation, it is necessary to constrain the number of output variables being computed by a single system. Yet, we do not want to severely limit the number of output variables which can be computed. One answer to this problem is to use a layered architecture.

For this type of system, we assume that the ASOCS system as previously defined (with some changes to be explained) is a single ***Adaptive Plane.*** The ASOCS system so far discussed is basically a 2-dimensional architecture. In the layered architecture we build a 3-dimensional architecture by stacking some number of adaptive planes contiguously. Figure 4.12 show the representation of the layered ASOCS architecture.

With the layered architecture it is possible to divide up the total number of output variables amongst a number of adaptive planes. If their are n current output variables and m adaptive planes, then each plane could be responsible for n/m output variables. Each plane can still receive as many input variables as are used in defining the output variables since this does not cause a critical increase of connectivity between the nodes. In this case, each plane will usually compute a converging function with more input variables than output variables, while the overall ASOCS system can be converging or diverging.

There are a couple of basic architectural differences. The adaptive plane in this methodology is simpler than the system discussed earlier in the chapter. The input

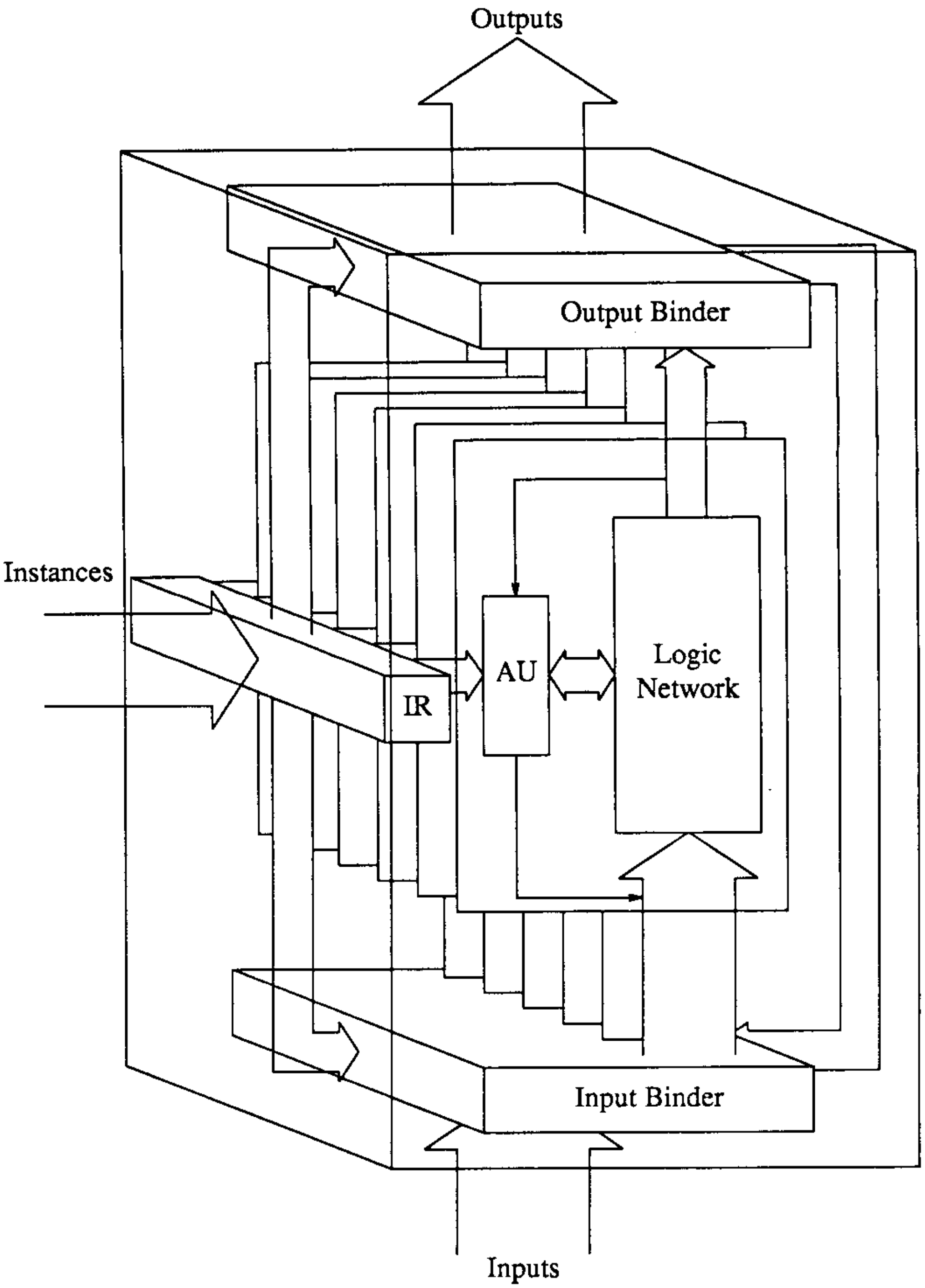


Figure 4.12 - Layered Architecture

and output binders no longer reside within the adaptive plane. There is a single input and output binder which serve all of the adaptive planes in the system. The feedback path is also no longer in the adaptive plane, and one feedback path from the output binder to the input binder is sufficient to serve all planes. The AU is relieved of the duties of variable binding. This function is taken over by the ***Instance Router.*** The instance router receives instances from the outside, causes variable bindings when necessary, and routes the instances to the adaptive plane which is responsible for the output variable defined by the new instance. Since each plane is responsible for a non-intersecting subset of the output variables, all of the planes can undergo adaptation simultaneously. Also, any plane which is not undergoing adaptation, may continue in the execution mode.

There are many other methods for decomposing the ASOCS system due to its regularity. One intuitive method which would fit in with the layered architecture would be to have a layer of *primitive functions* between the input binder and the adaptive planes. These would consist of commonly occurring values which could in turn be shared by many of the adaptive planes so that redundant work within planes could be diminished. One could also use hierarchical decomposition to solve this problem, by having layered ASOCS systems at a lower level feeding into systems at a higher level.

Chapter 5

PRIMITIVE MECHANISMS IN ASOCS SYSTEMS

Given a brief understanding of the architecture and its capabilities, we can now discuss the basic *primitive mechanisms*, or methods, used to make the system work in a functional manner. These mechanisms are techniques used in solving a goal specified by the constraints of the system. The following methods are the building blocks out of which useful *algorithms* can be built. In this sense, the algorithm is the overall strategy used to implement a desired functionality on the ASOCS architecture. Not every method is used with every algorithm, but the majority are used with each.

We first discuss the communication capabilities necessary to build the basic primitives. These communication abilities allow the system to adapt in a self-organizing and concurrent fashion. System communication is divided into the categories of *Global* and *Local* commands. Using these techniques, four basic primitives are defined, out of which all current algorithms are built. These four are

1. Instance Introduction
    a. *Presentation*
    b. *Instance Broadcast.*
2. *Node Selection.*
3. *Node Addition and Combination.*
4. *Self-Deletion*

Each of these topics will be discussed in separate sections. The last section will cover the general methodology used for fault detection and recovery.

5.1. Global Commands

In order to allow the ASOCS structure to adapt there must be a method by which all nodes can simultaneously be working towards a specific goal. Although each node can accomplish most of its function using only local knowledge, there must be a method by which all nodes can know what they should be working on at a specific time. For example, a node should not change its function during execution mode. Thus, there must be a global message to tell nodes when the system is in the adaptation mode, at which time changes are permissible. The communication abilities discussed in this section are used only during the adaptation mode of ASOCS operation.

These goals are accomplished by the use of ***Global Commands.*** A global command is a message written onto the global broadcast bus. The adaption unit initiates all global commands. By placing a predefined message on the broadcast bus, the AU can allow all nodes in the network to execute specific procedures. The action which a node takes will depend upon its current state. Thus, the system is not "single instruction multiple data", rather the global instruction restricts the independence of the network nodes to actions compatible with the common goal. It is in this sense that the AU *guides* the network through adaptation.

A node in the network, when uniquely *selected*, may also place a message on the broadcast bus which can in turn be read by the AU and all other nodes in the network. In these cases it is essential with the single node be uniquely chosen, so that bus conflicts due to multiple nodes simultaneously trying to write on the bus do not occur. This selection is performed by a primitive process called the *selection wave.* For some commands, any node may have the ability to hold a ***ready*** line inactive until it has completed the action defined by the global command. Thus, the AU can time global command issuance in an orderly sequence.

5.2. Local Commands

The other type of communication, ***Local Commands,*** occur under the restrictions placed by the global commands. Local commands consist of a message originating at one node and sent to one or more of its directly connected neighbors. A global command can allow execution of one or more types of local commands. Each of the primitive mechanisms discussed below is achieved by a combination of global and local commands.

5.3. Instance Introduction

There are two basic ways in which the AU can send information to the logic network about specific instances. They are ***Presentation*** and ***Instance Broadcast.***

5.3.1. Presentation

The AU *presents* an instance to the network by setting the network data lines to the values matching the instance which is being presented. In this way, each node can detect what its output currently is for that instance. The node can then use that information to evaluate its ability, in terms of the instance, to intervene in the network adaptation.

Since instances are incomplete (they do not contain all the variables in the environment) there are many different environment states which could match the instance being presented. Those input variables which occur in the instance being presented are set to a 1 or a 0, (i.e. asserted), and all remaining variables are set as don't knows, since their value in a given environment state does not affect what the final output of the network must be. Thus the ASOCS must use 3-state DPLM's if presentation is to be used as the instance introduction mechanism.

Assume, for instance, that there are currently 8 bound input variables entering the network from the environment: $X_1 - X_8$. The AU presents the instance

$$X_1 \bar{X}_3 X_7 \longrightarrow Z$$

to the network, through the input router. The router asserts the variables X_1 and X_7 to 1 and the variable X_3 to 0. The other 5 variables are set to don't know. This data then flows combinatorially through the network data paths (DPLM's). This situation is shown in Figure 5.1.

The AU would then send a global command to the network signaling the nodes that a presentation is taking place. Each control unit can detect and store the output of its particular DPLM. The AU can also read the output of the top node relating to the output variable Z on the test bus, which is either 1, 0, or ?. If the example instance were a new instance, and the output of the top node corresponding to Z were 1, then the network would require no adaptation because it already fulfills the new instance for all states of the environment. If the output were a 0 or a ?, then network adaptation would have to take place.

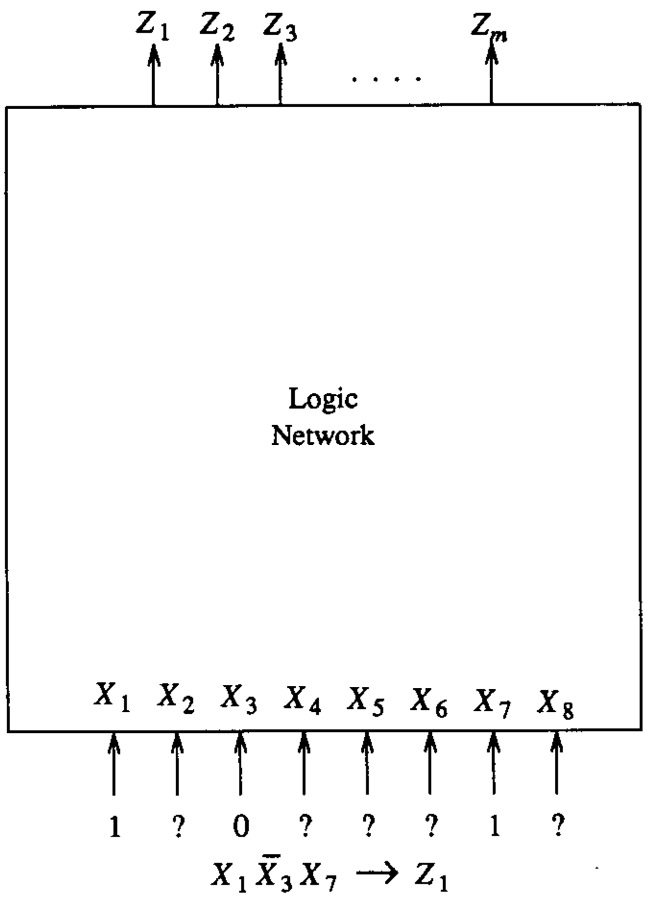

Z_1 Z_2 Z_3 .... Z_m
Logic
Network
X_1 X_2 X_3 X_4 X_5 X_6 X_7 X_8
1 ? 0 ? ? ? 1 ?
$X_1\bar{X}_3X_7 \rightarrow Z_1$

Figure 5.1 - Instance Presentation

Without 3-state logic each possible state of the environment matching the instance presented would have to be sequentially presented to the logic network to achieve the same results. For a given instance, 2^{n-m} states of the environment would have to be shown to the network, where n is the number of bound variables in the environment, and m is the number of variables in the instance. For the above example, 2^{8-3}, or 32, enironment states are represented by the single instance presentation. Thus, the problem which is exponential with the number of instance variables for a 2-state scheme, can be solved in a single iteration using 3-state presentation.

5.3.1.1. Presentation and Learning by Example

In the foregoing, it was assumed that the AU would cause the input variables to be set to a 3-state value. However, it is equally reasonable to allow the environment to to set the value of the variables during presentation. This is a another form of learning by example.

For example, assume that the logic network is to be used in controlling robot motion. Assume that control of a part of the robot, the arm, is the current concern. Sensors could be connected directly from the arm into the input binder. Inputs not critical for arm motion would be disconnected and could thus be don't cares. The arm could then be put through sets of motions, and only the output required for successive arm positions would be entered into the ASOCS system. The system could then adapt, and the arm could be moved to a new position and different outputs could be specified.

This technique could be carried one step further if the outputs are also connected to a mechanical device, rather than input manually. Thus relations between input and output could be entered into the ASOCS without having to manually derive or specify the rule base.

5.3.2. Instance Broadcast

The alternate method of communicating information to the network about a single instance, ***Instance Broadcast,*** uses the broadcast bus. The actual instance is placed on the broadcast bus in a symbolic form. For example, using the same instance defined in the previous section, an encoding of the exact instance $X_1 \bar{X}_2 X_7 \rightarrow Z_1$, or a list of the variable names, could be broadcast on the broadcast bus. In this case, a node must know what its function is in terms of the input variables of the system. For example, figure 5.2 shows a node receiving an instance broadcast.

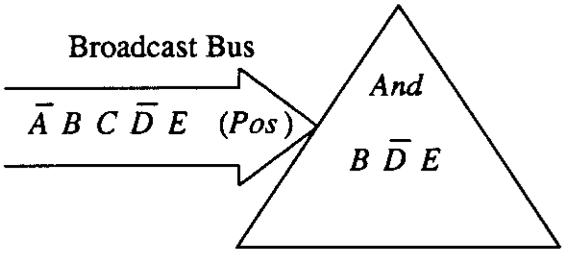


Figure 5.2 - Instance Broadcast

In this case the node knows that it does the *And* function of the variables B, $\bar{D}$, and E. The instance broadcast consists of a list of the input variables occuring in the instance, together with whether the instance is positive or negative. By comparing its function and variables, with the variables written on the broadcast bus, the node can determine its potential involvement in network modification relating to the broadcast instance.

5.3.3. Combination of Presentation and Instance Broadcast

The technique used to inform the network about a new instance has a strong effect on the type of ASOCS algorithm used. In the presentation method a node need know nothing about the *semantics* of the variables it is acting upon, rather the nodes need only know the *function* it produces for different states of its inputs. AA1 uses the presentation technique. The network node need no nothing about the variables coming into it, it is only concerned with what it outputs (functions) for different input patterns.

AA2 and AA3 use instance broadcast as the method of instance introduction. In this case each node must know something about its function in terms of the semantics of its variables (i.e. A node would know if the variables A and B were part of its function). However, AA2 and AA3 use DPLM's which only compute the *And* function and thus a simple list of the variables computed at a node can be stored.

A variation of instance broadcast can be used to allow more general functions in the DPLM. If the function becomes complex, then the memory required to represent the function will increase. Complex functions occur when inversions are allowed between nodes in the network. For example, figure 5.3 shows two nodes with their outputs going to a new node.

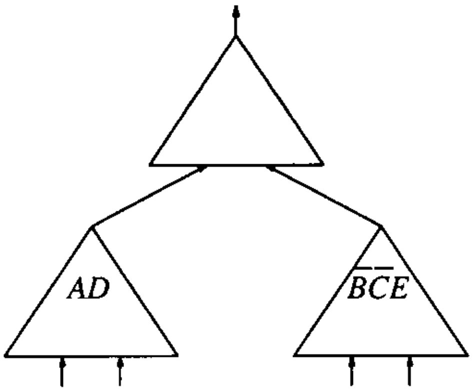


Figure 5.3 - Network Representation

If the function of the top node is the *And* function, then the top node is the conjunction of the union of the bottom nodes, which is $A\bar{B}\bar{C}DE$. When the instance is broadcast the node need only compare its variable list with that of the instance. However if the right input to the top node was inverted, and the top node were still an *And* gate, then its function would be

$$AD\overline{\bar{B}\bar{C}E} = AD(\bar{B} + \bar{C} + E)$$

This type of function would become increasingly "messy" as it passed through more inversions. It would also be more difficult to compare this type of function with the list of variables given in the instance broadcast.

There is a simple way of solving this problem. Rather than store the function of the node, just store the list of variables. Together with the NI broadcast of a list of variables, do a *presentation* of the instance to the logic network. Thus, instead of comparing to ascertain if it matches the complex function, the node need simply look at its output during presentation, to see if it matches the instance. In order to allow this extension, a 3-state DPLM must be used.

5.4. Selection Waves

It is often necessary to select a node or nodes in the network for a specific action. It is desirable that this node selection accomplish three basic conditions.

1. It must select "useful" nodes.
2. It must be able to select a single network node.
3. It should use the network concurrency to speed up the selection.

The method of ***Selection Waves*** fulfills all of the above criteria.

For ease of explanation of this method, and the other methods to follow in this chapter, we assume without loss of generality that there is only one output, and thus one top node in the logic network. The methods discussed are general and can be used for multiple output networks with nodes having responsibility for more than one output.

When a selection wave is to take place, a global command is given. Upon receipt of this command, each node calculates a locally derived *criterion*, or *score*, defined by the algorithm and global command. After calculation of its own score, which is symbolized as a discrete value, a node waits for its two child nodes to send their own scores with a local command. The node then chooses the maximum score between its own and that of its two children. The node then sets its ***Selection State.*** The selection state is a ternary value which can be set to ***Self, Left,*** or **Right.** In summary,

$$\textit{selection state(node)} = \begin{cases} \textit{self} & \textit{if node has maximum score} \\ \textit{right} & \textit{if right child has maximum score} \\ \textit{left} & \textit{if left child has maximum score} \end{cases}$$

Ties are decided depending on system goals. After setting its state the node passes up to its parents either its own value if its state is *self*, or the value of the maximum of its children. After the data has flowed upwards, the top node receives the maximum value that exists in the network. The time for this to happen is proportional to the maximum depth of the network.

The wavelike action is equivalent to a heap sort where the top node will always be sent the highest score in the network. This wave, being half of the total selection wave is called the ***Sort Wave.***

Once the upward sort wave is completed, the downward flow may optionally be initiated. The downward wave is called the ***Arbitration Wave.*** A binary token is passed down to obtain the selection of a single node. The top node may start the passing down of this token, or it can be started from the AU[1], if nodes do not test whether they are the top node.

Each node follows the same steps.

1. If a 0 is received from all parents, then pass a 0 to both children.
2. If a 1 is received from any parent, then
   a. If state is *self*, then this node is selected. Send a 0 to both children.
   b. If state is *left*, send a 1 to the left child and a 0 to the right child.
   c. If state is *right*, send a 1 to the right child and a 0 to the left child.

Thus, one node is always uniquely selected during the arbitration wave. The total time necessary to select a node in the selection wave is proportional to 2 times the number of levels in the network. Thus, selection time $O(d)$ and $\approx O(log(n))$ where d is the depth of the network and n is the number of nodes.

Following is an example of a selection wave in a logical network. Figure 5.4 shows a logical network where each node has calculated its own local score. Each bottom node sets its selection state to *self* and passes up its value to the second layer. In this case the first node in the second layer sets itself to *right* and the second node sets itself to *left*. Both of these nodes pass an 8 up to the top node. The top node can

[1] Recall that the AU can monitor the result of the sort wave.

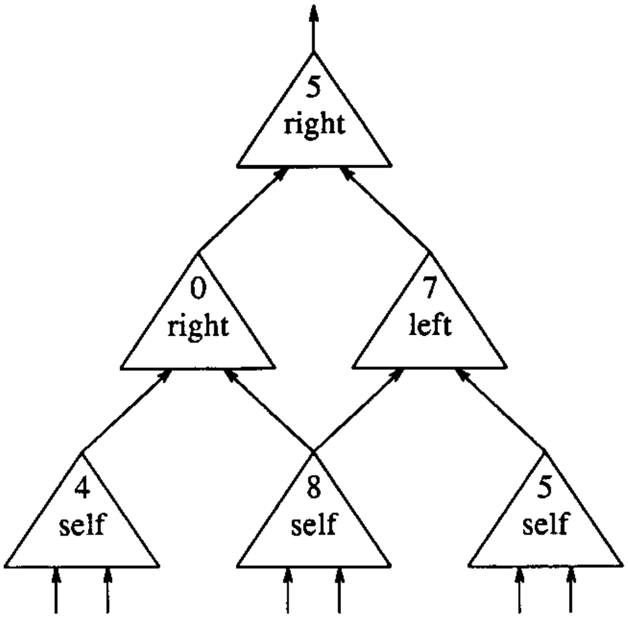


Figure 5.4 - Network with Deduced Scores

arbitrarily decide to set its self to either *right* or *left*. Assume that the top node sets itself to *right*. Then it passes a 1 to its right child and a 0 to its left child. The right child has a state of *left*, so it would pass a 1 to its left child, which is the bottom middle node. Upon receipt of the 1, this node would know that it had been selected.

If the left node on the second layer had been an eight, rather than a 0, it would have decided during the sort wave whether to set its state to *self* or *right*. The decision regarding this choice in case of ties could be made (a priori) on the basis of whether deeper or shallower nodes were best for the overall goal of the selection. Depth and other decision factors could also be part of the criterion evaluation.

5.5. Node Addition and Combination

When a new instance is entered which causes the network to no longer fulfill the instance set, network *modification* or *update* must take place. The first step in the modification of the network is the ***Selection of Nodes*** which will participate in the growth process. When a node is chosen to participate in the growth process, it is labeled as a ***Growth Node.***

5.5.1. New Node Addition

The nodes selected during the selection phase, are not always sufficient to bring the network to complete fulfillment of the instance set. In this case, ***New Node Addition*** is necessary. New node addition refers to the situation when new nodes are allocated for the network, under the guidance of the AU. These are also labeled as growth nodes.

5.5.2. Node Combination

Once a sufficient group of growth nodes has been selected, (or added by new node addition), the nodes are combined. ***Node Combination*** is the method by which growth nodes are connected together until the top node of the modified network is a complete discriminant node. The order of connection and the decision about which nodes should combine is dependent on the algorithm. Node combination consists of

1. Allocation of a new node.
2. Connection of two growth nodes as children of the new node.
3. Setting of the new nodes function and memory by the children.

Figure 5.5 is a representation of a logical network in which nodes A and B and E have been selected during the selection phase.

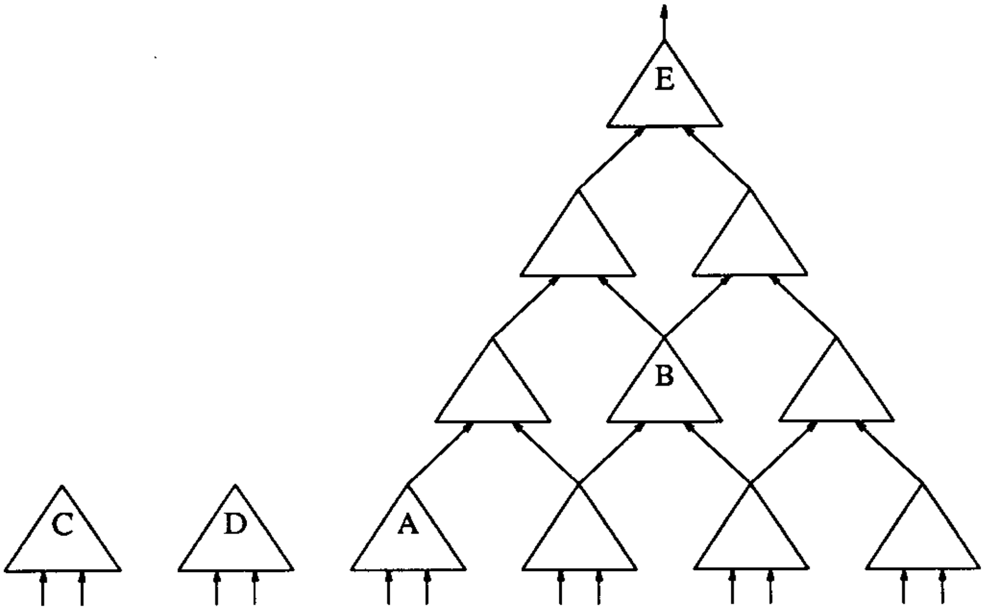


Figure 5.5 - New Node Addition

Nodes C and D are new nodes which have been allocated by new node addition, given variable inputs and functions, and have also been set as growth nodes. Figure 5.6 is a possible node combination which could take place when the AU gives the global growth command. The nodes which are labeled with an "x" are the nodes which had to be allocated to allow the combination of the growth nodes. There will always be $n-1$ new nodes allocated for n growth nodes, since that is the number of connections necessary to reduce n inputs to 1 output.

The order in which the nodes combine may or may not be important, depending on the algorithm. The function of the new parent node is defined locally by the child nodes in each case. In a local growth sequence, two nodes become siblings, a new

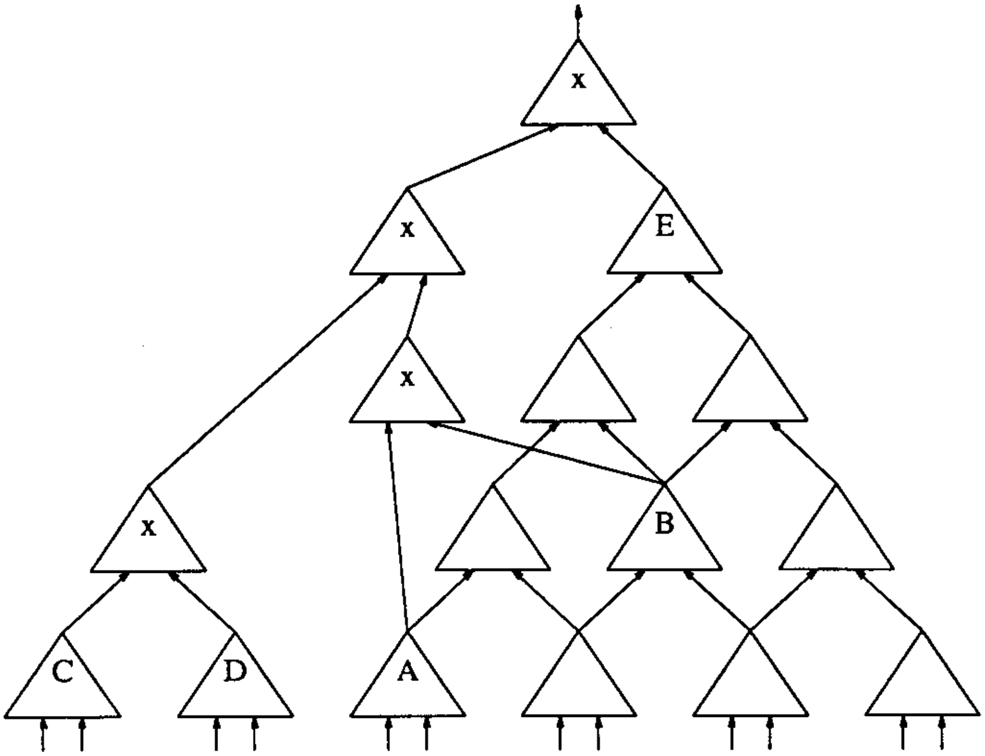


Figure 5.6 - Node Combination

node is allocated to which they send their outputs, and each sibling node sends a ***Set Function*** local command to the newly allocated node, which causes the new node to be set to the correct function. Details of the set function command are algorithm dependent. It may also be necessary at this time for the memory of the new node to be loaded with some information from its children. This loading of information is called ***Memory Inheritance.***

A key issue is the method by which two nodes are selected to combine together and how a new node is allocated for them. The method used to select which nodes combine together are both algorithm and implementation dependent. However, a few possible methodologies are outlined here.

1. They could be chosen according to *topographical* location, depending on shortest distance or shortest path between nodes, or other related measures. Local search wave commands would be used to find closest neighbors.

2. A simple method would be *chronological combination.* The first node selected would automatically join with the second node, etc.

3. A daemon, called the *Allocator,* having extended knowledge of the network structure, could be in charge of both selection of nodes to join, and allocation of new nodes.

4. The AU directs the network through node selection and monitors information on the broadcast bus about nodes which have been selected. The AU could then select which nodes should join together, using the information garnered during node selection.

5. Combination can also be performed on a purely local basis by using the information output by selected nodes. Since all nodes can read the broadcast bus, each node can collect information about each node selected for growth. Since each node has the same deterministic algorithm, each growth node could locally decide which growth node it must combine with, and no outside intervention is needed.

5.6. Self-Deletion

Theoretical optimality, in terms of the number of nodes necessary to fulfill the instance set, is not the goal of this system. However, ***Self-Deletion*** is performed to make the network *relatively* optimal. If a node is no longer playing an active role in the functioning of the network, it can be deleted or removed from the network. It will be seen in the following chapters that it is possible for a node, on a local basis, to decide whether it is useful or not. This section discusses the mechanics of node deletion assuming it has been determined that the node can be deleted.

Self deletion occurs only when the *self-delete* global command is issued, after which actions are undertaken by the network nodes through the use of local commands. A node which determines that it is deletable sends local commands to those of its neighbors which should also be deleted because of their relationship to the original node.

5.6.1. Node States

There are three basic states of nodes in the network, with respect to the instance set. A node which if deleted would not prevent the network from fulfilling the instance set, is called a ***Non-Discriminant Node***. A non-discriminant node can be deleted from the logic network.

A node which gives the correct output for a given bound output variable, for all states of the environment, is a ***Complete-Discriminant Node***. Every top node corresponding to a given output must be a complete discriminant.

Any other node which is not one of the two above types, is simply called a ***Discriminant Node.*** A discriminant node discriminates between at least two discordant instances in the instance set. A node discriminates between two discordant instances when it asserts one truth-value level for all environment states matching one of the instances, and it asserts the opposite level for all environment states matching the other instance.

Proofs showing that nodes can be deleted from the network without changing the network function will be given in the context of a given algorithm.

5.6.2. Non-Discriminant Deletion

The exact method of non-discriminant deletion is detailed for each algorithm. The basic procedure is for the non-discriminant node to delete itself and to send commands to its two children for them to delete themselves. The children recursively send this command to their children and a complete subtree can be deleted. If a child, receiving the delete command, has more than one parent, then the link to the deleted parent is removed.

5.6.3. Complete-Discriminant Deletion

If a node is determined to be a complete discriminant, then the following steps can be made:

1. The node sends a local command to its two children which sets them as *live* nodes. The child nodes recursively send this same command to all of their children.

2. The node sends a local command to all of its parent causing them to self-delete.

3. Before each parent self-deletes they send a local command to both their parent and child nodes, causing them to recursively send the delete message and then self-delete. Using this method, all nodes in the network will receive the self-delete command. If a node has been set as *live*, then it ignores the self-delete command and will not send it to its neighbors. Thus all nodes in the network will self-delete except the nodes in the directed graph rooted at the new complete discriminant node.

4. The output of the complete-discriminant node is bound to the output variable for which it is a complete discriminant. This is done by having the node send a local command up to the former complete discriminant, or to the output binder, which causes the change in binding. The complete discriminant node becomes the new top node in the network for the given output variable.

5.6.4. Locally Redundant Deletion

Another type of self-deletion can occur when a node is determined to be ***Locally Redundant***. A locally redundant node is a discriminant node, which by passing of local information between itself and its neighbors, or other network nodes, is found not to be necessary to the overall fulfilling of the instance set. In other words, the nodes directly around it already compute a superset of the information computed by the locally redundant node. The processing required to discover if a node is locally redundant can be carried out during execution mode since the control unit is not used during that time. The methods for discovering local redundancy are very diverse and are briefly mentioned in the algorithm chapters.

5.7. Fault Detection and Recovery

In a system which is self-organizing, the ability for the system to repair itself without outside intervention is critical. There is no method for an outside agent to address or test specific modules within the system. In the ASOCS system there is a very natural way for both ***Self-Testing*** and ***Self-Recovery*** to occur without any outside awareness of the occurrence. Since during execution mode, the control unit of a network node is not used in the execution process, it can constantly be used for the purpose of self test and recovery.

5.7.1. Self-Testing

Self-testing in the ASOCS system does not really consist of a single node testing itself for faults. Rather, testing of a given node is done by the neighbors to that node. This is because if a node is already faulty, then the self-test mechanism could also be faulty. So, rather than leave the testing responsibilities to a single node, test responsibility is given to all neighbors of a given node. Neighbors to a node include children, parent, and sibling nodes. Depending on implementation the neighbors could also be physically connected nodes which are not logical neighbors. Each neighbor can store information concerning the function of a given node when the node is allocated or modified. Then, the neighbor nodes can constantly check each other, in a *round-robin* fashion, to assure that the tested node is doing the function that it is supposed to. The type of stored information necessary for testing is dependent on the algorithm used.

There are two parts of each node which need testing. They are the control unit and the DPLM. The communication paths associated with each unit are considered part of the unit. The control units can be tested by a derived protocol which in essence asks the tested unit to tell its current function to the asking neighbor. If the answer no longer matches what it was doing, and if no adaptation phase has occurred to cause a change then the tested node can be marked as bad by all of its neighbor nodes. Again, since the node is assumed faulty, it cannot be counted on to mark itself as a bad node.

