In *Journal of Parallel and Distributed Computing,* vol. **11**, No. 4, pp. 303-313, 1991.

A Self-Adjusting Dynamic Logic Module

Tony R. Martinez and Douglas M. Campbell

Computer Science Department
Brigham Young University
Provo, Utah 84602

Abstract

This paper presents an ASOCS (Adaptive Self-Organizing Concurrent System) model for massively parallel processing of incrementally defined rule systems in such areas as adaptive logic, robotics, logical inference, and dynamic control. An ASOCS is an adaptive network composed of many simple computing elements operating asynchronously and in parallel. This paper focuses on Adaptive Algorithm 2 (AA2) and details its architecture and learning algorithm. AA2 has significant memory and knowledge maintenance advantages over previous ASOCS models. An ASOCS can operate in either a data processing mode or a learning mode. During learning mode, the ASOCS is given a new rule expressed as a boolean conjunction. The AA2 learning algorithm incorporates the new rule in a distributed fashion in a short, bounded time. During data processing mode, the ASOCS acts as a parallel hardware circuit.

Keywords: ASOCS, Self-organization, Neural Networks, Connectionist Computing, Parallel, Rule-based, Machine Learning.

1. Introduction

This paper gives an ASOCS (Adaptive Self-Organizing Concurrent System) architecture [7,10] which guarantees learning for boolean rule based systems in bounded time. It can also be used for learning by examples and handles contradictory rules and/or examples. This particular ASOCS uses Adaptive Algorithm 2 and has significant improvements in both memory requirements and knowledge maintenance over previous ASOCS models [8]. Target applications include rule based systems for logical inference, robotics, adaptive logic, fault-recovery, and real-time dynamic control.

The search for fast and robust computation has increased research in highly parallel systems with both traditional [2,4] and connectionist [5,14] views. Researchers of massively parallel systems have sought speed both during processing and learning (programming). But programming and updating massively parallel systems have tremendous overhead and complexity.

The goal of ASOCS is to train (program) a parallel digital network to solve problems defined by rule based propositional logic. A system is trained (programmed) through the incremental input of rules expressed as conjunctions of boolean variables. Real world applications using rule-based propositional logic are increasing [3].

An ASOCS is an adaptive network of many simple computing elements operating in a parallel, asynchronous fashion. An ASOCS operates in both data learning and data processing modes.

During learning the system reconfigures itself in a distributive manner to accommodate new (and perhaps conflicting) rules. The AA2 learning algorithm guarantees adaptation in $O(log(n))$ time for any new rule. Through its learning process ASOCS can discover critical variables and use these to generalize and classify large input spaces.

During data processing the system acts as a parallel hardware circuit; it asynchronously maps input data to output data in $O(max(d,log(n)))$ time, where d is the maximum depth (longest path) of the network, and n is the number of network nodes, as is typical for hardware circuits.

Initial ASOCS research has focused on three basic learning models labeled Adaptive Algorithm 1, Adaptive Algorithm 2, and Adaptive Algorithm 3. Details for AA1 can be found in [8]; details for AA3 can be found in [6, 11]. These three algorithms vary dramatically, although AA3 shares some important aspects of AA2.

ASOCS arose from reexamining perceptron [13] related ideas. The basic building block, however, is that of digital programmable nodes, an idea spawned by the notion of a universal logic module (ULM) [16]. Verstraete [15] sought methods of programming fixed ULM structures to solve arbitrary boolean mappings. ASOCS departs drastically from these efforts by having a non-passive network which adapts in a self-organizing fashion [6, 9]. ASOCS models offer parallel inference, high speed adaptation, and internal consistency control [6,7,8]. Proof of concept VLSI fabrication of ASOCS devices has been completed [1] and other implementation efforts are currently underway.

Although ASOCS research was initiated with a neural network emphasis, the proposed mechanisms differ extensively from standard neural network paradigms. The authors make no claim to be modelling neural functionality with this model. Rather, distributed, parallel, and self-organizing paradigms are used in order to attain an improved *computational* mechanism, offering speed, fault tolerance, and ease of use. The model also differs from standard AI machine learning techniques, such as the Aq algorithm [12], since AA2 does not do heuristic *search* over an inductive space.

This paper seeks to overview the AA2 processing and learning algorithms. Discussion of specific implementations is beyond that scope and can be found elsewhere [1, 6].

The outline of the paper follows. Section 2 defines the mechanism of knowledge input. Section 3 describes ASOCS during processing mode. Section 4 describes ASOCS during learning mode. Section 5 works in detail a concrete example of ASOCS learning. Section 6 extends the model to multiple outputs. Sections 7 and 8 overview simulation results, comparisons with AA1 and AA3, and current research efforts.

2. Knowledge Input

The atomic knowledge element is the *instance*. Each instance is a (partial) function from a set of boolean variables to a boolean variable. Thus, each instance is a propositional production rule.

Following are examples of instances:

I. A' B' ==> C

II. A B' C ==> A'

III. A' B' ==> C'.

Instance I forces C to become true whenever A and B are false. Instance II forces A to become false whenever A and C are true and B is false. Instance III forces C to become false whenever A and B are false.

Instances I and III are *inconsistent* with each other; when A and B are false, instance I tries to set C to true while instance III tries to set C to false.

When the variables of an instance are not matched, the instance says nothing about the output of the function. (An implementation may set the output variable to true, to false, or even to don't know.) Instances are incomplete and partial functions by definition.

An instance with a non-negated variable on its right hand side is said to be a *positive* instance or to have a positive polarity; an instance with a negated variable on its right hand side is said to be a *negative* instance or to have a negative polarity. Thus, each instance has either negative or positive *polarity*.

Two instances with the same polarity are *concordant* with respect to each other. Two instances with opposite polarity are *discordant* with respect to each other.

An instance which has both a Boolean variable and its complement on its left hand side can never be fulfilled. Therefore, without loss of generality, we require that *no instance has both a variable and its complement on its left hand side*.

Let I_1 and I_2 be two discordant instances and let v be a boolean variable. If v occurs in the left hand side of I_1 and v' occurs in the left hand side of I_2 (or vice versa), then v is a *discriminant variable* for I_1 and I_2. For example, A is a discriminant variable for the pair of instances A B' ==> D and A' C D ==> D', but neither B, C, or D are discriminant variables for the pair.

Let S be a set of instances (an *instance set*). We say an instance set is *consistent* if and only if S does *not* contain any two instances X and Y where X is a positive instance and Y is a negative instance such that 1) X and Y have the same right hand variable and 2) there is a set of boolean values which simultaneously matches the left hand sides of X and Y. Thus, in a (consistent) instance set S, no two instances have a common set of boolean values for which the function defined by S produces inconsistent values. *We require instance sets to be consistent*. The union of a set of instances defines a partial function. It is a (partial) function defined by a set of rules, a functional rule base.

Two fundamental types of inconsistencies exist. The first type is obvious; two instances of opposite polarity have the same left hand variables as in AC ==> B and AC ==> B'. The second type is more subtle. Although the instances AD ==> B and C ==> B' have opposite polarity and different left hand variables, nevertheless when A, D, and C are true, B becomes both true and false.

Fortunately, there is a basic computational test for inconsistency. Its proof is omitted.

Lemma. Two instances are inconsistent if and only if they are of opposite polarities and have no discriminant variable. Two instances having a discriminant variable can never be simultaneously *matched*.

I. A B ==> C

II. B C ==> C'

III. B' C ==> C'

For example, instance I and instance II are inconsistent because there is no discriminant variable. Instance I is consistent with instance III because of the discriminant variable B. Instances II and III cannot be inconsistent because they have the same polarity.

In the ASOCS model, instances are added incrementally. Let NI be a new instance and S be an instance set. Either NI is consistent with S or else there is at least one instance of S with which NI is inconsistent.

An NI 1) may be redundant to information already contained in the instance set or 2) may contain new information.

