JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 5, 26–58 (1988)

Adaptive Parallel Logic Networks*

TONY R. MARTINEZ

Computer Science Department, Brigham Young University, Provo, Utah 84602

AND

JACQUES J. VIDAL

Computer Science Department, University of California, Los Angeles, California 90024

Received July 12, 1986

This paper presents a novel class of special purpose processors referred to as ASOCS (adaptive self-organizing concurrent systems). Intended applications include adaptive logic devices, robotics, process control, system malfunction management, and in general, applications of logic reasoning. ASOCS combines massive parallelism with self-organization to attain a distributed mechanism for adaptation. The ASOCS approach is based on an adaptive network composed of many simple computing elements (nodes) which operate in a combinational and asynchronous fashion. Problem specification (*programming*) is obtained by presenting to the system if–then rules expressed as Boolean conjunctions. New rules are added incrementally. In the current model, when conflicts occur, precedence is given to the most recent inputs. With each rule, desired network response is simply presented to the system, following which the network adjusts itself to maintain consistency and parsimony of representation. Data processing and adaptation form two separate phases of operation. During processing, the network acts as a parallel hardware circuit. Control of the adaptive process is distributed among the network nodes and efficiently exploits parallelism. © 1988 Academic Press, Inc.

1. INTRODUCTION

This paper presents a novel class of special purpose processors referred to as ASOCS (adaptive self-organizing concurrent systems). Intended applications

* This research was supported in part by the Aerojet–UCLA Cooperative Research Master Agreement No. D841211 and NASA NAG 2-302.

0743-7315/88 $3.00
Copyright © 1988 by Academic Press, Inc.
All rights of reproduction in any form reserved.

include adaptive logic devices, robotics, process control, system malfunction management, and in general, applications of logic reasoning. ASOCS combines massive parallelism with self-organization to attain a distributed mechanism for adaptation.

It is becoming apparent that the sequential "fetch and execute" model of von Neumann computing is inefficient for many classes of computation. Our research has sought computational paradigms benefiting from parallelism in a *connectionist* fashion. A further goal is to discover systems whose functionality can adapt as the problem is changed. However, parallel adaptive systems are highly complex, are difficult to program, and suffer from slow adaptation mechanisms. Highly dense systems are also limited due to the disparity in I/O bandwidth relative to internal processing capacity (as seen in current VLSI technology where communication on and off chip is an increasingly severe bottleneck). We have found that self-organizing mechanisms solve these problems by avoiding prescriptive programming, allowing substantial parallelism in the adaptation process, and by requiring limited control information from the outside environment.

The basic problem is to design concurrent digital networks able to solve problems defined by propositional logic. Problem specification (*programming*) is obtained by presenting to the system if–then rules expressed as Boolean conjunctions and referred to as *instances*. New rules (instances) are added incrementally to the rule base. In the present model, when conflicts occur, the most recent entry has priority over previously introduced information. Real-world applications using rule-based propositional logic have been forthcoming, including a detailed system for in-flight fault maintenance for the Space Shuttle [2].

The ASOCS 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 among the network nodes. Figure 1 displays a high-level view of the model. Adaptation and data processing form two separate phases of operation. During processing, the network acts as a hardware circuit of Boolean gates, mapping input states to output states. Inputs and outputs of the network are Boolean values. Logic predicates (instances) which have been implicitly "stored" by the connections of the network are processed in data-flow fashion to extract logical inferences.

During adaptation, the network structure and the node functions can change to update the overall network function to match the incrementally entered instances. As new instances 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. The internal configuration is hidden from observation, and

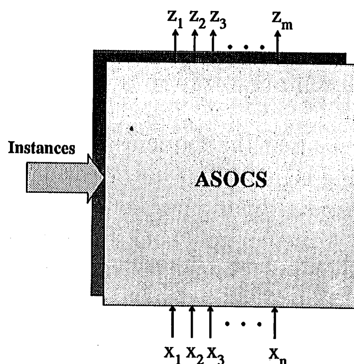

FIG. 1. ASOCS model.

its natural redundancy (many possible correct configurations for a given function) allows for fault tolerant mechanisms.

The control of the adaptive process is distributed among the network nodes 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.

The basic operational sequence for the adaptation process is as follows. A new instance is entered and the rule base is made consistent and minimal. The new instance is then broadcast to all the nodes of the logic network. Nodes within the network contain some memory and are able to locally determine their ability to aid in potential modifications of the network. These nodes are then uniquely selected by a mechanism called *selection waves*. During a *combination* phase, selected nodes recombine and a correct network is obtained. Finally, nodes which are no longer necessary for correct functioning of the network *self-delete.*

Section 2 of this paper briefly describes previous work leading to the ASOCS model. In Section 3, the method of incremental knowledge input is detailed. Section 4 describes the basic architectural model. The fifth section develops the internal algorithm that the system uses for adaptation. The final sections present simulation results and discuss ongoing research.

The ASOCS engine, as discussed in this paper, is a kernel abstract machine around which larger systems could be built. We restrict ourselves to discussion of the basic model. Implementation issues, multi-ASOCS systems, and methods of integrating ASOCS into complete systems, are beyond the scope of this presentation. Three models of ASOCS, each sharing much of the same overall goals, have been proposed [4]. This paper gives a detailed discussion of the first model.

2. Previous Work

This work is part of an effort centered on a reexamination of perceptron-related ideas [6, 9]. However, we have studied the use of digital building blocks, rather than the traditional *threshold unit.* The search for VLSI compatible implementations of parallel pattern recognizers spawned the method of implementing general logic functions with multiplexor circuits known as universal logic modules or ULMs [10]. Two-input ULMs form sufficient building blocks to compose general-purpose, programmable, logic processors.

In [7], Verstraete modifies the ULM scheme in a subtle but central way, by reversing the conventional roles of data and control. In the process, multi-input ULM networks become highly redundant structures. The modified structures were referred to as dynamically programmable logic modules. These networks are created by connecting two-input Boolean modules in tree structures. VLSI layout and testing of this concept have been initiated [5]. More recently, effective algorithms for programming multilayered programmable logic networks have been developed [8].