The other part of the system to be tested is the DPLM. This test is done by cooperation of the children and a parent to the tested node. The children know what the outputs of their DPLM's currently are. They send this information to the parent node (their grandparent). The parent node knows the function of the tested DPLM and what its current output is. If it is incorrect then the node can be marked as bad. If the outputs of the child nodes are changing at too fast a rate to allow this sort of test, then the AU would have to intermittently enter a DPLM test mode where each gate could be tested while the inputs were held steady.

One other important consideration is the use of *Majority Logic* to decide whether a node is faulty. It is not sufficient to mark a node as bad because one neighbor node makes that claim. Either a unanimous or a majority decision is necessary to set a node as bad. If only one single node claims a neighbor is bad, and if this claim is unsubstantiated by the other neighbors, then the accusing node is marked as a bad node, since its testing ability must then be faulty.

For this same purpose of majority decision making, it would be possible to expand the definition of the neighbors of a given node to those nodes within 2 or more immediate connections from a tested node. In this way, *second opinions* can be used before the decision to mark a node as bad.

5.7.2. Self-Recovery

If a faulty node is found then *Self-Recovery* must take place. One method of self-recovery which could be done without leaving execution mode is that of ***Horizontal Self-Recovery.*** If a node is marked as bad by its neighbors, then any unused available node on the same physical level can be chosen as a replacement node. The two child nodes connect their outputs to the newly allocated node, and load it with the old function and memory of the bad node. The output of the new node is then connected to any nodes which used to have a connection from the bad node. Figure 5.7 show an example of this type of self-recovery. The solid lines represent the initial configuration of the connections of the middle node. When the node is discovered as faulty, its connections are routed by its neighbors to the unused node to its right. The new interconnections are shown with dashed lines. It is possible that a *glich* in the output could occur during this self-recovery phase, yet an incorrect output is possible during the entire time after a node has gone bad, until it is replaced.

Another type of self-recovery, which could also be combined with horizontal self-recovery, is ***Guided Self-Recovery.*** In this case, when a node has been discovered

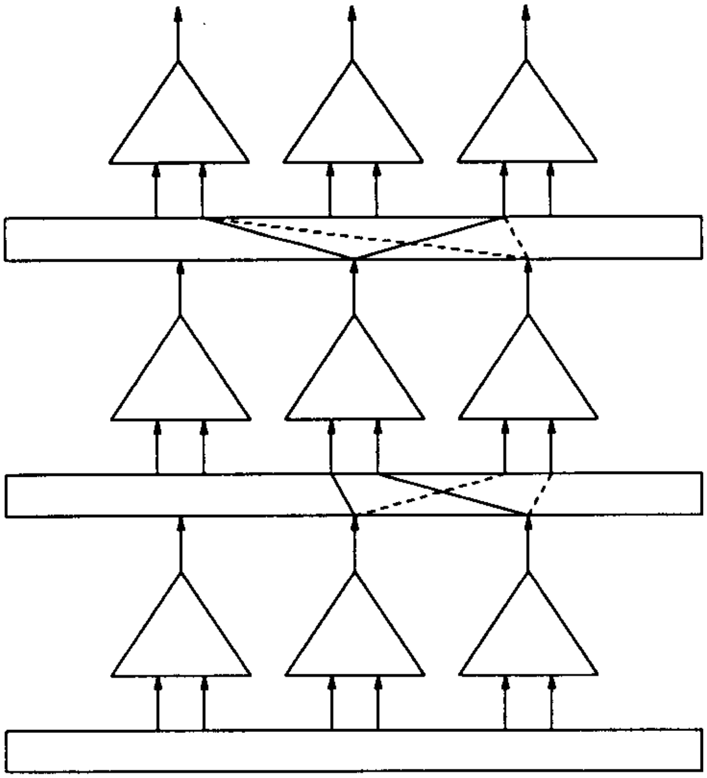


Figure 5.7 - Horizontal Self-Recovery

to be bad, any neighbor node can assert an interrupt line to the AU, which will cause the AU to enter a self-recovery phase. A neighbor node can put information regarding the function of the bad node onto the broadcast bus, and the AU can guide the network through recovery by replacing the function, or by other methods similar to the normal adaptation mechanisms.

Chapter 6

ADAPTIVE ALGORITHM 1

In this chapter we discuss one particular algorithm which can be implemented on the ASOCS architecture, which for identification we call ***Adaptive Algorithm 1 (AA1)***. The first section gives a description of the specific ASOCS architectural requirements needed for AA1. Section 2 discusses the memory structure of the system and the detailed layout of the network nodes. Section 3 gives an informal description of the algorithm. The following section gives a formal description of the algorithm in the form of a structured language. Next an example is shown. The final sections discuss the multiple-output case and gives a summary of AA1.

It is assumed that the primitive mechanisms and architecture discussed in earlier chapters are understood, and they will not be reproduced in each chapter discussing an adaptive algorithm.

For simplicity, we assume that there is only one output variable, Z. However, the algorithm is suitable for any number of outputs, and the multi-output case is discussed later.

In AA1, there is always a single top node for each output variable which is a complete-discriminant node for that variable. When a new instance is added to the set, and if the top node does not already fulfill the new instance, a new node is created which discriminates the new instance from all old discordant instances. This node is created by combining nodes within the network which already discriminate a subset of the old discordant instances from the new instance. If the union of these selected nodes is not sufficient to discriminate the new instance from all discordant instances, then new nodes are allocated which discriminate the remaining instances. Once the new node (which discriminates the new instance from all discordant instances) is built, it can be combined with the old top node (which discriminates all old instances, but not the new instance) creating a new top node which is a complete discriminant node for the given output variable.

6.1. Architectural Constraints

Each adaptive algorithm requires a particular version of the basic ASOCS architecture. AA1 uses the complete architecture discussed in Chapter 4, while the two algorithms to be defined later use a subset of the basic ASOCS architecture. AA1 uses *presentation* as the mode of instance introduction, and thus it requires 3-state DPLM's and use of the presentation bus. Interconnection switching between layers of the nodes is also necessary.

6.2. Memory and Node Descriptions

6.2.1. Instance Table

The instance set is maintained consistent and minimal in the adaption unit and is stored in a data structure called the ***Instance Table (IT)***. The instance table is structured as a table with three columns. In the first column, all positive instances are symbolically stored. This column is called the *Positive (P)* column. The second column contains all negative instances and is called the *Negative (N)* column. The third column contains a one-bit discrimination flag and the column is labeled with *(D)*. This column is used during the selection and new node addition phases. There is no relation between positive and negative instances in the same row.

Initially each cell in the instance table is empty, which state is represented by a "-" (empty marker). When a new instance is input to the AU, it is stored in the first empty location in the instance table. For example, if the NI is a positive instance, then it is stored in the first empty cell in the positive column. Each cell in the positive or negative column of the table corresponds to one instance of the instance set. An example of an instance table is shown in figure 6.1.

P	N	D
$A\bar{B}D$	$\bar{A}\bar{B}\bar{E}$	
BCD	$B\bar{C}\bar{D}F$	
$\bar{A}E\bar{F}$	$AB\bar{D}$	
-	$AB\bar{C}$	
$A\bar{B}\bar{C}$	$B\bar{C}\bar{D}\bar{E}F$	
$\bar{A}BC$	$B\bar{C}\bar{E}G$	
$\bar{A}BD$	-	

Figure 6.1 - Instance Table

6.2.2. Node Table

Each node control unit in the network contains a ***Node Table (NT)***. The function of the NT is to store the value of its DPLM output (1, 0, or ?) when the environment matches a given instance in the instance set. The structure of the NT (i.e., its indexing), in terms of what each cell represents, is the same as that of the instance table. Thus, the third positive cell of every NT, corresponds to the instance in the third positive cell of the instance table. A location in the NT stores only its DPLM output for the instance stored in the same location of the IT. When a new instance is presented to the network, all nodes allocate the next free space in their node table, which is identical for every node, to correspond to the new instance.

Consequently, in the NT there is a four state cell corresponding to each instance in the IT. The four states are represented by "1", "0", "?", and "-". In the presentation phase a node determines its output for all environment states matching any given instance. If a node outputs a 1 for all states of the environment which match the instance assigned to a given cell, then a 1 is put in that cell. If the node outputs 0 for all states of the environment which match the instance, then a 0 is placed in the cell. If the node outputs a 1 for some states and a 0 for other environment states matching the instance, then a "?", signifying *don't know*, is put in the cell. If no instance is currently assigned to a cell in the table, then a "-" is placed in the cell.

Figure 6.2 shows an example of a NT which corresponds to the instance table shown in figure 6.1.

When a global command corresponding to a given instance is broadcast to the

P	N	D
1	0	
?	0	
0	?	
-	1	
?	0	
0	1	
0	-	

Figure 6.2 - Node Table

network, only an index value is necessary to specify which cell in each NT corresponds to the instance. Thus, if the third positive instance in the instance table were to be deleted, a global message "Delete third positive instance" would be sent out. Each node would then set the third cell of its positive column to empty.

6.2.3. Instance Discrimination

At this point we show how a node in AA1 can ***discriminate*** between instances. If a node always outputs one asserted level for all states of the environment matching a particular positive instance, and the opposite level for all states of the environment matching a particular negative instance, then that node discriminates between those two particular instances. Refer to the node table of figure 6.2 for an example. The node always outputs a 1 when the environment matches the positive instance represented in the first cell in the positive column. The node always outputs a 0 when the environment matches the negative instance represented in the first cell of the negative column. Thus, this node discriminates between these two instances. Just because the node outputs a 1 does not mean the positive instance represented by the first cell has been matched, but it does mean that the negative instance in the first negative cell has not been matched. This node discriminates the first positive instance from the first, second, and fifth negative instances. It also discriminates other positive instance from other negative instances.

For any given cell in the node table which is asserted, (a 1 or a 0), that node discriminates the instance represented by that cell from all discordant instances which have the opposite assertion. Thus, the next asserted positive instance, in the third cell of the positive column in figure 6.2, is discriminated from the negative instances represented in the fourth and sixth cells.

Note that it is never necessary to discriminate between concordant instances, since they can never contradict each other. Thus, addition of a NI can never cause an old concordant instance to be modified (except for minimization). This also follows from the partiality of the network which allows it to output any value when an instance is not matched. Thus the system is a discrimination network which is responsible for discriminating positive instances from negative instances.

To fulfill the instance set, every positive instance must be discriminated from every negative instance. Thus, when a NI is input, a node need only compare the NI with cells corresponding to discordant instances.

6.2.4. Node Status

In this section the four basic states of nodes in AA1 are described. A node, by looking at its node table, can categorize and mark itself as one of the four following node states. When discussing values of cells within node tables, only non-empty cells are considered.

6.2.4.1. Discriminant Node

A ***Discriminant Node*** was defined in chapter 5 as a node which discriminates at least one positive instance from one negative instance. In terms of a node table, this means that there must be at least one asserted cell in the positive column, and at least one opposite asserted cell in the negative column. In other words, the node must discriminate between at least one pair of instances. Figure 6.2 is an example of a discriminant node.

6.2.4.2. Non-Discriminant Node

A ***Non-Discriminant Node*** is a node which does not discriminate at least one positive instance from one negative instance. In terms of the node table, a non-discriminant node is any node that does not contain an asserted cell in the positive column and an opposite asserted cell in the negative column. Figure 6.3 is an example of a non-discriminant node.

P	N	D
1	1	
?	?	
1	?	
-	1	
?	?	
1	1	
1	-	

Figure 6.3 - Non-Discriminant Node

6.2.4.3. Complete Discriminant Node

A ***Complete Discriminant Node*** is one which discriminates every positive instance from every negative instance. In terms of the node table, it is one in which all positive cells are asserted at the same value, and all negative cells are asserted at the opposite value. Figure 6.4 is an example of a complete discriminant node.

P	N	D
0	1	
0	1	
0	1	
-	1	
0	1	
0	1	
0	-	

Figure 6.4 - Complete Discriminant Node

The top node of the network must always be a complete discriminant node in order to correctly fulfill the instance set. In a multi-output system, there is typically one top node for each output variable.

6.2.4.4. One-Sided Discriminant Node

A ***One-Sided Discriminant Node,*** which was not mentioned in chapter 5, is a discriminant node which always asserts one value for either all positive or all negative instances, but which can output any value for the discordant instances. The one-sided discriminant node can discriminate a single instance from all discordant instances and is the type of node built to discriminate a new instance from all old discordant instances. Using the node table, a one-sided discriminant node is one in which either the positive or negative column is set to one asserted value. Figure 6.5 is an example of a one-sided discriminant node.

P	N	D
1	1	
?	1	
0	1	
-	1	
?	1	
0	1	
0	-	

Figure 6.5 - One-Sided Discriminant Node

6.2.5. Extensions to Memory Tables

One variation to the way in which node tables are stored, which would be advantageous for some instance sets, is now discussed. The purpose of node tables is to allow each node to know which instances it discriminates between. Don't know values do not contribute to the discrimination to be made. Thus it is not really necessary to store any cells which contain a don't know value.

In order to avoid storage of don't know cells, it is necessary to use a more flexible storage structure. Rather than have each NT in the image of the instance table, each instance must be given a label or identifier by which it can be addressed. By doing this, each node can only store those cells for which it always outputs an asserted value. Thus, when the AU sends a command corresponding to a given instance, it sends an identifier rather than an offset.

The price to be paid is that an identifier must be stored along with each cell. The size of the identifier must be at least n bits, where n is the total number of instances in the IS. The total memory needed at each node may be greater than that needed when the original storage method is used. If each node contains many don't know cells, and this is dependent on the instance set, then this second methodology would be more parsimonious. In this chapter we continue to have node tables in the image of the instance table.

6.2.6. DPLM Functions

DPLM's, in principle, can be set to any one of the 16 boolean functions of two inputs. In AA1 the *Exclusive-Or* and the *Equivalence* functions have not been exploited, although it is possible to conceive schemes where they could be used to minimize the number of nodes in a network. The remaining 14 functions are necessary for an AA1 implementation (some for actual processing and others for communication purposes). In the virtual network, we are not concerned about physical interconnections, and the only necessary functions are the *And* and *Or* functions with all possibilities of inverted inputs. Thus, only the 8 functions shown in table 6.1 are used.

6.2.7. Node Subparts

Besides the node table described above, the memory of each node must contain other information. Each node has a left child and a right child. The children can be either another node or a literal variable. Each node also has a set of parent nodes which is empty for the top node. The node memory contains the current function of

x_1x_2	x_1x_2	x_1x_2	x_1x_2	
00	01	10	11	Function
0	0	0	1	$x_1 \cdot x_2$
0	0	1	0	$x_1 \cdot \bar{x}_2$
0	1	0	0	$\bar{x}_1 \cdot x_2$
0	1	1	1	$x_1 + x_2$
1	0	0	0	$\bar{x}_1 \cdot \bar{x}_2$
1	0	1	1	$x_1 + \bar{x}_2$
1	1	0	1	$\bar{x}_1 + x_2$
1	1	1	0	$\bar{x}_1 + \bar{x}_2$

Table 6.1 - *And* and *Or* Function Permutations

the DPLM and two information flags: *growth-flag* marking whether the node is a growth node, and a *growth-polarity-flag* marking whether the node is a positive or negative growth node.

6.3. Informal Description

In this section we discuss AA1 in an informal less formal manner. Section 4 gives a formal description which is meant to clear any ambiguities or questions arising from the informal description. The most effective way to understand the algorithm is to integrate the reading of the formal section with the informal. This same structural order is used when explaining the other two algorithms in the next chapters.

Again, the current system is dealing with only one output variable. We signal the end of one algorithm cycle, that of processing a new instance, by stating that the system returns to execution mode, or to the input of a new instance.

6.3.1. Instance Set Maintenance

In AA1 the instance is made both consistent and minimal in the adaption unit. Upon receipt of the new instance the instance set is first made consistent. This consistency modification, explained in chapter 3, can cause deletions and modifications of old instances discordant to the NI. Minimization can also cause both deletion and modification to instances which are concordant to the NI.

The algorithmic method used to maintain the instance set is not critical to AA1 and is not discussed here. The result of the maintenance process is two lists of instances. The *delete-list* is a list of old instances which have been deleted from the instance set. The *add-list* is a list of instances which have been modified by the NI. The NI is also a member of the add-list. It is possible that either or both of these lists be empty, in which case the modification cycle would be complete for the given NI. The add-list can only be empty when the NI was deleted by superset minimization.

Following is an example. Assume a current instance set containing the the single instance $A\ B \longrightarrow Z$. The NI $C \longrightarrow \bar{Z}$ is given. The old instance $A\ B \longrightarrow Z$ is now contradicted and will be changed to $A\ B\ \bar{C} \longrightarrow Z$. Thus, the delete-list will contain $A\ B \longrightarrow Z$ and the add-list will contained $A\ B\ \bar{C} \longrightarrow Z$ and $C \longrightarrow \bar{Z}$.

6.3.2. Network Update: Instance Deletion Broadcast

An old instance which is deleted brings about ***Instance Deletion Broadcast.*** Old instances can be deleted by subset contradiction or subset minimization. One instance deletion broadcast is necessary for each instance deleted during the process of making the IS consistent and minimal. In an instance deletion broadcast, the cell location of the deleted instance is broadcast to the network. Every node in the network places an empty marker in the cell corresponding to the deleted instance. An empty marker is also placed in the instance table.

The network will always fulfill the instance set after instance deletion broadcast, because the top node of the network will still remain a complete discriminant node, since making a cell within a complete discriminant node empty can never cause inconsistency. It may be possible for nodes to self-delete after instance deletion broadcast, even though the add-list is empty. In either case, the self-deletion phase will occur just before the return to execution mode.

6.3.3. Network Update: Instance Presentation

All instances in the add-list can be treated the same way. However, instances which are not the NI can typically be handled with much less processing. The mechanisms of AA1 reduces the processing for these cases. In this section we explain the overall modification process for a NI. In a later section we discuss the other types of instances that could be part of the add-list and how their processing can usually be trivially carried out.

If the NI is to be added to the set, then ***Instance Presentation*** takes place as follows.

1. Set network inputs to the values of the NI.
2. Send *Present* global command with the polarity (positive or negative) of the NI.

3. If the output of the top node correctly matches the NI then no further processing is necessary.

4. Otherwise, go to the *Selection* phase.

If there are any new input variables in the NI, the AU binds these variables to a port, through the input binder.

The AU, through the execute/adapt router, causes the input variables which occur in the instance to be asserted, and the rest are set to don't knows as explained in chapter 5. The data then flows up through the network, and each node can record the output of its DPLM.

The global present command is sent to the network, together with the polarity (positive or negative) of the NI. The NI is added to the instance table in the AU. The AU places the NI in the first empty cell in the column corresponding to the polarity of the NI. Each cell in the network also allocates one cell under the same criteria. Thus, each node selects the same cell position in its own NT.

The AU monitors the output of the top node in the network. If the output after presentation matches the consequent of the NI, then the network already fulfills the NI and no modification is necessary.

If the output of the top node does not match the NI[1] then the network must be modified and the system will go on to the selection phase.

[1] If the output is either a don't know or the opposite assertion of the consequent of the new instance, then the output does not correctly match the instance.

6.3.4. Network Update: Node Selection

If network modification must be done, then selection waves are used to select nodes within the network to be part of the modification. The steps for the selection process follow.

1. Set all cells in the discriminate column of the instance table and node tables to 0.
2. Each node calculates its *discriminant count.*
3. A sort wave is initiated.
4. If the result of the sort wave, as seen by the AU through the top node, is a 0, then exit the selection process and proceed to *New Node Addition.*
5. Otherwise, select a node with an arbitration wave.
6. The selected node broadcasts its *discriminant-vector* and marks itself as a growth node.
7. All nodes and the instance table *Or* their discriminate columns with the *discriminant vector.*
8. If the discriminate column still contains 0's, goto 2.
9. Otherwise, go to the *Combination* Phase.

This process is now illustrated by an example. Assume that one node in the network has a node table as shown in figure 6.6 after the presentation of a new negative instance. The bottom cell in the negative column (shown in bold) was the first empty cell in the negative column. The output of the DPLM for this instance is a 1, so that

P	N	D
1	0	0
?	0	0
0	?	0
-	1	0
?	0	0
0	1	0
0	1	0

Figure 6.6 - Node Table after New Instance Presentation

value is placed in the cell.

At the beginning of the selection process, each node sets the values in its discriminate column to 0.

The node in figure 6.6 discriminates the NI from the old positive instances in the third, sixth, and seventh cells of the positive column. Thus, the node discriminates the NI from 3 of the old instances. This value is called the ***Discriminant Count*** for that node. Each node calculates its own discriminant count by summing the number of cells in the column opposite of the new instance which have an opposite assertion to the value in the cell representing the NI, and which have a 0 in the discriminate cell in of their corresponding row. Thus, a node outputting a don't know for a NI always has a discriminant count of 0 since it cannot discriminate any instances from the NI.

Thus with AA1, the discriminant count is the *local score* used during the selection wave as explained in chapter 5. Using the discriminant count, a single node having the highest value is selected during one selection wave. Once the node is selected, it sets itself as a ***Growth Node.*** At this point it puts a 1 in the discriminate column wherever it lines up with an old instance which the node can discriminate from the new instance. For the current example node the discriminate column would appear as in figure 6.7.

P	N	D
1	0	0
?	0	0
0	?	1
-	1	0
?	0	0
0	1	1
0	1	1

Figure 6.7 - Discriminate Column Setting

The node then places the contents of this column, which is called the ***Discriminant Vector,*** on the broadcast bus. Each node in the network then does the logical **Or** of the discriminant vector on the broadcast bus with its own discriminant vector in its discriminate column. After each selection wave, all nodes have the exact same discriminant vector.

The adaption unit is also able to read the broadcast bus, and it also must **Or** the broadcast discriminant vector with its own. Any 0's remaining in the discriminant vector represent instances which are not yet discriminated from the new instance. If this is the case, another selection wave is started. The discriminant count is again calculated at each node, and cells are not counted if they have a 1 in their corresponding discriminate column cell. Thus, only those instances which have not been discriminated yet can be counted towards the discriminant count of a node. An instance which is already discriminated by an earlier selected growth node, is not included in the new discriminant count of the node, even though it may also discriminate it.

Another node is then selected from within the network. The AU checks the maximum discriminant count of the network at the output of the top node after the upward sort wave. As long as the count is greater than 0, a downward arbitration wave is initiated, which selects a new growth node. The selected node then broadcasts its updated discriminant vector and again all nodes update their discriminant vectors accordingly. Once the discriminant vector contains only 1's, then enough nodes have been selected to discriminate the new instance and *combination* can be started. By contrast, if the outcome of the sort wave is 0 and there are still one or more zeros in the discriminant vector (instances which have not been discriminated), then *new node addition* must be done before node combination.

Because the selection process is distributed, there is no guarantee that the minimal set of nodes will be selected. Although all OI's are always discriminated, it is possible that a smaller set of nodes could have discriminated the same number of OI's. A more complex scheme could be used to assure optimal selection.

It should be noted that nodes which have already been selected as growth nodes, and nodes which output a don't know for the NI, still participate in the selection process, although these nodes always calculate a discriminant count of 0 and thus will never be selected. Yet, they must still pass information to parent and children nodes during the sort and arbitration waves, and it is simpler to treat them like any other node.

6.3.5. Network Update: New Node Addition

If after the selection phase there are still old instances which cannot be discriminated by nodes currently within the network, then it is necessary to add new nodes to the network. This addition of new nodes is called ***New Node Addition.***

If there are still non-discriminated instances, the AU will have a 0 in the discriminant column across from each non-discriminated instance. New nodes are added to the network as follows. For each non-discriminated instance in the instance table, one discriminant variable is chosen. New nodes are then allocated and the inputs to these nodes come from the set of selected discriminant variables. Following is an example.

Assume that after the selection process the instance table appears as in figure 6.8, where the NI is the bottom negative cell.

P	N	D
$A\bar{B}D$	$\bar{A}\bar{B}\bar{E}$	0
BCD	$B\bar{C}\bar{D}F$	1
$\bar{A}E\bar{F}$	$AB\bar{D}$	0
-	$AB\bar{C}$	1
$A\bar{B}\bar{C}$	$B\bar{C}\bar{D}\bar{E}F$	1
$\bar{A}BC$	$B\bar{C}\bar{E}G$	1
$\bar{A}BD$	$A\bar{B}\bar{D}\bar{E}$	1

Figure 6.8 - Instance Table after Selection

The possible discriminant variables which can be used are the complement of the variables in the NI. Thus, the possible discriminant variables are: $\overline{A}$, B, D, and E. It is sufficient to choose one of these variables from each of the non-discriminated instances. We know that each discordant instance must contain at least one of these variables, otherwise there would be a contradiction and the instance set would not be consistent.

We must choose one of the above discriminant variables from each of the two instances: $A\ \overline{B}\ D$ and $\overline{A}\ E\ \overline{F}$. From the first instance we can use the variable D and from the second instance we could use either $\overline{A}$ or E. If a discriminating variable were shared amongst the two non-discriminated instances, then one variable would have been sufficient to discriminate both instances. Thus a minimal set of discriminating variables, where at least one variable is found in each non-discriminated instance, is sufficient to build the new nodes.

Assume that D and $\overline{A}$ are chosen. Since there are two variables, one new node will be added. The new node must assert either 1 or 0 when the discriminating variables are both matched by the current state of the environment, and the complement when either is not matched. For two inputs this can always be done using either an *And* gate or an *Or* gate (Figure 6.9).

The AU could either broadcast a message to the network or send a message to the router to cause an unused node to be allocated for this function. The method depends upon the implementation. The node table of the new node must be initialized to correct values. The AU can calculate the positive and negative columns for the new node, since the inputs to the new node are literal input variables, which is the way that

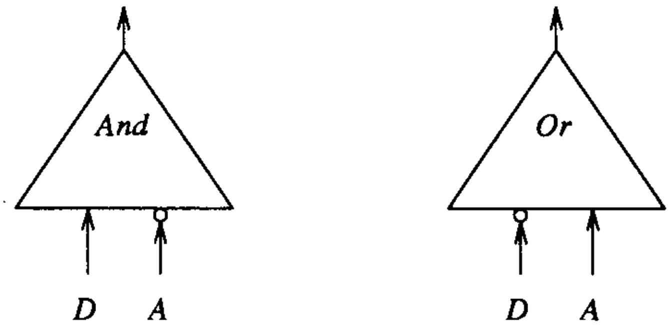


Figure 6.9 - Possible New Node Implementations

instances are stored in the instance table. The AU could then send this information to the node. The new node(s) is then set as a growth node, and it is ready for the combination phase.

If only one variable need be used to do discrimination, or if the number of discriminant variables is odd, it is not necessary to add a new node for the single variable. The variable may be directly combined with a growth node during the combination phase.

6.3.6. Network Update: Node Combination

Node Combination is the process by which growth nodes combine such that a complete discriminant node is formed. The method used to select which nodes to combine is implementation dependent. Some of the options were briefly discussed in chapter 5. For this section, we assume that two nodes are grouped as growth siblings, and an unused node is allocated to be the new parent node. Below are the basic steps.

1. While there are growth nodes in the network
    a. Growth nodes are paired and a new node is allocated

for each pair of growth nodes.

b. The function and the memory of the new node are set by the child nodes.

3. The last created growth node (a one-sided discriminant node) and the old top node combine.

4. The function and memory of the new top node are set by the child nodes.

The order in which nodes are combined, or which nodes are connected to which, is not critical except in terms of the depth of the network. The same number of combinations take place regardless of the connection order. For this section we assume that any two growth nodes can be combined, since the combination criteria is not here defined. The top node of the network is set as a growth node after all other growth nodes have combined.

Every growth node in the network, except possibly the top node, outputs an asserted value for the NI. A growth node which outputs a 1 when the NI is matched is called a ***Positive Growth Node.*** A growth node which outputs a 0 when the NI is matched is called a ***Negative Growth Node.*** A positive growth node outputs a 0 when any of the old instances which it discriminates from the NI is matched by the environment, and vice versa for a negative growth node. The function of the newly allocated parent node is determined by the *growth polarity* of the children nodes. For each combination of children growth polarities, the parent node can be set in two possible manners: one which makes the parent node a positive growth node and one which makes it a negative growth node. Except for the top node, it does not really matter whether a new node is a positive or a negative growth node. It would be sufficient to force all nodes to a single growth polarity. This information is passed to the new parent by the ***Set Function*** local command. Table 6.2 shows the functions which a new parent node can take on depending on whether the children nodes are positive or negative growth nodes.

Left Child	Right Child	Parent Function	Parent Growth Polarity
Negative	Negative	$x_1 + x_2$	Negative
Negative	Negative	$\bar{x}_1 \cdot \bar{x}_2$	Positive
Negative	Positive	$x_1 + \bar{x}_2$	Negative
Negative	Positive	$\bar{x}_1 \cdot x_2$	Positive
Positive	Negative	$\bar{x}_1 + x_2$	Negative
Positive	Negative	$x_1 \cdot \bar{x}_2$	Positive
Positive	Positive	$\bar{x}_1 + \bar{x}_2$	Negative
Positive	Positive	$x_1 \cdot x_2$	Positive

Table 6.2 - Function Settings for New Nodes

The same functions are used, as shown in the table, regardless of whether the NI is a positive or a negative instance.

After the function of the new node is set, it can have its node table loaded by its two children nodes. This is done by applying the new function to the cell values of the children cells, and then putting the output into the node table of the new node. This, as before mentioned, is known as ***Memory Inheritance.*** Once the new node's table is filled, it sets itself as a growth node, and can then be combined with some other growth node. These combinations of nodes can go on concurrently. For *n* original growth nodes in the network, there are exactly $n-1$ combinations that take place, adding $n-1$ new nodes to the network.

Following is an example of the combination of two growth nodes. We continue to assume for the following examples that the NI is represented by the bottom cell of the negative column. Figure 6.10 shows the node tables of the two growth nodes which are about to be combined.

P	N	D
1	0	0
?	0	0
0	?	1
1	1	0
?	0	0
0	1	1
0	**1**	1

P	N	D
?	1	0
1	?	1
0	0	0
1	0	1
1	0	1
1	?	1
?	**0**	0

Figure 6.10 - Left and Right Child Tables

If the parent node is chosen to be a positive growth node, then the function of the parent node must be $x_1 \cdot \bar{x}_2$, as shown in table 6.2. Figure 6.11 shows the table of the new parent node after combination has taken place. After combination of these two nodes, only the first positive instance remains non-discriminated. The discriminate column was filled in for explanation purposes. It would not actually be updated during combination. Note that the number of instances discriminated by the NI cell always increases as nodes are combined.

When all of the growth nodes have been combined, with the exception of the top node, a *one-sided discriminant node* has been created which discriminates the NI from

P	N	D
?	0	0
0	0	1
0	?	1
0	1	1
0	0	1
0	?	1
0	1	1

Figure 6.11 - New Parent Node with DPLM Function $x_1 \cdot \bar{x}_2$

all of the old discordant instances. The one-sided discriminant node is either a positive or negative growth node, and it has a form as shown in figure 6.12.

P	N	D
0	0	1
0	0	1
0	?	1
0	1	1
0	0	1
0	?	1
0	1	1

Figure 6.12 - New One-Sided Discriminant Node

This one-sided discriminant node is then combined with the old top node to form a new complete discriminant node.

The top node, which was a complete discriminant node before presentation, will now be a one-sided discriminant node with the node table shown in figure 6.13, since only the NI cell will have changed. The cell corresponding to the NI in the old top node is always either a don't know, or the opposite assertion of the consequent of the

P	N	D
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	0	0
1	?	0

Figure 6.13 - Old Top Node

NI. If it had been the same assertion as the consequent of the NI, then the network would not have needed to be modified, and this would have been detected in the presentation phase. Note that the discriminant vector of the old top node always contains all 0's because the NI cell is either a don't know, or it is the same as all the discordant instances. Thus, it cannot discriminate any OI from the NI, although it discriminates all of the other positive instances from all negative instances. The new one-sided discriminant discriminates all OI's from the NI and thus by combining these two, all instances can be discriminated.

The way in which the new one-sided discriminant node and the old top node are combined is not immediately obvious. There are four ways in which the nodes can combine, depending on whether the new one-sided discriminant is a positive or a negative growth node, and whether the NI is positive or negative (Table 6.3).

To explain this intuitively, assume that we have the old top node and new one-sided discriminant (NOSD) as shown in the above figures. If the output of the NOSD is a 0, then we know nothing, in terms of the NOSD, about what the network output

New Instance	New One-Sided Discriminant	New Top Node Function
Negative	Negative	$x_1 \cdot x_2$
Negative	Positive	$\bar{x}_1 \cdot x_2$
Positive	Negative	$\bar{x}_1 + x_2$
Positive	Positive	$x_1 + x_2$

Table 6.3 - New Top Node Function

should be, because the NOSD can output 0 for either positive or negative instances. However, if the output of the NOSD is a 1, then we know that the network output must be a 1, since the NOSD outputs 0 for all negative instances matched by the environment. Now, if the NOSD outputs a 0, then we know that the NI has not been matched, since the NOSD always outputs a 1 when the NI is matched. So, when the NOSD outputs a 0, then the output of the old top node gives the correct network output, since it outputs correctly for all instances except the new instance. For this case, having a negative new instance and a negative NOSD growth node, the function is $X_1 \cdot X_2$ as shown in table 6.3.

At this point the network again has a complete discriminant as the top node. The system will then go to the self-deletion phase.

6.3.7. Network Update: Self-Deletion

The basic steps for self deletion are as follows.

1. Do complete-discriminant deletion.
2. Do non-discriminant deletion.
3. Do locally redundant deletion.

4. Return to execution mode.

After the network has been modified, or if the node tables were modified by minimization or subset contradiction, the self-deletion global command is issued. Each node tests its node status to see if it can be deleted from the network.

In AA1 the self-deletion phase is initiated after all network modifications have taken place. Alternatively, it is possible to do the self-deletion after the instance deletion phase and before the modification phase. This variation causes trade-offs in the overall network efficiency and will be evaluated in the simulation chapter.