By definition, an NI is *redundant with respect to an instance set S*, if the partial function defined by the NI is already contained in the partial function defined by S. For example, let the instance set S be

D ==> C

B ==> C

and let the NI be AB ==> C. Clearly, the fact that the NI AB forces C is already contained by the second instance of the original system. Since the NI is redundant, parsimony suggests that we delete it

By definition, an NI *contains new information with respect to an instance set S,* if the partial function defined by the NI modifies the partial function defined by S. If an NI contains new information then it may a) contain no inconsistencies, b) contain inconsistencies with one or more old instances, and c) may cause old instances to become redundant.

An NI adding information, but containing no inconsistencies, follows. Let the instance set S be

A ==> C

B == > C

and let the NI be A' B' ==> C. In this case, the NI tells us to set C to true when A and B are false, an extension of the partial function defined by S.

The principle of parsimony may also apply when new information causes an old instance to become redundant. For example, let the instance set S be

AB ==> C

AC' ==> C

B ==> C

and let the NI be A ==> C. The NI contains new information; it tells us what to do when A is true and B is false, as well as what to do when A is true and C is false. But more than that, the original instances AB ==> C and AC' ==> C are now redundant as they are but special cases of A ==> C. If we can detect such redundancies quickly, then parsimony suggests they be deleted. Parsimony, or minimization, is not essential to the proper functioning of an ASOCS.

If the new instance is inconsistent with an old instance, then we give precedence to the new instance and *remove the contradicted portion of the old instance*.

Theorem 1. Let S1 ==> Z be a new instance and S2 ==> Z' be an old instance. Suppose there is no discriminant variable for the new and old instance. If S1 is a subset of S2, then every part of the old instance contradicts the new instance.

Proof. Since S1 is a subset of S2, every set of variables that realizes S2 (and forces Z to be false by the old instance) extends to variables that realize S1 (which forces Z to be true by the new instance). Therefore, every part of the partial function defined by S1==> Z is a rewrite of S2 ==> Z', that is, the new instance contradicts *everything* for which the old rule stood.

Theorem 2. Let S1 ==> Z be a new instance and S2 ==> Z' be an old instance. Suppose there is no discriminant variable for the new and old instance. Suppose S1 is not a subset of S2. Let S3 be those variables in S1 that are not in S2. Then the part of the old instance which is not contradicted by the new instance is given by the following set of instances {I' S2==>Z': I is in S3}.

Proof. Since S1 is not a subset of S2, there is a variable in S1 that is not in S2. Thus, the set S3 is not empty. Let I belong to S3. The instance S2 ==> Z' is the union of the two rules: I S2 ==>Z' and I' S2==>Z'. Since I belongs to S1 and since there is no discriminant variable for S1 and S2, we can find boolean values for S1, I, and S2 so that all hold, forcing Z to be set inconsistently. On the other hand, S1==> Z and I' S2==>Z' now share the discriminant variable I and are therefore not inconsistent. That is, we may save the I' S2 ==> Z' part of the old instance for each variable in S3. This concludes the proof of the theorem.

To illustrate theorem 2, consider the new instance ACD ==> Z' confronting the old instance AB ==> Z. Since ACD is not a subset of AB we form the variables in the new instance that are not in the old instance, namely C and D. Theorem 2 says to replace AB ==> Z by the pair of instances C'AB ==> Z and D'AB ==> Z.

The addition of discriminant variables to recover the uncontradicted portion of an old instance is called DVA (*discriminant variable addition)*. DVA plays a critical role in reconfiguring a network during the learning algorithm discussed in section 4.

Theorems 1 and 2 allow us to take any consistent instance set and any new instance and produce a new instance set that contains as much of the old instance set as can be saved with the new instance being given precedence. The number of instances in the new instance set may be greater than, less than, or equal to the number of instances in the original instance set (depending on the amount of redundancy or contradiction). In the new instance set, all instances have equal priority and order is again inconsequential.

Instances may come from human intervention or automated mechanisms. Typical learning has more general instances (those with fewer antecedent variables) entered first followed by refinement through more specific instances (those with more antecedent variables).

In the AA2 ASOCS implementation of section 4, the system maintains consistency in a manner invisible to the user. By dynamically modifying the instance set, the system discovers which variables are critical in making decisions. This leads to natural algorithmic *generalization* through critical variables when the system receives novel inputs. In the AA2 ASOCS model, the system does not explicitly store the instance set; instead, the system *stores the information implicitly in a distributed fashion*.

3. The ASOCS AA2 Model

In this section we show how the AA2 ASOCS dynamic parallel network implements a consistent instance set.

We say that an ASOCS implementation *fulfills* a consistent instance set if whenever an input forces the instance set to logically output Z (or Z'), then the implementation physically outputs Z (or Z'). The network output for an input not matching any of the instances can be either a don't know or a generalized output.

Figure 1 gives the architecture of the AA2 ASOCS implementation. In this section we only discuss those parts of the implementation needed for execution. In section 4 we discuss how the implementation represents itself (how it learns).

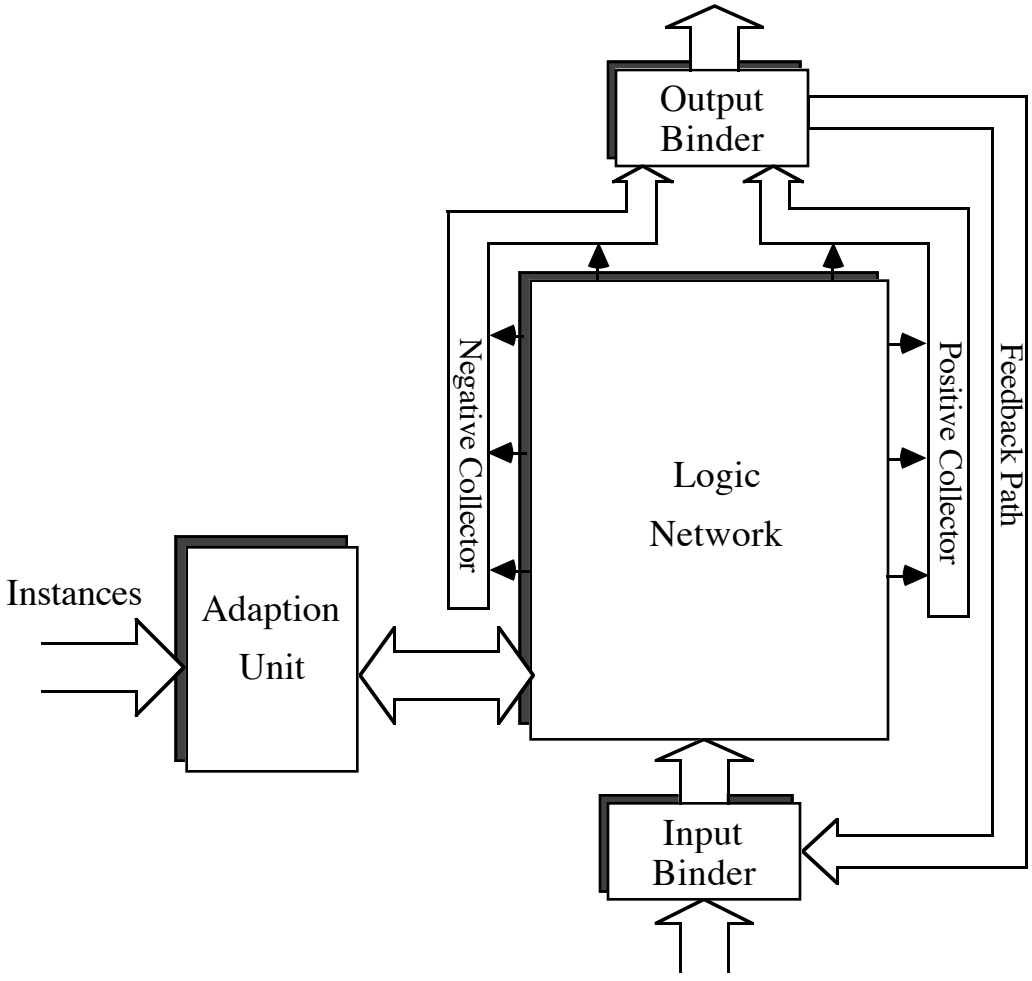

Figure 1 - AA2 Architecture

During execution mode, only the *logic network* is active. Input data flows asynchronously through the network with only propagation delays. The *feedback path* allows the system to use an

output variable as an input variable. The *input* and *output binders* dynamically connect input and output variables to the network. Variables are defined when first seen in a new instance. (If a clocked register is added at the output binder, then the system is similar to a dynamically adaptable finite state machine.)