These approaches contrast with that of ASOCS in that they are directed to the algorithmic assignment of node functions to a network of fixed topology when a global goal function is provided. They are not meant to adapt to a changing goal function. ASOCS on the other hand, which was first introduced by Martinez in a Ph.D. dissertation [3], deals with the synthesis of topologically adaptive logic networks with distributed control. A VLSI design of the ASOCS model is currently in fabrication [1].

3. Instances and Adaptation

3.1. *Instance Set*

The atomic knowledge element of the system is the *instance,* which is a Boolean rule defining what a given output should be when confronted with a given (Boolean) input vector. Each instance is an if–then statement where the antecedent is a conjunction of Boolean variables, and the consequent is a single Boolean variable. Thus, the instance is a propositional production rule. The following are examples of instances:

$$X_1X_2 \rightarrow Z_1$$

$$X_1\bar{X}_2X_3 \rightarrow \bar{Z}_2$$

$$\bar{X}_2\bar{X}_3Z_2 \rightarrow Z_3.$$

We define $\rightarrow$ as the implication operator that states "if the antecedent is true, then the consequent must become true." Note that an output variable could be used in the antecedents of other instances. In this case the variable

is known as a *feedback variable.* In the current model a feedback variable can be treated like any other input variable. For ease of explanation in this paper we will consider ony a single Boolean output variable, Z. The system is easily extended to multiple variables and this is discussed later. The input to the system is a vector of Boolean variables.

An instance does not need to contain (and rarely will) all variables in the input. Instances are *incomplete* by nature. An instance, not matched by the input environment, says nothing about what the output of the system should be. The instance $A \rightarrow Z$ states that Z should be 1 when A is 1 regardless of all other variables. Variables not appearing in an instance are *don't care* variables for that instance.

In the ASOCS model, instances are input to the system *incrementally* and the current collection of instances is called the *instance set.* The instance set is the rule base of the system and as new instances are entered the instance set is modified so as to remain *consistent* and *minimal.* By consistent it is meant that no two instances can specify an opposite output for the same input. Newer instances are given precedence and the contradicted portion of an old instance is deleted from the set. Minimality means that the instance set is stored using a minimal representation.

An input variable occurring in one instance, and occurring in its complemented form in another instance, is a *discriminant variable* with respect to the two instances.

An instance implying a positive output ($Z = 1$) is a *positive* instance, while an instance implying a negated output is a *negative* instance. This is known as the *polarity* of an instance. Two instances with the same polarity are *concordant,* while those with opposite polarities are *discordant.*

We now consider how an instance set is processed in a network of logic devices computing a single output. A logic network *fulfills* the instance set if, when any positive instance is matched, the network outputs a positive value (1), and when any negative instance is matched, it outputs a negated value (0). The next section shows that a positive and negative instance cannot be matched simultaneously in a consistent instance set. If the environment does not match any instance in the set, the network can output either a positive or a negative value.

3.2. *Consistency*

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. For example, the two instances

$$A \rightarrow Z$$

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

contradict each other when both A and B are 1. *Contradictions occur when discordant instances exist without discriminant variables in their antecedents.* Two instances with no discriminant variables can always for some value of the input both be matched. A corollary to the above is that all pairs of instances of opposite polarity must contain at least one discriminant variable with respect to each other.

Let us now consider adaptation of the instance set when a new instance is entered. In this paper we adopt the rule that the new instance has precedence over older instances. Thus, when a new instance contradicts old instances, the old instances (or those portions of the old instances which are contradicted) are removed from the instance set. In a complete system, other rules for instance set modification could be supported (i.e., verify new instances causing large modifications, etc.).

There are two basic ways by which the instance set is modified due to a contradiction. First, when the variables of a new instance (NI) are a subset or equal to those of a discordant old instance (OI), the OI is deleted from the IS. For example,

$$ABC \rightarrow \bar{Z} \qquad \text{(OI)}$$

$$AB \rightarrow Z \qquad \text{(NI).}$$

Second, when there is no discriminant variable between the NI and the discordant OI and the OI is not a subset of the NI, the OI is replaced by one or more modifications of itself. The modification includes all of the OI which is not contradicted by the NI. Thus, the added instances contain the OI concatenated with one discriminant variable of the NI. For example,

$$AB \rightarrow Z \qquad \text{(OI)}$$

$$ACD \rightarrow \bar{Z} \qquad \text{(NI)}$$

will be replaced by

$$AB\bar{C} \rightarrow Z \qquad \text{(OI modification)}$$

$$AB\bar{D} \rightarrow Z \qquad \text{(OI modification)}$$

$$ACD \rightarrow \bar{Z} \qquad \text{(NI).}$$

One of the above two types of modification will occur with each discordant OI in the set which does not contain a discriminant variable with respect to the NI.

When considering the logic network, it should be noted that only the NI can cause a network change to be necessary. Modified instances added to the set in the manner of the second example are already fulfilled by the network, since the root instance from which they are constructed was previously fulfilled.

3.3. *Minimization*

Minimization can occur only between concordant instances. Minimization is based on repeated application of the following three Boolean identities:

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

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 functionally equivalent sets having the same number of instances with differing variable combinations.

Thus, we define an instance set as *minimal* if no two concordant instances can be equivalently represented by one instance, or by two instances with fewer variables.

The environment can match more than one instance, as long as the matched instances are concordant. For example, $A \rightarrow Z$ and $B \rightarrow Z$ 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.

In terms of correct functioning of a logic network (i.e., that it fulfill the instance set), it is not necessary that the instance set be minimal. Minimality can reduce costs in terms of memory and number of necessary nodes. However, making the instance set minimal requires more time, so this time/space tradeoff is a decision left to the designer. In this paper, we assume that the set is made minimal.

