In *IEEE Transactions on Systems, Man, and Cybernetics*, vol. **24**, No. 3, pp. 503-510, 1994.

Proof of Correctness for ASOCS AA3 Networks[1]

Cory Barker and Tony R. Martinez

Computer Science Department, Brigham Young University

Abstract—This paper analyzes Adaptive Algorithm 3 (AA3) of Adaptive Self-Organizing Concurrent Systems (ASOCS) and proves that AA3 correctly fulfills the rules presented. Several different models for ASOCS have been developed. AA3 uses a distributed mechanism for implementing rules so correctness is not obvious. An ASOCS is an adaptive network composed of many simple computing elements operating in parallel. An ASOCS operates in one of two modes: learning and processing. In learning mode, rules are presented to the ASOCS and incorporated in a self-organizing fashion. In processing mode, the ASOCS acts as a parallel hardware circuit that performs the function defined by the learned rules.

I. Introduction

Many connectionist computing or neural network architectures have been developed including backpropagation techniques [9], Boltzmann machines [1], and spontaneous learning systems [3, 10]. Connectionist models are characterized by many simple computing elements (nodes) operating in parallel with a large number of interconnections. Most models use a static network topology and learn by changing node functions. An Adaptive Self-Organizing Concurrent System (ASOCS) [4, 6] is a connectionist model that learns both by selecting node functions and by dynamically changing the network topology. An ASOCS, like most connectionist models, operates in both data processing and data learning modes.

During data processing, the ASOCS acts as a parallel hardware circuit. As is typical for hardware circuits, 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.

During data learning, the ASOCS reconfigures itself in a distributive manner to accommodate new (and perhaps conflicting) rules. ASOCS potential comes from its ability to 1) guarantee correct learning of any new rule, and 2) adapt to any new rule in time bounded by $O(log\ n)$, where n is the number of network nodes.

A number of formal ASOCS models have been developed, with initial research focusing on adaptive algorithm 1 (AA1) [5], adaptive algorithm 2 (AA2) [7], and adaptive algorithm 3 (AA3) [8]. These three algorithms vary dramatically, although AA3 has some similarity to AA2. AA3 improves on other ASOCS models in simplicity, implementability, and cost.

AA3 has a number of potential advantages over other learning models. Typical connectionist learning models require many presentations of the training set while AA3 learns a set of rules in one pass. Some learning models can fail to converge on the training data. AA3 will always learn the training data correctly. Judd [2] has shown that learning in a network with fixed topology (where learning is done by changing node functions) is intractable. Since AA3 modifies the network topology during learning the system is able to learn a set of rules in polynomial time. A weakness of AA3 is its lack of good generalization. Work is currently being done to combine ASOCS ideas with other techniques to improve generalization. Potential applications for AA3 include adaptive logic, robotics, and real-time dynamic control.

AA3 uses a distributed, as opposed to localist, approach to fulfilling a rule; i.e. for a given rule no single node can be identified that implements the rule. For this reason it can be difficult to

---

[1]Manuscript received

The authors are with the Computer Science Department, Brigham Young University, Provo, Utah 84602. This research has been supported in part by the Air Force Office of Scientific Research.

intuitively see how AA3 correctly implements a set of rules. The goal of this paper is to prove that AA3 is correct and also to support intuitive understanding of the model. A brief description of the AA3 model is given here. A detailed description and motivation for the model can be found elsewhere [9].

The outline of the paper is as follows. Section 2 defines the mechanism of ASOCS knowledge input. Section 3 describes AA3 architecture and operation in processing mode. Section 4 describes AA3 operation during learning mode. Section 5 defines the concepts of Boolean domains and subdomains. Section 6 proves that an AA3 network is a total function. Section 7 proves that an AA3 network correctly fulfills the rules presented to it. Section 8 gives conclusions and summary. Section 9 contains references.

II. Knowledge Input

The function to be performed by the network is defined by if-then rules called *instances.* Each instance is a partial function from a set of Boolean variables to a Boolean variable. An instance is written as a *variable set* followed by an implication arrow followed by an *output variable*. A variable may include a negation which is indicated by a prime symbol. For example, A' is the negation of the variable A. The variables A' and A are different variables, but A' and A are related in that the value of A' is always the opposite of the value of A. When reference is made to a negated variable, the variable is considered to include its negation. A variable set is *matched* when the conjunction of its variables is true. An instance specifies that, when its variable set is matched, the output variable must be set true. When the variable set of an instance is not matched, the instance says nothing about the output of the function.