Each node within the logic network has the structure of figure 2. During execution mode, only the dyadic AND gate is active.

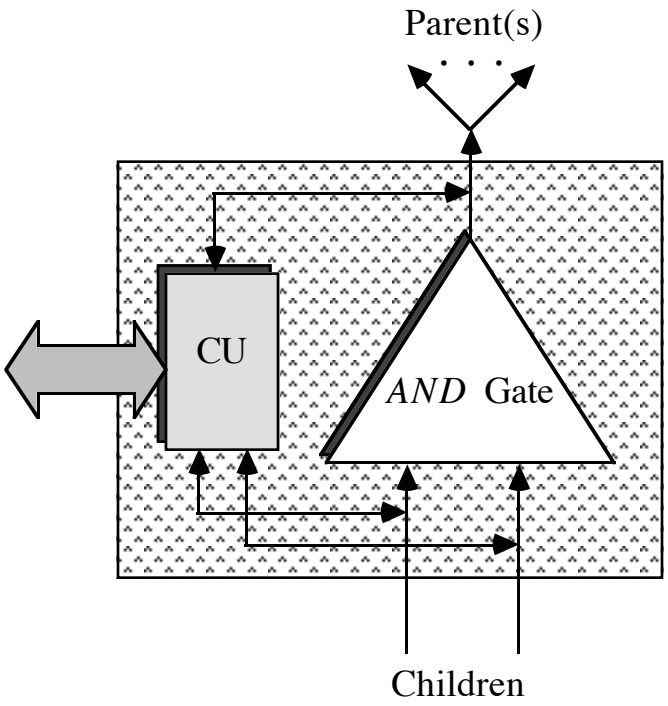

Figure 2 - AA2 Network Node

A node CU (control unit) can send local commands on the links to its immediate neighbors. A node which gives input to another node is known as a *child* node. A node which receives the output of a node is known as a *parent* node. A node may have more than one parent node, but a node has exactly two children. Child nodes of the same parent are said to be *sibling* nodes with respect to each other.

In AA2, unlike AA1 and AA3, there is a network node directly corresponding to each current instance. Since a single node has only two inputs, a node representing an instance with many variables must be built from the conjunction of nodes below it.

A node which corresponds directly to an instance is called a *Dnode (discriminant node)*. A node is a Dnode if and only if it has no parents. A Dnode can be positive (D+) or negative (D-) corresponding to positive and negative instances.

The nodes from which a Dnode receives its inputs are called *Pnodes* (*primitive nodes)*, denoted by the symbol P. A Pnode may be shared by many Dnodes.

Each node CU has memory for a 3-state polarity flag with value: D+, D-, or P (for positive Dnode, negative Dnode,or Pnode). A node N can be thought of as a record with a set of variables V and a 3 state *polarity flag* PF. Thus, N.V refers to the set of variables corresponding to node N, and N.PF is the polarity flag of node N.

Each node CU has memory to store the total variables over which the node does a conjunction; known as its *variable list*. In figure 4 and throughout the paper, a node's variable list appears at the bottom of the node. Although this model assumes a variable list at each node, it is possible to implement functionally equivalent broadcast mechanisms without any variable memory at the node.

One alternate implementation [1] requires a short sequence of presentations [6,8] which are sufficient for a node to determine its relation to the new instance.

The overall structure of the AA2 network is that of a DAG (Directed Acyclic Graph) as in figure 3. In terms of the DAG, a Dnode is a top node (a parentless node), while a Pnode is an internal node (with parent(s)). Each Dnode is the root of a directed tree whose other nodes are Pnodes (and the Pnodes may be shared by other Dnodes).

Figure 3 illustrates a possible AA2 configuration for the following instance set. Each node's variable list is at the bottom of the node. The symbol at the top of the node is the three-state polarity flag.

A B D' E ==> Z'
A B C' D ==> Z
C E ==> Z'

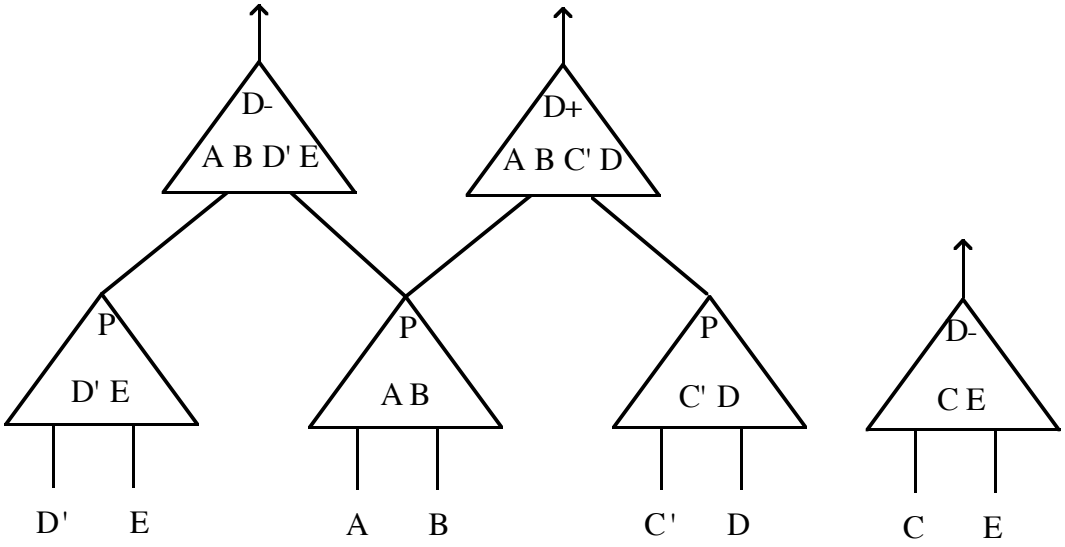

Figure 3 - Example Network Configuration of the above instance set

The output of all positive Dnodes is sent to the *positive* collector; the output of all negative Dnodes is sent to the *negative* collector. Each collector performs a logical *OR* of its received values. A three node structure handles the collector outputs as in figure 4. The middle node is the output node; it outputs Z if the positive collector is active, and outputs Z' if the negative collector is active. If both positive and negative collector are inactive, then the don't know output is active. If both collectors are active, then the network has an error (inconsistency).

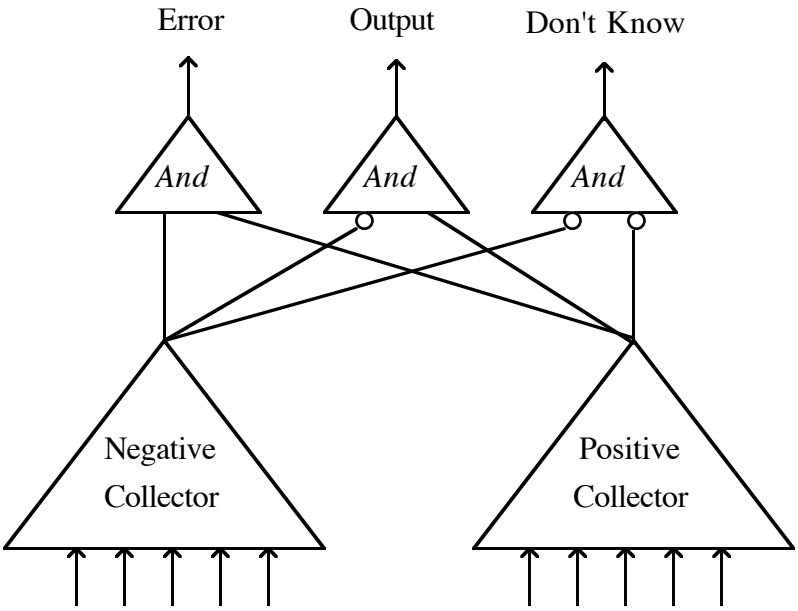

Figure 4 - Collector Structure

AA2 implements a sum-of-products expressions. The products are the instances represented by the Dnodes in the network; each Dnode is equivalent to one instance in the current instance set. The products represented by the Dnodes are summed by the collectors in a fashion similar to the or-planes of a programmable logic array (PLA).