4. ASOCS Architecture

The *adaptive self-organizing concurrent system* (ASOCS) is made up of Boolean inputs and outputs with an input channel provided for entering instances (Fig. 1). There is no logical constraint on the total or relative number of input and output variables. Figure 2 shows the ASOCS internal structure. The two main components are the *adaption unit* (AU) and the *logic network.* During execution mode, only the logic network is active. The input data flows asynchronously through the network with only propagation delays. During adaptation mode, the other parts of the system become active. The AU and the logic network are connected by a *broadcast bus.* (Broadcast bus is a name for an abstract component performing the broadcast mechanism, which could be implemented as a wired bus, optical broadcast, message passing between nodes, etc. Names given in this paper are not meant to predefine a specific implementation methodology.) The AU can *broadcast* to the entire logic network by placing a message on the broadcast bus, but it cannot address a

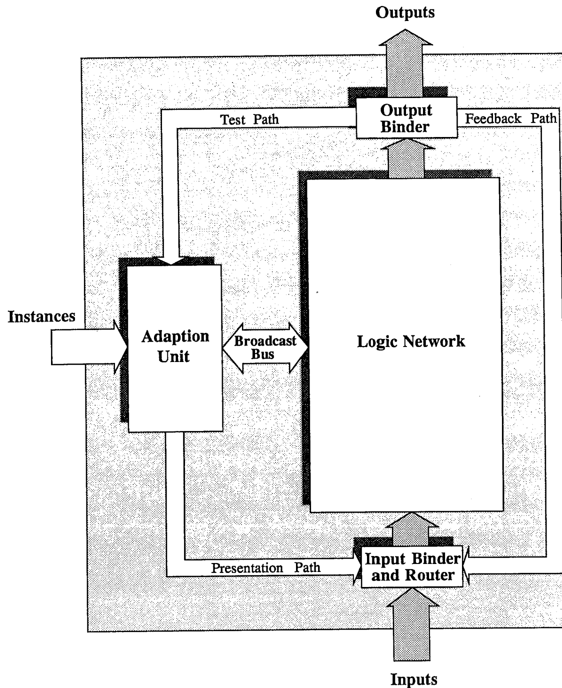

FIG. 2. Overall system structure.

specific node in 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.

The AU can feed test data into the network on the *presentation path* and through the *input router,* which is controlled by the AU. During adaptation, the input router gets its input from the AU. During execution, the input router passes Boolean inputs from the environment into the network.

It is also necessary to bind variables, in a flexible manner, to data 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 can also monitor the outputs of the network through the *test path.*

4.1. *Adaption Unit*

The adaption unit (AU) guides the logic network through adaptation by broadcasting commands to network nodes. This allows all nodes within the

network to work cooperatively by simultaneously executing procedures triggered by the command.

Instances are input incrementally to the adaption unit. The instance set is maintained consistent and minimal, as described in previous sections, in the AU and is stored in a data structure called the *instance table* (IT). The instance table may be structured as a table with three columns. The first column stores all positive instances and is called the *Positive* (P) column. The second column contains the negative instances and is called the *Negative* (N) column. The third column contains a one-bit discrimination flag and is labeled with (D). This column is used during the adaptation phase and is discussed later. The ordering of the instances is inconsequential and there is no relation between the positive and negative instances that appear in the same row.

Initially each cell in the instance table is empty (represented by a "—"). 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, 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 Fig. 3.

4.2. *Network Node*

A single node within the logic network is represented in Fig. 4. Each node in the network has two basic parts: The *control unit* and a dyadic *dynamic programmable logic module* (DPLM). During execution mode the DPLM is the only active element, and the network functions like a hardwired logic circuit.

During adaptation, the control unit can 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. Each network

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

FIG. 3. Instance table.

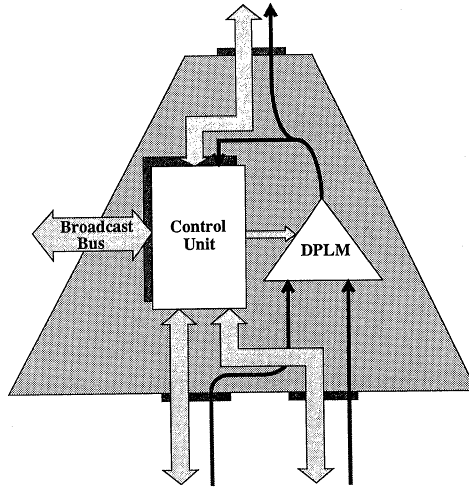

FIG. 4. Single network node.

node has identical structure, but differs from other nodes in the contents of its control unit memory, and the current function of its corresponding DPLM.

4.2.1. *Dynamic programmable logic module.* The DPLM, as used in this model, is a two-input single-output programmable logic gate. During the processing phase, the DPLM functions like a normal Boolean gate. In the adaptation phase, an input or output variable can have any one of three values which will be called *positive, negative,* and *don't know.* These three values are represented as "1," "0," and "?." Thus, the DPLM is a three-state device.

For this model, it is only essential that the DPLM be programmable to 8 of the possible 16 Boolean functions of two inputs. These are the AND and OR functions with all permutations of variable inversions as shown in Table I.

The three state functions are defined in terms of the eight Boolean functions which the DPLM can perform. Table II shows the truth table of the three-state DPLM for the AND, OR, and NEGATION functions. The truth table for any other function can be extrapolated from these three.

4.2.2. *Node control unit.* The node control unit contains a small amount of memory, and has the ability to communicate with the nodes to which it is directly connected. The control unit is only active during the adaptation phase of the ASOCS model.

The control unit can send commands on a bidirectional path, which connects it to its immediate neighbors. Each node receives inputs from exactly two *child* nodes, and sends output to one or more *parents.* The two nodes which input to a parent node are called *sibling nodes* with respect to each other.

TABLE I
BOOLEAN FUNCTIONS

x_1x_2 00	x_1x_2 01	x_1x_2 10	x_1x_2 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	$\bar{x}_1 + \bar{x}_2$

Each node control unit in the network maintains a *node table* (NT) in its local memory. The purpose of the NT is to store the value which its DPLM outputs (1, 0, or ?) for each instance in the instance set when that instance is matched by the environment.

(In the following we assume a specific structuring of the tables for purpose of discussion. However, other functionally equivalent implementations could be used.) The position of each instance in the NT (i.e., its indexing) is that of the same instance stored in the instance table of the AU. Thus, the third positive cell of every NT corresponds to the instance in the third positive cell of the instance table. When a new instance is presented to the network, all nodes allocate the next free space in their node table to the new instance.