The following are examples of instances:

I. $A'B' \rightarrow C$

II. $AB'C \rightarrow Z'$

III. $A'B' \rightarrow C'$

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

An instance with a non-negated output variable true is called a *positive* instance. An instance with a negated output variable is called a *negative* instance. This characteristic of an instance is called *polarity.*

A set of instances S is *consistent* if S does *not* contain any two instances X and Y where X is a positive instance, Y is a negative instance and 1) they have the same output variable and 2) there is a set of Boolean values which can simultaneously match the variable sets of X and Y. If a new instance is inconsistent with an instance set, then we give precedence to the newer instance and remove any contradicted portions of old instances.

The reasoning for giving precedence to newer instances is based on the concept of *incremental learning*. Incremental learning assumes that general rules are learned first. Specific rules are learned later and override portions of the general rules as exceptions. When the specific rules do not apply, the system falls back on the general rule. For example, when learning rules for English language plural, the first and most general rule learned is to add *s* to a base noun. The rule works for a large majority of words. Later specific exceptions are learned such as with the word *mouse* where the plural is *mice* instead of *mouses*.

A Boolean variable occurring in the variable set of one instance and occurring in its complemented form in another instance is said to be a *discriminant variable* for the two instances. A discriminant variable is a necessary and sufficient condition for consistency between two instances with the same output variable and opposite polarity.

If $X\colon V1 \rightarrow O1$, $Y\colon V2 \rightarrow O2$ are two instances, then the relationship between $V1$ and $V2$ may be one of *superset*, *equal*, *subset*, *discriminated* or *nondiscriminated*. Superset, equal, and subset are standard set relationships. Two variable sets are discriminated if they have a discriminant variable. Two variable sets are nondiscriminated if one of the other four relationships does not hold. Superset, equal, and subset are actually special cases of nondiscriminated. Table 1 shows examples of the five relationships between variable sets.

Variable Set	**Relationship to $AB'D$**
A	Subset
$AB'D$	Equal
$AB'DE$	Superset
$B'E'$	Nondiscriminated
CG	Nondiscriminated
$AB'D'$	Discriminated

Table 1 - Variable Set Relationships

III. Network Structure

This section describes the structure of the network and how Boolean inputs are mapped to Boolean outputs in processing mode. Nodes that perform a two input AND function are dynamically allocated and connected to build conjunctions of input variables. The constructed conjunctions are not identical to the conjunctions given by the variable sets of the input instances. The implementation of a single instance's conjunction may be distributed over multiple network conjunctions. Conjunctions are combined using two multiple-input OR gates called OR-planes, one for positive conjunctions and one for negative conjunctions. This overall structure is shown in Figure 1.