6.3.7.1. Complete Discriminant Deletion

The deletion process for a complete-discriminant node was described in chapter 5. However, one remark about complete discriminant deletion in AA1 is necessary.

If a node discovers that it is a complete discriminant, it can begin the deletion process of all nodes above it. The output of the complete-discriminant node will replace the output of the old top node. The node must then check itself to be sure that it outputs a 1 for positive instances. If it currently outputs 1 for negative and 0 for positive, then the function must be inverted by inverting its inputs and changing from *And* to *Or* or vice versa. This also inverts its node table. The node would then correctly fulfill the output.

6.3.7.2. Non-Discriminant Deletion in AA1

This section is divided into two parts. First we show what type of nodes are non-discriminant and prove that the network can function correctly without them.

Next, the method of non-discriminant deletion, once a node is determined to be non-discriminant, is shown.

6.3.7.2.1. Proof of Non-Discriminant Nodes

The proof of whether a node is non-discriminant in AA1 can be obtained exhaustively. That is, we show that for each type of non-discriminant node, combined with a discriminant node, that the parent node can never discriminate more than the discriminant child node. The parent node can therefore always be replaced by the discriminant child. In the following examples the left columns of the table represent the non-discriminant node (NDN). The next two columns represent the discriminant node (DN), and the 4 right columns the parent node for the *And* (PA-and) and the *Or* (PA-or) functions. It is only necessary to show the *And* and the *Or* functions since an inversion of a line only complements a node table and this can be undone by an inversion of the output of the PA. We show the positive and negative columns of a node but do not show the discriminate column.

The first example is the case when the non-discriminant node contains only one type of value (Table 6.4).

Children				Parent			
NDN		DN		PA-and		PA-or	
P	N	P	N	P	N	P	N
1	1	1	1	1	1	1	1
1	1	0	0	0	0	1	1
1	1	?	?	?	?	1	1

Table 6.4 - NDN with One Value

A unary value NDN combines to either make a copy of itself or of the DN in the parent. It is not necessary to show the unary value NDN which is all 0's since the two cases are complementary. The same complementary notion applies for the following examples.

Table 6.5 shows an example of a binary NDN, containing two values.

Children				Parent			
NDN		DN		PA-and		PA-or	
P	N	P	N	P	N	P	N
1	?	1	1	1	1	1	1
?	1	0	0	0	0	?	1
1	1	?	?	?	?	1	1

Table 6.5 - NDN with Two Values

The result of combining a binary NDN with a DN is either a copy of the DN or another NDN.

The last case is the NDN with all three values (Table 6.6).

Children				Parent			
NDN		DN		PA-and		PA-or	
P	N	P	N	P	N	P	N
0	?	1	1	0	?	1	1
0	?	0	0	0	0	0	?
0	?	?	?	0	?	?	?
1	?	1	1	1	?	1	1
1	?	0	0	0	0	1	?
1	?	?	?	?	?	1	?

Table 6.6 - NDN with Three Values

This case is a little more tricky. We can just look at the PA-and since the PA-or is complementary. The instances discriminated in the PA are a subset of those discriminated in the DN. In the DN each of the 4 asserted positive cells discriminates 2 discordant cells. In the PA, only one of those 4 positive cells discriminates any longer, and it discriminates the same two instances that it did in the DN. Thus, the PA always discriminates a subset of the DN, and can be replaced by the DN.

6.3.7.2.2. Non-Discriminant Deletion Procedure

If a node is determined to be a non-discriminant node (NDN) then:

1. The NDN sends a local command telling its children to self-delete. Each child node will self-delete unless it has another parent link, in which case only the connection to the deleting parent is removed. This process repeats for the child nodes and a deletion wave will continue recursively until all descendants either have another parent link, or until the bottom of the network is reached. In the latter case any input variables entering the node can be removed and unbound unless used in another node.

2. The NDN sends a local delete command to its parent nodes. The sibling of the NDN is then combined with each parent of the NDN's parent nodes (i.e. all of NDN's grandparent nodes). The function of the grandparent nodes is then reset by the ***modify function*** command which independently originates from each parent of the NDN, and is sent to the parents of each parent.

3. The non-discriminant node, and the parent node self-delete.

For example, figure 6.14 shows a logical network (where each node is numbered only for identification).

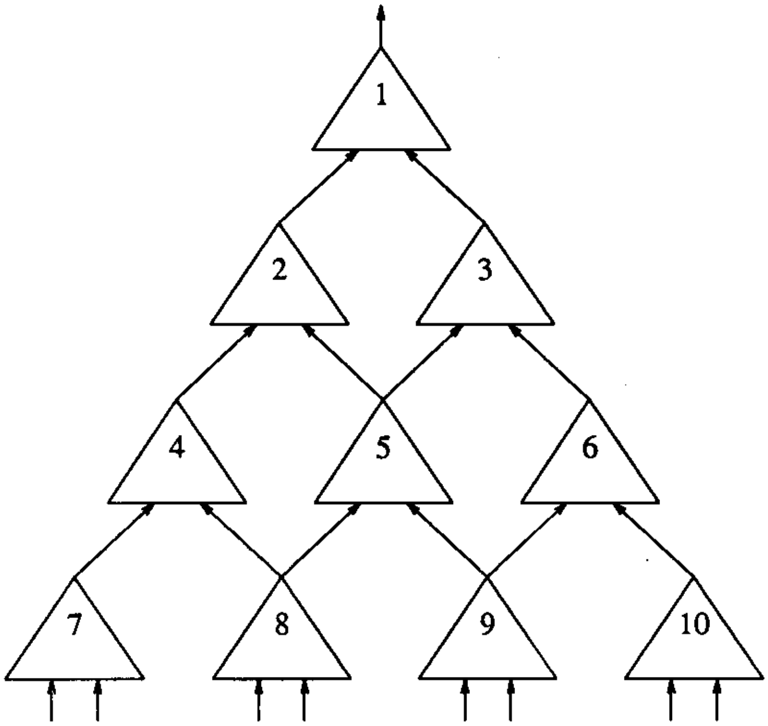


Figure 6.14 - Logical Network

Assume that node 6 has been determined to be a non-discriminant node. Following the above steps:

1. Node 6 is deleted.

2. Node 10 and all nodes under it are deleted, except nodes with multiple parents. Node 9 is not deleted since it has an output to node 5.

3. Node 3 is deleted.

4. Node 5 connects to node 1 using the modify function local

command which passes the old function of node 3 and which of node 3's children were deleted, to node 1.

The modified network would appear as in Figure 6.15.

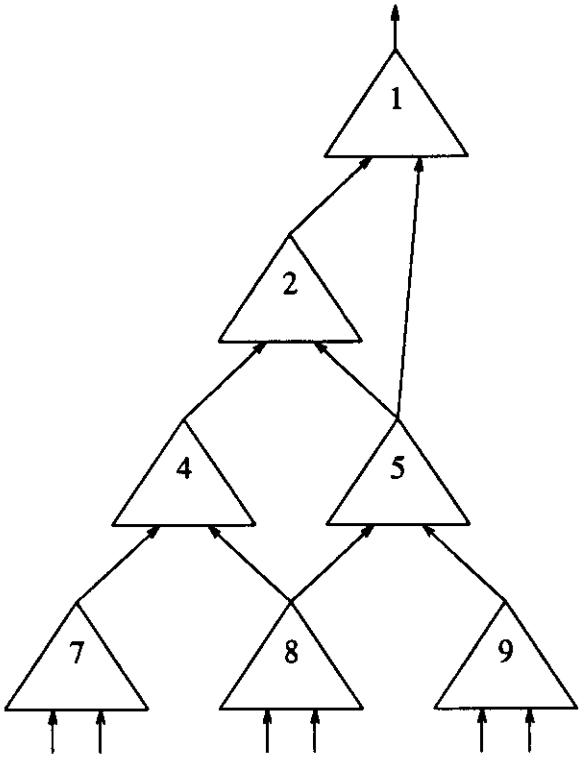


Figure 6.15 - Modified Network

6.3.7.2.3. Modify-Function Local Command

The modify-function command is only needed if a non-discriminant node has one or more parent nodes. When the deletion of a node has taken place, the parent of the deleted node has only one input going into it from the sibling of the deleted node. The parent node then becomes a one input-one output node which can do no more than either invert or not invert its input. Thus, the parent can be deleted and the output of the sibling node can be sent to the parent of the parent node. However, the output of the sibling may need to be inverted before entering its grandparents. Whether or not it is to be inverted depends on the old function of the parent node. If that function caused the input of the sibling to be inverted, then it must also be inverted. This inversion must now be done at the grandparent nodes, rather than at the sibling, since the sibling could have other parent nodes. These inversions must be done by changing the function of the grandparent nodes.

In the following explanation the numbers of the nodes in figure 6.14 are used to aid in understanding. When a non-discriminant node (6) is to be deleted, it sends a message to its parent node (3) which then sends the modify-function command to its parent (1). Parameters of this command are a) the function of the parent node (6), and b) which of its children is the sibling (5) to the non-discriminant node (6). This allows the grandparent (1) of the non-discriminant node to know whether and which variable it should invert. Table 6.7 summarizes for which functions of the parent node (3), must the grandparent (1) change its own function. The function column is the function of the parent node (3). The next two columns tell whether the grandparent (1) would have to change its function if the sibling node of the non-discriminant node were the left input to the parent or the right input; x_1 or x_2 respectively. The new function columns show what the new function of the grandparent (1) to the non-discriminant node, which will become the parent to the sibling, would be changed to if the sibling were either the x_1 or the x_2 input to the parent (3) node.

In the special case when the sibling to the non-discriminant node is a child of the grandparent of the non-discriminant node, then the parent of the non-discriminant

Function of Parent	Invert if Sibling is		New Grandparent Function if Sibling is	
	x_1	x_2	x_1	x_2
$x_1 \cdot x_2$	no	no	No Change	No Change
$x_1 \cdot \bar{x}_2$	no	yes	No Change	$x_1 \cdot x_2$
$\bar{x}_1 \cdot x_2$	yes	no	$x_1 \cdot x_2$	No Change
$x_1 + x_2$	no	no	No Change	No Change
$\bar{x}_1 \cdot \bar{x}_2$	yes	yes	$x_1 \cdot \bar{x}_2$	$\bar{x}_1 \cdot x_2$
$x_1 + \bar{x}_2$	no	yes	No Change	$x_1 + x_2$
$\bar{x}_1 + x_2$	yes	no	$x_1 + x_2$	No Change
$\bar{x}_1 + \bar{x}_2$	yes	yes	$x_1 + \bar{x}_2$	$\bar{x}_1 + x_2$

Table 6.7 - Modify Function Table

node is recursively flagged as a non-discriminant node also.

6.3.7.3. Locally Redundant Deletion in AA1

As discussed in chapter 5, locally redundant deletion can be as general as desired. For this algorithm, we discuss two examples of locally redundant deletion. Neither of these methods is shown in the formal algorithm although they could easily have been.

One method follows the same steps as non-discriminant deletion. If the discrimination done by a parent node is a subset of one of its children, then the parent and the sibling branch can be deleted. However in this case, none of the three nodes independently are non-discriminant nodes. The rule for whether a parent node is locally redundant is as follows; If a node does not assert a don't know cell of a child, and it does not invert an asserted cell of a child using its function (*And, Or*) and not the inversion of a line, then the node is locally redundant. If a parent node table has not matched one of the two conditions in the above rule, then it is discriminating a subset, possibly proper, of the child node and can be replaced by the child node. This follows from the fact that the only way in which any cell in a node table can discriminate more[2] instances in a modified node table, is if at least one discordant cell, which was not asserted opposite before, has been asserted opposite to the discriminating cell.

Another type of redundant deletion which could be carried out periodically, like garbage collection, is to have each node, in a round robin fashion, place the contents of its node table on the broadcast bus. It would also send a token to its children nodes marking all its descendants. Then, any node which is not a descendant of the broadcasting node which has a node table which is a subset, in terms of discrimination, of the broadcast node, can mark itself as a non-discriminant node. It could then delete itself and all of its children. The output of the broadcasting node would then be input to the parent of the deleted node at the input which used to come from the deleted node.

There are many methods by which locally redundant deletion could be implemented in order for the network to approach theoretical optimality.

6.4. Formal Algorithmic Description

In this section we give a formal description of AA1 in the form of a structured programming language. The goal of this section is to have a description which is unambiguous and which explains completely the algorithm.

The language used to describe the algorithm follows the generic syntax of a

[2] Or a different set of instances.

Pascal-like programming language. Some unique language constructs are here defined in order to show where parallelism is inherent in the problem. Language constructs are shown in bold type. Procedures and functions are italicized. Procedures which return arguments as results have those arguments marked by "result". **For each** is a sequential operator and **for all** is a parallel operator. * signifies all elements of an array dimension. ε signifies element of a set.

The data types and data structures of the language are first defined. A high-level algorithm then follows giving the overall flow of AA1. The procedures used in the high level description are then defined in detail. The empty marker for a cell is represented as "nil".

Detailed procedural descriptions for ***Present, Sort Wave, Arbitration Wave,*** and *Complete-Discriminant Deletion* are not included below, since they were detailed in chapter 5.

Data Types

set:

list:

boolean:

integer:

function: One of the 16 dyadic boolean functions.

variable: Variable which occurs in an instance (i.e. A , $\bar{B}$, etc.)

instance: Represents a single instance. Its subparts are:

instance.variables is a list of the variables in an instance.

instance.polarity is positive or negative. $\overline{\textit{polarity}}$ is the complement of polarity.

Data structures

instance-table 3 column array. Columns are labeled: negative, positive, and discriminated.

Each cell contains the instance.variables for a single instance.

node-table 3 column array with the same structure as the instance-table.

Cell values in the node-table can be 1, 0, ?, or nil.

broadcast-bus Represents the broadcast bus.

node Represents a single network node. Its subparts are:

node.left-child - Either a node or a variable.

node.right-child - Either a node or a variable.

node.parents - Set of nodes.

node.dplm-function - Function of the node's DPLM.

node.node-table - Node-table of node.

node.dplm-output - Current output of the node's DPLM.

node.growth-flag - Tells if node is a growth node or not.

node.growth-polarity - Either positive or negative growth polarity.

top-node Top node of the network

High Level Description

procedure: *AA1-Update*(NI)

declare NI: instance

```
declare         Delete-List: list of instances
declare         Add-List: list of instances
begin
    Make-Consistent-and-Minimal(NI , result: Delete-List , result: Add-List)
    Delete-Instances(Delete-List)
    Add-Instances(Add-List)
    Self-Deletion
end
```

Individual Procedures

Make-Consistent-and-Minimal receives a new instance (NI) as a parameter. It assumes the instance set as a global structure and can modify it. It makes the instance set consistent and minimal. It returns a list of instances which have been deleted (Delete-List), and a list of new instances to be added because of consistency and minimization modifications (Add-List). The Add-List also includes the NI. Either or both lists may be nil. Implementations for this routine are independent of AA1 and are discussed in chapter 3.

```
procedure: Delete-Instances(Delete-List)
declare         Delete-List: list of instances
declare         row: integer
begin
    for each instance in Delete-List
    begin
        row ← index of cell of instance in the instance-table
        *Broadcast* (instance.polarity , row)[3] of instance
        **for all** nodes node.node-table(instance.polarity , row) ← nil
        instance-table(instance.polarity , row) ← nil
    **end**
**end**


**procedure:** *Add-Instances(Add-List)*
**declare**          Add-List: list of instances
**declare**          row: integer
**begin**
    **for each** instance in Add-List
    **begin**
        row ← index of the first cell in the instance-table with value nil
          in column instance.polarity.
        instance-table(instance.polarity , row) ← instance.variables
        *Present*(instance)[4]
        **for all** nodes node.node-table(instance.polarity , row) ← node.dplm-output
        **if** top-node.dplm-output ≠ instance.polarity **then** *Reconfigure*(instance , row)[5]
    **end**
**end**


**procedure:** *Reconfigure*(instance , row)
```

[3] Instance.polarity specifies the positive or negative column, while row specifies which cell in the column.

[4] Presentation causes dplm-output to be set to 1, 0, or ? for each node.

[5] This can only occur if the instance is the NI, or if it is discordant to the NI and it was modified due to *minimization*.

```
declare          instance: instance
declare          row: integer
begin
    Node-Selection(instance , row)
    Node-Addition(instance)
    Node-Combination(instance)
end

procedure: Node-Selection(instance , row)
declare          instance: instance
declare          row: integer
declare          done: boolean
declare          Selected-Node: node
declare          Discriminant-Count: integer
declare          Wave-Result: integer
begin
    for all nodes node.node-table(discriminated , *) ← 0
    Done ← false
    while not Done
    begin
        for all nodes
        begin
            Discriminant-Count ← 0
            if node.node-table(instance.polarity , row) ≠ "?" then
            begin
```

```
                for each index (i) if node.node-table($\overline{\textit{instance.polarity}}$ , i) =
                  Complement(node.node-table(instance.polarity , row)) then
                    Discriminant-Count ← Discriminant-Count + 1
            end
        end
        for all nodes do Sort-Wave(Discriminant-Count , result: Wave-Result)
        if Wave-Result > 0 then
        begin
            for all nodes do Arbitration-Wave(result: Selected-Node)
            Selected-Node.growth-flag ← growth-node
            broadcast-bus ← Selected-Node.node-table(discriminated , *)
            for all nodes node.node-table(discriminated , *) ←
              [node.node-table(discriminated , *) OR broadcast-bus]
            instance-table(discriminated , *) ←
              [instance-table(discriminated , *) OR broadcast-bus]
        end
        else Done ← true
    end
end


procedure: Node-Addition(instance)
declare          instance: instance
declare          discriminant-set: set of variables
declare          discriminant-variables: set of variables
declare          discriminant: variable
declare            new-node: node
begin
    if instance-table(discriminated , *) $\neq$ all 1's then
    begin
        discriminant-variables ← Complement(instance.variables)
        discriminant-set ← $\phi$
        for each row in instance-table
        begin
            if instance-table(discriminated , row) = 0 then
            begin
                discriminant ← any one variable where
                  variable $\varepsilon$ instance-table($\overline{instance.polarity}$ , row) AND
                  variable $\varepsilon$ discriminant-variables
                discriminant-set ← add-to-set(discriminant)
            end
        end
        while size-of(discriminant-set) $\geq 2$
        begin
            new-node ← allocate-new-node()
            new-node.left-child ← select-and-delete-one-element(discriminant-set)
            new-node.right-child ← select-and-delete-one-element(discriminant-set)
            new-node.dplm-function ← function according to Table 6.2[6]
            load-node-table(new-node.dplm-function , new-node.left-child ,
```

[6] The tables are found in the corresponding section of the informal discussion.

```
                new-node.right-child)
            new-node.growth-flag ← growth-node
        end
        if size-of(discriminant-set) = 1 then[7]
        begin
            new-node ← allocate-new-node()
            new-node.right-child ← select-first-element(discriminant-set)
            new-node.dplm-function ← right
            load-node-table(new-node.function , nil , new-node.right-child)
            if instance.polarity = new-node.right-child.polarity then
              new-node.growth-polarity ← positive
            else new-node.growth-polarity ← negative
            new-node.growth-flag ← growth-node
        end
    end
end
```

```
procedure: Load-node-table(function , left , right)[8]
declare         function: function
declare         left: node or variable
declare         right: node or variable
begin
```

[7] This method is not necessary. Rather than allocate a node for a single variable, it is sufficient to let that variable act as if it were a growth node when the *node combination* phase is reached. However, in order to have a uniform *node combination* routine, a node is allocated.

[8] If a child is a variable, then the values are taken from the instance-table

```
    for all rows
      node-table(negative , row) ←
        [(left.node-table(negative , row))  function  (right.node-table(negative , row))]
    for all rows
      node-table(positive , row) ←
        [(left.node-table(positive , row))  function  (right.node-table(positive , row))]
end


procedure: Node-Combination(instance)
declare             instance: instance
declare             new-node: node
begin
    while number of growth nodes in the network ≥ 2
    begin
        new-node ← allocate-new-node()
        new-node.left-child ← any single growth node
        new-node.left-child.growth-flag ← non-growth
        new-node.right-child ← any single growth node
        new-node.right-child.growth-flag ← non-growth
        new-node.dplm-function ← function according to Table 6.2
        load-node-table(new-node.function , new-node.left-child , new-node.right-child)
        new-node.growth-flag ← growth-node
        new-node.right-child.parents ← new-node.right-child.parents + new-node
        new-node.left-child.parents ← new-node.left-child.parents + new-node
    end
    new-node ← allocate-new-node()
    new-node.left-child ← last growth node[9]
    new-node.right-child ← top-node
    new-node.dplm-function ← function according to Table 6.3
    load-node-table(new-node.function , new-node.left-child , new-node.right-child)
    top-node ← new-node
    new-node.right-child.parents ← new-node.right-child.parents + new-node
    new-node.left-child.parents ← new-node.left-child.parents + new-node
end

procedure: Self-Deletion
begin
    for all nodes
    begin
        if (node.node-table(negative , *) = (all 1's or all 0's)) and
          (node.node-table(positive , *) = Complement(node.node-table(negative , *) then
        begin
            Complete-Discriminant Deletion(node)
            if node.node-table(negative , 1) $\neq$ 0 then
              node.dplm-function ← invert(node.dplm-function)
        end
    end
    for all nodes
```

[9] The last growth node will be a one-sided discriminant node.

```
        begin
            if there does not exist (node.node-table(negative , i) = (1 or 0) and
              (node.node-table(positive , j) = Complement(negative , i)) then
                Non-Discriminant-Deletion(node)
        end
    end


    procedure: Non-Discriminant-Deletion(nd-node)
    declare            nd-node: node
    begin
        Delete-Wave(nd-node.left-child , nd-node)
        Delete-Wave(nd-node.right-child , nd-node)
        for all nd-node.parents  (nd-node-parent)
        begin
            for all parent-nd-node.parents  (grand-parent)
            begin
                if grand-parent ε Sibling(nd-node).parents then
                    Non-Discriminant-Deletion(nd-node.parent)
                else if parent-nd-node.dplm-function inverts nd-node according to Table 6.7 then
                    Set function of grand-parent according to Table 6.7
            end
            Remove-Node(nd-node-parent)[10]
        end
```

[10] *Remove-Node* removes the node from the network.

```
    Remove-Node(nd-node)
end


Delete-Wave(d-node , parent-node)
declare         d-node: undetermined[11]
declare         parent-node: node
begin
    if d-node is a node then
    begin
        if Size-of(d-node.parents) > 1 then
            d-node.parents ← d-node.parents - parent-node
        else
        begin
            Delete-Wave(d-node.left-child , d-node)
            Delete-Wave(d-node.right-child , d-node)
            Remove-Node(d-node)
        end
    end
end
```

6.5. Add-List Instances other than the New Instance

Any instance which has been modified but not deleted will pass through the same modification mechanism as the NI. One instance modification cycle is necessary for

[11] It could be either a node or a variable.

each modified old instance. These modified instances are placed in the add-list by either overlap contradiction with the NI, or one-difference superset minimization. However, although these instances are in the same add-list as the NI, the effort of the modification cycle will usually be much simpler. Following is an explanation of how the two types of modified instances are handled.

A modified instance caused by overlap contradiction always has one more variable than it did before and thus its representation space is cut in half. Following the proof in the knowledge base chapter, a cell in a node can either not change, or go from a don't know to an asserted value, when the modified instance is presented. Since the top node cannot then be changed by this method,[12] the network never needs modification following the presentation. Thus the modification cycle does not proceed beyond presentation phase, since network modification is never needed. The only action is to update the node tables within the already extant nodes.

A modified instance caused by one-difference superset minimization has one less variable than it did before. Thus, its representation space is double what it was before. Thus, a cell in a node can either not change, or go from an asserted value to a don't know value. This type of change could cause the network to no longer fulfill the instance set, since a top node could lose its complete discriminant status. In this case, the rest of the modification cycle must be completed to restore the complete discriminant top node.

[12] This is because all cells in the complete discriminant node are already asserted.

6.5.1. Extension to Modification Cycle

Because of the situation explained in the previous section, it is possible that more than one modification cycle will be necessary in order to change the network. These multiple cycles are called ***Modification Iterations.*** Simulations indicate that modification iteration is not typically a major concern. However, there is a way to avoid them completely. This extension is explained in this section.

Addition and presentation of instances in the add-list, with the exception of the NI, cannot cause the network to cease to fulfill the instance set. This is trivially shown above for instances modified by overlap contradiction.

As we saw, for instances modified by superset one-difference minimization, it is possible that the top node no longer be a complete discriminant node. However, once the NI has been added to the set and any modifications made to the network, the network will fulfill the entire instance set, regardless of whether network modification followed the presentation of minimized old instance. Yet, the top node could still contain a "?" for the cell corresponding to the modified old instance. This is best shown by an example.

Assume the instance $A\bar{B}C \longrightarrow Z$ exists in the IS. The instance $AB \longrightarrow Z$ is then input. The old instance is changed to $AC \longrightarrow Z$ by superset one-difference minimization. Assume that the cell corresponding the OI in the top node changes from a 1 to a "?". In the new top node created after network adaption (as explained in this chapter) the cell for the NI will contain a 1 and the cell for the modified OI will still show a "?". Thus, the top node would not be a complete discriminant node as defined in the node status section. However, we know that the network will still output a 1 when $A\bar{B}C$ is matched because no modification has taken place to the network under the old top node; only the node table has been changed. The pattern ABC which the old cell does not correctly represent is correctly fulfilled by the NI since it covers that case. Thus, the network will still function correctly although the top node has a don't know cell.

This approach is parsimonious in terms of processing, but has a cost in the representation of the instance table. The AU must still maintain a correct copy of the instance set in order to make future consistency changes correctly. Two alterations are needed.

First, the AU must store two different instance tables. One is a copy of the instance set. The other is an image of the tables which are stored in all the nodes.

Second, the AU must mark some cells of the image copy with a flag stating that while a top node may contains a "?" for that cell, it still fulfills the instance set. Thus, the overall tradeoff is between adaptation time and memory overhead.

6.6. AA1 Example

In this section we give an example of the algorithm. We start with a null instance set and begin adding instances and building the network. When starting the algorithm works in a slightly different manner than when the instance set is already large. This is because more effort is needed to add new input variables, and there are none or few nodes in the network available to discriminate new instances. We avoid restating fine detail which has already been discussed. For example, if it is necessary to select a discriminating variable for new node addition, one will simply be chosen for use, rather than to go through looking at each possible variable.

Assume that we are concerned with one output variable Z. The first instance input to the system is $A\ \bar{B}\ C \longrightarrow Z$. Since there are currently no nodes in the network the delete-list is empty and the add-list is just the NI. The algorithm will go to the new node addition phase. If we choose A as the discriminating variable, then a node could be added to the network as shown in figure 6.16, where only the positive and negative columns are shown, and the function of the DPLM of the node is shown under the node table.

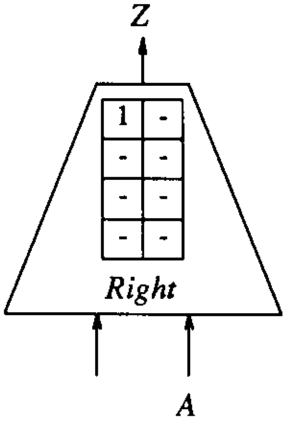


Figure 6.16 - Initial Node

This node would indeed fulfill the instance set, but it is more than necessary. In fact, since the node is a non-discriminant node, as earlier defined, it will delete itself from the network.

Rather than have a network to fulfill the current set, the system would simply set the Z output to a 1. This could not be done by a node, since any node would delete itself at this point. Thus, this is a special boundary condition in which the output could be set by the AU, the output binder, or any way which was most efficient for a given implementation.

Two more positive instances are then added to the system: $B\ \bar{C}\ D\ E \longrightarrow Z$ and $\bar{A}\ B\ C\ \bar{E} \longrightarrow Z$. Since these are positive instances, they cannot contradict the positive instance in the IS. In this case, they also cannot be further minimized. No change to the system can be made. Any node which was added to the network to fulfill this IS would still self-delete, since it would be a non-discriminant node. At this point the instance set is as follows.

$$A\ \bar{B}\ C \longrightarrow Z$$

$$B\ \bar{C}\ D\ E \longrightarrow Z$$

$$\bar{A}\ B\ C\ \bar{E} \longrightarrow Z$$

When displaying the instance set during this example positive instances are shown first, followed by the negative instances. We assume that the image in the node tables match the order in which we show the instances.

The next instance added to the system is $A\ B\ \bar{C}\ \bar{D} \longrightarrow \bar{Z}$. This instance does not contradict any of the earlier instances, and thus it is added to the IS with no modification to the OI's. Since this is a negative instance, and since Z is currently always positive, there must be an update. Because there are still no nodes in the network, the system proceeds to new node addition. Since the negative NI is not yet discriminated from any of the positive OI's, it is necessary to add sufficient nodes to do this discrimination. Variables C and D are discriminating variables which together are sufficient to discriminate the NI from the three original OI's. A new node is allocated with these two variables as input. The new node can be either a positive or a negative growth node. If the node became a negative growth node, it would switch to positive upon recognizing that it is a complete discriminant. The allocated node is shown in figure 6.17.

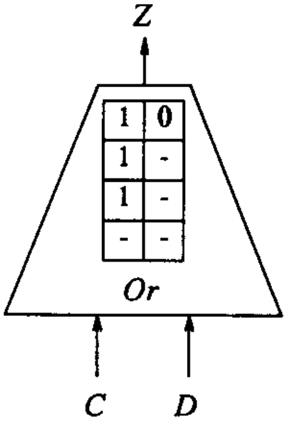


Figure 6.17 - Modified Node

The cell representing the NI is shown in bold.

The next instance input is $\bar{A}\ \bar{C}\ \bar{D}\ \bar{E}\ \longrightarrow\ \bar{Z}$. This instance does not cause either contradiction or minimization, and thus it is added directly to the IS. Now that there is a node in the network, the algorithm will go through all of its steps. The NI is presented to the network and the output is a 0. Thus, the network already fulfills the NI and no modification to the network is necessary. At this point the instance set appears as follows.

$$A\ \bar{B}\ C\ \longrightarrow\ Z$$

$$B\ \bar{C}\ D\ E\ \longrightarrow\ Z$$

$$\bar{A}\ B\ C\ \bar{E} \longrightarrow Z$$

$$A\ B\ \bar{C}\ \bar{D} \longrightarrow \bar{Z}$$

$$\bar{A}\ \bar{C}\ \bar{D}\ \bar{E} \longrightarrow \bar{Z}$$

The next instance added is $A\ \bar{B}\ C\ F \longrightarrow \bar{Z}$. This instance does not cause minimization, but it does contradict the first instance in the set. Thus, the NI is added, the first positive instance is put in the delete-list, and the overlap modification $A\ \bar{B}\ C\ \bar{F} \longrightarrow Z$ is put together with the NI in the add-list. The new instance set is shown below.

$$A\ \bar{B}\ C\ \bar{F} \longrightarrow Z$$

$$B\ \bar{C}\ D\ E \longrightarrow Z$$

$$\bar{A}\ B\ C\ \bar{E} \longrightarrow Z$$

$$A\ B\ \bar{C}\ \bar{D} \longrightarrow \bar{Z}$$

$$\bar{A}\ \bar{C}\ \bar{D}\ \bar{E} \longrightarrow \bar{Z}$$

$$A\ \bar{B}\ C\ F \longrightarrow \bar{Z}$$

The index of the deleted instance is broadcast and an empty marker placed in the first positive cell. The modified instance is presented to the network, and a 1 is placed in the empty cell. The node continues to be a complete discriminant node. When the NI is presented, the output of the network is 1. Since this is incorrect, network modification must take place. One selection wave results in a 0 from the top node, since the single current node does not discriminate the NI from any OI. New node addition must then take place. Variables B and F can be used as discriminating variables, as explained in the new node addition section, to discriminate the NI from the 3 positive OI's. A new node is added, combining these two variables. This new node is the only original growth node and it is a one-sided discriminant node. It is then combined with the old top node and the new top node is given a function according to table 6.7. The modified network is shown in figure 6.18.

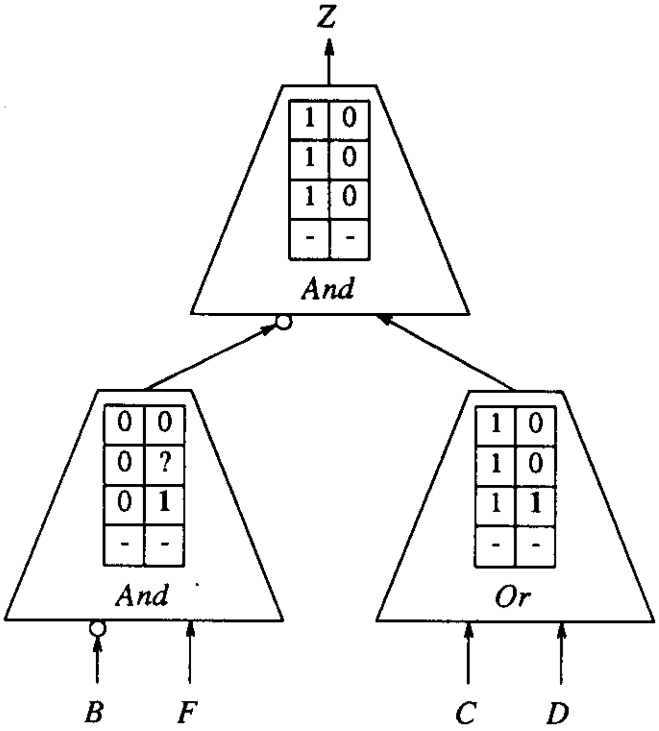


Figure 6.18 - Modified Network

The next instance added is $\bar{A}\ C\ D\ F \longrightarrow Z$. This NI can not be minimized with any concordant instance, nor does it contradict a discordant instance. When the NI is presented to the network, the output of the top node is a don't know. After the

NI presentation, the state of the network would be as shown in figure 6.19.