Consequently, in the NT there is a four-state cell corresponding to each cell of the IT. The four states are represented by "1," "0," "?," and "—." During the adaptation phase, the node table is updated. Thus, if a cell contains a 1, then that node outputs a 1 for all states of the environment which match the corresponding instance.

TABLE II
THREE-STATE TRUTH TABLES

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
?	?	?	?	?

Figure 5 shows an example of a NT which corresponds to the instance table shown in Fig. 3.

Thus, when a command corresponding to a given instance is broadcast to the network, only an index value is needed to specify the proper cell for each node. For example, if the third positive instance in the instance table were to be deleted, a global message "Delete third positive instance" could be sent to the network. Each node would then set the third cell of its positive column to "—" (empty).

4.2.3. *Instance discrimination.* We now examine how a node *discriminates* between discordant instances. It is never necessary to discriminate between concordant instances, since they cannot contradict each other. The information stored in the node table of each node indicates how that node can be used for discrimination. If a node outputs one level for all states matching a particular positive instance, and the opposite level for all states matching a particular negative instance, then that node discriminates between those two instances. For example, the node table of Fig. 5 shows that the node always outputs a 1 when the environment matches the positive instance in the first row, and a 0 when the environment matches the negative instance represented in the same row. Thus, this particular node discriminates between these two instances. In this case, the node discriminates the first positive instance from the first, second, and fifth negative instances. It also discriminates the positive instance in the third column from the negative instances in the fourth and sixth columns, and so on.

The fact that a given node outputs a 1 does not mean that a positive instance in its node table with the entry "1" has been matched, but it does mean that the negative instances with entries of "0" have not been matched. If a cell in the node table 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.

4.2.4. *Node status.* The discriminant function of a node discussed above shows that a node can be in different states. The node control unit, by looking

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

FIG. 5. Node table.

at its node table, can categorize and mark itself as being in one of the following four states.

A *discriminant node* is a node which discriminates at least one positive instance from one negative instance. In the node table there must be at least one asserted cell in the positive column, and at least one cell with the opposite level in the negative column. Figure 6a is an example of a discriminant node.

A *nondiscriminant node* is a node which does not discriminate at least one positive instance from one negative instance. The node table of a nondiscriminant node does not contain an asserted cell in the positive column and a cell with the opposite polarity in the negative column (Fig. 6b).

A *complete discriminant node* is one which discriminates every positive instance from every negative instance. In the node table all positive cells are asserted at the same polarity, and all negative cells are asserted with the opposite polarity (Fig. 6c). The top node of the network must always be a complete discriminant node in order for the network to fulfill the instance set.

A *one-sided discriminant node* is a discriminant node which always asserts one value for either all positive or all negative instances, and the opposite value for at least one instance in the discordant column. The one-sided discriminant node discriminates at least one instance from all discordant instances and is the type of node built to discriminate a new instance from all old discordant instances (Fig. 6d).

5. Adaptation Algorithm

In this section we discuss the adaptation algorithm for the present ASOCS structure. There is always a single top node for the output variable which is a complete discriminant node. 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

a)

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

b)

P	N	D
1	1	
?	?	
1	?	
—	1	
?	?	
1	1	
1	—	

c)

P	N	D
0	1	
0	1	
0	1	
—	1	
0	1	
0	1	
0	—	

d)

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

FIG. 6. Normal, non- , complete, and one-sided discriminant nodes.

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 this new node (a one-sided discriminant 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. The steps needed to complete this process are discussed in the next six subsections. They are:

1. *Instance set maintenance,* whereby the instance is made consistent and minimal after receipt of the new instance.

2. *Instance presentation,* which causes the broadcast of the new instance to the network, whereupon nodes can add a new value to their node tables.

3. *Node selection,* where each node calculates its ability to aid in discrimination of the new instance and the "best" discrimination nodes are selected to participate in the modification process.

4. *New node addition,* optionally adds new nodes to the network, if those available are not sufficient to completely discriminate the new instance.

5. *Node combination,* when selected and new nodes combine to form a new top node which is a complete discriminant for the entire instance set.

6. *Self-deletion,* where all nodes no longer necessary for correct functioning of the network are removed.

5.1. *Instance Set Maintenance*

The algorithmic method used to maintain the instance set is not critical to the implementation and is not discussed here. Instance set maintenance was discussed in Section 3. 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 lists be empty, in which case the modification cycle is complete for the given NI. The add-list can only be empty when the NI can be deleted, i.e., when the current instance set already fulfills it.

For example, let us assume that the current instance set contains the single instance $AB \rightarrow Z$. Next, $C \rightarrow \bar{Z}$ is input. The old instance $AB \rightarrow Z$ is now contradicted and will be replaced by $AB\bar{C} \rightarrow Z$. Thus, the delete-list will contain $AB \rightarrow Z$ and the add-list will contain $AB\bar{C} \rightarrow Z$ and $C \rightarrow \bar{Z}$.

The index of any deleted instance is broadcast to the network. Each node

places an empty token in the corresponding cell. The instance is also deleted from the instance table in the AU.

5.2. *Instance 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. Each node can then independently detect what it currently outputs for that instance. The node can then use this information to evaluate its ability to be involved 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 any instance being presented. The adaption unit sets the network input through the input router. 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 the network output.

Assume, for instance, that there are currently eight bound input variables entering the network from the environment: $X_1 - X_8$. The AU presents the instance $X_1\bar{X}_3X_7 \rightarrow Z$ to the network. The input router asserts the variables X_1 and X_7 to 1 and the variable X_3 to 0. The other five variables are set to don't know. These data then flow combinatorially through the network data paths (Fig. 7).

The NI is added to the instance table in the AU. The AU enters the NI in the first empty cell in the column corresponding to its polarity. A global presentation command is sent to the network that contains the polarity (positive or negative) of the NI and the index of the presented instance. Each control unit then detects the output of its particular DPLM and stores the value in the indexed cell. Once presented, the AU can then read the output variable Z at the top node, which is either 1, 0, or ?. If the new instance polarity and the Z output match, then the network requires no adaptation because it already fulfills the new instance for all states of the environment. If the output were discordant or a ?, network adaptation would have to take place.