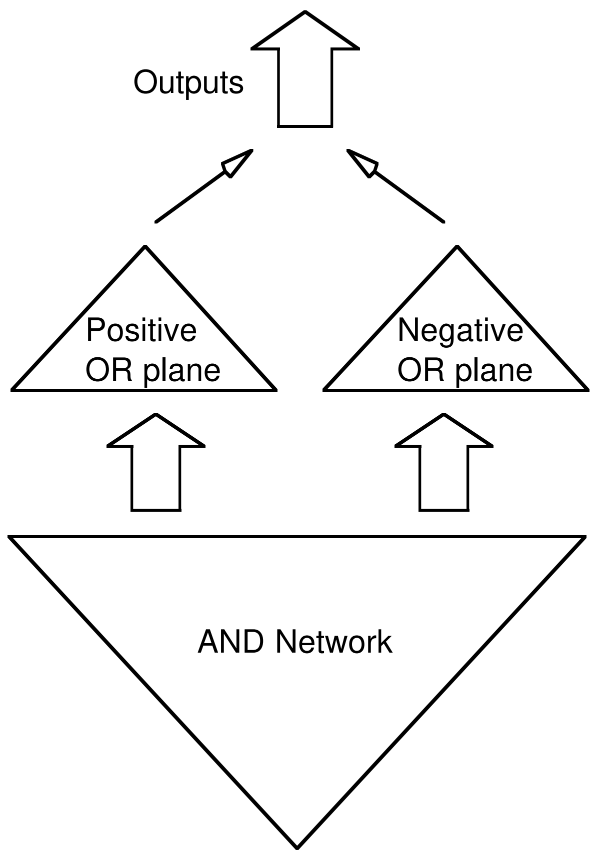

Figure 1 - AA3 Structure

The AND nodes have several important characteristics illustrated in Figure 2. At the base of the network of nodes is a *root* node with no inputs. All other nodes have two inputs or *children*, one from a node and the other from an input variable. The *variable set* of a node is the union of the variable set of the node's child and the variable directly input to the node. The variable set of the root is empty. The output of the root is always true, while the output of any other node *N* is the conjunction of the variables in the variable set of *N*. Each node is labeled with its variable set.

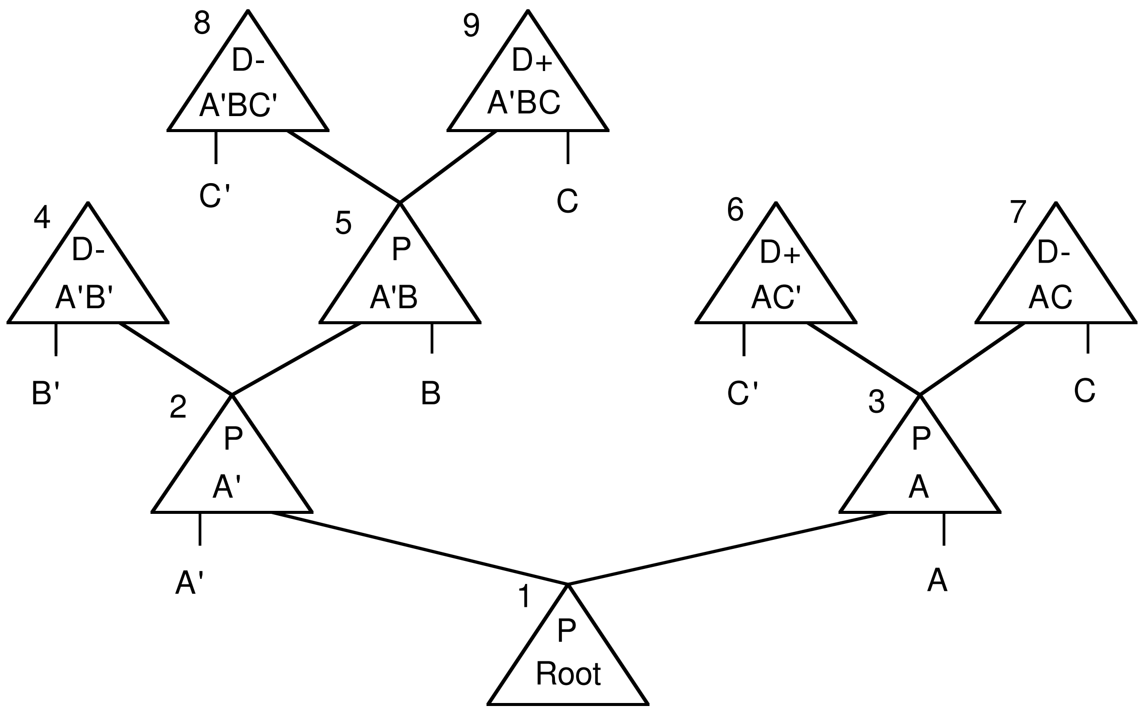

Figure 2 - AA3 Network

Each node is either a Primitive node (Pnode) or a Discriminant node (Dnode). Pnodes are building block nodes and output to other nodes while Dnodes are terminal or leaf nodes and output to the OR-planes. Dnodes can be either *positive Dnodes* (D+) or *negative Dnodes* (D-). Positive Dnodes output to the positive OR-plane while negative Dnodes output to the negative OR-plane. If the variable set of any positive Dnode is matched, then the positive OR-plane outputs true. If the variable set of any negative Dnode is matched, then the negative OR-plane outputs true.