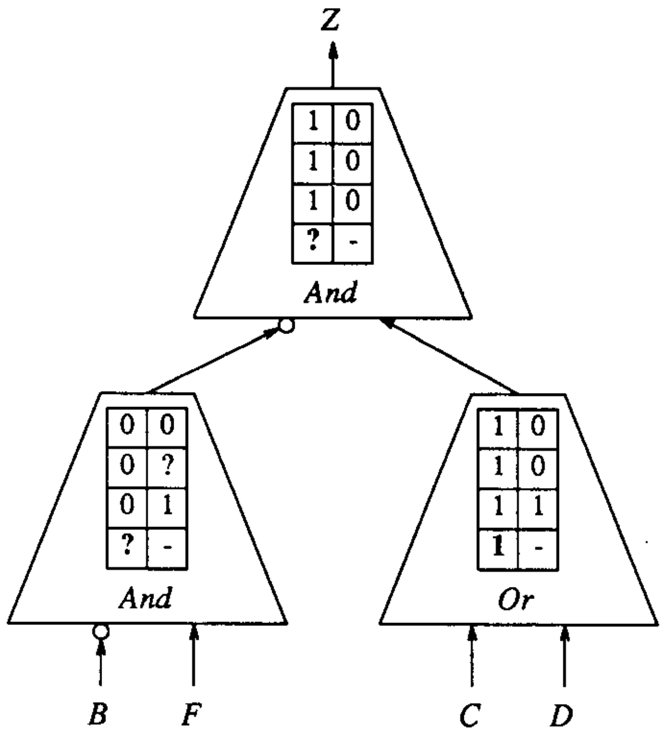


Figure 6.19 - Modified Network

A selection wave is then initiated, and the bottom right node is selected as a growth node, having a discriminant count of 2. Another selection wave is then initiated, but 0 reaches the top node, since no other node can discriminate the NI. The AU chŏoses one discriminant variable from the single instance, the third negative instance, which is not yet discriminated. The only possible variable is *A*. The variable *A* is combined with the selected growth node, and the resulting one-sided discriminant node is combined with the old top node, to form a new complete discriminant node. Figure 6.20 shows one possible combination scheme where the first new node was made a negative growth node. The self-modification phase is entered, but no nodes are deleted, since they are all discriminant nodes.

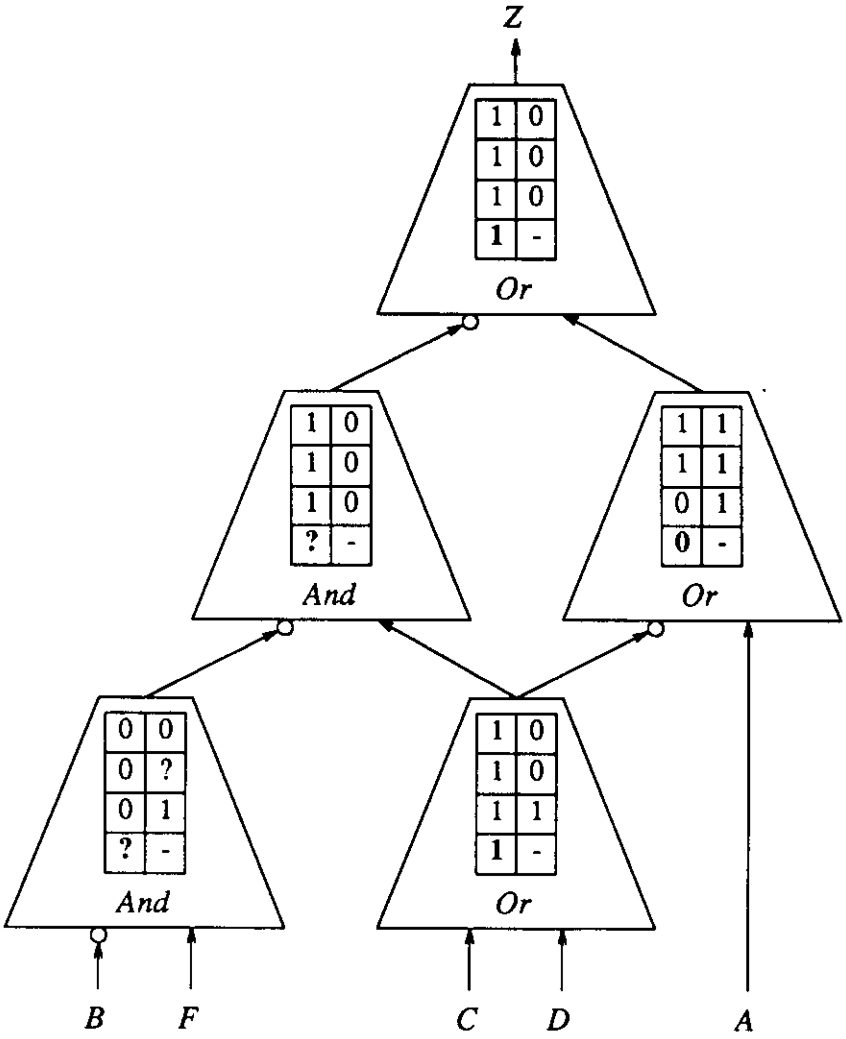


Figure 6.20 - Modified Network

The last instance input to the system is $A\ C \longrightarrow Z$. This instance causes both minimization and contradiction. It does a subset contradiction of the second negative instance, causing that instance to be deleted. This fact is broadcast to the network and an empty marker is placed in the cell of each node which corresponded to that instance. The NI also does a subset minimization with the first positive instance. It is also deleted and the same information is sent to the network. The NI also causes one-difference minization with the last concordant instance. This is a case where a modification iteration would be necessary since the top node will no longer be a complete-discriminant. This modification iteration is left as an exercise to the reader since the final network outcome will be the same regardless. The NI can then be added to the IS. The new instance set is shown below.

$$A\ C \rightarrow Z$$

$$B\ \bar{C}\ D\ E \rightarrow Z$$

$$\bar{A}\ B\ C\ \bar{E} \rightarrow Z$$

$$A\ B\ \bar{C}\ \bar{D} \rightarrow \bar{Z}$$

$$\bar{A}\ \bar{C}\ \bar{D}\ \bar{E} \rightarrow \bar{Z}$$

$$\bar{A}\ C\ D\ F \rightarrow \bar{Z}$$

At this point, the NI is presented to the network. The NI is placed in the first cell of the positive column, since that cell just became empty when the OI was deleted. After presentation, the state of the network is as shown in figure 6.21. In this case the output from the top node is correct and it would not be necessary to modify the network. However, since minimization was done to the instance set, the system enters the self-deletion phase before it returns to execution mode. The bottom right node has now become a complete discriminant node. Everything above it will be recursively deleted when the complete discriminant node sends a local command up to its parent. After the network finishes the self-pruning process the total network fulfilling the current instance set is as shown in figure 6.22.

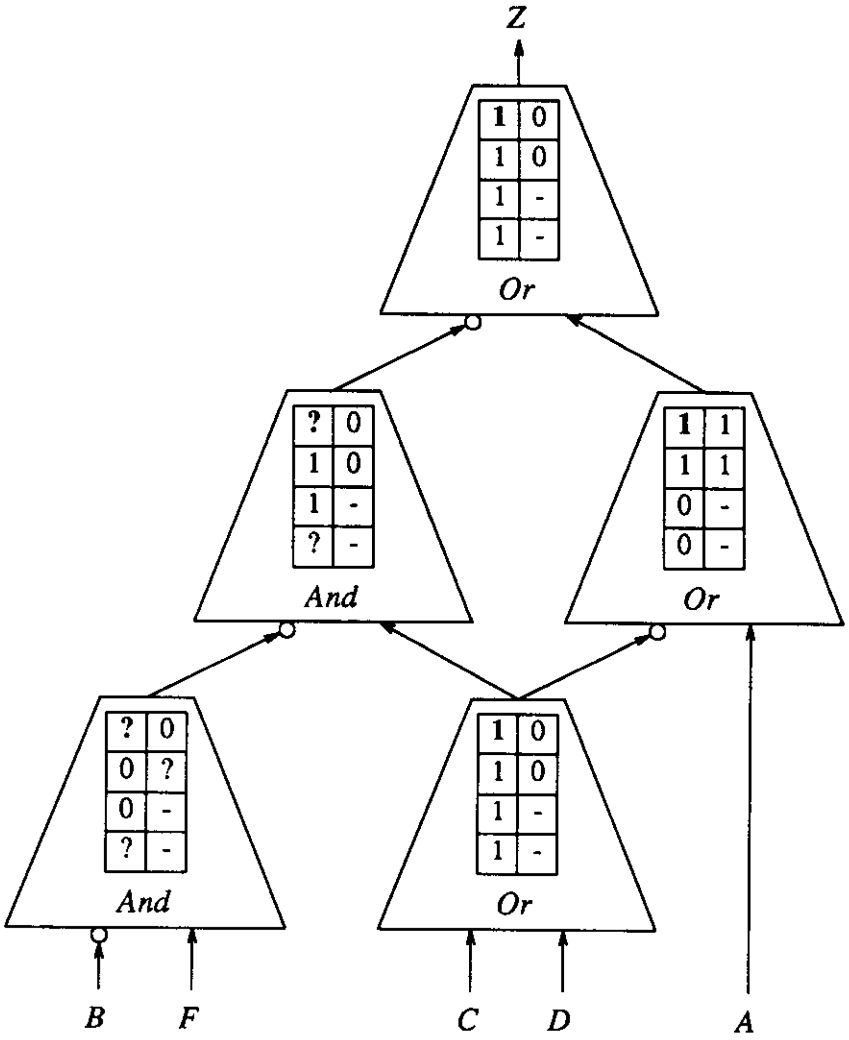


Figure 6.21 - Modified Network

This example has shown each of the major steps which occur during network reconfiguration. The network acts as a discrimination network which keeps itself relatively optimal. In the example there were 7 variables, but only a few instances, and the number of nodes necessary to perform the correct discrimination for all 2^7, or 128, environment states is quite small.

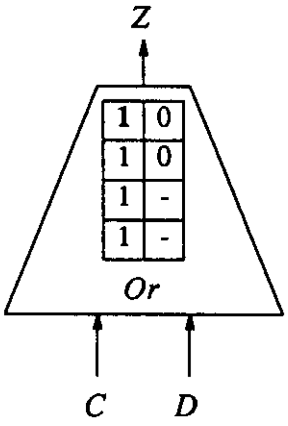


Figure 6.22 - Final Network

6.7. Multiple Outputs

The basic change needed for multiple outputs is for each node to maintain one node table for each output. In this way, a single node can be used in the discrimination of any number of output variables. Global commands include which output variable they affect. The node status depends on the output variable. If a node becomes non-discriminant for one output variable, but it is still a discriminant for a different variable, then it cannot be deleted. If it were a NDN for all of its output variables then it would be deleted.

Instances for each output variable need to be kept in independent instance sets within the AU. There is no minimization or contradiction possible between instances defining different output variables. The same is true for feedback variables. In each independent instance set, a feedback variable is treated exactly like any other input variable.

6.8. Comments on AA1

In this section we make some comments on the features and problems of AA1. Statistical analysis is given in the simulation chapter.

In AA1, discrimination is measured only in terms of the function allocated to the DPLM. There is no local knowledge at a node about the overall function, or the variables involved in the function.

In AA1 the discrimination is truly distributed, since there is no particular node which has responsibility for a given instance. The discriminatory responsibility for a single instance can be spread out over a large number of nodes.

There are two main problems with AA1. The first and most obvious is memory use. AA1 needs a node table which can grow as large as the instance set. The instance set used in any system would have to remain smaller than the fixed size of the network node tables. Larger instance sets could be processed by using hierarchical decomposition. Extensions of the algorithm allowing nodes to process over a subset of the instance set, in order to maintain a small node table size, is a current research effort.

It is also necessary to have one node table for each output variable. This shows the need for the *layered architecture* discussed in chapter 4. Each logic plane, and thus the nodes that are in the logic plane, would be limited to some fixed number of output variables. By using multiple output planes, a large number of output variables can be computed.

The other problem comes up when considering implementation of interconnection. In selecting nodes, AA1 allows any discriminant node to be chosen. If the nodes selected are far apart topographically it becomes difficult to efficiently combine those nodes. If interconnection multiplexers were as wide as the network, then this would not be an obstacle. But, in order to have high speed through these routers it will be necessary to limit their width, and use many non-connected or lightly-connected multiplexers. The algorithmic way to solve this problem is to make topography an important part of the locally computed criterion used during selection. Thus, nodes can be selected, not just by their discrimination ability, but also according to the ease by which they can be connected to other growth nodes.

As will be seen in later chapters, the first problem (node memory) is alleviated in both AA2 and AA3, and the second (node interconnections) in AA3.

Chapter 7

ADAPTIVE ALGORITHM 2

In this chapter we discuss the second adaptive algorithm developed for the ASOCS architecture: ***Adaptive Algorithm 2 (AA2).*** The chapter presents the basic aspects of AA2, with some comments on possible extensions.

The first section discusses the architectural requirements for AA2. The third section gives a formal description of AA2. Section 2 gives a informal overview of each phase of the algorithm. Section 4 illustrates the algorithm with a simple example. The last sections discuss multiple outputs and summarize AA2.

To simplify the exposition, we assume only one output variable Z.

AA2 uses a method much different than that of AA1. Instead of having a single top node for an output variable, there can be a number of top nodes for an output. These are divided into *positive* and *negative* nodes. If any positive node outputs a 1, then the output of the network must be a 1. If any negative node outputs a 1, then the output of the network must be a 0. Outputs from top nodes are *OR*'ed together. This same methodology is used in AA3. However, in AA2 there is one top node matching each instance in the instance set. Thus the network is maintained as a consistent and minimal instance set. There is no need to maintain the instance set in the AU, since the network implicity maintains it. All nodes use a gate which does only the *And* function. Nodes are combined until the overall conjunction matches a single instance.

This node then becomes a top node and is set as positive or negative depending on the polarity of the instance being matched. When a new instance is broadcast to the network, each node is able to compare its function with that of the new instance and consequently deduce its potential participation.

AA2 does not have the memory constraints found in AA1. The memory at a single node does not increase as the size of the instance set increases. Neither is there a significant increase in node memory for multiple-outputs. Another feature of AA2 is that the network can tell if the current environment state does not match any instance of the instance set. In many applications it is not desirable that the output be arbitrary when the environment is not matched by an instance. The don't know output allows the output to be controlled differently when no instance is matched by an environment. The maximum depth of the AA2 network is equal to the number of bound input variables.

7.1. Architectural Requirements

The architecture is basically the same as used for AA1. However there are some important differences which are explained in this section.

7.1.1. Network Nodes

In AA2, each node of the logic network knows its logic function in terms of the variables which are part of its function. Figure 7.1 is a representation of a node which does the *AND* function of the variables A, $\bar{B}$, and D. In AA2 all node DPLM's do the dyadic *And* function. A node knows its overall function by storing a list of the

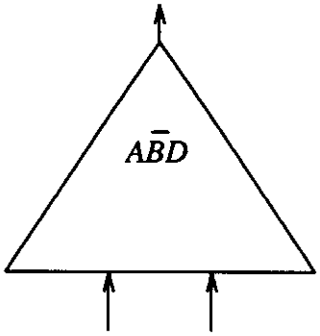


Figure 7.1 - Node Representation

variables which it conjuncts. Each child of the node does a conjunction of a subset of the variables stored at the node. A node does not know which variables correspond to which child.

A node can be in one of two basic states: ***D-Node (Discriminant Node)*** or ***P-Node (Primitive Node).***

D-nodes can be either ***Positive D-Nodes*** or ***Negative D-Nodes.*** We compact the terminology by calling the current state of any node its ***Polarity.*** A positive D-node has positive polarity. A negative D-node has negative polarity. The P-node has nil polarity; i.e. no polarity at all.

When any node in the network has a positive output for a certain instance, it is called an ***Active Node*** in terms of that instance. If any positive D-node is active then the output of the network must be positive. If any negative D-node is active then the output of the network must be negative. A negative and a positive D-node cannot be simultaneously active, but it is possible that more than one D-node of the same polarity be simultaneously active.

P-nodes do not directly control the output of the network, but they are the nodes which make up the inputs to D-nodes.

Because presentation of the NI is not necessary in AA2, the DPLM need only be a binary gate. The DPLM also needs to realize only a subset of the functions which it was able to do in AA1. In fact, if inversions of literal variables are done in the input binder, then the DPLM does not have to be programmable. It can be a simple two-input *And* gate. In this chapter we continue to refer to the logic gate of the node as a DPLM, although it need not be programmable.

AA2 is simpler in its mechanism than AA1. More importantly, AA2 does not require that the memory at the nodes grow with the number of instances. The memory of a node contains the list of variables which its function covers, and that list will not change as the IS grows. The largest possible list size at a single node is equal to the number of bound input variables. Once a node is set to conjunct a list of variables, the memory at the node need never increase. This list of variables, unique for each node, is known as the *variable-list.*

Each node has a ***Polarity Flag,*** marking whether it is a P-node or a D-node. If it is a D-node, then the flag states whether it is positive or a negative. It is also possible that the node is unused, which is another state. Thus the flag is a 4 state, or 2 bit, memory cell.

Each node has a left and right child. The children can be either nodes or input variables. Each node also has a set, possibly empty, of parent nodes. Finally, each node has a ***State Flag*** which stores information regarding the current state of the node in reference to an instance.

7.1.2. Or-Planes

The outputs of all positive D-nodes enter into the ***Positive Or-Plane.*** The outputs of all negative D-nodes enter into the ***Negative Or-Plane.***

Figure 7.2 shows a general representation of the structure of the logic network.

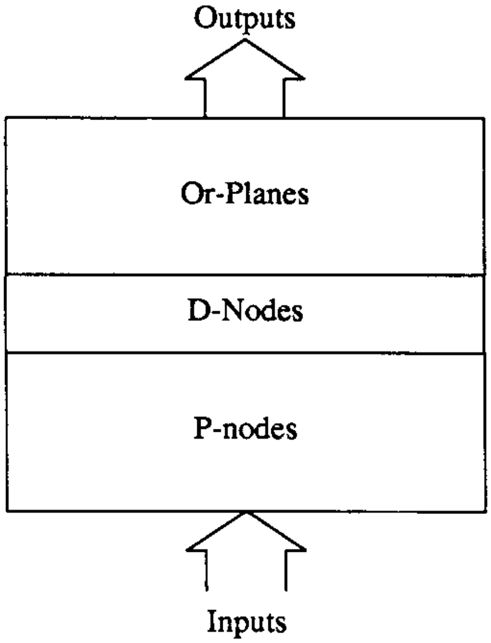


Figure 7.2 - General AA2 Structure

Inputs to D-nodes must be from either a P-node, or a literal variable (i.e. A, B, etc.). The D-nodes are the top nodes of the network. However, D-nodes can be found at any depth of the network. The inputs to the or-planes must be from D-nodes. Any time a D-node is created or modified, a local command is sent to the or-plane, and the output of the D-node is placed into the correct or-plane.

The or-planes need not be adaptive, The logical structure of the or-planes, which includes the positive and the negative planes, is shown in figure 7.3. A 3-node

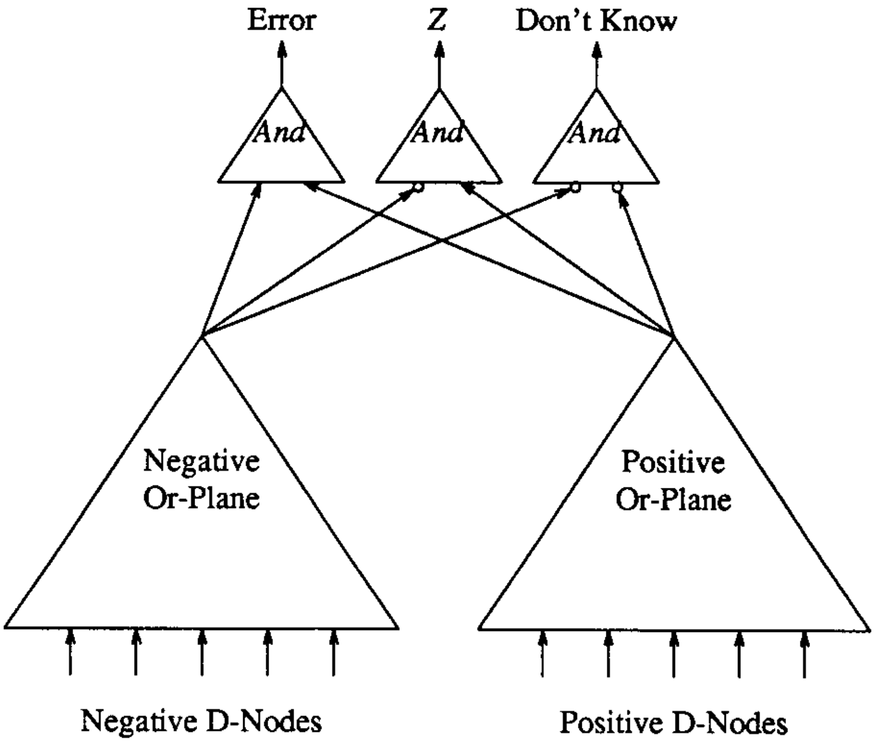


Figure 7.3 - Or-Planes

structure handles the or-plane outputs. The middle node is the output node. It outputs 1 if the positive plane outputs a 1, and it outputs 0 if the negative plane outputs a 1. If both or-planes output a 0 then Z is set to 0, since the node is set to an *And* function. (The node could also be set to an *Or* function, making the default value a 1, when no instance is matched.)

If both planes output a 0, then the don't know output is active. With this output, one can always know whether an instance is matched or whether the environment state is a don't know state. If both planes output a 1, then there is an error in the network.

As can be seen by viewing the AA2 structure, the algorithm implements a *sum-of-products* expression. The products are the instances represented by the D-nodes in the network and each D-node is equivalent to one instance in the instance set. The products represented by the D-nodes are summed by the or-planes.

Figure 7.4 shows a representation of the top of the logic network, which is the place where the main difference with AA1 occurs.

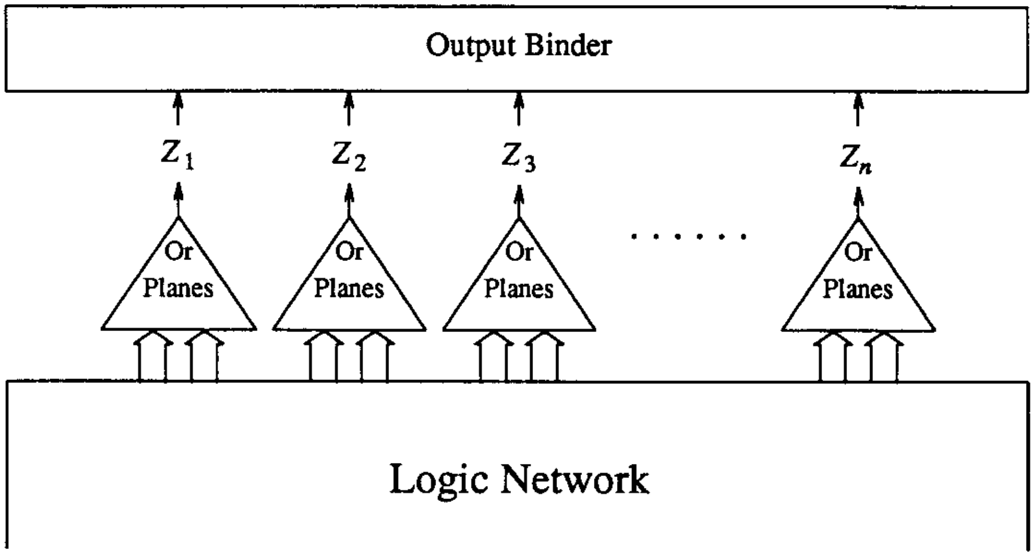


Figure 7.4 - AA2 Or-Plane Architecture

For each of the or-planes the left input arrow corresponds to the negative D-node outputs and the right input arrow to the positive D-node outputs.

A more general architecture with the or-planes is to have an adaptive layer between the or-planes and the output binder. Or-planes can then be dynamically bound to output variables. Thus, if the number of D-nodes corresponding to a given output variable becomes greater than the number of inputs to an or-plane, another or-plane can be allocated for that output variable. The outputs of the two or-planes would then be gated together with the *Or* function. In this way the number of instances defining an output is not limited by the width of the or-planes.

Finally, in terms of implementation, the or-planes are placed alongside the logic network and run parallel to it. If a *wired-or* technology is used for the or planes, then two single wired-or lines could be used as the or-plane for any output variable. These lines run vertically along the side of the logic network. Outputs from D-nodes run immediately perpendicular to the D-nodes and join the vertical or-plane. This would resolve the problem of horizontal connectivity due to D-node outputs to the or-plane. (Nodes within the network would not have to be used as simple communication buffers passing output variables up to the or-planes.) The input binder could also be placed parallel to the logic network, such that variables could easily be entered at any level of the network. Figure 7.5 gives a representation of the ASOCS structure (excluding the AU) for AA2.

7.2. Informal Description of AA2

This section gives the informal description of AA2. Refer to the formal description (in the next section) for the final word on ambiguities.

7.2.1. Broadcast and Instance Set Maintenance

In AA2 it is not necessary to store the instance set separately from the network, and therefore it is not necessary to do instance set maintenance in the adaption unit. The instance set is implicitly represented and maintained consistent and minimal within the logic network. When a new instance is broadcast, the nodes in the network adjust such that the network represents a consistent and minimal instance set.

Instance broadcast is the method of instance introduction in AA2. The broadcast

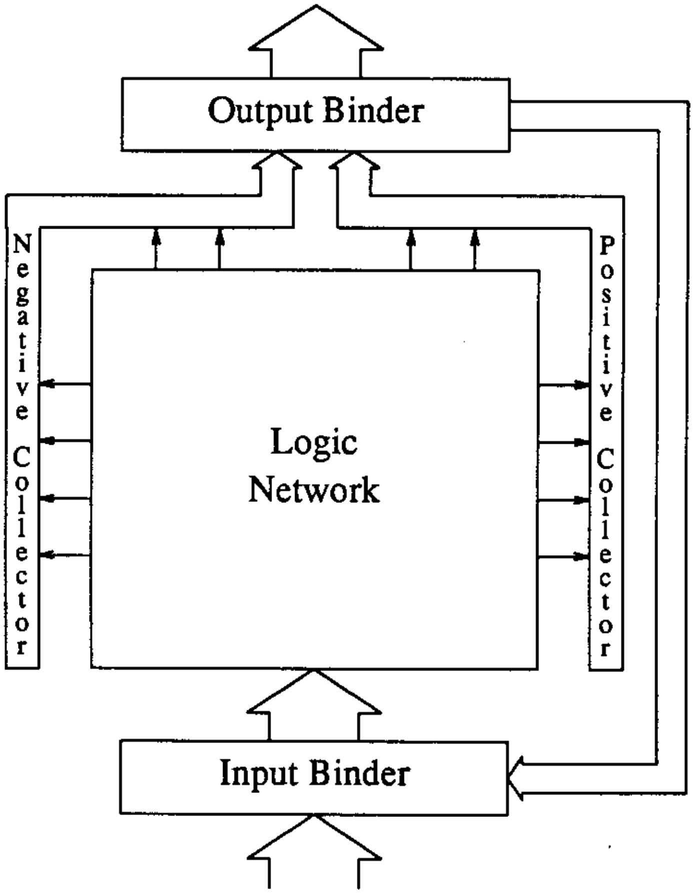


Figure 7.5 - AA2 Implementation Structure

to the network is simply a list of the variables (including their polarity) contained in the antecedent of the NI, together with the polarity of the new instance.

7.2.2. Node and New Instance Relationships

When a NI is broadcast, each node[1] relates with the instance in one of six possible ways. They are ***Subset, Equal, Superset, Overlap, One-Difference,*** or **Discriminated.** These are the same terms defined it chapter 3 regarding the relationships between instances. In the following explanations variables in the NI and in a node are

[1] The relation is between the variables of the instance and the variable list of each node.

only considered to be the same if they are concordant variables. Figure 7.6 shows examples of the six basic new instance to node relationships which can occur after the instance broadcast.

$A\ \bar{B}\ D \rightarrow \bar{Z}$	New Instance
$A \rightarrow \bar{Z}$	Subset Node
$A\ \bar{B}\ D \rightarrow Z$	Equal Node
$A\ \bar{B}\ D\ E \rightarrow \bar{Z}$	Superset Node
$\bar{B}\ \bar{E} \rightarrow Z$	Overlap Node
$C\ G \rightarrow \bar{Z}$	Overlap Node
$A\ \bar{B}\ \bar{D} \rightarrow \bar{Z}$	One-Difference Node
$A\ \bar{B}\ \bar{D} \rightarrow Z$	Discriminated Node

Figure 7.6 - Node - New Instance Relationship Examples

A node is a *subset* node if its own variable-list is a subset of the variables of the NI. A subset node always outputs a 1 for all environment states matching the NI.

A node is *equal* if its variables are the same as those of the NI. An equal node always outputs a 1 for all environment states matching the NI.

A node is a *superset* node if it has more variables than the NI, but every NI variable occurs in the node's variable list. A superset node outputs 1 for some states of the environment that match the NI, and outputs 0 for others.

A node is an *overlap* node if there is no discriminant variable between the NI and the node, and the node is not a subset, equal, or superset node. An overlap node outputs 1 for some states of the environment matching the NI, while it outputs a 0 for others.

A node is a *one-difference* node if the NI and the node are concordant, and there is exactly one discriminant variable between the two, and except for the discriminant variable, one is a subset of the other.

If there is one or more discriminant variables between the NI and the node variables, and the node is not a one-difference node, then the node is *discriminated.* A discriminated node outputs 0 for all states of the environment matching the NI.

7.2.3. Node Action and Priority

The action that a node can take depends on both its relation with the NI and its polarity. The permutations of these two values are similar to the types of minimizations and contradictions which can take place in an instance set when a NI is added, as explained in chapter 3.

After broadcast of the instance, each node compares the NI with its own variable list, and then sets its state flag to a corresponding priority. The priority will be used to determine the order in which the nodes should be selected. For each NI-node relationship there is a specific action which can take place. In AA2 some actions must take place before others, and thus a priority is given to each type of relationship. Nodes with the highest priority are chosen first to cause the action specified by the NI-node relationship. A node only causes its specified action to take place when it is selected. It is possible that the action of a high priority selected node is sufficient to cause correct network update, in which case no more nodes are selected.

Selection in AA2 must be implemented different than for AA1 since there are multiple top nodes for a single output variable. These differences are implementation dependent and will be explained in a later section. For ease of explanation, the same selection terms used in Chapter 6 are used in the remainder of this chapter.

There are 15 different combinations of NI-node relationship and node polarity. Table 7.1 gives an overall summary of the potential action states together with their priority.

Node is	Subset	Equal	Superset	Overlap
Discordant D-Node	DVA (3)	Polarity Inversion (1)	Self Delete (6)	DVA (3)
Concordant D-Node	New Instance Fulfilled (1)	New Instance Fulfilled (1)	Self Delete (6)	No Action (7)
One Difference D-Node	Modify New Instance (2)	Modify New Instance. (2)	Modification Iteration (4)	Impossible State
P-node	Potential Growth Node (5)	Complete Discriminant Deletion Becomes D-node (1)	No Action (7)	No Action (7)

Table 7.1 - Node Actions

DVA is explained in a later section.

The priorities are the criteria used during the selection process which uniquely selects nodes. Priority 1 is the highest. The only nodes which can initiate an action during the main update phase are those with priorities 1-4. Priority 5 nodes can be selected during node combination. Priority 6 nodes will initiate self-deletion after the update phase. A priority 7 node can never be selected.

The priority 6 nodes could be allowed to immediately self-delete after new instance broadcast. However, for efficiency as explained later, the self-deletion is not allowed to occur until after the network modification. AA2 Self-deletion, a type of *non-discriminant deletion,* is explained in a later section.

The basic AA2 algorithm is as follows:

1. Input a New Instance.
2. Broadcast the New Instance.
3. Each node compares its variable list with the NI and sets its priority.
4. A sort wave is performed and the AU views the output score.
5. If the score is a 1 then

    a. Do arbitration wave and selected node broadcasts its relation.

    b. If the node is concordant-subset or concordant-equal then exit.

    c. If the node is an equal p-node, then do *complete-discriminant* deletion, then goto 4.

    d. If the node is discordant-equal, then do *polarity-inversion,* then goto 4.

6. If the score is a 2 then

    a. Do arbitration wave and allow selected node to broadcast its variables.

    b. Minimize the NI in the AU and rebroadcast it to the network.

c. Goto 3.

7. If the score is a 3 then

    a. While there are still nodes with priority 3 select one node with an arbitration wave, and do *discriminant variable addition.*

    b. Goto 4

8. If the score is a 4 then

    a. do arbitration wave to select the node.

    b. recurse to 1 with the new instance built from the variable list of the node and the node polarity.

    c. Goto 4

9. If no previously selected node had a priority of 1 do *Node Combination* to create a new D-node matching the NI.

10. Do Self-Deletion.

Each node action is now discussed in detail.

7.2.3.1. Priority 1 Nodes

There are four node states in the table which have priority 1. If a node has priority 1, then that node is sufficient to fulfill the new instance. Only one node in the network can be a priority 1 node for any given instance. This follows from the consistent and minimal structure of the network.

A subset or equal concordant D-node, which can never cause a contradiction, already fulfills the NI. In this case, no change need be made to the network. This corresponds to a NI immediately deleted by subset minimization.

If the node is a discordant D-node[2] that is equal to the NI, then the node will change its polarity from positive to negative, or vice versa. This change must be accompanied by a local command sent to the or-plane, causing the output to be switched to the discordant or-plane. This method of change is called ***Polarity Inversion.*** The update cannot terminate at this point since it is possible that other nodes contradict the NI. The algorithm continues by initiating another selection wave.

An equal P-node causes complete discriminant deletion. The procedure is the same as that in AA1. Since the P-node is an exact match of the NI, it can become the new D-node responsible for that instance. All nodes above the equal P-node can be deleted from the network. The polarity of the new D-node is set to the polarity of the NI. After this deletion the node sets its priority to 7 so that it will not be selected in a subsequent selection wave, and another sort wave is initiated. This is to see if any priority 2 nodes had been set during the NI broadcast, which could now cause further minimization to take place in the network.