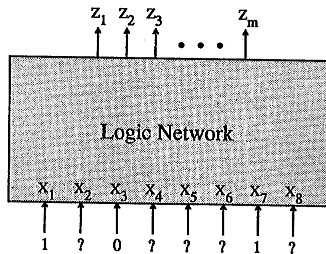

FIG. 7. Instance presentation.

Without three-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-j} states of the environment would have to be shown to the network, where n is the number of bound variables in the environment, and j is the number of variables in the instance.

5.3. *Node Selection*

If the network must be modified, some applicable nodes must be selected to be part of the modification. The method used is referred to as *selection waves*.

When a global selection command is given, each node calculates a locally derived value representing its ability to discriminate the new instance from old instances, called the *discriminant count*. This process is shown by example. Assume a node in the network with the node table shown in Fig. 8 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 value is placed in the cell during presentation.

At the beginning of the selection process, each node sets the values in its discriminate column to 0. The node in Fig. 8 discriminates the NI from the positive instances in the third, sixth, and seventh cells of the positive column. Thus, the node discriminates the NI from three old instances. This value is the discriminant count for the 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 assertion opposite to the value in the cell representing the NI, and which have a 0 in the discriminate column of the same row. A node outputting a don't know for the NI always has a discriminant count of 0 since it cannot discriminate any instances from the NI.

After calculation of its own discriminant count, a node waits to receive the

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

FIG. 8. Node table after new instance presentation.

discriminant counts of its two child nodes. The node then chooses the maximum discriminant count 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,

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

Ties can be decided arbitrarily. After setting its selection state, each node passes the maximum value up to its parents. When the upward data flow is complete, the top node has received the maximum value existing in the network.

Once the upward wave is completed, a downward flow may optionally be initiated. A binary token is passed down to obtain the selection of a single 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 downward wave. The total time necessary to select a node in the selection wave is proportional to two times the depth of the network. Thus, selection time is $O(d)$ and $\approx O(log(n))$, where d is the depth of the network and n is the number of nodes.

An example of a selection wave is given below. Figure 9 shows a logical network where each node has calculated its own discriminant count. 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 arbitrarily decide to set itself to either *right* or *left.* Assume that the top node sets itself to *right.* It then passes the active token (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 has been selected.

Once a node is selected, it sets itself as a *growth node,* and it will be combined with other growth nodes during the subsequent *combination* phase. At this point the selected node puts a 1 in the discriminate column in each row containing an old instance which it can discriminate from the new instance. For the current example node, the discriminate column would appear as in Fig. 10. Since a unique node is selected during a selection wave, it can then

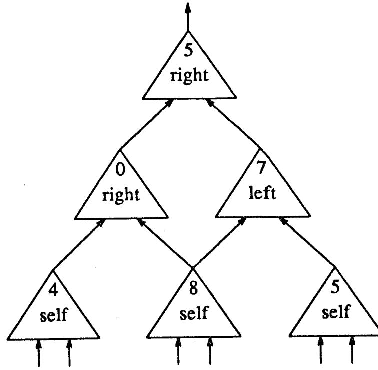

FIG. 9. Network with deduced scores.

place information on the broadcast bus without possibility of contention. The node places the contents of its discriminate column, which is called the *discriminant vector,* on the broadcast bus. All other nodes in the network then do a logical OR of the discriminant vector on the broadcast bus with their own discriminant vectors. After each selection wave, all nodes have the exact same discriminant vector.

The adaption unit also reads the broadcast bus and performs the same operation with its own discriminant vector in the instance table. Any 0's remaining in the common 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, but cells are not counted if they have a 1 in the discriminate column. Thus, only those instances which have not yet been discriminated can be counted toward the new discriminant count of a node.

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

FIG. 10. Discriminate column setting.

Another node is then selected. The AU checks the maximum discriminant count of the network at the output of the top node after the upward wave. As long as the count is greater than 0, a downward 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 upward wave is 0 and there are still one or more zeros in the discriminant vector (instances which have not been discriminated), then new nodes must be added.

5.4. *New Node Addition*

If after the selection phase there are still old instances which cannot be discriminated by nodes currently within the network, then *new node addition* takes place. If there are still nondiscriminated instances, the AU will have a 0 in the discriminant column across from each nondiscriminated instance. New nodes are added to the network as follows. For each nondiscriminated instance in the instance table, one discriminant variable, occurring in itself and the new instance, is chosen. New nodes are then allocated with inputs taken from the set of selected discriminant variables.

Assume, for example, that after the selection process the instance table appears as in Fig. 11, where the NI is the negative instance in the last row. The possible discriminant variables which can be used are the complement of the variables in the NI: $\bar{A}$, B, D, and E. It is sufficient to choose one of these variables from each of the nondiscriminated instances. We know that each of these discordant instances contains 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: $A\bar{B}D$ and $\bar{A}E\bar{F}$. From the first instance we can use the variable

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

FIG. 11. Instance table after selection.

D and from the second instance we could use either $\bar{A}$ or E. If a discriminating variable is shared between the two nondiscriminated instances, then one variable is sufficient to discriminate both instances. Thus, a set of discriminant variables, where at least one variable is found in each nondiscriminated instance, is sufficient to build the new nodes.

The new node must assert either 1 or 0 when the discriminant variables are both matched by the current state of the environment, and the complement when either is not matched. This node can then be combined, during *combination,* with the other growth nodes to create a node which discriminates all discordant instances from the new instance.

Assume that D and $\bar{A}$ are chosen. Since there are two variables, one new node will be added. For two inputs this can always be done by setting the DPLM to either the AND or the OR function (Fig. 12).

The node table of the new node must then be initialized to correct values. The AU calculates the positive and negative columns for the node table of the new node, since the inputs to the new node are literal input variables, as stored in the instance table. The AU then broadcasts this information to the node. The new node(s) is then set as a growth node, and it is ready for the combination phase.

5.5. *Node Combination*

At this point sufficient growth nodes have been selected to make creation of a complete discriminant node possible. *Node combination* is the process by which growth nodes combine such that a complete discriminant node is formed. The method used to pair up growth nodes for combination is implementation dependent and not discussed here. 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 (created) for each pair of growth nodes.
   b. The function and the memory of the new node are set by the child nodes, and the new node is set as a growth node.