The *maximum depth of the network* is equal to the number of bound input and feedback variables.

4. The AA2 Learning Algorithm

We now explain what it means for a network to learn when a new instance appears. Let the network represent the (consistent) instance set S and let NI be a new instance. We say that a network representing the instance set S' *has learned the NI* if and only if

1) S' represents the NI
2) If I is an instance of S, then S' represents all of I that is not contradicted by NI.
3) If I is an instance of S, then S' represents no part of I that is contradicted by the NI.
4) If I' is an instance of S', then I' is a subrule of either NI or S.
5) S' is consistent.

Thus the network must

a) resolve any contradictions created by the NI (theorem 1),
b) retain all non-contradicted information from the previous instance set (theorem 2),
c) represent the NI itself,
d) not add any other information.

The AA2 learning algorithm has four phases. First, if the NI is already present, then exit the learning algorithm. Second, if the network partially contradicts the NI (theorem 2), then the network must recover the uncontradicted portion. Third, a representation of the NI must be put into the network. Fourth, any portion of the network that is in complete contradiction with the NI (theorem 1) must be deleted, while any part of the network that is redundant, may be deleted.

Communication from/to the AU and the network is done through a broadcast/gather mechanism. Time necessary to broadcast in any balanced hierarchical structure (such as a tree) is log(n) where n is the number of nodes in the tree. Gather is also log(n) in time. Thus the AU can broadcast a command to the network, nodes can respond, and the AU can tell if any node (although it need not know which) fulfills the command, in log(n) time. Details of this communication are implementation dependent.

The simplified AA2 learning algorithm follows:

I. Is the NI already present.
The AU broadcasts the quickexit flag, NI.V, and NI.PF.
Each node executes the quickexit function and reports its result to the AU.
If any node returns true for Quickexit, then exit;

II. Recover any partially contradicted information.
The AU broadcasts the recovery flag, NI.V, and NI.PF.
Each node executes the recovery procedure;

III. Add the NI to the network.
The AU executes Node combination to add the NI to the network;

IV. Delete contradicted and redundant information.
The AU broadcasts the self-deletion flag, NI.V, and NI.PF. Each node executes the self-deletion procedure.

4.1 Node Partitioning

The AA2 learning algorithm makes use of the fact that each NI partitions the set of nodes into one of five blocks: *ProperSubsetNodes, EqualNodes, ProperSuperSetNodes, DiscriminatedNodes,* and *OverlapNodes*. Let N.V denote the variables of a node N and NI.V denote the variables of the new instance NI. The definition of these five blocks follows.

1. N is a *ProperSubsetNode* with respect to the NI if and only if N.V is a subset of NI.V, but N.V is not equal to NI.V.
2. N is an *EqualNode* (with respect to the NI) if and only if N.V is equal to NI.V.
3. N is a *ProperSupersetNode* (with respect to the NI) if and only if N.V is a superset of NI.V, but N.V is not equal to NI.V.
4. N is a *DiscriminatedNode* (with respect to the NI) if and only if there is a variable v such that either
   1) v in N.V and v' in NI.V or
   2) v in NI.V and v' in N.V.
5. N is an *OverlapNode* (with respect to the NI) if and only if N is not a DiscriminatedNode, a ProperSubsetNode, an EqualNode, or a ProperSupersetNode.

Lemma. The ProperSubsetNodes, the EqualNodes, the ProperSupersetNodes, the DiscriminatedNodes, and the OverlapNodes partition the set of nodes, that is, every node belongs to at least one and to at most one of the above five sets.

Proof. To see that every node belongs to at least one of the five sets it suffices to note that OverlapNodes acts as a catch 22.

To see that a node belongs to at most one of the five sets, we need only show the five sets are mutually disjoint. By definition, the ProperSubsetNodes, the EqualNodes, the ProperSupersetNodes, and the OverlapNodes are mutually disjoint. It therefore suffices to show that a DiscriminatedNode can not be a ProperSubsetNode, an EqualNode, or a ProperSupersetNode. Let N be a DiscriminatedNode. Without loss of generality, let v belong to N.V and v' belong to NI.V. Since a variable list can not include both a variable and its complement, the presence of v in N.V and v' in NI.V proves that N.V can not be a subset of N.V, a superset of N.V, or equal to N.V. This concludes the proof of the lemma.

Each node can execute boolean predicates with parameters N and NI to test for ProperSubsetNode, EqualNode, ProperSupersetNode, DiscriminatedNode, and OverlapNode. Each

node can execute boolean predicates with parameters N and NI to test for concordancy and discordancy.

4.2 QuickExit

The network already fulfills an NI if the NI is already in the network, or if the NI is a special case of a more general pattern. In such cases the learning algorithm can stop. Specifically, if the network contains a concordant Dnode whose variables are a subset of or the same as the variables of the NI, then the network already fulfills the NI and remains consistent without further action or reconfiguration.

```
function Quickexit: boolean;
begin        {each node sends its Quickexit result to the AU}
   Quickexit:=
        (N.PF = Dnode)          and
         Concordant(N,NI)    and
        (ProperSubsetNode(N, NI) or EqualNode(N, NI) )
 end;
```

4.3 Recovery

An NI may contradict only part of a Dnode; for example, suppose the NI is ABCD ==> Z and the Dnode is AC==> Z'. Then by theorem 2, the part of AC==>Z' that is not contradicted by the NI is given by applying DVA and replacing AC==> Z' by ACB'==>Z' and ACD'==>Z'.

An NI causes a partial contradiction if and only if the network contains a discordant Dnode which is either a ProperSubsetNode or an OverlapNode with respect to the NI. The partial contradiction is recovered by applying DVA as described in theorem 2.

```
Procedure Recovery;
begin
   if   (N.PF =  Dnode) and
        (discordant(N,NI) ) and
        ( ProperSubsetNode(N,NI) or OverLapNode(N,NI) ) then
           DVA;
end;     {recovery procedure}

Procedure DVA;
begin
   let S = NI.V - N.V;
   For each v in S do begin
       allocate(node);
       if NI is positive then
            node.PF       := D-
       else
            node.PF       := D+ ;
       node.child1    := N;
       node.child2    := complement of v
    N.PF                  := Pnode;
end;     {DVA procedure}
```

4.4 Node Combination

Node Combination is the mechanism by which a new Dnode is created which represents the NI. Node combination may also require creation of Pnodes essential to building up the variable list of the new Dnode. Where possible, node combination uses Pnodes already existent in the network in

building the new Dnode for sake of parsimony. The particular physical implementation of node combination is highly dependent on the technology and architecture of a physical ASOCS implementation and is not covered here. Examples are found in [1,6]. A formal discussion of node combination is found in appendix I.

4.5 Self-deletion

An NI may completely contradict an instance of the network as indicated by theorem 1. For example, let the NI be AC==> Z. Then the instances ABC ==> Z' and AC ==> Z' are completely contradicted. Nothing can be salvaged from an instance represented in the network as a discordant, ProperSupersetNode Dnode. Nor can anything be salvaged from a discordant, EqualNode Dnode.

In addition, some nodes are redundant with respect to an NI. For example, let the NI be AC==> Z. Then the instances ABC==>Z and ACEF ==> Z are redundant with respect to the NI. A concordant, ProperSupersetNode Dnode is redundant with respect to an NI.