When a priority 1 node is selected, node combination will not be necessary, regardless of whether further node selection is necessary. This is because the selected priority 1 node will be the node corresponding to the NI.

7.2.3.2. Priority 2 Nodes

Both node states with priority 2, the one-difference subset and one-difference equal nodes, cause the same action: to minimize the NI by removing one variable. When the AU receives a 2 at the end of a sort wave, it selects the node. The node then

[2] Meaning that it contradicts the new instance.

broadcasts its variable list. The AU is then able to minimize the NI, and rebroadcast it to the network.

After the rebroadcast, all nodes will recalculate their priority and the algorithm will continue as if the rebroadcast were the initial broadcast. However, because of consistency, no node can ever contradict a new instance which has been minimized. If one could, then the network would have been inconsistent.

7.2.3.3. Priority 3 Nodes

The discordant subset and overlap nodes have a priority of 3. They both cause the same action, *discriminant variable addition (DVA).* DVA is explained in a later section.

7.2.3.4. Priority 4 Nodes

The priority 4 node is the superset one-difference node. This node state is the one which can cause efficiency problems for AA2. Each time one of these nodes is encountered it is necessary to recurse through the whole modification cycle with the new instance being built of the variables and polarity of the node.[3] Each one of these recursions causes a *modification iteration*, an occurrence that is shown in chapter 9 to be fairly costly.

This iteration is necessary because a more compact representation is sought in this case. The node itself will always be deleted. However, it is not usually possible to simply delete part of the subtree to obtain a D-node representing the correct

[3] Which polarity must be the same as the polarity of the new instance.

instance. For example, assume a D-node with a variable list of $ABCD$ built from two nodes with lists AB and CD respectively. The concordant NI $B\overline{C}$ is then broadcast. The D-node must be changed to ABD. The only way to do this is to combine the node with AB together with the single variable D. This must be done by a modification iteration.

A simple variate which allows real-time adaptation[4] is to ignore priority 4 nodes. In this case the network will still fulfill the instance set, although the set it stores will not be minimal.

This variate could then be extended by allowing the network to minimize itself at some later time when no processing was being done. This minimization, or "garbage collection," would be a type of locally redundant deletion in which each node could, in a round robin fashion, broadcast its variable list. Nodes able to do one-difference minimization with the broadcast list could then cause modification iterations.

7.2.3.5. Priority 5 Nodes

The only priority 5 node is the subset P-node. It is the basic building block used in building new D-nodes to match a NI. Subset P-nodes can be conjunctively combined to create a new node having all the variables in the NI antecedent. Their use and creation are described in the section on *Node Combination.*

[4] Adaptation time guaranteed within a small upper bound.

7.2.3.6. Priority 6 Nodes

Priority 6 nodes are those nodes which will initiate self-deletion after the network update phase. Both discordant and concordant superset D-nodes fall into this category.

7.2.3.7. Priority 7 Nodes

The priority 7 nodes are those which cannot be selected for any update function. These nodes are the concordant overlap D-node, the overlap and subset P-nodes, and the discriminated node (a node having at least one discriminant variable). The subset P-node will always be deleted by a deletion wave initiated by its parent(s). The other nodes may be deleted by a complete discriminant deletion initiated by an equal P-node.

7.2.4. Discriminant Variable Addition

In AA2, D-nodes are only created to match a newly created instance. In the case of ***Discriminant Variable Addition (DVA)*** these are new instances created by overlap contradiction. Each discriminant variable of the NI is combined with the output of a D-node to create new D-nodes. The old D-node becomes a P-node. The new D-nodes take on the complement of the polarity of the new instance.

For example, assume there is a Positive D-node with a variable list of A and $\bar{B}$. The new instance $A\bar{B}CD \longrightarrow \bar{Z}$ is then broadcast. The node becomes a priority 3 node, and it will have to do DVA. There are 2 discriminant variables in the NI: C and D. Thus, two new nodes will be created by combining the output of the old D-node with the inversion of each of discriminant variables. Figure 7.7 shows the change made to the old D-node after this process.

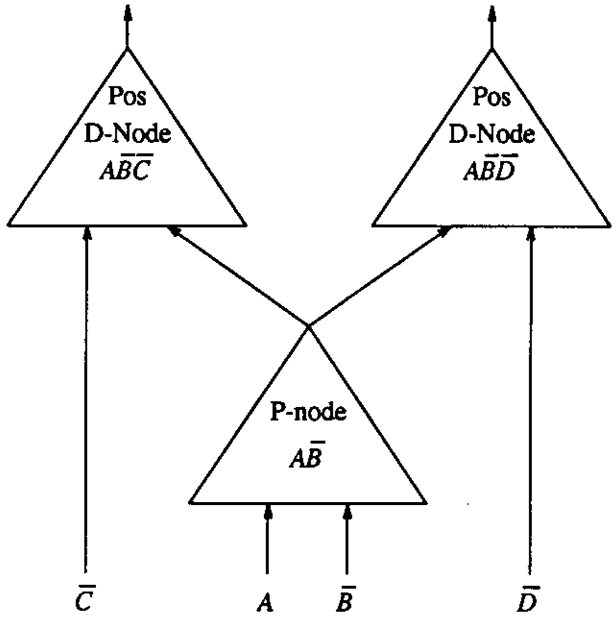


Figure 7.7 - Discriminant Variable Addition

The old D-node becomes a P-node. Note that there is still no new D-node matching the NI. This is not accomplished by DVA. It will be accomplished by the later step of *node combination.*

The above explanation of DVA is not quite complete, due to minimization constraints. It is not sufficient to simply add a D-node for each discriminant variable of the NI. Unlike node combination, it is possible that the new D-node created by DVA will not be minimal. The problem is solved as follows. When a priority 3 node is selected, it broadcasts its variable list on the broadcast bus. The AU can then calculate what new instances (D-nodes) have to be added before actually initiating the DVA. The AU broadcasts, one at a time, each of the new instances. Each node sets its action state, and a sort wave takes place. If the priority returned by the wave is "1", then some other node is already a subset of the new instance, and it is not necessary to add it to the network. Following is an example of this case.

Assume that we have the same example as above with a positive D-node (1) with a list of $A\bar{B}$. Assume also that there is a negative D-node (2) with a list of $\bar{C}$. The new instance $A\bar{B}CD \longrightarrow \bar{Z}$ is then input to the system. This instance is broadcast to the network, and node 1 sets itself as a subset node with a priority of 3. Node 2 does not match the NI, thus it has a priority of 5. DVA must then take place.

However, before creation of any nodes, the node broadcast its variable list. The AU calculates that $AB\bar{C} \longrightarrow Z$ and $AB\bar{D} \longrightarrow Z$ are the two new necessary instances. The AU then broadcasts each of these instances to the network to see whether they are already fulfilled by some node. When $AB\bar{C} \longrightarrow Z$ is broadcast to the network, node 2 signals that it is set to state 1, because it is a concordant subset node. Thus, the AU knows not to add the instance $AB\bar{C} \longrightarrow Z$ to the network. When $AB\bar{D} \longrightarrow Z$ is broadcast, no 1 is produced by the sort wave. In this case the DVA will create only one new node by combining the output of node 1 with the variable $\bar{D}$ to create a new positive D-node.

When the variable list of a node to be created by DVA is broadcast to the network, it is possible that concordant D-nodes are superset-one-difference nodes in relation to the newly created node. These nodes set their state to 4 and will be processed during the phase of the algorithm serving priority 4 nodes. Only nodes which are initially discriminated (have a priority of 7) are eligible to have their priorities changed to a 4. This is because they must contain a discriminant variable between themselves and the NI in order to be superset one-difference with a created DVA node.

7.2.5. Node Combination and New Node Addition

The new D-node created for a NI, which is not fulfilled by any node already existent in the network, is formed by combining P-nodes which are a subset of the NI; i.e. those with a priority of 5. Node combination will only take place if no priority 1 node was selected for the current instance. If combination of already existent subset P-nodes is not sufficient to create a complete D-node, then the AU will cause new P-nodes to be created by *new node addition.*

There is more than one way in which node combination can be performed. One simple method is to use a *Combination* selection wave, which causes the subset P-nodes to use the number of variables in their list which are also in the NI list as the criterion. The selected node broadcasts the variables in its list. The AU can then, if necessary, start another selection wave, having broadcast the abbreviated list. If the P-nodes are not sufficient to build the new D-node then the AU will allocate new nodes doing the conjunction of the variables not contained in the selected P-nodes. The selected P-nodes and any newly created P-nodes then combine together to form the new D-node corresponding to the NI. The D-node is set to the polarity of the NI.

Assume that we use the example as shown above in figure 7.7. After DVA, it is still necessary to create the new D-node to match the NI $A\bar{B}CD \longrightarrow \bar{Z}$. A selection wave is started and the P-node with $A\bar{B}$ is selected as one P-node. Assume that another selection wave does not result in any other priority 5 nodes. Then the AU would cause a new P-node with a list of *CD* to be created. The outputs of these two P-nodes would then combine to form the new negative D-node representing the NI. The final configuration is shown in figure 7.8

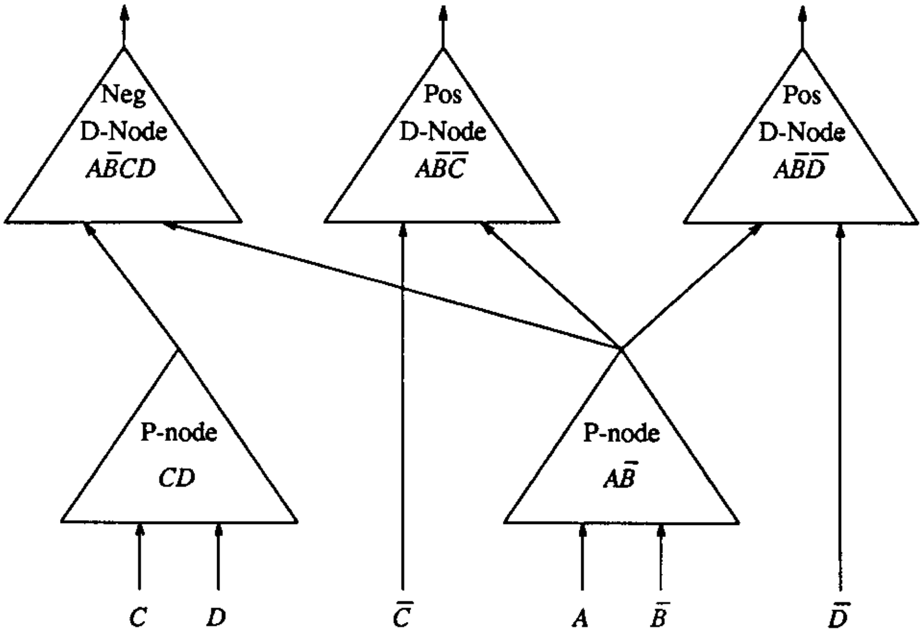


Figure 7.8 - Node Combination

7.2.6. Self-Deletion

The concordant and discordant superset D-nodes initiate self-deletion. Their deletion will take place after the network has been modified. The method of self-deletion is carried out like non-discriminant[5] deletion in AA1, except that there is never a parent node to the node initiating the self-deletion. Any superset P-node, a node which has parent(s), will be deleted by a parent which must also be a superset node.

[5] Complete discriminant deletion was explained earlier in the discussion of priority 1 nodes.

The self-deleting node sends a local command to both of its children nodes for them to also self-delete. This continues recursively, until a child has more than one parent node, at which time it simply removes the link to the deleted parent.

Self-deletion is done after network modification so that no P-node which might be used in node combination, will be recursively deleted because it was a descendent of a self-deleting node.

A D-node which self-deletes also sends a local command to the or-plane signifying that it is no longer an output.

When looked at closely, it is seen that for AA2 complete discriminant deletion is not really essential. The same goals are accomplished by using non-discriminant deletion, together with node addition. However, it is more efficient to use both deletion processes. In AA1, the two types of deletion are independent, and it is essential to have both.

7.2.7. Node Selection in AA2

The logic network in AA1, for any single output variable, converges to a single top node. This is very natural for the selection process, since there is a single termination and initiation point for the sort and arbitration waves respectively. This convergence is not found in AA2. This could be handled by augmenting the or-planes such that the D-nodes sent their sorted criteria to the or-plane which could then do a linear maximization of the criteria, which could then be sent back down the highest D-node.

However, we note that the need for selection in AA2 is different than that for AA1. In AA1 there can be any number of eligible nodes for selecting. The main purpose of selection is just to make sure a subset of then are uniquely chosen. In AA2 each matched node, within a priority class, already knows it must be used, and the purpose of selection is just to make sure there are no resource conflicts. This is because sorting is no longer critical. Four distinct selection waves could be used, one for each the four priorities which can initiate an action. Once within a class, it makes no difference in which order they are selected, since they must all be selected. Thus, rather than a sorting selection wave, a simple bus contention protocol could be used until each node has its turn. The type of protocol used is an implementation decision.

7.3. Formal Description of AA2

In this section we use a structured language to give a formal description of AA2. The language used is the same as for AA1 and its operators will not be redefined in this section.

Data Types

set:

list:

boolean:

integer:

polarity: positive, negative, or nil.

variable: variable which occurs in an instance (i.e. A , $\bar{B}$, etc.)

instance: instance.variables is a list of the variables in an instance.

instance.polarity is positive or negative. $\overline{\textit{polarity}}$ is the complement of polarity.

Data structures

broadcast-bus   Represents the broadcast bus.

node   Network node whose subparts are:

node.left-child - Node or a variable

node.right-child - Node or a variable

node.parents - Set of nodes

node.polarity - Positive (D-node), negative (D-node), or nil (P-node)

node.state - Integer priority between 0-6

node.variables - List of variables which the node conjuncts

```
program: AA2-Update(NI)

declare          NI: Instance

begin

    Add-New-Instance(NI)

    Non-Discriminant-Deletion

end


procedure: Add-New-Instance(NI)

declare          NI: Instance

declare          new-node-needed: boolean

declare          done: boolean

declare          current: node

begin

    new-node-needed ← true
```

```
for all nodes Set-State(node , NI)
done ← false
while not done
begin
    if Sort-Wave(result: highest-priority) ≠ 1,2,3, or 4 then done ← true
    else
    begin
        Arbitration-Wave(result: current)[6]
        if current.state = 1 then
        begin
            new-node-needed ← false
            if broadcast-bus = concordant-subset-d-node then done ← true
            else if broadcast-bus = concordant-equal-d-node then done ← true
            else if broadcast-bus = discordant-equal-d-node then
              current.polarity ← NI.polarity
            else if broadcast-bus = equal-p-node then
            begin
                Complete-Discriminant-Deletion(current)
                current.polarity ← NI.polarity
            end
        end
        else if current.state = 2 then
        begin
```

[6] The selected node places its node state on the broadcast bus.

```
                done ← true
                Add-New-Instance(Make-Minimal-Instance(
                  NI.variables , current.variables , NI.polarity))
            end
            else if current.state = 3 then Discriminant-Variable-Addition(current,NI)
            else if current.state = 4 then Add-New-Instance(Make-Minimal-Instance(
              current.variables , NI.variables , NI.polarity))
        end
    end
    if new-node-needed = true then Node-Combination(NI)
end

procedure: Set-State(node , NI)
declare         node: node
declare         NI: instance
begin
    node.state ← priority as shown in Table 7.1
end

procedure: Make-Minimal-Instance(vset1 , vset2 , polarity)
declare         vset1: set of variables
declare         vset2: set of variables
declare         polarity: polarity
declare         new-instance: instance
begin
```

```
        new-instance.polarity ← polarity
        new-instance.variables ← vset1 minus (discordant variable in vset2)
        return(new-instance)
end

procedure: Discriminant-Variable-Addition(NI , current)
declare         NI: instance
declare         current: node
declare         new-node: node
declare         dvar: variable
begin
        for all NI.variables (dvar) not ε current.variables
        begin
                Broadcast(current.variables + $\overline{dvar}$)
                if no node.state = 1 for (current.variables + $\overline{dvar}$) then
                begin
                        new-node ← Allocate-New-Node()
                        new-node.left-child ← $\overline{dvar}$
                        new-node.right-child ← current
                        new-node.variables ← current.variables + $\overline{dvar}$
                        new-node.polarity ← current.polarity
                        Set-State(new-node)
                        current.parents ← current.parents + new-node
                        for all nodes with state = 7 Set-State(node)[7]
                end
```

```
    end
    current.polarity ← nil
end

procedure: Node-Combination(NI)
declare        NI: instance
declare        done: boolean
declare        selected-node: node
declare        share-set: set of variables
declare        new-node: node
declare        node1: node
declare        node2: node
begin
    share-set ← $\phi$
    done ← false
    while (share-set $\neq$ NI.variables) and (not done)
    begin
        selected-node ← priority 5 node largest nonzero (node.variables $\overline{\textit{intersect}}$ share-set)
        if selected-node = nil then done ← true
        else
        begin
            selected-node.state ← growth[8]
            broadcast-bus ← selected-node.variables
```

[7] This sets any node which is superset-one-difference with the newly created node to a state of 4.

[8] Growth is an integer constant unique from the other priorities.

```
        share-set ← share-set union broadcast-bus
    end
end
if share-set ≠ NI.variables then
begin
    share-set ← share-set $\overline{\textit{intersect}}$ NI.variables
    while share-set ≠ $\phi$
    begin
        new-node ← Allocate-New-Node()
        new-node.left-child ← Extract-Element(share-set)
        new-node.right-child ← Extract-Element(share-set)
        new-node.polarity ← nil
        new-node.state ← growth
        new-node.variables ← new-node.left-child and new-node.right-child
    end
end
while there exists ≥ 2 growth nodes
begin
    node1 ← any growth node
    node2 ← any other growth node
    new-node ← Allocate-New-Node()
    new-node.left-child ← node1
    new-node.right-child ← node2
    node1.state ← 0
```

```
        node2.state ← 0
        node1.parents ← node1.parents + new-node
        node2.parents ← node2.parents + new-node
        new-node.state ← growth
        new-node.polarity ← nil
    end
    new-node.polarity ← NI.polarity[9]
end


procedure: Non-Discriminant-Deletion
declare         node: node
begin
    for all nodes if (node.state = 6) and (node.polarity ≠ nil)[10] then
    begin
        Delete-Wave(node.left-child , node)
        Delete-Wave(node.right-child , node)
        Remove-Node(node)
    end
end


Procedure: Delete-Wave(d-node , parent-node)
declare         d-node: undetermined[11]
declare         parent-node: node
```

[9] At this point the last growth node will become the new D-node.
[10] If the node is not a D-node it will be deleted by it's parent.
[11] It could be either a node or a variable.

```
begin
    if d-node is a node then
    begin
        if Size-of(d-node.parents) > 1 then
          d-node.parents ← d-node.parents - parent-node
        else
        begin
            Delete-Wave(d-node.left-child , d-node)
            Delete-Wave(d-node.right-child , d-node)
            Remove-Node(d-node)
        end
    end
end
```

7.4. Example

We now give a short example which demonstrates the basic aspects of AA2. Assume that the first instance input to the system is $AB\overline{C}D \longrightarrow Z$. There are no nodes yet in the network. In this case the AU must guide the building of the new D-node. A and B are connected to the first node, $\overline{C}$ and D to the second, and the two outputs are combined. Figure 7.9 shows the initial network. In the figures, all nodes which have a parent are P-nodes. A top node is always a D-node, marked P or N to signify whether it is a positive or a negative D-node. The connection into the or-plane is not shown.

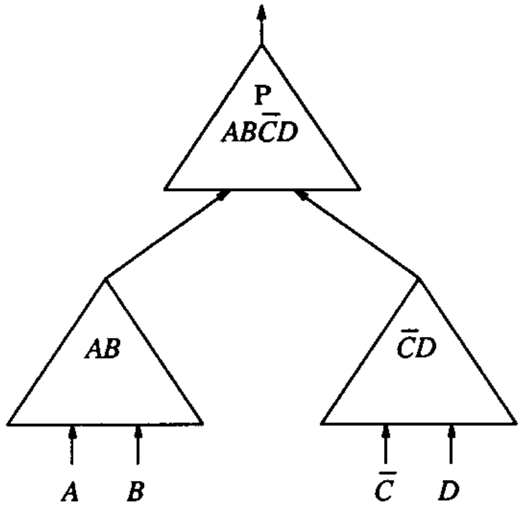


Figure 7.9 - Initial Network

The next instance added is $AB\bar{D}E \longrightarrow \bar{Z}$. After broadcast and the sort wave, a 5 is produced at the top of the network telling the AU that there is no contradiction or minimization possible, that the NI will be entered, and that there are subset P-node(s) to aid in building the new D-node. After arbitration the single subset P-node AB is selected. It broadcasts its variable list, sets itself as a growth node, and sets its priority to 7. Another arbitration returns 7, meaning that there are no more subset P-nodes. A new node is created with $\bar{D}$ and E as the inputs. This new node is combined with the selected P-node, creating a new D-node. The new D-node's polarity is set to negative, and its output is forwarded to the or-plane. Figure 7.10 shows the network structure after this NI addition.

The next instance is $AB\bar{C}D \longrightarrow \bar{Z}$. The sort wave returns a 1 from the right D-node. This node then undergoes *polarity-inversion* and the modification is complete since there are no priority 3 nodes. The right D-node is now a negative D-node.

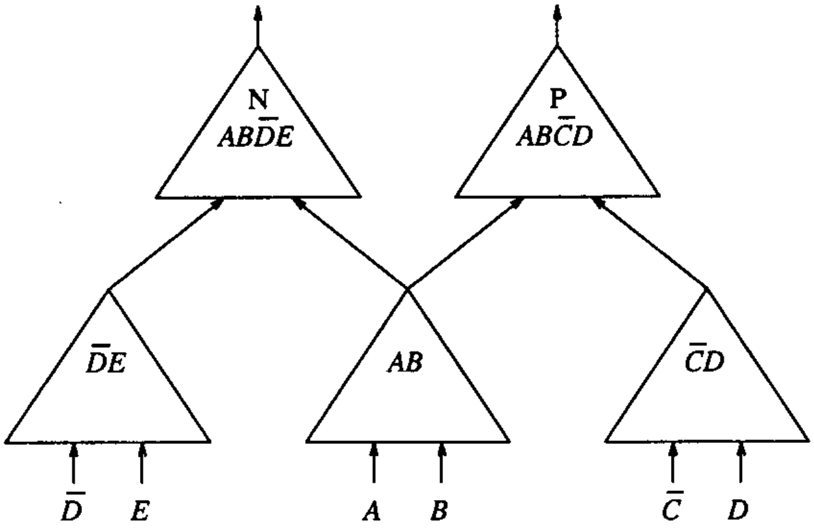


Figure 7.10 - Modified Network

Next, $AB\bar{C} \longrightarrow \bar{Z}$ is input. The right D-node sets itself to self-delete since it is a superset D-node, and the highest output from the sort wave is a 5 from the middle subset P-node. The priority 6 node must not begin self deletion until after any other network modification is complete to insure that no children of the deleted nodes are deleted before they could be used in node combination. The subset P-node is selected and the AU causes it to be combined with the variable $\bar{C}$. After this combination the two nodes set to self-delete begin the self-deletion phase. The network now appears as in figure 7.11.

If self-deletion had been allowed to take place before node combination, and if the middle P-node had not had an output going to the left, that P-node would have been deleted during the self-deletion phase and would have to have been subsequently reallocated.

$ABC \longrightarrow \bar{Z}$ is the next input to the system. The sort wave produces a 2 from

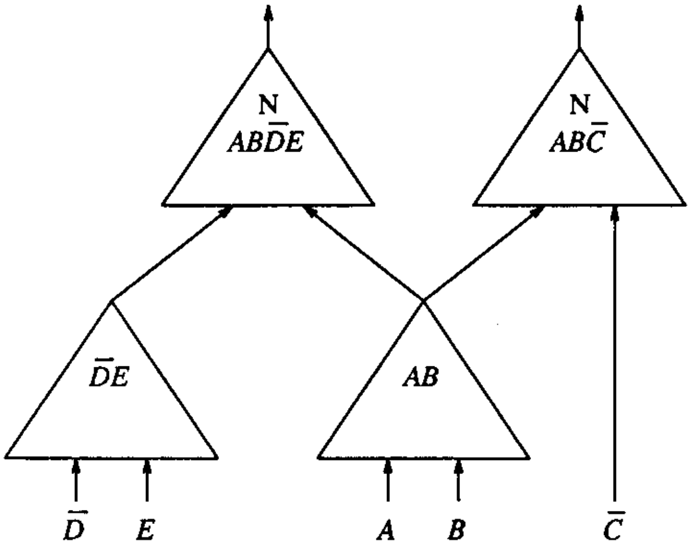


Figure 7.11 - Modified Network

the right P-node since it is one-difference equal. The selected node broadcasts its variable list and the AU modifies the NI to $AB \longrightarrow \bar{Z}$. The AU then rebroadcasts the modified NI to the network. A 1 is produced by the middle P-node since it is equal to the NI. This causes complete discriminant deletion, and all nodes of the network are deleted except the middle P-node. The P-node then changes itself to a negative D-node.

The final instance is $AB\bar{E} \longrightarrow Z$. A 3 is returned by the sort wave meaning that overlap contradiction has taken place and that DVA must be performed. The node is selected, it broadcasts its list, and then sets its priority to 0. At this point the AU causes the selected node to be combined with a discriminant variable from the NI. Since there is only one discriminant variable, only one new node is created. The variable list of the new node is first broadcast by the AU, to see if any node in the network already fulfills it. The variable E is combined with the old D-node, and a new negative D-node is created. The old D-node becomes a P-node.

New node addition after DVA, when necessary, always consists of combining the old selected D-node with the conjunction of the discriminant variables of the NI. This list of discriminant variables is broadcast, and selection waves are used to select subset P-nodes which can build the conjunction. Since no subset P-nodes exist in the current network, the AU causes the old D-node to be combined with $\bar{E}$, creating a new positive D-node. The final network is shown in figure 7.12.

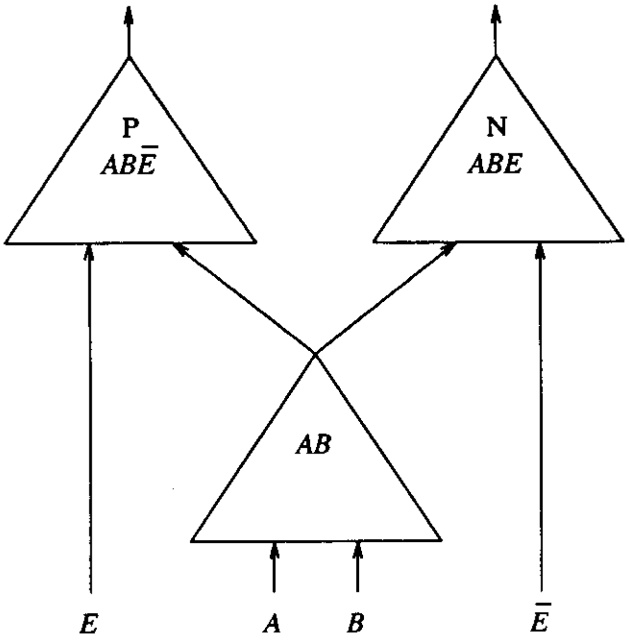


Figure 7.12 - Final Network

7.5. Multiple Outputs

AA2 is easily extendible to multi-variate output. For multiple variables it is only necessary that each D-node have one specific polarity-flag for each variable for which it is a D-node. The variable list stays the same, regardless of the number of output variables. Only 2 bits are required for each additional output variable for which a D-node participates. A P-node can be used for any number of output variables without any memory modification. This is because the extra polarity-flag is only necessary for D-nodes of the output variables.

During any modification phase, only one output variable is under scrutiny, and each D-node must check the polarity flag for that variable, if it exists, and act accordingly. Nodes that do not have a polarity flag for the variable are still involved in the modification process. If a node is a P-node, it can be used for node combination or complete discriminant deletion. If it is a D-node for one or more other variables, it can simultaneously be used as a P-node for the current output variable. if it becomes a D-node for the current variable, it must create a corresponding polarity flag.

Thus, a node could be a P-node for one output variable, a positive D-node for another, and a negative D-node for yet others. If a node is a D-node for more than one variable, it requires separate outputs into the or-plane for each variable.

7.6. Conclusion

A possible variation of AA2 is to limit the system to instances of one polarity of output assertion. This is a case of using a ***Single-Valued Logic***. For example, the system could accept only positive instances and guarantee a 1 whenever a positive instance is matched. Otherwise a 0 would always be output. With this mechanism, the size of the network is greatly reduced. However, it becomes impossible to determine whether the output of the network should be a don't know.

The depth of AA2 network will never exceed the total number of bound input and feedback variables. Thus, the worst cast for execution time is $O(n)$ where n is the number of bound input and feedback variables. If minimization is not performed during the regular adaptation process then the adaption time can also be $O(n)$. Otherwise, the possibility of modification iterations prevent a guarantee that adaptation time will be within that bound.

A unique aspect of AA2 is that the network always knows if an instance has been matched, and it has the ability to output don't know for unmatched environment states. This aspect is only maintained when AA2 keeps the instance set minimal.

Chapter 8

ADAPTIVE ALGORITHM 3

In this chapter we discuss ***Adaptive Algorithm 3 (AA3)***. The layout of this chapter is similar, albeit shorter, than the previous chapter on AA2. The algorithms are alike in many ways. Most of the constructs in AA2 are used unchanged in this chapter. AA3 uses instance broadcast as does AA2. However, AA3 does not use selection waves or modification iterations.

The first section describes the architectural and node requirements of AA3. Section 2 gives an informal description of the algorithm, and section 3 gives the formal description. The next section gives an example of the functioning of AA3. Section 5 discusses multiple-outputs. The following sections give extensions and a summary.

AA3 is similar, although simpler than AA2. Rather than have one D-node for each instance in the instance set, it is only necessary to have sufficient D-nodes to properly discriminate the instance set. When a new instance is broadcast, only discordant D-nodes compare themselves with the new instance, thus the only test is for contradiction. The main features of AA2 and AA3 are the same except for two notable exceptions. AA3 is not able to discern when the environment state does not match any instance, as is done in AA2. However, AA3 does not require flexible interconnections between nodes, allowing for an easier physical implementation.

8.1. Architecture and Node Requirements

In AA3 network nodes are similar to those in AA2. The DPLM function of each node is always the *And* function. There are positive D-nodes, negative D-nodes, and P-nodes. All nodes have one ***Variable Child*** and one ***Node Child.*** A variable child is a literal variable, possibly inverted, which inputs to the node's DPLM. The node child is a network node. Each P-node has two parent nodes: ***Left Parent*** and ***Right Parent.*** A D-node has no parent nodes. All nodes also have a *variable-list* and a *polarity-flag*. The or-plane methodology explained in chapter 7 is used for AA3. Outputs from D-nodes exit the network to the sides and join the or-planes. Since inputs enter at all levels of the network the variable inputs also enter the networks from the side. Figure 8.1 gives a representation of the AA3 network structure.

This *regularity* of the node interconnections is unique to this algorithm and is not found in the previous ones. Because of regularity, implementation can be done with a *fixed* node interconnection structure. There is no need for interconnection multiplexers between layers of nodes, since each node can be permanently attached to its two parents.

The overall structure is that of a binary decision tree (BDT). The bottom node of the network is called the ***Root Node*** and it is unique from all other nodes in the network. It has no inputs, and its DPLM always outputs a "1". Otherwise it is identical to any other node. The root node can be a D-node or a P-node. Figure 8.2 gives an illustration of a possible internal network structure of AA3. The connections from the D-nodes into the or-plane and from the input binder to the nodes are not shown.

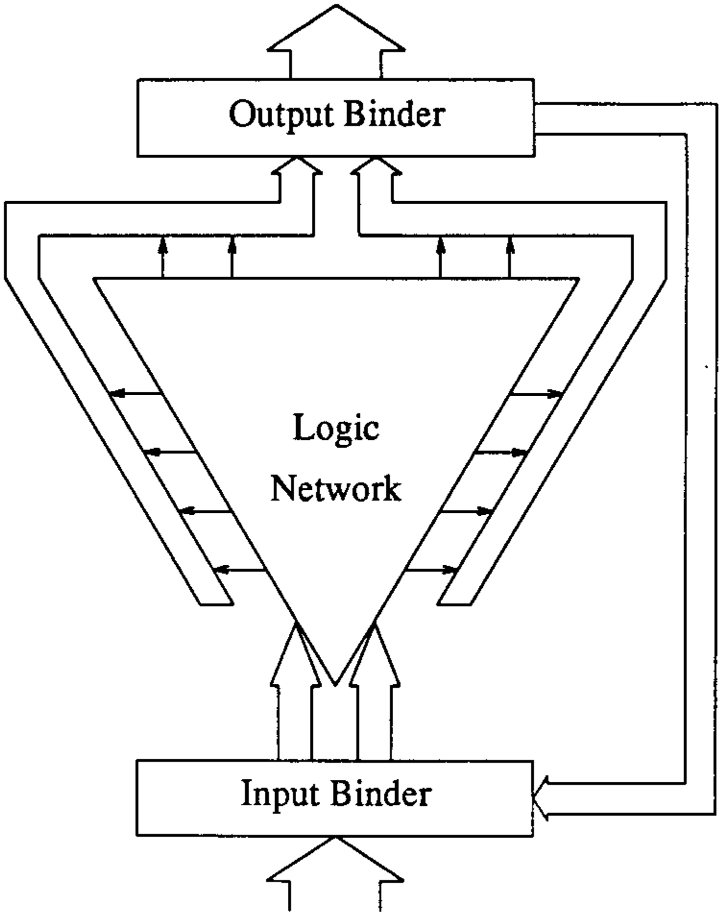


Figure 8.1 - AA3 Network Structure

In a physical implementation a hardwired tree structure would be used. Nodes which are functioning would be *Active* nodes and unallocated nodes are *Inactive* nodes. A deleted node becomes inactive and an allocated node becomes active. The logical structure shown in figure 8.2 could reside on a fixed physical structure as is shown in figure 8.3. Nodes in the figure marked with an "x" represent the active nodes, while all others are currently inactive.