2. The last created growth node (a one-sided discriminant node) combines with the old top node.

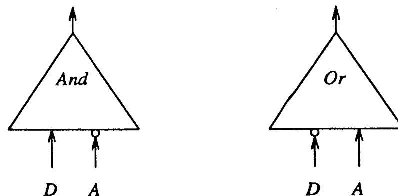

FIG. 12. Possible new node implementation.

3. 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 order.

Every growth node in the network outputs an asserted value of either polarity 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. 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 to be either a positive or a negative growth node. Except for the top node, it does not matter which growth polarity is chosen. The information is passed to the new parent by its children. Table III shows the possible functions which a new parent node can receive, depending on whether the children nodes are positive or negative growth nodes. The same functions are used, as shown in the table, regardless of whether the NI is a positive or a negative instance. The new node will discriminate the union of the instances discriminated by the two child nodes.

After the function of the new node is set, its node table must be loaded by its children nodes. This is done by applying the new function of the parent node to the cell values of the children cells, and then putting the output into the new node table. Once the table is filled, the node sets itself as a growth node, and can then be combined with some other growth node. These combinations of nodes can take place independently and concurrently. For n original growth nodes in the network, exactly $n - 1$ combinations take place, adding $n - 1$ new nodes to the network.

TABLE III

FUNCTION SETTINGS FOR NEW 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

The following is an example of the combination of two growth nodes. We continue to assume for the following examples that the NI is the negative instance in the last row. Figure 13 shows the node tables of the two growth nodes which are about to be combined. 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 III. Figure 14 shows the table of the new parent node after combination. Now only the first positive instance remains nondiscriminated. (The discriminate column was filled for clarity although it would not actually be updated during combination.) The number of instances discriminated by the NI cell always increases as nodes are combined.

When all of the growth nodes have been combined a *one-sided discriminant node* will be created which discriminates the NI from all of the old discordant instances. The one-sided discriminant node is either a positive or a negative growth node, and it has a form as shown in the left table of Fig. 15.

This one-sided discriminant node is then combined with the former top node, shown in the right table of Fig. 15, to form a new complete discriminant node. The function setting of the created node is different than that used for the previous growth nodes. The top node, which was a complete discriminant node before presentation, will now be a one-sided discriminant node 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 complement of what it should be. 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 other positive instances from all negative instances. By contrast, 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 polarity sensitive. There are four ways in which the nodes can combine, depending on whether the new one-sided discriminant

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

FIG. 13. Left and right child tables.

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

FIG. 14. New parent node with DPLM function $x_1 \cdot \bar{x}_2$.

is a positive or a negative growth node, and whether the NI is positive or negative (Table IV).

To explain this intuitively, assume the old top node and new one-sided discriminant (NOSD) as shown in Figure 15. If the output of the NOSD is a 0, then we know nothing, in terms of the NOSD, about what the network output 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. Conversely, 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 positive NOSD growth node, the function is $\bar{x}_1 \cdot x_2$ as shown in Table IV. At this point the network again has a complete discriminant as the top node, and it fulfills the instance set.

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

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

FIG. 15. New one-sided discriminant node and old top node.

TABLE IV
NEW TOP NODE FUNCTION

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	$x_1 + x_2$

5.6. *Self-Deletion*

After a correct network is fashioned, the *self-deletion* command is broadcast. There are three basic types of self-deletion, explained briefly in the following sections.

5.6.1. *Complete discriminant deletion.* When a node discovers from its node table that it is a complete discriminant node, all nodes above it can be deleted and it can become the new top node. A command is sent from the complete discriminant node to its parents telling them to self-delete. The parents in turn, send the same command to their parents and to the child which did not initiate the deletion command. This continues recursively until only the new top node and its descendants remain in the network.

5.6.2. *Nondiscriminant deletion.* A node which discovers, by observing its node table, that it is a nondiscriminant node, can self-delete. The node also sends delete commands to both of its child nodes. If a child has more than one parent, then the link to the nondiscriminant node is removed. If the nondiscriminant node was its only parent, then the child self-deletes and forwards the deletion command to its two children. This continues recursively, deleting the subtree rooted at the nondiscriminant node. The parent of the nondiscriminant node is also deleted and the sibling of the nondiscriminant node will connect to the grandparent of the nondiscriminant node.

5.6.3. *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 that is not necessary to the overall fulfilling of the instance set because other nodes in the network 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 diverse and are not covered here.

5.7. *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 each modified old instance. These modified instances are

placed in the add-list due to either contradiction or minimization. However, although these instances are in the same add-list as the NI, the amount of effort in the modification cycle will usually be less.

An add-list instance created by contradiction always has gained one more variable. The network will always fulfill the modified instance since it is a subset of what the network already fulfills. Thus, the modification cycle does not proceed beyond the presentation phase, since modification is unnecessary. The only action is to update the node tables within the already extant nodes.

There is one way by which minimization can cause an instance to be placed in the add-list. This happens when there is one discriminant variable between the new instance and a concordant old instance, and the old instance contains all the variables of the new instance plus at least one variable not contained in the new instance. For example, assume the old instance $ABC \rightarrow Z$ and the new instance $A\bar{B} \rightarrow Z$. The old instance would be minimized to $AC \rightarrow Z$. The modified instance contains one less variable than before. After presentation, the cell corresponding to this modified instance in the top node of the network could become a don't know. In that case the top node ceases to be a complete discriminant node and the system would continue through the other phases of adaptation. This is known as a *modification iteration.*

This iterative process can be avoided in two basic ways. First, even though the top node can contain a don't know value in a cell after such a presentation, it still correctly fulfills the instance set. Extra memory can be used to flag this situation and no extra processing will be necessary. The other approach is not to minimize in this case. This would not affect the system's ability to fulfill the instance set; it would only cause the instance set to be slightly larger than optimal. Either of these methods would ensure that the adaptation time could be kept within a bounded value linear with the depth of the network.

6. Example

In this section we give an example of the adaptation algorithm at work. We start with a null instance set and begin adding instances and building the network. 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 arbitrarily be chosen.