If a network contains instances that are completely contradicted by the NI, then they must be removed. If a network contains instances that are redundant with respect to the NI, then they can be removed. These removals are done by the self deletion procedure.

```
Procedure SelfDeletion
Begin
   If (N.PF = Dnode) and Discordant(N,NI) and EqualNode(N,NI)
   or
   (N.PF = Dnode)  and ProperSupersetNode(N,NI) then
   send the self-delete command to each child node and return N to the free pool.

   If N is a parentless child node, then
   send the self-delete command to each child node and return N to the free pool.
        {Note that this causes children of deleted nodes to recursively delete unless they have
other parents.}
end;
```

4.7 Node Reductions

In this simplified AA2 learning algorithm we have omitted optional procedures that may lead to a smaller network. For example, we could add a node reduction strategy based on *one-difference* nodes. A node N is a one-difference node with respect to the NI if and only if

1. NI and N are concordant,
2. NI and N share exactly one discriminant variable,
3) NI.V is a subset of or equal to N.V.

For example, the node AB'C ==> Z is a one-difference node with respect to the NI ABC ==> Z. In this case, the node and the NI can be reduced to a single instance AC==> Z. Likewise, the node AB'C ==> Z is a one-difference node with respect to the NI AB ==> Z. In this second case, the node and the NI are equivalent to the (shorter) node pair AC==>Z and AB ==> Z.

One-difference nodes arise naturally from conservative, small corrections to past knowledge.

Additional node reduction mechanisms can be found in [6].

5. Example

In this section we present an example that illustrates different parts of the AA2 learning algorithm. We assume that the network begins without any nodes and that the following six instances appear in the given order.

I. ABC'D ==> Z

II. ABD'E ==> Z'
III. ABC'D ==> Z'
IV. ABC' ==> Z'
V. ABC'E' ==> Z'
VI. ABE' ==> Z

I. ABC'D ==> Z. Since the network begins without nodes, the quickexit test fails. Since there are no nodes, there is no recovery. The AU adds the positive Dnode ABC'D to the network (1). Since there are no other nodes, there is no self-deletion. The network ends as in figure 9.

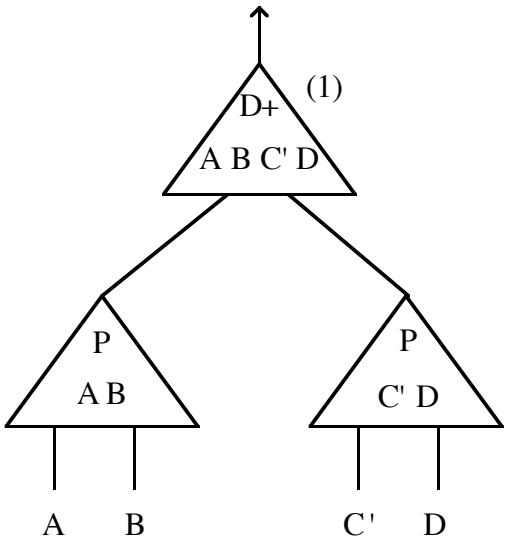

Figure 9 - Network after first node combination

II. ABD'E ==>Z'. The Dnode (1) is a discordant DiscriminatedNode with respect to the NI. Therefore, quickexit fails. There is no recovery. The AU adds the negative Dnode ABD'E to the network (2). There is no self deletion. The network ends as in figure 10.

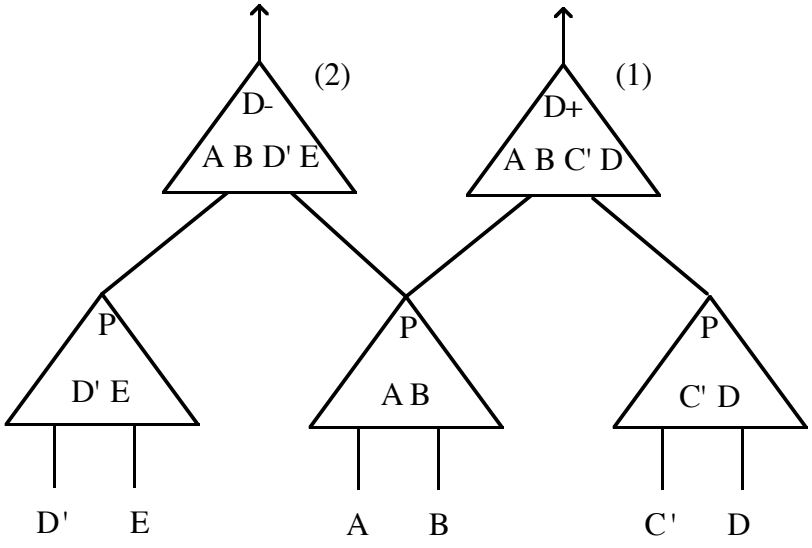

Figure 10 - Modified Network

III. ABC'D ==>Z'. The Dnode (1) is a discordant Equalnode with respect to the NI, and the Dnode (2) is a concordant DiscriminatedNode with respect to the NI. Therefore, quickexit fails. There is no recovery. The AU adds the negative Dnode ABC'D to the network (3) as in figure 11. The network then self-deletes (1), the positive Dnode ABC'D. The network ends as in figure 12.

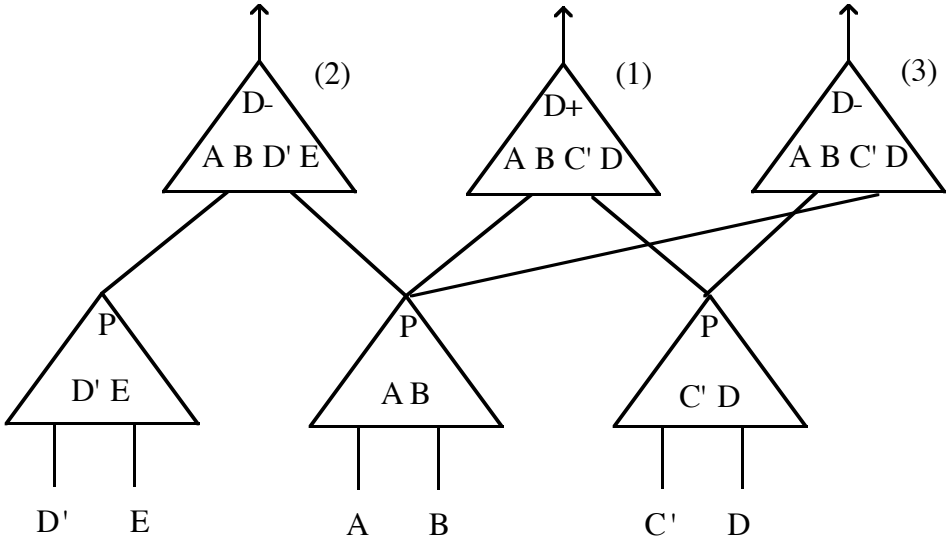

Figure 11 - NI Added

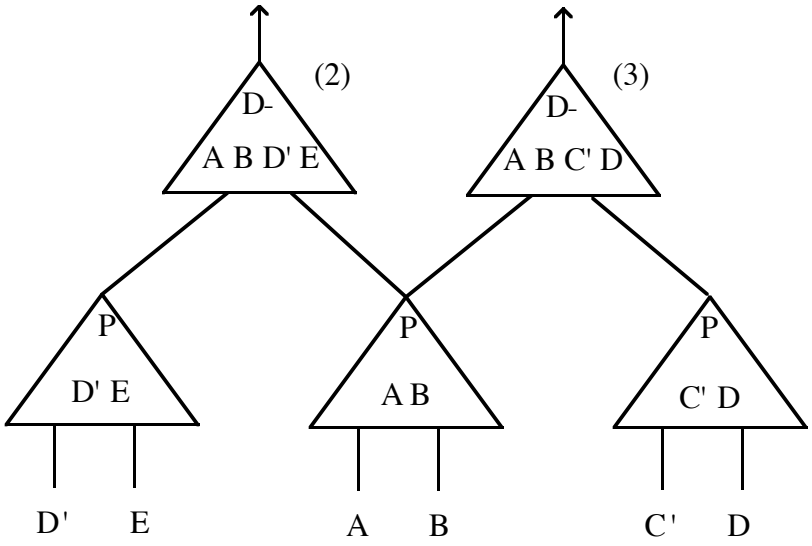

Figure 12 - After Self-Deletion

IV. ABC' ==> Z'. The Dnode (2) is a concordant OverlapNode with respect to the NI and the Dnode (3) is a concordant ProperSupersetNode with respect to the NI. Therefore, quickexit fails. There is no recovery. The AU adds the negative Dnode ABC' to the network (4) as in figure 13. The network then self-deletes (3), a concordant ProperSupersetnode. The network ends as in figure 14.