The flexible portion of the network is at the input and output binders. The perpendicular traffic (variable inputs and outputs) needs to be dynamically modifiable. It

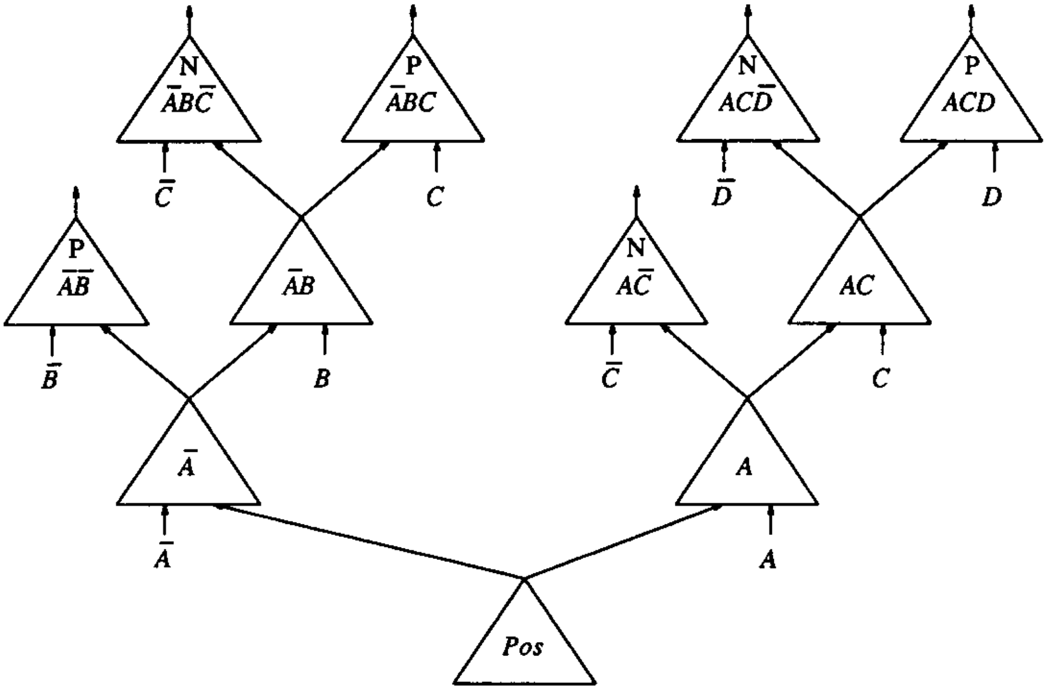


Figure 8.2 - AA3 Internal Network

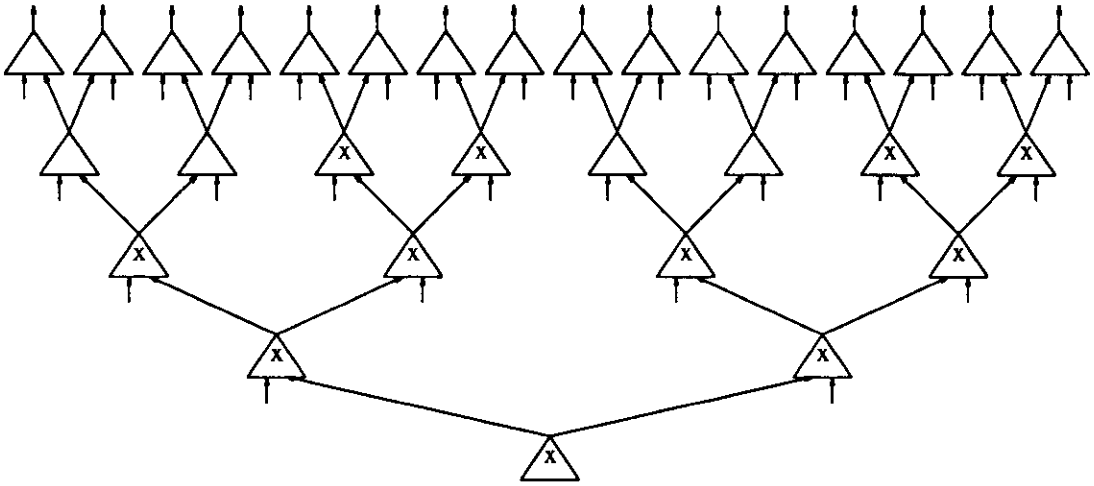


Figure 8.3 - AA3 Fixed Network Implementation

is by changing the variable inputs to the nodes, and the polarity of the nodes, that the network adapts.

The maximum depth of the network is equal to the number of bound input and feedback variables, plus the root node.

8.2. Informal Description

8.2.1. Overview

The overall functioning of AA3 is much simpler than the previous two algorithms. The basic steps are outlined below.

1. Input a New Instance.
2. Broadcast the New Instance.
3. Each discordant D-node tests its relationship to the New Instance.
    a. If the node is Equal or Superset, do *Polarity Inversion.*
    b. If the node is subset or overlap, do *DVA.*
4. Self-Deletion.

There are no loops or modification iterations. No selection waves are required and the actions specified in step 4 can occur concurrently.

The difference between AA2 and AA3 is the discrimination paradigm. When a new instance is entered, AA3 considers only discordant D-nodes, like AA1. It ignores concordant nodes that are matched, as long as no discordant node matches the NI. Table 8.1 illustrates the contrast between AA3 and AA2. It has the same format as the corresponding table in the AA2 chapter, but fewer action entries. An "X" signifies that no action is needed for that node state.

Node is	Subset	Equal	Superset	Overlap
Discordant D-Node	DVA (2)	Polarity Inversion (1)	Polarity Inversion (1)	DVA (2)
Concordant D-Node	X	X	X	X
One Difference D-Node	X	X	X	Impossible State
P-Node	X	X	X	X

Table 8.1 - Node Actions

Nodes which are discordant equal or discordant superset D-nodes undergo *polarity inversion*. This is accomplished by setting the nodes polarity to its complement.

Subset and overlap discordant D-nodes cause *discriminant variable addition (DVA)*. DVA for AA3 is explained in the next section.

All other nodes take no independent action. They may be deleted as part of the self-deletion phase.

8.2.2. Discriminant Variable Addition

Discriminant Variable Addition (DVA) is the only method by which new nodes are created in AA3. The goal of DVA is to create a new D-node with the polarity of the new instance which will always output high when the NI is matched. This node need not be equal to the NI. The other nodes created by DVA will cause the network to output the complement of the NI polarity when environment states previously matching the contradicted node, but not contradicted by the NI, are matched by the environment.

DVA is a recursive procedure which combines the contradicted D-node with discriminant variables from the NI. Each variable in the NI which does not occur in the variable list of the contradicted D-node is considered a *discriminant variable*. The steps of DVA are then as follows.

1. Variable-list ← list of all variables in the NI which do not occur in the variable-list of the D-node.

2. Call DVA with the contradicted D-node as the node argument, and the variable-list as the list argument.

DVA(*node*,*list*)

a. If *list* is empty then exit.

b. Allocate the two parent nodes of *node*.

c. Choose any one discriminant variable from *list* and delete the variable from the *list*.

d. Make one parent a discordant D-node (in terms of the NI) and make the inversion of the chosen variable the nodes variable-child.

e. Make the other parent a concordant D-node (in terms of the NI) and make the chosen variable the nodes variable-child.

f. Set the polarity of *node* to nil. This makes it a P-node.

g. Call DVA with the concordant D-node and *list* as the arguments.

The number of nodes added is equal to 2 times the number of discriminant variables in the NI. There will be one iteration in the procedure above for each discriminant variable. We illustrate DVA with an example.

Assume that a positive D-node exists which has a variable list of $A\bar{B}$. The new instance $A\bar{B}CD \longrightarrow \bar{Z}$ is then input to the system. This instance is broadcast to the network, and the node sets itself as a subset discordant D-node. DVA must take place. One of the discriminant variables, C or D must be chosen for the first iteration. Assume that the variable C is chosen first. Figure 8.4 shows the state of the network after the first iteration.

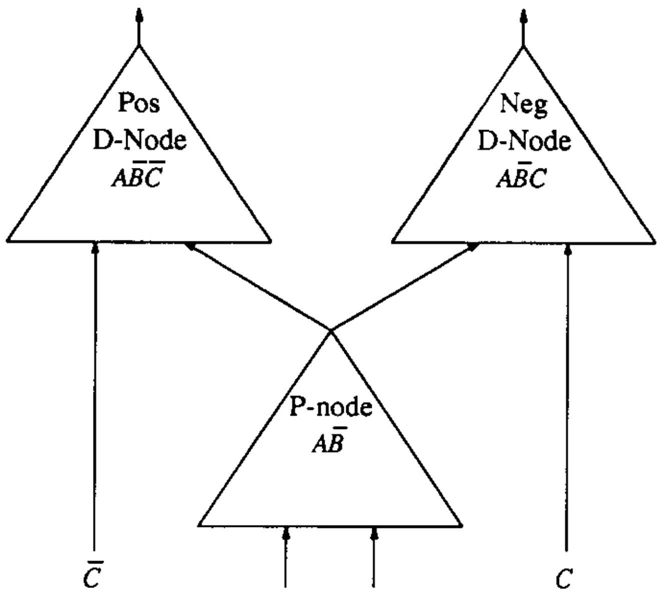


Figure 8.4 - Discriminant Variable Addition

In the figure the old D-node has become a P-node. The node which combined with $\bar{C}$ is now a positive D-node. The right node is now a negative D-node.

Variable D must be chosen for the second iteration. The node to undergo DVA is the *concordant D-node.* This contrasts with the first DVA iteration which is always with a discordant D-node. Figure 8.5 shows the final structure of this piece of the network.

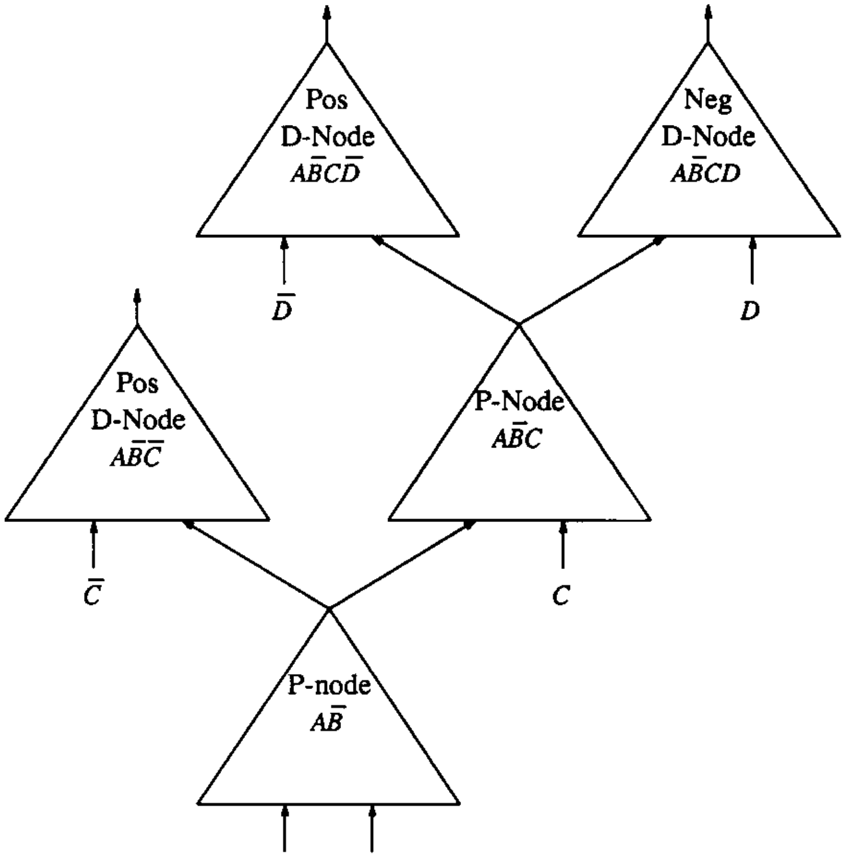


Figure 8.5 - Final Network

It is necessary to continue DVA for each discriminant variable in order to properly fulfill the instance set. If we had left the network as it was after the first DVA then the instance set[1] which the network fulfilled would be

[1] We can never deduce the original instance set by simple observation of the network.

$$A\,\bar{B}\,\bar{C} \rightarrow Z$$

$$A\,\bar{B}\,C \rightarrow \bar{Z}$$

However, according to the rules defined in the knowledge base chapter, the correct IS representation should be

$$A\,\bar{B}\,\bar{C} \rightarrow Z$$

$$A\,\bar{B}\,\bar{D} \rightarrow Z$$

$$A\,\bar{B}\,C\,D \rightarrow \bar{Z}$$

This second and correct IS representation is fulfilled by the final network arrived at when both iterations are done.

8.2.2.1. Alternate Network Implementation for DVA

This section describes an alternate network implementation which allows a two-fold saving of traffic from the input binder. In figure 8.4, the variable B comes in on two separate lines, one for B and one for $\bar{B}$. Figure 8.6 shows an alternative where B can enter the network on one line, and then be inverted by the function of the new discordant D-node. In this case, all network nodes which are left-parents would have a set DPLM function of $x_1 \cdot \bar{x}_2$, while each right-parent would continue to do $x_1 \cdot x_2$.

8.2.3. Self-Deletion

After all polarity inversions or DVA's have been accomplished, the network enters the self-deletion phase. There is only one mode of self-deletion in AA3, in

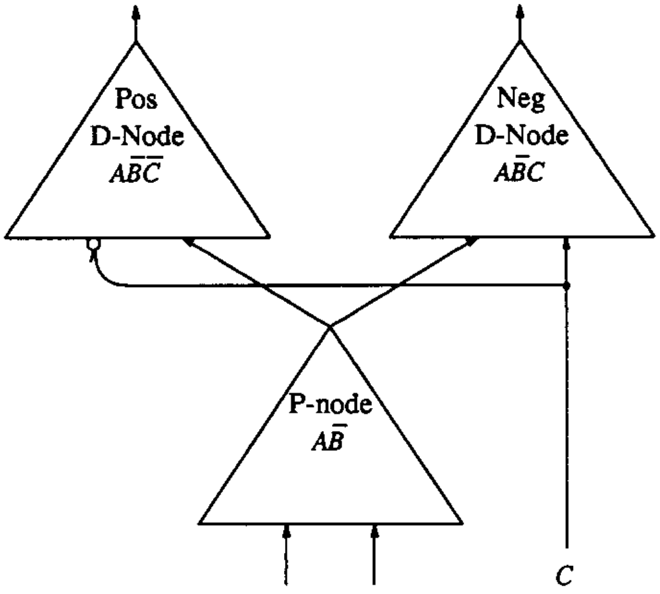


Figure 8.6 - Alternate Implementation of DVA

contrast to the multiple modes in the earlier algorithms. P-nodes do not initiate any self-deletion. It is also sufficient that only modified D-nodes apply the self-deletion test to themselves.

The self-deletion test and procedure is as follows: If the sibling of the D-node is a D-node of the same polarity, then both D-nodes self-delete and the child becomes a D-node with the polarity of its former parents. The new D-node then recursively applies this rule to itself.

Assume that one branch of the network appears as in figure 8.7. Each node is numbered for identification purposes. The new instance $\bar{A}\bar{B}C\bar{D}EG \longrightarrow Z$ is broadcast. This causes node 1 to do polarity inversion. After any other necessary network modification has taken place (in another part of the network), the AU issues the self-delete global command. Node 1 tests its sibling (2) and sees that they are both posi-

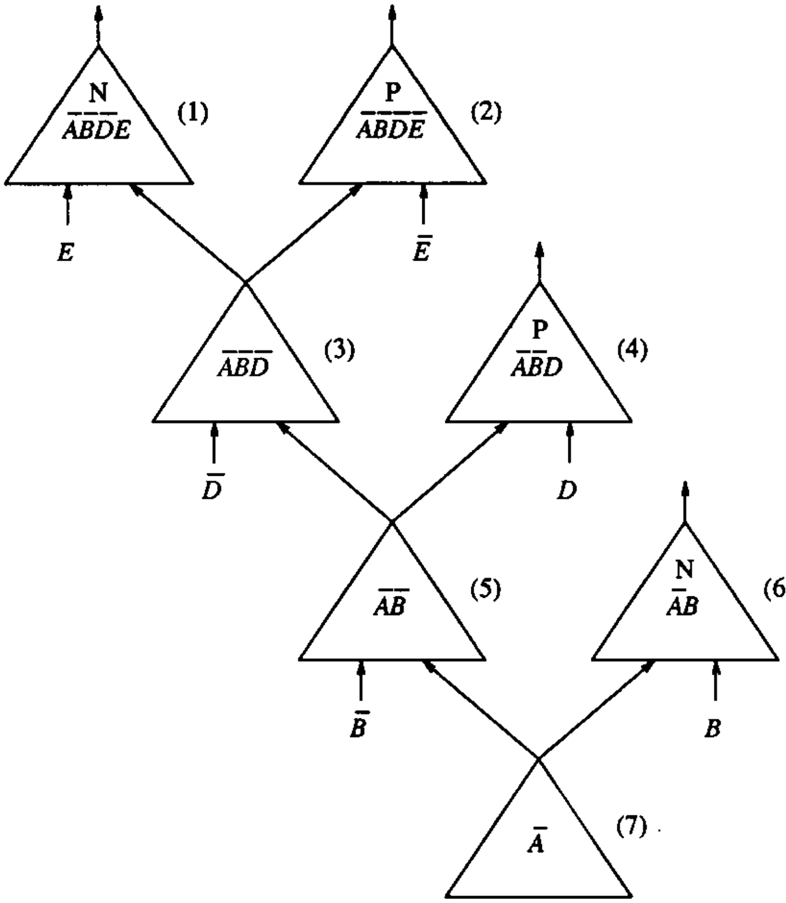

N
$\overline{A}\overline{B}\overline{D}E$
(1)
P
$\overline{A}\overline{B}\overline{D}\overline{E}$
(2)
E
$\overline{E}$
$\overline{A}\overline{B}\overline{D}$
(3)
P
$\overline{A}\overline{B}D$
(4)
$\overline{D}$
D
$\overline{A}\overline{B}$
(5)
N
$\overline{A}B$
(6)
$\overline{B}$
B
$\overline{A}$
(7)

Figure 8.7 - Self Deletion Example

tive D-nodes. Both nodes will then delete themselves and the child node (3) will become a positive D-node. Node 3 now test its sibling (4), which is also a positive D-node. They both self-delete and node 5 becomes a positive D-node. Since the sibling (6) to node 5 is not a positive D-node, no more self deletion can take place. The final network appears as in figure 8.8.

8.3. Formal Description

In this section we give the formal description of AA3 using the same format as in the previous two chapters.

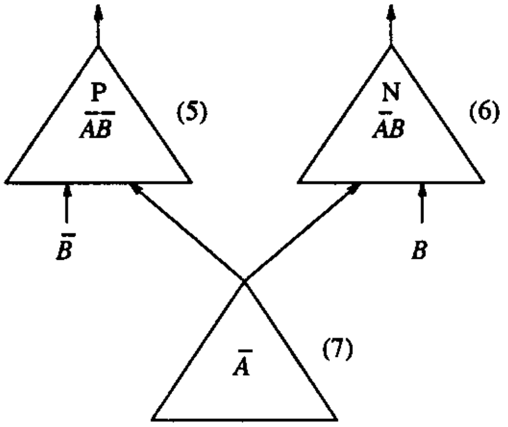


Figure 8.8 - Self Deletion Example

Data Types

set:

list:

polarity: positive, negative, or nil.

variable: variable which occurs in an instance (i.e. A , $\bar{B}$, etc.)

instance: instance.variables is a list of the variables in an instance.

instance.polarity is positive or negative. $\overline{\textit{polarity}}$ is the complement of polarity.

Data structures

node Network node whose subparts are:

node.variable-child - Variable

node.node-child - Node

node.left-parent - Node

node.right-parent - Node

node.polarity - positive (D-node), negative (D-node), or nil (P-node)

node.variables - List of variables which the node conjuncts

```
program: AA3-Update(NI)
declare          NI: Instance
begin
    Add-New-Instance(NI)
    for all nodes Self-Deletion-Test(node)
end


procedure: Add-New-Instance(NI)
declare          NI: Instance
declare          current: node
begin
    Broadcast(NI)
    for all nodes (current)[2]
    begin
        if current.polarity = $\overline{NI.polarity}$[3] then
        begin
            if current.variable-list is superset or equal to NI.variables then
                current.polarity ← NI.polarity
            else if (there are no discriminant variables between
                NI.variables and current.variable-list)[4] then
                   Discriminant-Variable-Addition(current , NI)
```

[2] *current* is the instantiation for each node used in the "for all" construct.

[3] Note that only discordant D-nodes can ever be involved in modification.

```
        end
    end
end

procedure: Discriminant-Variable-Addition(current, NI)
declare     current: node
declare     NI: instance
declare     concordant-node: node
declare     discordant-node: node
declare     dvar: variable
declare     discriminant-variables: set of variables
begin
    discriminant-variables ← Get-Shared-Variables(NI.variables, current.variables)
    if discriminant-variables $\neq \phi$ then
    begin
        dvar ← Select-Any-Element(discriminant-variables)
        discriminant-variables ← discriminant-variables minus dvar
        discordant-node ← Allocate-New-Node()
        discordant-node.variable-child ← $\overline{dvar}$
        discordant-node.node-child ← current
        discordant-node.polarity ← $\overline{NI.polarity}$
        discordant-node.variable-list ← current.variable-list + $\overline{dvar}$
        concordant-node ← Allocate-New-Node()
```

[4] If the node is a subset or overlap node.

```
            concordant-node.variable-child ← dvar
            concordant-node.node-child ← current
            concordant-node.polarity ← NI.polarity
            concordant-node.variable-list ← current.variable-list + dvar
            current.polarity ← nil
            current.left-parent ← discordant-node
            current.right-parent ← concordant-node
            *Discriminant-Variable-Addition*(concordant-node , NI)
        end
    end
```

```
procedure: *Get-Shared-Variables*(NI-variables , node-variables , returns:new-set)
declare         NI-variables: list of variables
declare         node-variables: list of variables
declare         new-set: set of variables
begin
    new-set ← $\phi$
    while NI-variables not empty
    begin
        if *First*(NI-variables) not an element of node-variables then
            new-set ← new-set + *First*(NI-variables)
        NI-variables ← NI-variables minus *First*(NI-variables)
    end
    return(new-set)
end
```

```
procedure: Self-Deletion-Test(node)
declare          node: node
begin
    if Sibling(node).polarity = node.polarity then
    begin
        node.node-child.polarity ← node.polarity
        node.node-child.left-parent ← nil
        node.node-child.right-parent ← nil
        Remove-Node(node)
        Remove-Node(Sibling(node))
        Self-Deletion-Test(node.node-child)
    end
end
```

9. Example

In this example each major aspect of AA3 is illustrated. Assume that the first instance input is $A\bar{B}D \longrightarrow \bar{Z}$. Since this is the first node, and it is impossible for any discrimination to take place, the root node, which always outputs a 1, is set as a negative D-node. Thus, the initial network configuration is shown in figure 9.

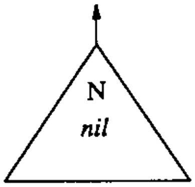


Figure 8.9 - Initial Network

In this and the following figures each node contains its variable-list and polarity. In terms of polarity, P is positive, N is negative, and no entry means the node is a P-node.

The next instance is $\bar{A}BDE \longrightarrow \bar{Z}$. Since there are no positive D-nodes in the network, no action can take place.

The following instance is $ABC \longrightarrow Z$. The lone network node is set to action state 2 for this case, and DVA must take place. Since there are three discriminant variables in the NI, and since the root node has a null variable list, three iterations take place. The order in which the three discriminant variables are processed is inconsequential. Assuming the order chosen is ABC then the network would appear as in figure 8.10. Each node is numbered for identification purposes.

The next instance input to the system is $\bar{A}D \longrightarrow Z$. Node 2 is a subset node to the NI so it must do DVA. The only discriminant variable is D so two new nodes are created in the single iteration. No self-deletion can take place, and the new network appears in figure 8.11.

$A\bar{B}\bar{C} \longrightarrow \bar{Z}$ is entered next. Since no positive D-node matches the NI, the NI is already fulfilled and no network modification takes place.

The final instance entered is $B \longrightarrow Z$. This instance matches discordant nodes 6 and 8. Node 6 is a superset instance and thus polarity inversion immediately takes place. Node 8 is an overlap node, and will be involved in DVA. B is the only discriminant variable so one DVA iteration takes place, creating two new D-nodes. At this point the self-deletion phase is started. The polarity inversion of node 6 has made

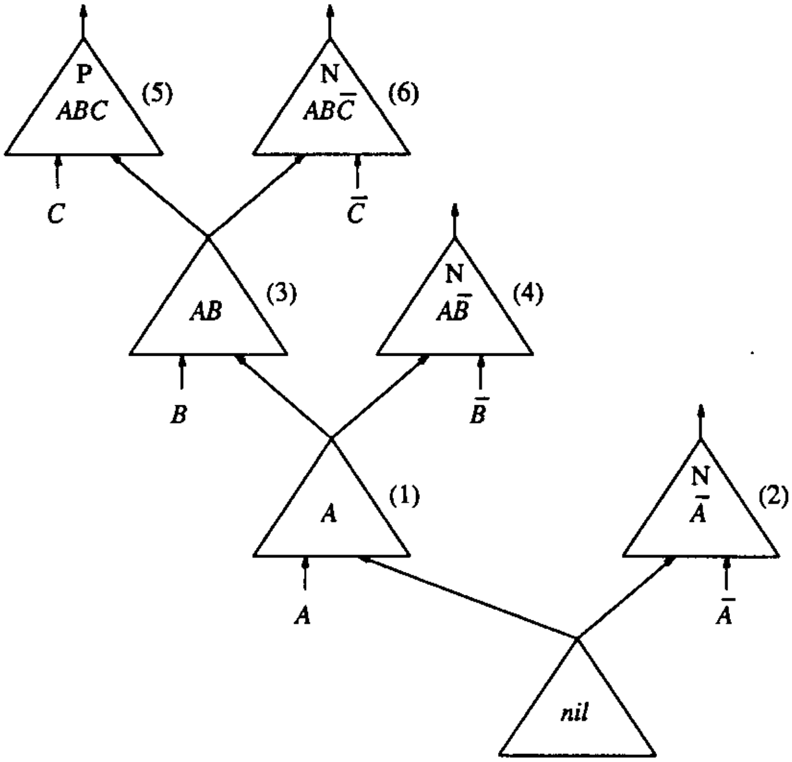


Figure 8.10 - Modified Network

it concordant with node 5. Thus, nodes 5 and 6 both self-delete, and node 3 becomes a new positive D-node. Since node 4 is a negative D-node, no further deletion takes place. The final network is pictured in figure 8.12.

We can now view the distributed nature of AA3. There is no single node in the network which is responsible to be active when B is asserted; corresponding to the just entered instance $B \longrightarrow Z$. Instead we see that the responsibility is spread across three positive D-nodes: 3, 7, and 9. At least one of these nodes will always be active when B is active. There is no single node which is always active when B is active. This contrasts with the method of discrimination in both AA2 and normal BDT structures. In these methodologies there is one node or path responsible to be active when-

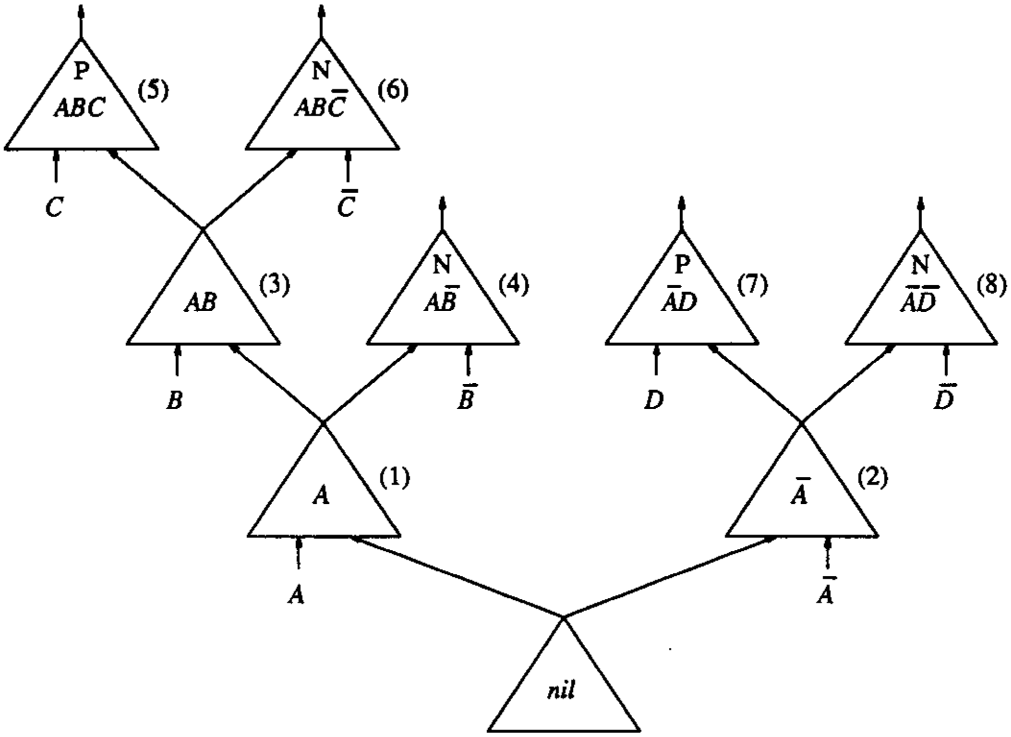


Figure 8.11 - Modified Network

ever a specific instance is matched. For a minimal BDT it is still the case that one node is always active whenever one of the set of minterms it is responsible for is matched.

9.1. Multiple Outputs

AA3 is amenable to multiple outputs by simply adding a polarity flag for each output variable at any node which is a D-node for that variable. For example, assume that we have the network constructed for variable Z as shown in figure 8.12.

If the instance $AB \longrightarrow \bar{Z}_2$ were added, then node 3 would become a negative D-node (defined by the new polarity flag corresponding to Z_2) for Z_2, while remaining a positive D-node for Z. Any instance additions for any other variable can never

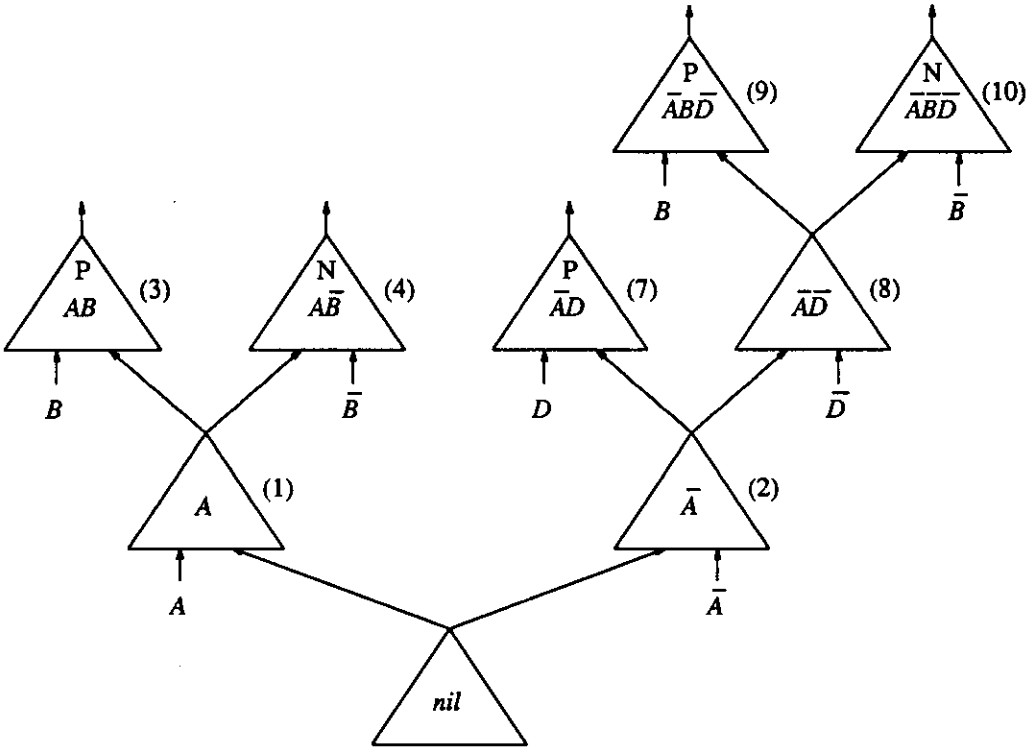


Figure 8.12 - Final Network

change the status of the variable Z D-nodes.

If the instance $\bar{A} \longrightarrow Z_2$ was then added, node 2, which is a P-node for Z will become a positive D-node for Z_2. The output of node 2 would be connected to the or-plane of Z_2.

Finally, assume that $ABG \longrightarrow Z_2$ is added. This contradicts with node 3 which was made a negative D-node for Z_2. DVA would take place at this node, creating two new D-nodes for Z_2. Node 4 would still remain a positive D-node for Z just as before, although it is now a P-node for Z_2.

9.2. Responsibility Lists

In AA3 no node (or any other agent) knows which instances it is responsible for. The instance set is not stored inside or outside of the network. Recall back to the example when the network was made up of only the root node (which was a negative D-node) after the input of 2 negative instances. The instance $ABC \longrightarrow Z$ was then entered. Since the node does not know which instances it is responsible for, it is necessary to discriminate all 3 of the discriminant variables. Realizing what the initial instances were, it is seen that only the variables A and B are necessary to do discrimination sufficient to fulfill the true instance set. Discriminating the variable C does not hurt, nor does it aid in this case. However, the state of the network, before the modification, could represent an infinite number of instance sets. At the point when $ABC \longrightarrow Z$ was input, any number of negative instances could already have been input to the system, causing no change.

The reason then, that DVA must iterate over all discriminant variables of a NI, is that the network has no memory of the instance set. It fulfills it without ever knowing what it is.