We will again be concerned with one output variable Z. The first instance input to the system is $A\bar{B}C \rightarrow Z$. Since there are currently no nodes in the network the delete-list is empty and the add-list is just the NI. The system will go to new node addition. If we choose A as the discriminating variable, then one node could be added to the network as shown in Fig. 16. (Only the positive and negative columns are shown and the function of the node is indicated below the node table). The function *Right* signifies that the DPLM

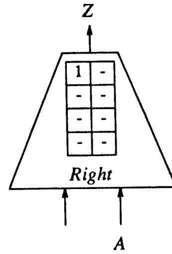

FIG. 16. Initial node.

simply passes its right input to the output. (This function was not mentioned as one of the eight essential, and indeed it could be replaced by the AND function with both inputs set to A). This node fulfills the instance set, but it is more than necessary. In fact, since the node is a nondiscriminant node it will delete itself from the network. Therefore, the system output is simply set to 1 at all times.

Two more positive instances are then added to the system: $B\bar{C}DE \rightarrow Z$ and $\bar{A}BC\bar{E} \rightarrow Z$. 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 would be added to the network to fulfill this IS would still self-delete, since it would be nondiscriminant. At this point the instance set is as follows,

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

$$B\bar{C}DE \rightarrow Z$$

$$\bar{A}BC\bar{E} \rightarrow Z.$$

(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 the instances are shown.)

The next instance is $ABC\bar{D} \rightarrow \bar{Z}$. 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, chosen from the three positive instances, 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 Fig. 17 and the cell representing the NI is shown in bold.

The next instance input is $\bar{A}\bar{C}\bar{D}\bar{E} \rightarrow \bar{Z}$. 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, except for adding a 0 to the cell in the node table representing the NI. At this point the instance set appears as follows,

$$A\bar{B}C \rightarrow Z$$
$$B\bar{C}DE \rightarrow Z$$
$$\bar{A}BC\bar{E} \rightarrow Z$$
$$ABC\bar{D} \rightarrow \bar{Z}$$
$$\bar{A}\bar{C}\bar{D}\bar{E} \rightarrow \bar{Z}.$$

The next instance added is $A\bar{B}CF \rightarrow \bar{Z}$. 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 substituted instance $A\bar{B}C\bar{F} \rightarrow Z$ is put together with the NI in the add-list. The new instance set is

$$A\bar{B}C\bar{F} \rightarrow Z$$
$$B\bar{C}DE \rightarrow Z$$
$$\bar{A}BC\bar{E} \rightarrow Z$$
$$ABC\bar{D} \rightarrow \bar{Z}$$
$$\bar{A}\bar{C}\bar{D}\bar{E} \rightarrow \bar{Z}$$
$$A\bar{B}CF \rightarrow \bar{Z}.$$

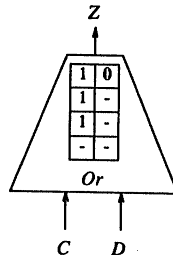

FIG. 17. Modified node.

The index of the deleted instance is then 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 the network must be modified. 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 to discriminate the NI from the three positive OI's. A new node is added, combining these two variables. This new node is the only original growth node and 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 III. The modified network is shown in Fig. 18.

The next instance added is $\bar{A}CDF \rightarrow Z$. This NI cannot 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 Fig. 19.

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 chooses 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 20 shows one possible

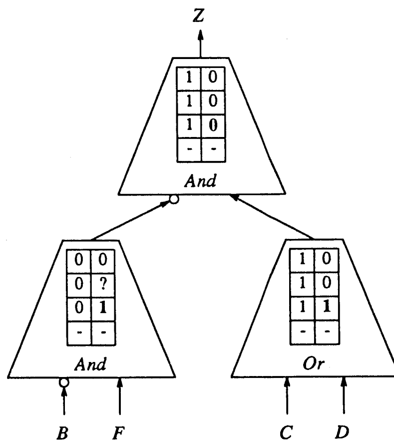

FIG. 18. Modified network.

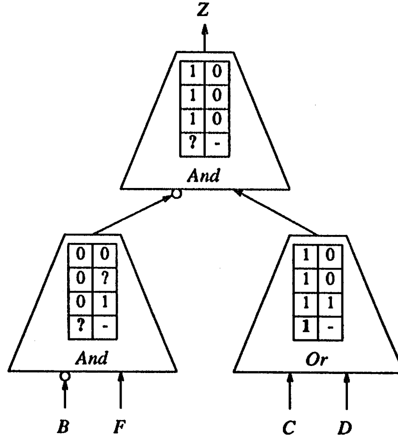

FIG. 19. Modified network.

combination scheme where the first new node was made a negative growth node. The self-deletion phase is entered, but no nodes are deleted, since they are all discriminant nodes.

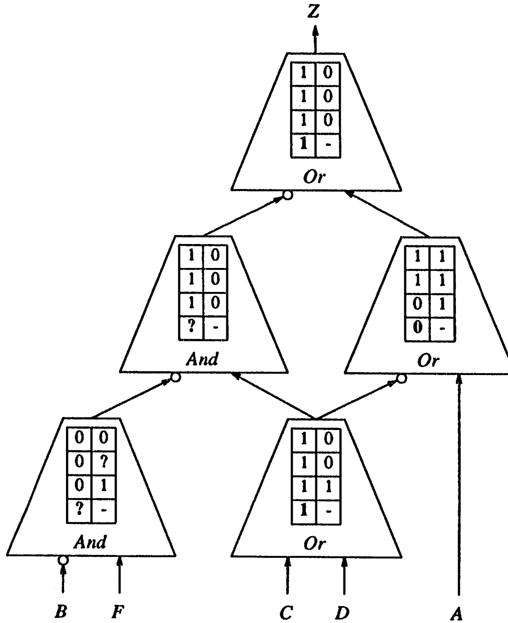

FIG. 20. Modified network.