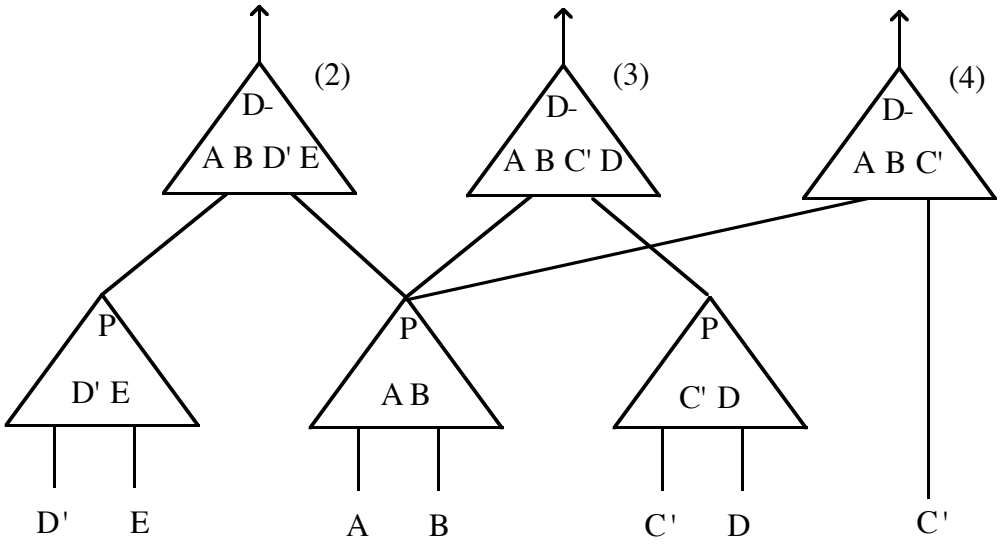

Figure 13 - After Node Combination

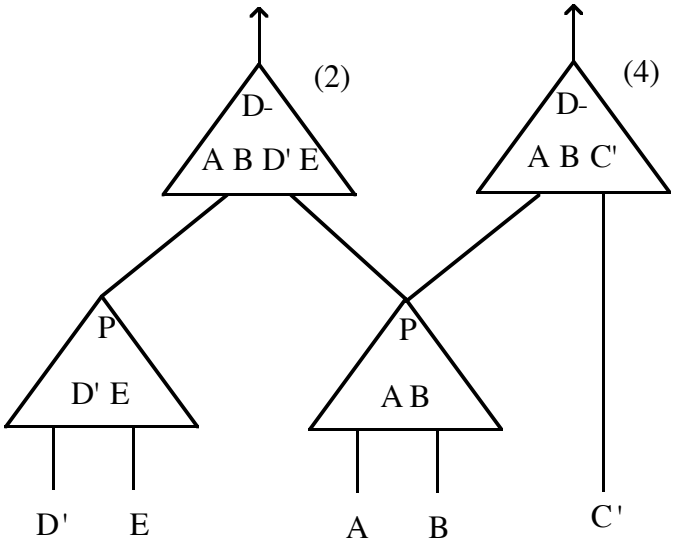

Figure 14 - After Self-deletion

V. ABC'E' ==> Z'. The Dnode (3) is a concordant ProperSubsetNodes with respect to the NI. Therefore, quickexit succeeds and no modification is made to the network. It remains as in figure 14.

VI. ABE' ==> Z. The Dnode (2) is a discordant Discriminated node with respect to the NI and the Dnode (4) is a discordant Overlap node with respect to the NI. Therefore, quickexit fails. There is recovery. DVA occurs for Dnode (4) with respect to E' = NI.L - N.L. The network adds the negative D node ABC'E (6) as in figure 15. The AU adds the positive Dnode ABE' to the network (7) as in figure 16. There is no self-deletion and the network ends as in figure 16.