A very powerful extension can be made to both AA3 and especially AA2, causing both to produce reduced networks. First we realize that the self-deletion rule of AA3 means that at any given node, if there is not both a positive and a negative D-node in the tree above the node, then the tree above the node can be pruned, and the node will become a D-node, having the polarity of the concordant D-node(s) in the tree above it. Thus, a single D-node can represent many instances. If a memory is kept at the node, containing a list of instances for which the D-node is responsible, then when DVA is done involving that node, a minimal DVA[5] can always be accomplished. This list of instances, stored at each D-node, containing the instances for which the D-node is responsible is called a ***Responsibility List.***

Of course, the memory, or the size of the list, has to be set at a limit. If the number of instances for which a single node is responsible grows beyond the memory size of the node, then it will sprout two new D-nodes using the DVA method, itself becoming a P-node, and the D-nodes will then each share a portion of the instance list small enough to fit within the memory constraints.

The main disadvantage to this method for AA2, besides the memory addition, is that it is no longer possible to ensure the don't know output. For AA3, the cost is also in node memory and the gain, as in AA2, is a smaller number of network nodes used to fulfill the same instance set.

9.3. Features and Summary

The maximum length path in an AA3 network is equal to the total number of bound input and feedback variables. Thus both execution and adaptation time are $O(n)$ where n is the total number of bound input and feedback variables. Because there are no modification iterations in AA3 the adaptation process can run in real-time.

Because the system is represented by a binary decision tree, it could also be represented as a ULM multiplexer tree. This is done by turning the tree upside down,

[5] One which only iterates over the necessary discriminant variables.

and allowing nodes in the bottom levels of the tree to dynamically set their DPLM's to *Positive* or *Negative*. For the single output case this would be a more efficient architecture, because it would obviate the need for the or-plane. However, the ULM technique breaks down for the multiple-output case. This is because the data flowing through the lines of the ULM network actually represent the output value. In the BDT structure the data represents control which selects D-nodes, which can then flexibly distribute data which manipulate the or-plane.

The perpendicular data lines on which input and output variables flow to and from layers of the network will have a limited bandwidth. AA3, like the other algorithms, could then use the layered architecture, and limit the number of output variables in an adaptive plane to a value which does not exceed the bandwidth of the I/O paths.

Of course, at the limit, one could use only one output variable per adaptive plane in the network. Because of the minimal amount of circuitry needed for AA3, this could still be a reasonable architecture. The advantage of this technique, is that perpendicular busses could be cut way down. All traffic to the or-planes could flow vertically through unused nodes of the network, since all D-nodes are at the top edge of the *used portion* of the network for the single output case.

In AA3 there is no notion of a minimal instance set, since there is no instance set stored. The network always computes a *consistent* form of the given instances. Observation of the current network structure is not sufficient to extract the true instance set. It is because of this lack of knowledge of the instance set that AA3 needs no modification iterations. This is because the perfect minimization of boolean functions is provably NP-complete.

AA3 has a number of advantages over earlier algorithms. One is its comparative simplicity. Another is the fact that adaptation time can be truly linear with the depth of the network due to the fact that no modification iterations are ever necessary. Perhaps most important in terms of implementation is the fact that it can operate on a network with fixed interconnections between nodes. It is still necessary to have flexible interconnections between nodes, which is obviated in AA3.

Chapter 9

ALGORITHM SIMULATIONS and EXAMPLE

In this chapter we discuss the implementation and results of computer simulations of the three adaptive algorithms discussed in the preceding chapters. The first section discusses why and how the simulations were implemented. Section 2 gives a detailed account of a specific simulation run on each of the algorithms and then compares the results. Section 3 gives a comparative summary of the three algorithms. Finally, section 4 illustrates the mapping of a concrete application into instances and an example of its implementation using one of the algorithms.

9.1. Why and How

Simulations were undertaken with two goals in mind. The first was to determine that the algorithms work as explained when the instance set grows large. The simulations empirically demonstrated that point and new understanding and insights were gained.

The second goal was to gather statistical data of the performance of the algorithms. This statistical analysis is the main purpose of this chapter. The statistics gathered are listed below. In the list, an *adaptation* means the overall modification to the system following addition of a single new instance.

1. Number of instances in the instance set.

2. Number of nodes in the logic network.

3. Maximum depth of the network.

4. Number of nodes deleted per adaptation.

5. Number of nodes created per adaptation.

6. Net change in number of nodes per adaptation.

7. Number of modification iterations per adaptation.

In order to obtain these statistics, three separate simulations were written, one for each algorithm. They were written in T, a dialect of Lisp, and run on an Apollo computing environment.

A separate program was provided to interface user and network. This program stores the instance set and maintains it consistent and minimal. For AA1 this program handles the instance set maintenance as would typically be done by the adaption unit. AA2 and AA3 do not require external instance set maintenance and the instance set in this program is only used for user aid and algorithm testing. A number of help and user aid commands can be called from the interface program. For instance, at any time the user can examine the nodes of the network, the instance set, or system statistics.

A large part of the effort was invested in the testing of the network. The network is tested after every adaptation to ensure that it correctly fulfills the instance set and maintains all other claimed features.

In order to allow large simulations, the interface program can access files. A separate program creates files of random instance sets of any given size. The input to this program is the maximum number of variables per instance, the number of instances, and a seed for the random number generator. The program then generates a file containing a list of random instances in a format compatible with the interface program. These instances are then extracted incrementally from the file. After each instance, statistics are written out to an output file.

A number of different instance files were generated and run on each of the three simulations. The next section shows in detail the input and output of one of these instance files, for each of the three simulations.

There are two main reasons to chose random lists of instances for simulation. First, it would take a long time to build up a large set of instances from the semantics of some real-world application. The second is that a random set tests a *worst case scenario* in terms of the amount of adaptation needed in the network. We use the term *worst case* to represent the class of adaptations requiring a large amount of computing effort. We do not intend the term to imply the theoretically worst single example. A set of random instances repeatedly causes contradictions to the instance set. Thus it is possible to observe the working of the network in situations requiring large amounts of adaptation. Real world applications should require much less network adaptation.

9.2. A Detailed Simulation

As mentioned in the previous section, many different instance files, varying in the number of instances and number of variables, were submitted to simulation. The same instance file could thus be tested against each algorithm and the results compared. For this chapter an instance file containing 50 instances, each limited to a maximum conjunction of 12 variables, is shown. This case is small enough to be easily followed and gives similar results to other simulations. In addition, it contains a couple *worst case* (degenerate) situations which show weaknesses in the first two algorithms.

The simulations were written for the single output case. Positive instances for the output are those that imply T (true), and negative instances are those the imply F (false). Each simulation receives the instances of the same file as input. The 50 instances in the file are shown below.

1. $\bar{f}\ \bar{l}\ \bar{a}\ \bar{i}\ \bar{k}\ h\ c\ d \longrightarrow T$
2. $e\ b \longrightarrow T$
3. $b \longrightarrow F$
4. $\bar{c}\ \bar{h}\ i\ \bar{b} \longrightarrow F$
5. $f\ \bar{i}\ c \longrightarrow T$
6. $i\ g\ a\ \bar{c}\ \bar{e}\ h\ j \longrightarrow T$
7. $f\ \bar{k} \longrightarrow T$
8. $\bar{j}\ i\ \bar{b}\ \bar{a}\ k\ e \longrightarrow F$
9. $\bar{c}\ i \longrightarrow F$
10. $\bar{k}\ d\ j\ \bar{b}\ f\ \bar{l}\ \bar{h}\ e\ c\ \bar{i} \longrightarrow T$
11. $b\ e \longrightarrow F$
12. $\bar{k}\ d\ i\ \bar{e}\ l\ \bar{j}\ \bar{h}\ \bar{f}\ c\ b\ \bar{a} \longrightarrow T$
13. $\bar{h} \longrightarrow F$
14. $\bar{b}\ k\ l\ a \longrightarrow T$
15. $c\ \bar{f}\ g\ \bar{k}\ l\ \bar{d}\ e\ \bar{b}\ a\ \bar{h}\ \bar{j}\ i \longrightarrow F$
16. $c\ \bar{b} \longrightarrow F$

17. $i\ \bar{d} \longrightarrow F$

18. $\bar{b}\ f\ g\ k\ \bar{l}\ \bar{c}\ a\ i\ h\ j\ \bar{d}\ \bar{e} \longrightarrow F$

19. $f\ b\ \bar{k}\ \bar{l}\ a\ \bar{j}\ c \longrightarrow T$

20. $\bar{h}\ \bar{j}\ \bar{f}\ \bar{l}\ g\ i \longrightarrow F$

21. $\bar{j}\ \bar{f}\ g\ \bar{b}\ i\ d\ \bar{c}\ \bar{k}\ \bar{l}\ \bar{a} \longrightarrow F$

22. $e\ \bar{d}\ \bar{j}\ g\ \bar{b} \longrightarrow T$

23. $c\ \bar{d}\ \bar{e}\ \bar{i}\ \bar{j}\ k\ \bar{g}\ \bar{a} \longrightarrow F$

24. $c \longrightarrow F$

25. $h\ \bar{j}\ d\ g\ \bar{b}\ l\ a\ \bar{e}\ f\ k \longrightarrow T$

26. $a\ \bar{b}\ e\ \bar{c}\ \bar{l} \longrightarrow F$

27. $\bar{a}\ \bar{i}\ k\ \bar{g}\ \bar{c} \longrightarrow F$

28. $h\ \bar{l}\ \bar{a}\ f\ \bar{k}\ \bar{i} \longrightarrow T$

29. $\bar{g}\ a\ l \longrightarrow F$

30. $g\ \bar{a}\ l\ k\ \bar{f} \longrightarrow T$

31. $i\ j\ \bar{k}\ \bar{b}\ e\ a\ h\ d\ \bar{f}\ c\ g\ l \longrightarrow F$

32. $\bar{c}\ a\ \bar{e}\ \bar{i}\ \bar{h}\ \bar{j}\ k\ \bar{d}\ l\ \bar{g}\ \bar{f}\ b \longrightarrow F$

33. $j\ l\ \bar{h}\ \bar{f}\ g\ \bar{a}\ c\ b\ i \longrightarrow F$

34. $f\ k\ l\ \bar{a}\ d\ c\ g\ \bar{j}\ \bar{h}\ i \longrightarrow F$

35. $\bar{d}\ \bar{c}\ g\ \bar{a}\ \bar{f}\ b\ \bar{k}\ \bar{l}\ i\ h\ \bar{j} \longrightarrow T$

36. $j\ \bar{h}\ k\ f\ \bar{l}\ \bar{e}\ d\ a\ \bar{c}\ \bar{g}\ i\ \bar{b} \longrightarrow T$

37. $j\ l \longrightarrow T$

38. $\bar{b}\ j\ \bar{k}\ e\ \bar{l}\ i\ T$

39. $a\ \bar{j}\ k\ \bar{l}\ \bar{g}\ h\ \bar{e}\ \bar{d}\ \bar{f}\ b \longrightarrow F$

40. $\bar{b}\ \bar{a}\ e\ \bar{l}\ \bar{i}\ h\ j\ c\ \bar{f} \longrightarrow F$

41. $j \longrightarrow F$

42. $\bar{j}\ e\ f\ g\ k\ b\ \bar{a} \longrightarrow F$

43. $\bar{l}\ \bar{a}\ \bar{c}\ \bar{j}\ f\ \bar{i}\ b\ k \rightarrow T$

44. $\bar{f}\ b\ \bar{g}\ \bar{j} \rightarrow T$

45. $b\ \bar{d}\ \bar{i}\ c\ \bar{a}\ e\ \bar{l}\ \bar{g}\ j \rightarrow T$

46. $\bar{l}\ c\ a\ j\ \bar{f}\ \bar{g}\ \bar{i}\ h\ e\ d \rightarrow F$

47. $i\ \bar{c}\ \bar{j}\ l\ e\ \bar{g}\ \bar{b}\ \bar{h}\ a \rightarrow T$

48. $\bar{l}\ \bar{j}\ \bar{i}\ d\ \bar{a}\ \bar{k}\ \bar{f}\ \bar{h}\ g\ b \rightarrow F$

49. $f\ i\ k\ \bar{a} \rightarrow F$

50. $\bar{k}\ \bar{j}\ \bar{i} \rightarrow F$

Since the instances are the same for each simulation, the final instance set will also be the same, regardless of what algorithm is used. Recall that the 3-state instance function is independent of the algorithm and network. The actual 2-state function implemented by the network structures of each algorithm is different in each case. The final consistent and minimal instance set contains 226 instances. Following is what the instance set looks like at the end of each simulation, after the sequential input and processing of each of the 50 instances.

$L\ A\ \bar{I}\ K\ \bar{C}\ \bar{B}\ G\ \bar{J} \rightarrow T$

$L\ A\ K\ \bar{C}\ D\ \bar{B}\ G\ \bar{J} \rightarrow T$

$F\ L\ A\ K\ H\ D\ \bar{E}\ \bar{B}\ G\ \bar{J} \rightarrow T$

$\bar{F}\ L\ \bar{A}\ K\ G\ \bar{J} \rightarrow T$

$F\ \bar{L}\ I\ \bar{E} \rightarrow F$

$\bar{L}\ I\ \bar{E}\ \bar{B} \rightarrow F$

$F\ I\ B \rightarrow F$

$F\ \bar{A}\ I\ \bar{E} \rightarrow F$

$F\ \bar{A}\ I\ D \rightarrow F$

$F\ I\ \bar{D}\ \bar{E} \rightarrow F$

$F\ L\ E\ B \rightarrow F$

$F\ L\ C\ \bar{D} \rightarrow F$

$F\ I\ C\ \bar{D} \rightarrow F$

$F\ L\ C\ B \rightarrow F$

$F\ L\ \bar{A}\ C \rightarrow F$

$F\ \bar{A}\ I\ C \rightarrow F$

$F\ L\ C\ E \rightarrow F$

$F\ I\ C\ E \rightarrow F$

$F\ I\ \bar{K}\ \bar{E} \rightarrow F$

$I\ \bar{K}\ \bar{E}\ \bar{B} \rightarrow F$

$\bar{L}\ \bar{H}\ \bar{E}\ \bar{B} \rightarrow F$

$\bar{L}\ \bar{H}\ D\ \bar{B} \rightarrow F$

$\bar{A}\ K\ \bar{B}\ \bar{G} \rightarrow F$

$F\ L\ \bar{A}\ \bar{H}\ \bar{E} \rightarrow F$

$F\ \bar{A}\ \bar{H}\ \bar{E}\ \bar{B} \rightarrow F$

$F\ L\ \bar{A}\ \bar{H}\ D \rightarrow F$

$F\ \bar{A}\ \bar{H}\ D\ \bar{B} \rightarrow F$

$F\ L\ K\ B \rightarrow F$

$F\ L\ \bar{H}\ B \rightarrow F$

$F\ \bar{L}\ \bar{A}\ \bar{I}\ K\ \bar{C}\ B\ \bar{J} \rightarrow T$

$\bar{L}\ K\ C\ \bar{B} \rightarrow F$

$\bar{L}\ K\ C\ G \rightarrow F$

$\bar{F}\ \bar{L}\ D\ B\ G \rightarrow F$

$\bar{F}\ \bar{L}\ \bar{I}\ B\ G \rightarrow F$

$F\ \bar{L}\ I\ D \rightarrow F$

$\bar{L}\,I\,D\,\bar{B} \rightarrow F$

$\bar{L}\,I\,D\,G \rightarrow F$

$F\,\bar{L}\,I\,\bar{G} \rightarrow F$

$\bar{L}\,I\,\bar{B}\,\bar{G} \rightarrow F$

$\bar{L}\,I\,K\,\bar{E}\,G \rightarrow F$

$F\,\bar{L}\,A\,I \rightarrow F$

$\bar{L}\,A\,I\,\bar{B} \rightarrow F$

$\bar{L}\,A\,I\,G \rightarrow F$

$I\,\bar{K}\,D\,G \rightarrow F$

$F\,\bar{A}\,I\,\bar{G} \rightarrow F$

$\bar{A}\,I\,\bar{B}\,\bar{G} \rightarrow F$

$F\,\bar{K}\,\bar{H}\,\bar{E} \rightarrow F$

$\bar{K}\,\bar{H}\,\bar{E}\,\bar{B} \rightarrow F$

$\bar{K}\,\bar{H}\,\bar{E}\,G \rightarrow F$

$\bar{K}\,\bar{H}\,D\,G \rightarrow F$

$F\,A\,E\,B \rightarrow F$

$A\,E\,B\,G \rightarrow F$

$F\,A\,C\,\bar{D} \rightarrow F$

$A\,C\,\bar{D}\,\bar{B} \rightarrow F$

$A\,C\,\bar{D}\,G \rightarrow F$

$F\,L\,C\,\bar{G} \rightarrow F$

$L\,C\,\bar{B}\,\bar{G} \rightarrow F$

$F\,A\,C\,\bar{G} \rightarrow F$

$A\,C\,\bar{B}\,\bar{G} \rightarrow F$

$K\,C\,\bar{B}\,\bar{G} \rightarrow F$

$F\,I\,C\,\bar{G} \rightarrow F$

$I\ C\ \bar{B}\ \bar{G} \rightarrow F$

$F\ A\ C\ B \rightarrow F$

$A\ C\ B\ G \rightarrow F$

$F\ A\ C\ E \rightarrow F$

$A\ C\ E\ \bar{B} \rightarrow F$

$A\ C\ E\ G \rightarrow F$

$F\ L\ \bar{K}\ C \rightarrow F$

$L\ \bar{K}\ C\ \bar{B} \rightarrow F$

$L\ \bar{K}\ C\ G \rightarrow F$

$F\ A\ \bar{K}\ C \rightarrow F$

$A\ \bar{K}\ C\ \bar{B} \rightarrow F$

$A\ \bar{K}\ C\ G \rightarrow F$

$F\ I\ \bar{K}\ C \rightarrow F$

$I\ \bar{K}\ C\ \bar{B} \rightarrow F$

$I\ \bar{K}\ C\ G \rightarrow F$

$F\ A\ K\ B \rightarrow F$

$A\ K\ B\ G \rightarrow F$

$A\ I\ B\ G \rightarrow F$

$F\ A\ \bar{H}\ B \rightarrow F$

$A\ \bar{H}\ B\ G \rightarrow F$

$\bar{K}\ \bar{H}\ B\ G \rightarrow F$

$A\ I\ \bar{D}\ \bar{E}\ \bar{B} \rightarrow F$

$A\ I\ \bar{D}\ \bar{E}\ G \rightarrow F$

$\bar{F}\ A\ B\ G \rightarrow F$

$A\ \bar{H}\ C\ \bar{B} \rightarrow F$

$A\ \bar{H}\ C\ G \rightarrow F$

$\bar{K}\,\bar{H}\,C\,\bar{B} \rightarrow F$

$\bar{K}\,\bar{H}\,C\,G \rightarrow F$

$\bar{F}\,C\,\bar{B}\,\bar{G} \rightarrow F$

$\bar{F}\,A\,C\,\bar{B} \rightarrow F$

$\bar{F}\,A\,C\,G \rightarrow F$

$\bar{F}\,\bar{K}\,C\,\bar{B} \rightarrow F$

$\bar{F}\,\bar{K}\,C\,G \rightarrow F$

$L\,\bar{K}\,E\,B\,G \rightarrow F$

$A\,I\,\bar{K}\,\bar{E}\,G \rightarrow F$

$L\,I\,\bar{K}\,\bar{E}\,G \rightarrow F$

$L\,I\,\bar{K}\,B\,G \rightarrow F$

$\bar{F}\,\bar{K}\,D\,B\,G \rightarrow F$

$\bar{F}\,L\,\bar{K}\,B\,G \rightarrow F$

$\bar{F}\,\bar{I}\,\bar{K}\,B\,G \rightarrow F$

$F\,\bar{L}\,A\,E \rightarrow F$

$\bar{L}\,A\,E\,\bar{B} \rightarrow F$

$\bar{L}\,A\,E\,G \rightarrow F$

$F\,\bar{L}\,A\,C \rightarrow F$

$\bar{L}\,A\,C\,\bar{B} \rightarrow F$

$\bar{L}\,A\,C\,G \rightarrow F$

$F\,\bar{L}\,I\,C \rightarrow F$

$\bar{L}\,I\,C\,\bar{B} \rightarrow F$

$\bar{L}\,I\,C\,G \rightarrow F$

$\bar{L}\,\bar{H}\,C\,\bar{B} \rightarrow F$

$\bar{L}\,\bar{H}\,C\,G \rightarrow F$

$\bar{F}\,\bar{L}\,C\,\bar{B} \rightarrow F$

$\bar{F}\,\bar{L}\,C\,G \rightarrow F$

$\bar{F}\,\bar{L}\,K\,B\,G \rightarrow F$

$\bar{L}\,I\,K\,B\,G \rightarrow F$

$\bar{F}\,\bar{L}\,\bar{H}\,B\,G \rightarrow F$

$\bar{L}\,I\,\bar{H}\,B\,G \rightarrow F$

$\bar{F}\,\bar{L}\,\bar{H}\,\bar{E}\,G \rightarrow F$

$\bar{L}\,I\,\bar{H}\,\bar{E}\,G \rightarrow F$

$\bar{F}\,\bar{L}\,\bar{H}\,D\,G \rightarrow F$

$F\,\bar{L}\,A\,\bar{H} \rightarrow F$

$\bar{L}\,A\,\bar{H}\,\bar{B} \rightarrow F$

$\bar{L}\,A\,\bar{H}\,G \rightarrow F$

$\bar{F}\,\bar{L}\,\bar{A}\,I\,\bar{K}\,H\,\bar{C}\,\bar{D}\,B\,\bar{J} \rightarrow T$

$\bar{B}\,J \rightarrow F$

$D\,J \rightarrow F$

$I\,J \rightarrow F$

$\bar{C}\,J \rightarrow F$

$A\,J \rightarrow F$

$\bar{E}\,J \rightarrow F$

$L\,J \rightarrow F$

$G\,J \rightarrow F$

$F\,\bar{H}\,C\,\bar{B} \rightarrow F$

$F\,\bar{H}\,C\,D \rightarrow F$

$F\,I\,\bar{H}\,C \rightarrow F$

$F\,A\,\bar{H}\,C \rightarrow F$

$F\,\bar{H}\,C\,\bar{E} \rightarrow F$

$F\,L\,\bar{H}\,C \rightarrow F$

$F\,\bar{H}\,C\,G \rightarrow F$

$F\,\bar{H}\,C\,\bar{J} \rightarrow F$

$F\,K\,C\,\bar{D}\,\bar{B} \rightarrow F$

$F\,K\,C\,\bar{D}\,\bar{E} \rightarrow F$

$F\,K\,C\,\bar{D}\,G \rightarrow F$

$F\,K\,C\,\bar{D}\,\bar{J} \rightarrow F$

$F\,\bar{A}\,K\,C\,\bar{B} \rightarrow F$

$F\,\bar{A}\,K\,C\,D \rightarrow F$

$F\,\bar{A}\,K\,C\,\bar{E} \rightarrow F$

$F\,\bar{A}\,K\,C\,G \rightarrow F$

$F\,\bar{A}\,K\,C\,\bar{J} \rightarrow F$

$F\,K\,C\,E\,\bar{B} \rightarrow F$

$F\,K\,C\,D\,E \rightarrow F$

$F\,K\,C\,E\,G \rightarrow F$

$F\,K\,C\,E\,\bar{J} \rightarrow F$

$F\,K\,C\,D\,B \rightarrow F$

$F\,K\,C\,\bar{E}\,B \rightarrow F$

$F\,K\,C\,B\,G \rightarrow F$

$F\,K\,C\,B\,\bar{J} \rightarrow F$

$F\,\bar{L}\,K\,C\,D \rightarrow F$

$F\,\bar{L}\,K\,C\,\bar{E} \rightarrow F$

$F\,K\,C\,D\,\bar{G} \rightarrow F$

$F\,K\,C\,\bar{E}\,\bar{G} \rightarrow F$

$F\,K\,C\,\bar{G}\,\bar{J} \rightarrow F$

$F\,\bar{K}\,\bar{H}\,\bar{C}\,B \rightarrow F$

$F\,\bar{K}\,\bar{H}\,B\,\bar{J} \rightarrow F$

$\bar{L}\ \bar{A}\ \bar{I}\ C\ \bar{D}\ E\ B\ \bar{G}\ J \longrightarrow T$

$\bar{I}\ \bar{H}\ \bar{B}\ \bar{G} \longrightarrow F$

$\bar{H}\ C\ \bar{B}\ \bar{G} \longrightarrow F$

$\bar{L}\ \bar{H}\ \bar{B}\ \bar{G} \longrightarrow F$

$\bar{H}\ \bar{E}\ \bar{B}\ \bar{G} \longrightarrow F$

$\bar{A}\ \bar{H}\ \bar{B}\ \bar{G} \longrightarrow F$

$F\ I\ \bar{K}\ H\ D \longrightarrow F$

$I\ \bar{K}\ H\ D\ \bar{B} \longrightarrow F$

$\bar{A}\ I\ \bar{K}\ D\ \bar{B} \longrightarrow F$

$F\ I\ \bar{K}\ H\ \bar{G} \longrightarrow F$

$I\ \bar{K}\ H\ \bar{B}\ \bar{G} \longrightarrow F$

$F\ \bar{I}\ \bar{K}\ \bar{H}\ D \longrightarrow F$

$F\ \bar{L}\ \bar{K}\ \bar{H}\ D \longrightarrow F$

$F\ \bar{K}\ \bar{H}\ D\ B \longrightarrow F$

$F\ \bar{A}\ \bar{K}\ \bar{H}\ D \longrightarrow F$

$\bar{I}\ \bar{K}\ \bar{H}\ D\ \bar{B} \longrightarrow F$

$\bar{A}\ \bar{K}\ \bar{H}\ D\ \bar{B} \longrightarrow F$

$F\ I\ H\ \bar{D}\ \bar{G} \longrightarrow F$

$I\ \bar{D}\ \bar{E}\ \bar{B}\ \bar{G} \longrightarrow F$

$I\ H\ \bar{D}\ \bar{B}\ \bar{G} \longrightarrow F$

$F\ L\ A\ \bar{I}\ \bar{G} \longrightarrow F$

$F\ L\ A\ \bar{E}\ \bar{G} \longrightarrow F$

$F\ L\ A\ B\ \bar{G} \longrightarrow F$

$F\ L\ A\ H\ \bar{G} \longrightarrow F$

$L\ A\ \bar{I}\ \bar{B}\ \bar{G} \longrightarrow F$

$L\ A\ \bar{E}\ \bar{B}\ \bar{G} \longrightarrow F$

$L\ A\ H\ \bar{B}\ \bar{G} \rightarrow F$

$F\ L\ \bar{I}\ \bar{H}\ \bar{G} \rightarrow F$

$F\ L\ \bar{H}\ \bar{E}\ \bar{G} \rightarrow F$

$F\ L\ \bar{A}\ \bar{H}\ \bar{G} \rightarrow F$

$F\ A\ \bar{I}\ \bar{H}\ \bar{G} \rightarrow F$

$F\ A\ \bar{H}\ \bar{E}\ \bar{G} \rightarrow F$

$F\ I\ \bar{H}\ \bar{E}\ \bar{G} \rightarrow F$

$F\ L\ \bar{I}\ K\ \bar{G} \rightarrow F$

$F\ L\ K\ \bar{E}\ \bar{G} \rightarrow F$

$F\ L\ K\ H\ \bar{G} \rightarrow F$

$F\ L\ \bar{A}\ K\ \bar{G} \rightarrow F$

$L\ \bar{I}\ K\ \bar{B}\ \bar{G} \rightarrow F$

$L\ K\ \bar{E}\ \bar{B}\ \bar{G} \rightarrow F$

$L\ K\ H\ \bar{B}\ \bar{G} \rightarrow F$

$F\ \bar{I}\ \bar{K}\ \bar{H}\ \bar{C}\ \bar{G} \rightarrow F$

$F\ \bar{L}\ \bar{K}\ \bar{H}\ \bar{C}\ \bar{G} \rightarrow F$

$F\ \bar{A}\ \bar{K}\ \bar{H}\ \bar{C}\ \bar{G} \rightarrow F$

$F\ \bar{L}\ \bar{K}\ \bar{H}\ \bar{G}\ \bar{J} \rightarrow F$

$F\ \bar{A}\ \bar{K}\ \bar{H}\ \bar{G}\ \bar{J} \rightarrow F$

$L\ A\ I\ \bar{H}\ \bar{C}\ E\ \bar{B}\ \bar{G}\ \bar{J} \rightarrow T$

$F\ \bar{A}\ I\ K \rightarrow F$

$\bar{F}\ K\ B\ \bar{G}\ \bar{J} \rightarrow T$

$\bar{F}\ I\ B\ \bar{G}\ \bar{J} \rightarrow T$

$\bar{F}\ \bar{A}\ K\ \bar{C}\ \bar{D}\ E\ \bar{B}\ G\ \bar{J} \rightarrow T$

$\bar{F}\ \bar{A}\ I\ \bar{C}\ \bar{D}\ E\ \bar{B}\ G\ \bar{J} \rightarrow T$

$\bar{A}\ \bar{I}\ K\ \bar{C}\ \bar{D}\ E\ \bar{B}\ G\ \bar{J} \rightarrow T$

$\bar{A}\ I\ \bar{K}\ \bar{C}\ \bar{D}\ E\ \bar{B}\ G\ \bar{J} \rightarrow T$

$\bar{F}\ L\ K\ \bar{C}\ \bar{D}\ E\ \bar{B}\ G\ \bar{J} \rightarrow T$

$\bar{F}\ L\ I\ \bar{C}\ \bar{D}\ E\ \bar{B}\ G\ \bar{J} \rightarrow T$

$L\ \bar{I}\ K\ \bar{C}\ \bar{D}\ E\ \bar{B}\ G\ \bar{J} \rightarrow T$

$L\ I\ \bar{K}\ \bar{C}\ \bar{D}\ E\ \bar{B}\ G\ \bar{J} \rightarrow T$

$L\ A\ I\ \bar{C}\ \bar{D}\ E\ \bar{B}\ G\ \bar{J} \rightarrow T$

$\bar{I}\ \bar{K}\ \bar{J} \rightarrow F$

$L\ A\ K\ \bar{C}\ E\ \bar{B}\ G\ \bar{J} \rightarrow T$

After the input of each instance in the instance set statistics are generated and output to a file. The statistics generated are the first 7 measures listed in the previous section. Four output files are shown in this section. The first table corresponds to AA1. Table 2 refers to a slight variant of AA1, which we here identify as AA1-b. The third and fourth are for AA2 and AA3 respectively.

Recall that for AA1, it is possible to enter the self-deletion phase at two different places. Self-deletion could be accomplished after all modification to the network has taken place. This is how it was explained in chapter 6. It is also possible that self-deletion occur after processing of the *delete-list* and before processing of the *add-list*. Either way is correct in that there is no further minimization[1] possible to the network. The first statistical file shows the original case, where self-deletion is the final operation. The second file shows the case where self-deletion takes place before node creation (AA1-b).

[1] In terms of complete discriminant and non-discriminant deletion. Locally redundant deletions was not programmed in the simulations.

The results for the four cases are shown on the following pages followed by a comparison and discussion of each.