The last instance input to the system is $AC \rightarrow Z$. This instance causes both minimization and contradiction. It completely contradicts 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 causes minimization with the first positive instance. It is also deleted and the same information is sent to the network. The NI also causes minimization with the last concordant instance. This is a case where a modification iteration would take place since the top node will no longer be a complete discriminant. The NI can then be added to the IS. The new instance set is

$$AC \rightarrow Z$$
$$B\bar{C}DE \rightarrow Z$$
$$\bar{A}BC\bar{E} \rightarrow Z$$
$$AB\bar{C}\bar{D} \rightarrow \bar{Z}$$
$$\bar{A}\bar{C}\bar{D}\bar{E} \rightarrow \bar{Z}$$
$$\bar{A}CDF \rightarrow \bar{Z}.$$

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 Fig. 21. In this case the output from the top node is correct and network modification is unnecessary. The bottom right node has now become a complete discriminant node. During the self-deletion phase everything above it will be recursively deleted. After the network finishes the self-pruning process, the total network fulfilling the current instance set is as shown in Fig. 22.

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 this example there were seven 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.

7. Multiple Outputs

The basic extension needed for multiple outputs is for each node to maintain a separate node table for each output. In this way, a single node can participate in the discrimination of any number of output variables. Commands broadcast by the AU specify the output variable targeted. The node status is also relative to output variables. For instance, if a node becomes nondiscriminant for one output variable, but it is still a discriminant for a different variable, it cannot be deleted.

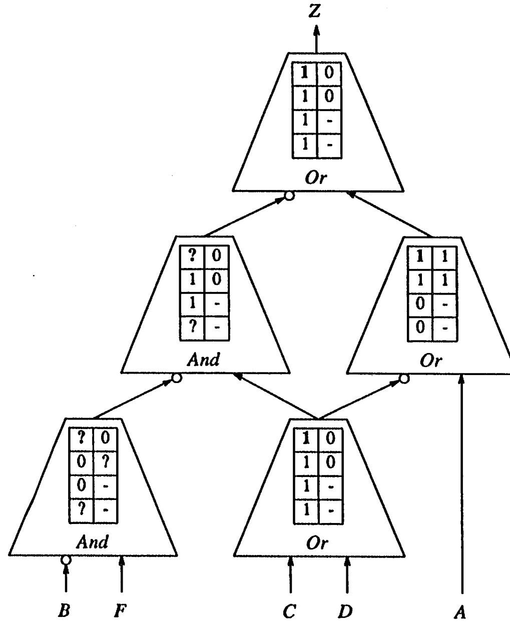

FIG. 21. Modified network.

Instances for each output variable need to be kept in independent instance sets within the AU. Minimization or contradiction does not take place between instances defining different output variables.

8. Simulation Results and Current Research

A software simulation of the described system confirmed the operation of the adaptation algorithm. The average number of nodes in a logic network

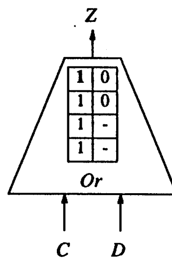

FIG. 22. Final network.

solving an instance set is equal to the number of instances in the set. It is also found that the time necessary to complete an adaptation step is $O(d)$ where d is the depth of the network and $O(log(n))$ where n is the number of network nodes. Since n is approximately equal to the number of instances, adaptation time is also $O(log(i))$ where i is the number of instances. The minimization preprocess in the adaption unit exceeds linear time, but this can be resolved by either decreasing or removing the minimization criteria, while still maintaining a correctly functioning system.

A detailed and formal exposition of the adaptation algorithm is found in [3]. It also discusses other ASOCS issues such as fault tolerance, physical implementation, and advanced architectures. The main drawback of this adaptation algorithm is that the amount of memory required at each node is proportional to the size of the instance set. The memory also grows proportional to the number of output variables in the system.

The aforementioned dissertation describes two more recent adaptation algorithms with improved features over the first. The main improvement is that the newer algorithms have a small fixed memory at each node which does not increase with the size of the instance set or the number of output variables. Both do not require the instance set to be maintained in the adaption unit. Rather, a new instance is simply broadcast to the network and the instance set is maintained implicitly within the logic network.

Work is currently funded and ongoing to develop prototypes and VLSI fabrication of the ASOCS model.

9. Conclusion

In this paper we have proposed an architectural model for adaptive combinational logic which uses concurrency in both the processing and the control stages. Functional specification is incrementally input to the system in the form of propositional rules. The current overall function is maintained as a consistent and minimal instance set. The rules are broadcast one by one to a network of identical nodes which then evaluate their ability to discriminate the new instance from the previous set. Useful nodes are selected and combination takes place without any outside agent knowing the location or structure of any internal node. Nodes can also detect when they are not actively involved in fulfilling the instance set and will subsequently self-delete. The overall adaptation is done in a self-organizing fashion and the time necessary for an adaptation is linear with the depth of the network.

References

1. Chang, J., and Vidal, J. J., Inferencing in hardware. *Proceedings of the MCC–University Research Symposium,* Austin, TX, July 1987.

2. Helly, J. J., Bates, W. V., and Kelem, S., A representational basis for the development of a distributed expert system for space shuttle flight control. NASA Technical Memorandum 58258, May 1984.
3. Martinez, T. R., Adaptive self-organizing logic networks. Ph.D. dissertation, Computer Science Department, University of California, Los Angeles, CA, May 1986.
4. Martinez, T. R., Models of parallel adaptive logic. *Proceedings of the IEEE Conference on System Man and Cybernetics,* October 1987.
5. Moore, D. W., General purpose perceptron. Rep. CSD-830817, Computer Science Department, University of California, Los Angeles, CA, June 1983.
6. Rosenblatt, F., *Principles of Neurodynamics.* Spartan Books, Washington, DC, 1962.
7. Verstraete, R. A., General purpose perceptrons: A Boolean treatment. M.S. thesis, Computer Science Department, University of California, Los Angeles, CA, Dec. 1982.
8. Verstraete, R. A., Assignment of functional responsibility in perceptrons. Ph.D. dissertation, Computer Science Department, University of California, Los Angeles, CA, June 1986.
9. Vidal, J. J., Silicon brains: Whither neuromimetic computer architectures. *Proc. IEEE Conference on Computer Design–VLSI in Computers,* 1983, pp. 17–20.
10. Yau, S. S., and Tang, C. K., Universal logic circuits and their modular realizations. *AFIPS Conference Proceedings,* 1968, Vol. 32, pp. 297–305.