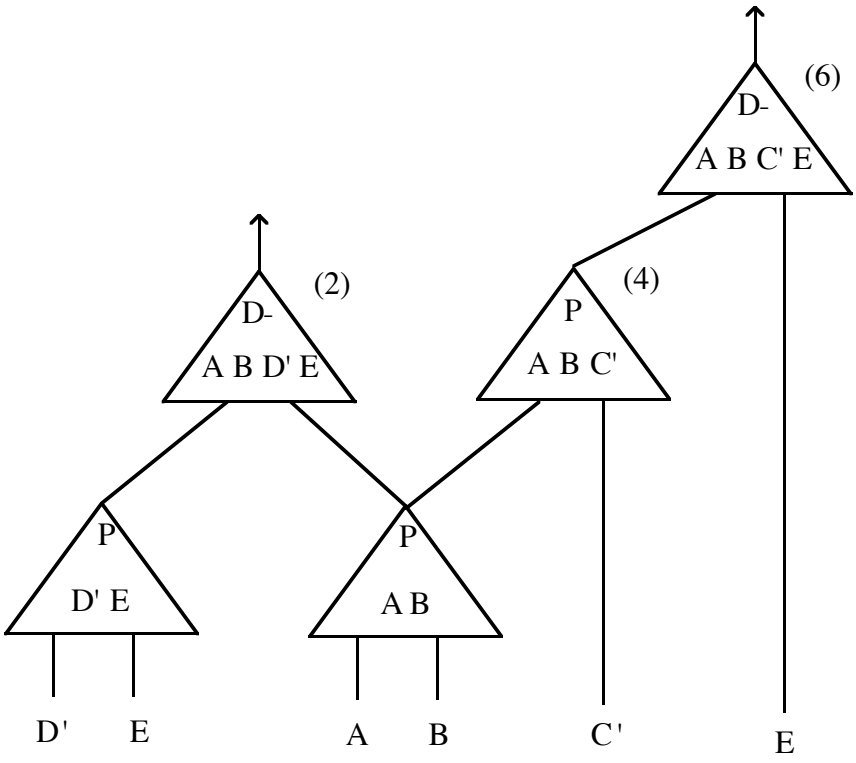

Figure 15 - After DVA

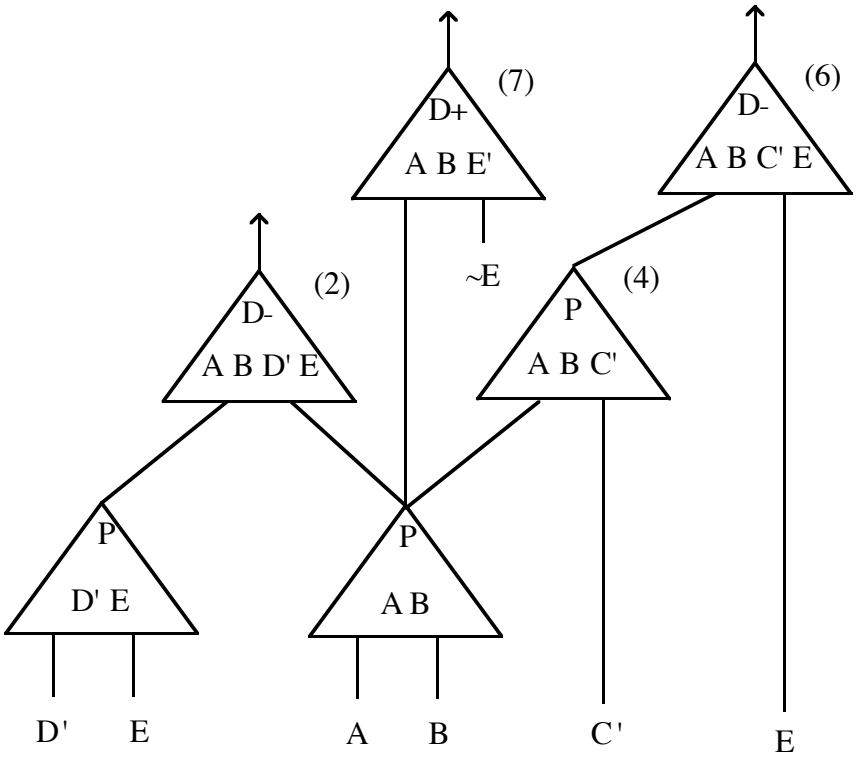

Figure 16 - Final Network

The reader probably noted an inefficiency when instance IV of section 5 was added. Namely, NodeCombination built a negative Dnode for ABC' although the positive Dnode ABC' was already present. The system then turned around and deleted the positive Dnode ABC'. *Polarity inversion* prevents this inefficiency. Namely, before NodeCombination is executed, the AU checks for the existence of a discordant EqualNode. If one is found, then instead of building a new node, the AU simply tells the old node to change its polarity flag and to switch its connection to the opposite collector.

6. Multiple Outputs

Having discussed the architecture and learning and deletion algorithm for a single output, we extend to multiple outputs. We first discuss the changes to the architecture and then the changes to the algorithms.

Architecturally we make two changes: 1) a node has an additional polarity flag for each output variable for which it is a Dnode; 2) each output variable has its own positive and negative collector. A physical node could be a positive Dnode for one variable, a negative Dnode for another variable, and a Pnode for a third variable.

The AA2 algorithm is changed so that whenever the AU makes a broadcast, it broadcasts the instance's variables, the instance's polarity, *and* the the instance's output variable name. Each node must check whether it is a Dnode for the output variable of the new instance. During DVA, a new Dnode must also set the corresponding polarity flag for the current output variable. When a Dnode self-deletes it simply sets its current output variable flag to nil. If a Dnode is not responsible for any other output variables, then a true node deletion is possible.

The real power and efficiency of the system becomes evident with multiple outputs; sharing of network nodes can be done with multiple outputs, leading to more efficient use of hardware.

7. Simulation, Implementation, and Comparison with Other Algorithms

Extensive software simulation has been carried out on the AA2 learning algorithm. These results are reported in [6]. For a single output variable, there are an average of two nodes per instance in the system. This statistic appears very promising. As discussed in section 6, this ratio should improve with the number of output variables in a system. Initial VLSI design and fabrication has already been accomplished for the AA2 algorithm [1], and testing is underway.

The depth (longest path) in an AA2 network is bounded by the number of defined input variables. Depth in a network is the restricting factor on speed, both during processing and adaptation. Therefore, given the size of the input space we can give an upper bound on speed for different applications.

AA2 improves over AA1 in two significant ways: distributed instance set consistency and memory requirements. AA1 requires the storage and maintenance of a consistent instance set to be processed in the adaption unit, while AA2 simply broadcasts the NI to the network where storage and consistency are done in a distributed manner throughout the network. AA1 requires node memory proportional to both the instance set size and the number of output variables, while AA2 requires at most 2 bits (polarity) per output variable per node. AA2 also has the advantage of being able to signal when an environment state does not match any instance in the instance set.

AA3 [6,11], the topic of a forthcoming article, is similar in many respects to AA2. Its simpler mechanism is based on an adaptive binary decision tree. It does not require flexible interconnections between network nodes. However, unlike AA2, the AA3 mechanism cannot signal when an environmental state does not match any instance in the instance set.

8. Conclusion and Future Work

This paper has introduced a model of computation to fulfill the ASOCS (adaptive self-organizing concurrent systems) mechanism. ASOCS features include guaranteed mapping of arbitrary boolean

functions, bounded linear learning time, stability of previously learned patterns, and extraction of critical features from a large environmental input. The AA2 system implements the ASOCS model with negligible memory requirements, with distributed storage, and with distributed maintenance of the knowledge base. Current research thrusts include 1) extension of ASOCS data and functional primitives to higher-order and more complex structures, 2) investigation and enhancement of the ASOCS feedback mechanisms to allow temporal dynamics and sequential algorithms, 3) preparation of new and improved models in terms of speed, programmability, and fault tolerance, together with studies of physical realization constraints, and 4) integration of ASOCS paradigms with other von Neumann and connectionist mechanisms.

Appendix I - Node Combination

This section deals with node combination via *n-variable DAGs*.

Let L be the set of 2n *literals* x1, x1', x2, x2', ... , xn, xn'. An *n-variable labeled DAG* is any directed acyclic graph such that

1. it has 2n source nodes;
2. the source nodes are labeled x1, x1', x2, x2', ... , xn, xn';
3. any non-source node is labeled as the union of its source vertices (see figure A1).

Parentless non-source nodes are called *D*nodes*. Non-source nodes with parents are called *P*nodes*.

Let N be any such n-variable labeled DAG. Let S be any non-empty subset of L. Suppose that S has the property that if v is in S, then v' is not is S. To be specific, let S={x1,x3,x4',x5',x6} and let N be as in figure A1.

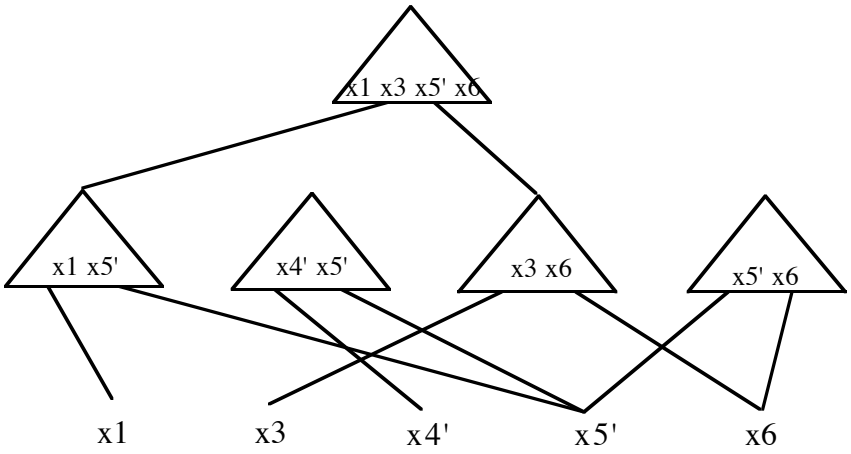

Figure A1 - n-variable labeled DAG

To add to N a D*node with label S, we could always combine |S|-1 source nodes in a trivial manner as in figure A2. However, such a construction fails to use any P*nodes that are already present.