Instance	# of Instances	# of Nodes	Depth	Nodes Deleted	Nodes Created	Net Change	Modification Iterations
1	1	0	0	0	0	0	0
2	2	0	0	0	0	0	0
3	2	0	0	0	0	0	1
4	3	1	1	0	1	1	1
5	6	4	3	0	3	3	1
6	23	11	4	0	7	7	1
7	25	13	5	0	2	2	1
8	26	13	5	0	0	0	0
9	15	10	6	6	3	-3	2
10	15	10	6	0	0	0	0
11	18	13	7	0	3	3	1
12	41	23	8	0	10	10	1
13	19	15	9	10	2	-8	2
14	26	20	10	0	5	5	1
15	26	20	10	0	0	0	0
16	25	25	12	0	5	5	2
17	27	28	13	0	3	3	1
18	27	28	13	0	0	0	0
19	51	35	14	0	7	7	1
20	51	35	14	0	0	0	0
21	51	35	14	0	0	0	0
22	98	41	15	0	6	6	1
23	102	45	16	0	4	4	1
24	38	58	23	21	34	13	14
25	48	66	24	0	8	8	1
26	51	70	25	0	4	4	1
27	52	73	26	0	3	3	1
28	73	79	27	0	6	6	1
29	76	82	28	0	3	3	1
30	102	88	29	0	6	6	1
31	102	88	29	0	0	0	0
32	102	88	29	0	0	0	0
33	107	93	30	0	5	5	1
34	107	93	30	0	0	0	0
35	118	106	32	0	13	13	2
36	170	122	33	0	16	16	1
37	175	128	35	13	19	6	5
38	201	132	36	0	4	4	1
39	201	132	36	0	0	0	0
40	201	132	36	0	0	0	0
41	100	133	46	21	22	1	13
42	100	133	46	0	0	0	0
43	113	164	53	0	31	31	7
44	160	166	53	7	9	2	2
45	194	176	54	0	10	10	1
46	194	176	54	0	0	0	0
47	222	187	55	0	11	11	1
48	222	187	55	0	0	0	0
49	228	192	56	0	5	5	1
50	226	186	53	15	9	-6	2

Table 1 - Adaptive Algorithm 1 - With Self-Deletion Last

Instance	# of Instances	# of Nodes	Depth	Nodes Deleted	Nodes Created	Net Change	Modification Iterations
1	1	0	0	0	0	0	0
2	2	0	0	0	0	0	0
3	2	0	0	0	0	0	1
4	3	1	1	0	1	1	1
5	6	4	3	1	4	3	2
6	23	11	4	0	7	7	1
7	25	13	5	0	2	2	1
8	26	13	5	0	0	0	0
9	15	10	6	9	6	-3	3
10	15	10	6	0	0	0	0
11	18	15	7	9	14	5	5
12	41	23	8	0	8	8	1
13	19	15	7	21	13	-8	5
14	26	21	9	1	7	6	3
15	26	21	9	0	0	0	0
16	25	21	9	21	21	0	7
17	27	31	11	6	16	10	4
18	27	31	11	0	0	0	0
19	51	41	13	3	13	10	3
20	51	41	13	0	0	0	0
21	51	41	13	0	0	0	0
22	98	48	15	11	18	7	5
23	102	53	16	3	8	5	2
24	38	24	10	36	7	-29	3
25	48	31	11	0	7	7	1
26	51	36	12	6	11	5	3
27	52	39	13	0	3	3	1
28	73	44	14	0	5	5	1
29	76	46	13	12	14	2	4
30	102	52	14	0	6	6	1
31	102	52	14	0	0	0	0
32	102	52	14	0	0	0	0
33	107	60	16	6	14	8	3
34	107	60	16	0	0	0	0
35	118	68	17	0	8	8	1
36	170	80	18	0	12	12	1
37	175	80	18	36	36	0	6
38	201	88	19	0	8	8	1
39	201	88	19	0	0	0	0
40	201	88	19	0	0	0	0
41	100	156	59	80	148	68	55
42	100	156	59	0	0	0	0
43	113	173	61	2	19	17	3
44	160	202	67	86	115	29	39
45	194	222	69	8	28	20	5
46	194	222	69	0	0	0	0
47	222	231	69	6	15	9	3
48	222	231	69	0	0	0	0
49	228	244	72	9	22	13	5
50	226	264	78	34	54	20	12

Table 2 - Adaptive Algorithm 1 - With Self-Deletion First (AA1-b)

Instance	# of Instances	# of Nodes	Depth	Nodes Deleted	Nodes Created	Net Change	Modification Iterations
1	1	7	1	0	7	7	1
2	2	8	1	0	1	1	1
3	2	9	10	1	2	1	1
4	3	11	10	0	2	2	2
5	6	15	9	1	5	4	2
6	23	40	9	0	25	25	1
7	25	55	9	0	15	15	1
8	26	59	9	0	4	4	2
9	15	50	9	12	3	-9	3
10	15	40	9	10	0	-10	1
11	18	44	9	3	7	4	1
12	41	81	11	0	37	37	2
13	19	48	9	40	7	-33	5
14	26	59	9	0	11	11	1
15	26	59	9	0	0	0	1
16	25	55	7	11	7	-4	2
17	27	59	8	0	4	4	1
18	27	59	8	0	0	0	1
19	51	91	8	0	32	32	1
20	51	91	8	0	0	0	1
21	51	91	8	0	0	0	1
22	98	162	8	0	71	71	1
23	102	171	8	0	9	9	2
24	38	123	7	50	2	-48	25
25	48	138	11	0	15	15	1
26	51	144	11	0	6	6	3
27	52	144	11	1	1	0	3
28	73	177	11	0	33	33	1
29	76	184	11	0	7	7	1
30	102	233	11	13	62	49	1
31	102	233	11	0	0	0	1
32	102	233	11	0	0	0	1
33	107	239	11	0	6	6	5
34	107	239	11	0	0	0	1
35	118	266	12	1	28	27	1
36	170	327	13	4	65	61	1
37	175	358	12	55	86	31	5
38	201	416	12	5	63	58	2
39	201	416	12	0	0	0	1
40	201	416	12	0	0	0	1
41	100	206	12	222	12	-210	83
42	100	206	12	0	0	0	1
43	113	239	12	0	33	33	1
44	160	331	11	21	113	92	2
45	194	381	11	0	50	50	1
46	194	381	11	0	0	0	1
47	222	415	11	0	34	34	1
48	222	415	11	0	0	0	1
49	228	423	11	0	8	8	1
50	226	415	11	22	14	-8	1

Table 3 - Adaptive Algorithm 2

Instance	# of Instances	# of Nodes	Depth	Nodes Deleted	Nodes Created	Net Change
1	1	1	1	0	1	1
2	2	1	1	0	0	0
3	2	3	2	0	2	2
4	3	9	5	0	6	6
5	6	15	5	0	6	6
6	23	41	10	0	26	26
7	25	59	11	0	18	18
8	26	79	11	0	20	20
9	15	69	9	16	6	-10
10	15	69	9	0	0	0
11	18	71	9	8	10	2
12	41	101	13	0	30	30
13	19	111	12	10	20	10
14	26	145	12	0	34	34
15	26	139	11	6	0	-6
16	25	125	10	14	0	-14
17	27	123	11	16	14	-2
18	27	109	11	14	0	-14
19	51	155	13	0	46	46
20	51	149	13	6	0	-6
21	51	141	13	8	0	-8
22	98	201	13	0	60	60
23	102	207	13	4	10	6
24	38	181	12	26	0	-26
25	48	197	12	0	16	16
26	51	197	12	24	24	0
27	52	185	12	20	8	-12
28	73	259	13	0	74	74
29	76	277	13	8	26	18
30	102	357	13	0	80	80
31	102	349	13	8	0	-8
32	102	345	13	4	0	-4
33	107	349	13	4	8	4
34	107	349	13	0	0	0
35	118	361	13	0	12	12
36	170	373	13	0	12	12
37	175	459	13	22	108	86
38	201	485	13	8	34	26
39	201	485	13	0	0	0
40	201	485	13	0	0	0
41	100	407	13	108	30	-78
42	100	357	13	50	0	-50
43	113	363	13	0	6	6
44	160	365	13	0	2	2
45	194	401	13	0	36	36
46	194	385	13	16	0	-16
47	222	403	13	2	20	18
48	222	397	13	6	0	-6
49	228	407	13	2	12	10
50	226	395	13	26	14	-12

Table 4 - Adaptive Algorithm 3

We first contrast the two versions of AA1. In terms of number of nodes AA1-b (which does deletion first) usually has about 10% less nodes. This statistic is not born out by the tables in this chapter, but it is over the course of many other simulations. Average statistics for all simulations are shown in the next section. For the current case AA1-b maintains a smaller network until instance 41 is reached.[2] After that AA1 maintains a smaller network. However, the difference in node count is negligible. The big difference is the larger amount of processing effort that AA1-b has to put forth in order to gain a small advantage in terms of number of nodes. The net change per modification is only slightly greater for AA1-b. The real difference is in the modification iterations. For instance 41, AA1 requires 13 modification iterations. In contrast, AA1-b needs 51 iterations.

Instance 41 is a worst case because it contradicts and is minimizable with a large percentage of the instance set. Instance 41 is $J \longrightarrow F$. It has only one variable and thus has a very large representation space. Instance 24 is another instance of this type and it can be seen that it also causes large changes to the network. These types of contradictory instances should be infrequent in real world applications.

When an instance such as instance 41 is entered, it can minimize a large portion of the instance set. When self-deletion is immediately undertaken, as in AA1-b, a large portion of the network gets deleted. However, the add-list is also very large, and a long series of network modifications are subsequently necessary. For instance 41, AA1-b deletes 80 nodes and then it creates 148 new nodes to fulfill the instance set.

[2] Instance 41 becomes infamous throughout the rest of this section, because it is a worst (degenerate) case.

AA1 processes the add-list before doing self-deletion. It only creates 22 nodes, after which it deletes 21. It was able to use most of the redundant nodes in the network to satisfy the add-list. In most cases, AA1-b can rebuild a more efficient network, with the cost of extra processing, than AA1.

We can now compare the three basic algorithms. The first two measures, number of nodes and depth, characterize the structure of the network. In terms of number of nodes, AA1 is better by a factor of 2 (AA2 and AA3 require approximately twice as many nodes per instance). However, the depth (which determines execution and adaptation time) is always greater in AA1. The network depth of AA2 and AA3 are both bound by the number of input variables.

The rest of the measures concern the amount of modification and processing time necessary to do network updates. In AA1 the net change, in terms of number of nodes, is less than for the other two algorithms. The worst case number of modification iterations in AA1 is also less than for AA2. Note however, that AA2 rarely has more than one modification iteration, but when they appear there can be many. AA3 never performs modification iterations, and thus this column is left out of the corresponding table. AA3 does have a large amount of net change compared to the other algorithms. However, on a fixed structure much of the changes are node memory updates rather than the interconnection modifications required in the other two algorithms.

AA2 and AA3 distinguish between P-nodes and D-nodes. For AA2, the number of D-nodes is always equal to the number of instances, since there must be one for each. Thus, for this case, AA2 ends up with 226 D-nodes (21 positive and 205 negative) and 191 P-nodes. AA3 has 201 D-nodes (41 positive and 160 negative) and 194 P-nodes.

Only one output variable was used in the simulations. The number of nodes per output variable would always be less with several outputs, since nodes would typically be shared amongst outputs.

9.3. Comparative Summary

Statistical measures for the three algorithms are empirical and thus tentative. Table 9.5 summarizes the garnered statistics for the three algorithms. The first two statistics represent the overall network structure at any time. The last three statistics show the amount of change necessary to update the network after the introduction of a new instance. These values represent the statistics gained from all simulations, not just those reported in the preceding section. The average values are approximate and have been rounded off. They are followed by the observed ranges in which they fall. In the second statistic n is the number of instances. For modification iterations it is assumed that one single iteration is required for the normal network update.

The number of nodes relative to the number of input variables was found to be fairly constant for any number of variables. There is a slight and constant increase of the percentage of nodes as the number of variables grow. The main factor affecting number of nodes is the number of instances in the set.

Measure	AA1	AA2	AA3
Nodes per Instance	1 (0-3)	2 (0-4)	2 (0-5)
Depth per # of Input Variables	n^x $0 \leq x < 1.5$	.9 (0-1)	.9 (0-1)
Average Nodes Deleted per Total Nodes	.02	.1	.2
Average Nodes Created per Total Nodes	.05	.15	.2
Average Modification Iterations	1.3 (1-n/5)	2 (1-n/2)	1 (none)

Table 9.5 - Statistical Summary

9.4. An Application to Fault Isolation

In this section we take a small real world application, map it into instances, and then show a possible network structure that fulfills the instance set. The example comes from the flight control of an experimental aircraft tested by NASA.

The system is originally described by the and/or/not graph shown in figure 9.1.

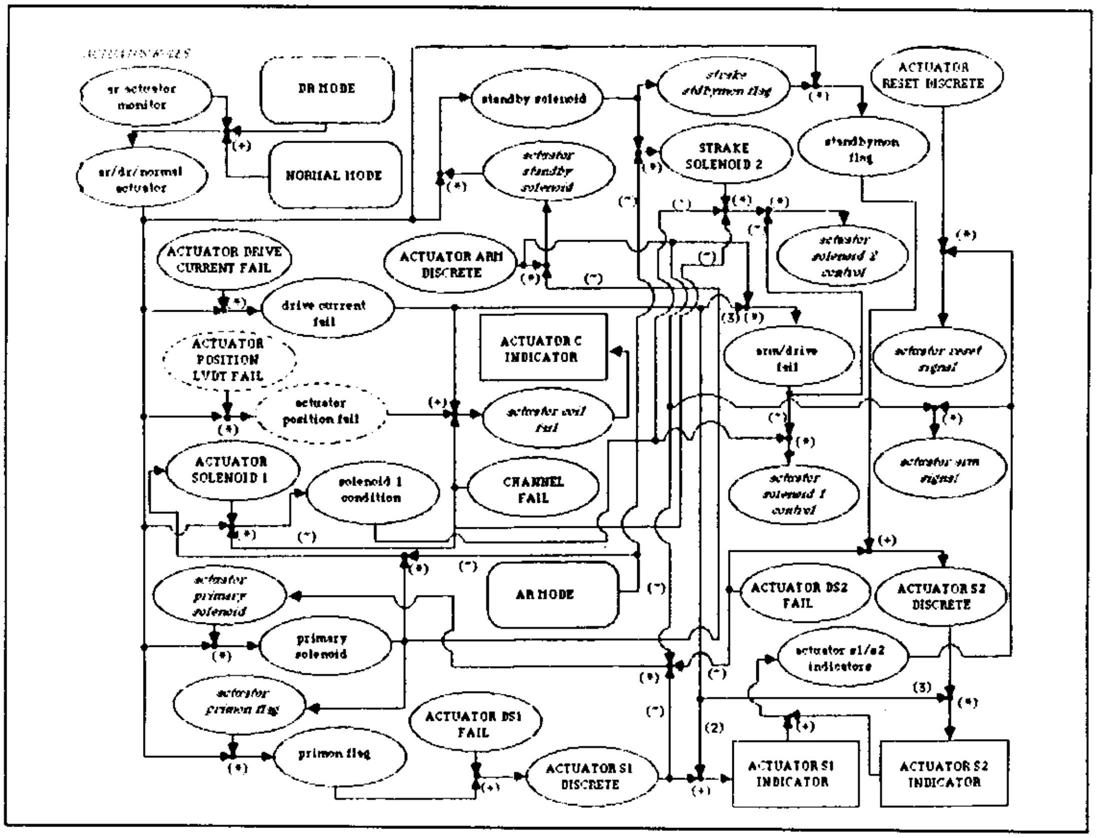


Figure 9.1 - And/Or/Not Graph Representation of Flight Control Sub-System

From this graph we choose one major output variable, *Actuator S2 Indicator.* The solution required for this output variable illustrates the two basic types of system feedback. A network can easily be extended to use all of the output variables.

The paths for the single output variable can be reexpressed by the cyclic tree structure of figure 9.2. The output variable *Actuator S2 Indicator* is at the top of the tree. Two other variables are also defined as output variables: *ar/dr/normal actuator* and *primary solenoid.* These two are shown in bold wherever they appear as output and in italics where they are input to other nodes.

The variable *ar/dr/normal* is an intermediate variable. Its definition (in the left-

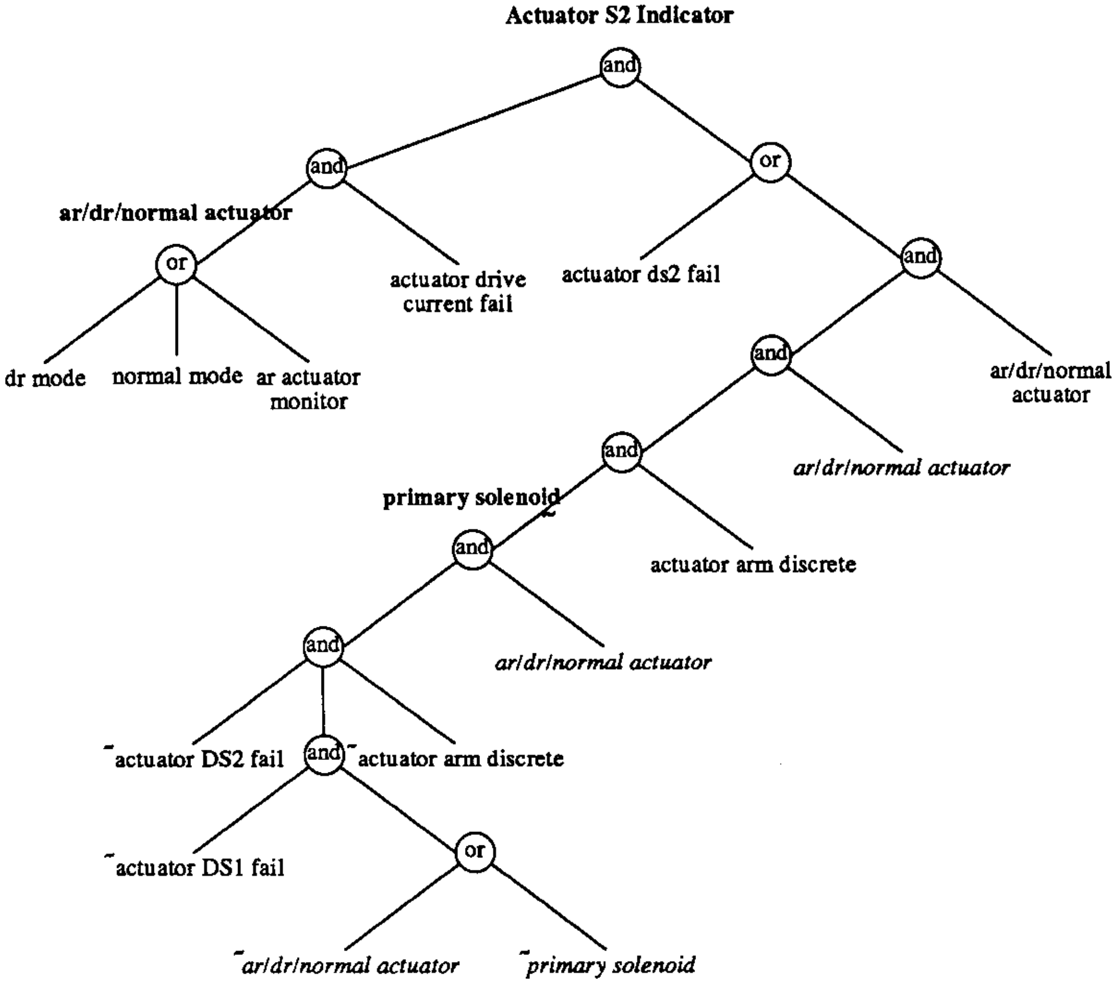


Figure 9.2 - Cyclic Tree Representation

hand part of the graph) is the disjunction of 3 variables. It is not strictly necessary to make this variable an intermediate value. The alternative however, once the tree is reduced to an instance set, is to replace each instance in which the variable occurs with three separate instances, each containing one of the three variables whose disjunction make up *ar/dr/normal actuator.*

The other defined output variable, *primary solenoid*, is an essential feedback variable. As can be seen in the tree representation, *primary solenoid* appears in the subtree which defines *primary solenoid.* Thus this variable is recursive, or cyclic, and it is necessary to have a feedback variable.

From the cyclic tree shown in figure 9.2 it is possible to derive the following three instance sets for the three defined output variables.

$\mathit{dr\ mode} \longrightarrow \mathit{ar/dr/normal\ actuator}$

$\mathit{normal\ mode} \longrightarrow \mathit{ar/dr/normal\ actuator}$

$\mathit{ar\ actuator\ monitor} \longrightarrow \mathit{ar/dr/normal\ actuator}$

****

$\mathit{ar/dr/normal\ actuator} \cdot \overline{\mathit{actuator\ ds1fail}} \cdot \overline{\mathit{actuator\ arm\ discrete}} \cdot$

$\overline{\mathit{actuator\ ds2\ fail}} \cdot \overline{\mathit{primary\ solenoid}} \longrightarrow \mathit{primary\ solenoid}$

****

$\mathit{ar/dr/normal\ actuator} \cdot \mathit{actuator\ ds2\ fail} \cdot \mathit{actuator\ drive\ fail} \longrightarrow$

$\mathit{actuator\ s2\ indicator}$

$\mathit{ar/dr/normal\ actuator} \cdot \mathit{actuator\ drive\ fail} \cdot \mathit{actuator\ arm\ discrete} \longrightarrow$

$\mathit{actuator\ s2\ indicator}$

We now show a possible network configuration for this example. For this case we use the single-valued logic AA2 scheme in which only positive instances are accepted. All states not matching the instances are set to negative by using the don't know comparison explained in chapter 7. Figure 9.3 shows one possible such AA2 configuration for the above three instance sets. Note that some nodes are single input

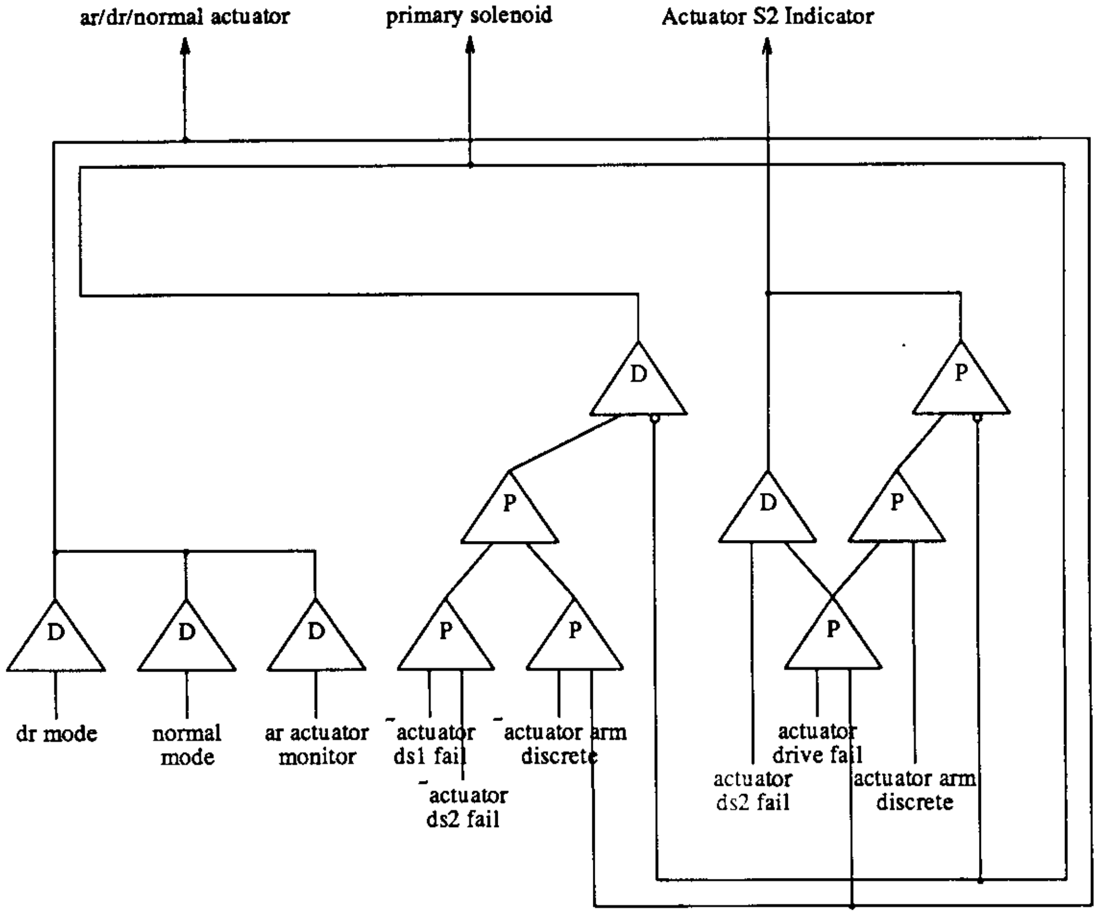


Figure 9.3 - AA2 Network Solution

nodes, since they match the instances having only one antecedent variable.

This example is a simple case of a multi-variate and feedback application. The structure of the final network is order dependent.

Chapter 10

FUTURE DIRECTIONS

In this chapter we briefly discuss some of the future research topics related with this effort. The first section discusses distributed control. Section 2 lists some research topics related to the ASOCS model. The conclusion is the last section.

10.1. Distributed Control

The main concept stressed during this dissertation is that of distributed control, based on self-organization and concurrency. The communication mechanism required for distributed control in this research is that of a *broadcast* capability coupled with local communication between neighboring nodes. The current model is directed to applications defined in terms of propositional rules. However, the present approach to distributed control should be extendible to systems using higher level functions and more complex data structures. A continuing research effort is directed at applying the same approach to such systems.

10.2. Research Extensions

This section briefly lists the immediate research efforts which could be studied as extensions to the current research model.

One possible extension is to relax the constraints on the way in which knowledge is entered to the system. Currently only narrowly defined "instances" can be used as system input. One immediate extension would be to allow simultaneous input of multiple instances. The instances could be pre-processed and ordered in an optimal fashion with respect to the update constraints.

Another current project is the physical implementation of prototype ASOCS systems. There are a number of promising technologies which could be used for each architectural component of ASOCS. For example, optical, wired, or non-wired broadcast mechanisms could be used in an implementation. At any rate, a conventional silicon layout and design appears to be immediately feasible. The government, under auspices of the Rome Air Development Center, has already given a research grant to develop a breadboard ASOCS demonstration.

As it is currently defined, at any given time ASOCS are dedicated devices structured to solve one overall function, even though that function can change in time. In order to allow a single ASOCS implementation to be multiplexed between functions, some method of *context switching* must be devised. This could be as simple as storing the network image (node functions and interconnections) for each different function. The image would be initially created using the adaptation mechanisms. A method of saving and restoring a network image would then be required.

An important addition to the current system is the integration of sequencing and memory. Although the current system can be used as a finite-state machine by the addition of a flip-flop layer at the output binder, it may be useful to allow higher degrees of sequencing. With a memory integrateded with the system some inputs may come from memory rather than the environment, and some outputs could trigger direct memory changes. This mechanism would allow an integration of Von Neumann and concurrent concepts. By controlling the contents of the memory in a time dependent fashion, basic sequential and iterative processes could be accomplished with the concurrent network.

Perhaps the most immediately critical research is to gain a better understanding of the types of real world applications which can be accomplished on a stand-alone or integrated ASOCS system.

10.3. Conclusion

This dissertation has described a novel approach whereby self-organization and concurrency can be used as methods of configuring parallel logic networks which solve an incrementally defined function, and which can do the adaptation in time linear with the depth of the network.

We initially gave a history of this research effort and motivation as to why ASOCS systems could be applicable to real world problems. The *instance* was defined as the atomic knowledge unit in ASOCS and the incremental addition of instances was shown to allow a universal representation of changing propositional logic functions. The basic architecture was then discussed. The primitive mechanisms which make up ASOCS algorithms are those of *instance introduction, node selection, node combination,* and *self-deletion.* Using variations of these mechanisms, three formal adaptive algorithms were presented. These algorithms underwent software simulations and a survey of the garnered statistics was shown.

The ASOCS research has been recognized and continues to be funded by government grants. A joint academic-industrial effort is currently ongoing to build prototype hardware for ASOCS systems.

Bibliography

References

Aker78. Akers, S.B., ''Binary Decision Diagrams,'' *IEEE Transactions on Computers* **C-27**(6) pp. 509-516 (June 1978).

Alex68. Alexander, I. and R.C. Albrow, ''Adaptive Logic Circuits,'' *Computer Journal* **11**(1) pp. 65-71 (May 1968).

Ande69. Anderson, J.L., ''Multiplexers Double as Logic Circuits,'' *Electronics* **42**(22) pp. 100-105 (October 27, 1969).

Arms79. Armstrong, W.W. and J. Gecsei, ''Adaption Algorithms for Binary Tree Networks,'' *IEEE Transactions on Systems, Man and Cybernetics* **SMC-9**(5) pp. 276-285 (May 1979).

Bart81. Barto, A.G., R.S. Sutton, and P.S. Brouwer, ''Associative Search Network: A Reinforcement Learning Associative Memory,'' *Biological Cybernetics* **40** (3) pp. 201-211 (May 1981).

Bloc62. Block, H.D., ''The Perceptron: a Model of Brain Functioning,'' *Reviews of Modern Physics* **34**(1) pp. 123-135 (January 1962).

Bobr78. Bobrowski, L., ''Learning Processes in Multilayer Threshold Nets,'' *Biological Cybernetics* **31** (1) pp. 1-6 (November 1978).

Bran83. Brand, D., ''Redundancy and Don't Cares in Logic Synthesis,'' *IEEE Transactions on Computers* **C-32**(10) pp. 947-952 (October 1983).

Buch84. Buchanan, B.G. and E.H. Shortliffe, *Rule Based Expert Systems, The MYCIN Experiments of the Stanford Heursitic Programming Project,* Addison-Wesley, Reading, Mass. (1984).

Cern79. Cerny, E., D. Mange, and E. Sanchez, ''Synthesis of Minimal Binary Decision Trees,'' *IEEE Transactions on Computers* **C-28**(7) pp. 472-482 (July 1979).

Cris80. Crist, S.C., ''Synthesis of Combinational Logic Using Decomposition and Probability,'' *IEEE Transactions on Computers* **C-29**(11) pp. 1013-1016 (November 1980).

Curt63. Curtis, H.A., ''Generalized Tree Circuit - The Basic Building Block of an Extended Decomposition Theory,'' *Journal of the ACM* **10**(4) pp. 562-581 (October 1963).

Feld81. Feldman, J.A., ''Memory and Change in Connection Networks,'' Technical Report 96, Computer Science Department, University of Rochester, Rochester, NY (December 1981).

Hans63. Hanson, W.H., ''Threshold-Logic Synthesis by Algebraic Methods,'' *IEEE Transactions on Electronic Computers* **EC-12**(4) pp. 401-402 (August 1963).

Haye83. Hayes-Roth, F., D.A. Waterman, and D.B. Lenat, *Building Expert Systems,* Addison-Wesley, Reading, Mass. (1983).

Hayn82. Haynes, L.S., R.L. Lau, D.P. Siewiorek, and D.W. Mizell, ''A Survey of Highly Parallel Computing,'' *COMPUTER* **15**(1) pp. 9-24 (Jan. 1982).

Hell84. Helly, J.J, W.V. Bates, M. Culter, and S. Kelem, ''A Representational Basis for the Development of a Distributed Expert System for Space Shuttle Flight Control,'' NASA Technical Memorandum 58258 (May 1984).

Hill84. Hillis, W.D., ''The Connection Machine: A Computer Architecture Based on Cellular Automata,'' *Physica* **10**(D) pp. 213-228 (1984).

Hint84. Hinton, G.E., T.J. Sejnowski, and D.H. Ackley, ''Boltzmann Machine: Constraint Satisfaction Networks that Learn,'' CMU-CS-84-119, Carnegie-Mellon Univ. (May 1984).

Holl78. Holland, J.H. and J.S. Reitman, ''Cognitive Systems Based on Adaptive Algorithms,'' pp. 313-329 in *Pattern-Directed Inference Systems*, ed. D.A. Waterman and F. Hayes-Roth,Academic Press, New York (1978).

Hu65. Hu, S.-T., *Threshold Logic,* University of California Press, Berkeley, CA (1965).

Kamm79. Kammozev, N.F. and A.N. Sychev, ''Spectral Method of Decomposition of Boolean Functions,'' *Automatic Control and Computer Sciences* **13**(2) pp. 46-50 (1979).

Mart83. Martinez, T.R., ''Convergence Algorithms for Fixed Structured Networks,'' M.S. Comprehensive Exam, Computer Science Department University of California, Los Angeles, CA (December, 1983).

Mins69. Minsky, M. and S. Papert, *Perceptrons, an Introduction to Computational Geometry,* MIT Press, Cambridge, MA (1969).

Moor83. Moore, D.W., ''General Purpose Perceptron,'' Report CSD-830817, Computer Science Department, University of California, Los Angeles, CA (June 1983).

More82. Moret, B.M.E., ''Decision Trees and Diagrams,'' *ACM Computing Surveys* **14**(4) pp. 593-623 (December 1982).

Muro71. Muroga, S., *Threshold Logic and Its Applications,* Wiley-Interscience (1971).

Quin83. Quinlan, J.R., ''Learning Efficient Classification Procedures and their Application to Chess End Games,'' pp. 463-482 in *Machine Learning, An Artificial Intelligence Approach,* ed. R.S. Michalski, J.G. Carbonell, and T.M. Mitchell,Tioga Publishing Co, Palo Alto, CA (1983).

Rose58. Rosenblatt, F., ''The Perceptron: a Probabilistic Model for Information Storage and Organization in the Brain,'' *Psychological Review* **65** (6) pp. 386-408 (November 1958).

Rose62a. Rosenblatt, F., *Principles of Neurodynamics,* Spartan Books, Washington, D.C. (1962).

Rose62b. Rosenblatt, F., ''A Comparison of Several Perceptron Models,'' pp. 463-484 in *Self-Organizing Systems 1962*, ed. M.C. Yovits, G.T. Jacobi, and G.D. Goldstein,Spartan Books, Washington, D.C. (1962).

Roth83. Roth, J.P., ''Logic Simplification,'' *IBM Technical Disclosure Bulletin* **25**(11B) pp. 5968-5969 (April 1983).

Roze79. Rozenfeld, T.K. and V.N. Silayev, ''Boolean Equations and Decomposition of Boolean Functions,'' *Engineering Cybernetics* **17**(1) pp. 85-92 (January-February 1979).

Shan49. Shannon, C.E., ''The Synthesis of Two-Terminal Switching Circuits,'' *Bell System Technical Journal* **28**(1) pp. 59-98 (January 1949).

Sing80. Singh, A.D. and F.G. Gray, ''Reducing Tree Depth in Combinational Universal Modular Trees by Functional Decomposition,'' *12th Annual Southeastern Symposium on System Theory,* pp. 108-112 (May 19-20, 1980).

Stra61. Stram, O.B., ''Arbitrary Boolean Functions of N Variables Realizable in Terms of Threshold Devices,'' *Proceedings of the IRE* **49**(1) pp. 210-220 (January 1961).

Vers83. Verstraete, R.A., D.W. Moore, and J.J. Vidal, ''Parallel Processing with Adaptive Logic: A Modular Architecture that Emulates Perceptrons,'' Unpublished Paper, Computer Science Department, University of California, Los Angeles, CA (August 19, 1983).

Vers86. Verstraete, R.A., ''Assignment of Functional Responsibility in Perceptrons,'' Phd Dissertation, Computer Science Department University of California, Los Angeles, CA (June, 1986).

Vida83. Vidal, J.J., ''Silicon Brains: Whither Neuromimetic Computer Architectures,'' *Proc. IEEE International Conference on Computer Design - VLSI in Computers*, pp. 17-20 (31 October-3 November 1983).

Wate78. Waterman, Donald A. and Frederick Hayes-Roth, *Pattern-Directed Inference Systems,* Academic Press, New York (1978).

Weav75. Weaver, C.S., ''Some Properties of Threshold Logic Unit Pattern Recognition Networks,'' *IEEE Transactions on Computers* **C-24** (3) pp. 290-298 (March 1975).

Widr64. Widrow, B., N.K. Gupta, and S. Maitra, ''Punish/reward: Learning with a Critic in Adaptive Threshold Systems,'' *IEEE Trans. System, Man, and Cybernetics* SMC-3(5) pp. 288-317 (1964).

Yau68. Yau, S.S. and C.K. Tang, ''Universal Logic Circuits and their Modular Realizations,'' *AFIPS Conference Proceedings* **32** pp. 297-305 (1968).

Yau70. Yau, S.S. and C.K. Tang, ''Universal Logic Modules and Their Applications,'' *IEEE Transactions on Computers* **C-19**(2) pp. 141-149 (February 1970).