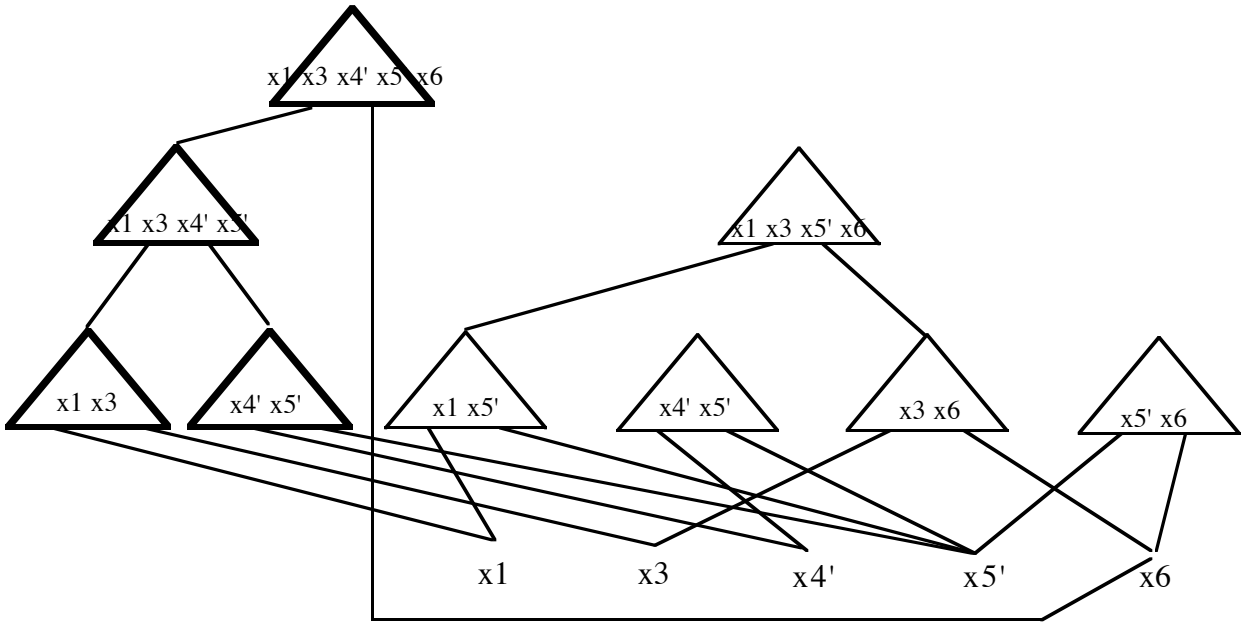

Figure A2 - Trivial addition of S

To use resources more efficiently we could identify all P*nodes of N that are subsets of S and combine them as in figure A3. But such a construction uses more P*nodes than necessary.

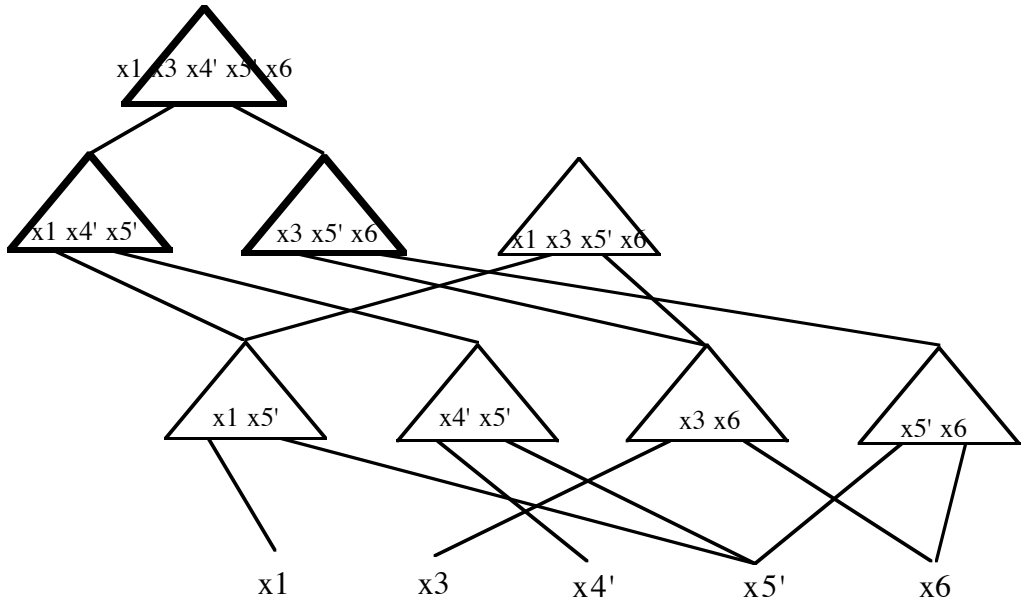

Figure A3 - P*node sharing with combination

A third mechanism would be to chose the P*nodes recursively. We could chose the first P*node which has the most variables as a subset of S. We could then form S1 by deleting from S the first P*nodes variables. We could then choose the second P*node as the P*node which has the most variables as a subset of S1, and so on. Any missing variables in the final Sn could be supplemented from the source nodes as in figure A4.

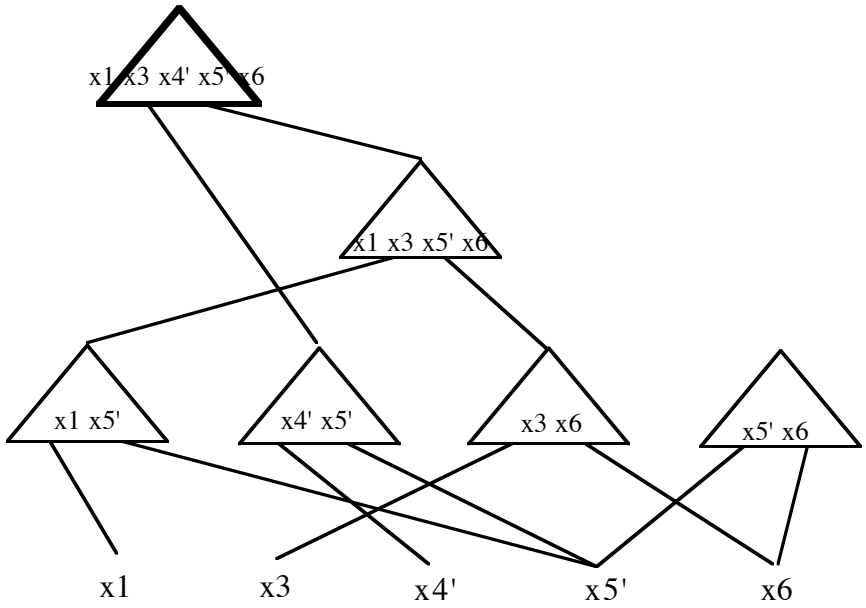

Figure A4 - Recursive P*node sharing

Clearly, many implementation strategies of node combination are possible. Suffice it to say, node combination is possible.

9. Bibliography

1. Chang, J. and J. J. Vidal, Inferencing in Hardware, *Proceedings of the MCC-University Research Symposium*, Austin, TX, (July 1987).

2. Haynes, L.S. , R. Lau, D. Siewiorek, and D. Mizell, A Survey of Highly Parallel Computing, *Computer,* vol. 15, No. 1, pp. 9-24, (1982).

3. 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.

4. Hwang, K., Advanced Parallel Processing with Supercomputer Architectures, *Proceedings of the IEEE*, Vol. 75, No. 10, pp. 1348-1379, (1987).

5. Kohonen, T., *Self-organization and associative memory*, Springer Verlag, New York, (1984).

6. Martinez, T. R., *Adaptive Self-Organizing Logic Networks*, Ph.D. Dissertation, Technical Report - CSD 860093, University of California, Los Angeles, CA (May 1986).

7. Martinez T. R., Models of Parallel Adaptive Logic, *Proceedings of the 1987 IEEE Systems Man and Cybernetics Conference*, pp. 290-296, (October, 1987).

8. Martinez, T. R. and J. J. Vidal, Adaptive Parallel Logic Networks, *Journal of Parallel and Distributed Computing*, Vol. **5**, No. 1, pp. 26-58, (1988).

9. Martinez, T. R., Digital Neural Networks, *Proceedings of the 1988 IEEE Systems Man and Cybernetics Conference*, pp. 681-684, (August, 1988).

10. Martinez, T. R., Adaptive Self-Organizing Concurrent Systems, in *Progress in Neural Networks*, Ablex Publishing, 1989.

11. Martinez, T. R. and Campbell, D.M., A Self-Organizing Binary Decision Tree for Arbitrary Functions, Submitted.

12. Michalski, R., J. Carbonell, and T. Mitchell, Eds., *Machine Learning*, Tioga Press, Palo Alto, CA, 1983.

13. Rosenblatt, F., Principles of Neurodynamics, Spartan Books, Washington, D.C. (1962).

14. Rumelhart, D. and McClelland, J., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. I, MIT Press, (1986).

15. Verstraete, R.A., *Assignment of Functional Responsibility in Perceptrons*, Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, CA, (June 1986).

16. Yau, S. S. and C. K. Tang, Universal Logic Circuits and their Modular Realizations, *AFIPS Conference Proceedings*, vol. 32, pp. 297-305, (1968).