Each Pnode has exactly two parents, the *left parent* and the *right parent*. Such parent nodes are said to be *siblings* with respect to each other. If a node has variable input *V*, its sibling always has variable input *V'*. The root node can be either a Pnode or a Dnode but has no sibling since it has no inputs. All other nodes must have a sibling.

A network *fulfills* an instance set if, when any positive instance is matched the positive OR-plane output is true and the negative OR-plane output is false. When any negative instance is matched, the negative OR-plane output is true and the positive OR-plane output is false. Conflicts between instances are resolved by order; the latest instance takes priority over the portion of any previous instances that it contradicts. For states of the environment that are not matched by any instance in the instance set, the network may arbitrarily choose the output of the OR-planes. The positive OR-plane may output true, the negative OR-plane may output true, or both OR-planes may output false indicating no output. Both OR-planes are never allowed to output true simultaneously since this is a contradiction.

An example of network execution is as follows. Suppose the network in Figure 2 is given the input *A'BC*. Node 1 will be active because the root is always active. Node 2 will be active because *A'* is active and the child of node 2 (the root) is active. Similarly nodes 5 and 9 are active. Since node 9 is a positive Dnode the positive OR-plane will be active. Note that all other nodes and the negative OR-plane are inactive.

IV. Learning

We describe the AA3 learning algorithm which tells how a consistent network reconfigures itself when faced with a new instance and we give an example of learning. When the system receives a new instance, each Dnode is sent the *variable set* and the *polarity* of the new instance. Each Dnode then independently executes the learning algorithm. Pnodes may be created by learning but remain inactive during the learning algorithm.

Let N be a Dnode and let I be the new instance. Let V_n be the variable set of N and let V_i be the variable set of I. Let P_n be the polarity of N and P_i be the polarity of I. The learning algorithm first compares P_n with P_i. If P_n=P_i, then the node makes no change since the node and the instance agree in output.

If $P_n \neq P_i$, then V_n is compared to V_i. If V_n is discriminated from V_i, then the node makes no change because the node and the instance can never cover the same state of the environment. If V_n is superset or equal to V_i then I covers all of N (N is completely contradicted by I), so P_n is changed to agree in polarity with P_i. Otherwise the input spaces of N and I overlap and the procedure DVA changes the contradicted part of N to agree with I. After the new instance has been incorporated, the node checks to see if it can self-delete. The procedures DVA and Self-Delete are described later.

```
Procedure Learn-New-Instance(N, I);
        If N is a Dnode and Pn≠Pi and Vn is not discriminated from Vi then
                If Vn is superset or equal to Vi then
                        Invert the polarity of N.
                else
                        Let S = Vi–Vn.
                        Call procedure DVA(N, S).
                end.
        end.
        Call procedure Self-Delete(N).
end.
```

The procedure DVA changes the contradicted part of N to agree with I. For each variable in V_i that is not in V_n, DVA adds two new nodes effectively splitting the input space of the old node in half. The final split creates a node covering the contradicted part of the old node. The polarity of this node is set to agree with the new instance.

```
Procedure DVA(N, S);
        Allocate two new nodes, N1 and N2 as parents of N.
        Let V be a variable in S.
        Make N1 a Dnode with polarity Pi and variable input V.
        Make N2 a Dnode with polarity Pn and variable input V'.
        Make N a Pnode.
        If |S| > 1 then
                Call procedure DVA(N1, S–V).
        end.
end.
```

The procedure Self-Delete allows two sibling Dnodes that output the same polarity to be removed. If two sibling Dnodes have the same polarity as a result of polarity inversion, the two nodes combined perform the same function as their child Pnode. The two nodes are removed and the child node is changed to a Dnode with the polarity of the deleted parent Dnodes. The algorithm is as follows:

Procedure Self-Delete(N);
  Let N be a node, S be the sibling of N, and C be the child of N and S.
  If N is a Dnode and the polarity of N is the same as the polarity of S then
    Change C to a Dnode with same polarity as N.
    Delete N and S.
    Call procedure Self-Delete(C).
  end.
end.

We now give an example of learning. Suppose the instance $B \rightarrow Z$ is presented to the network in Figure 2 (in section 3). Nodes 7 and 8 are the only nodes that are modified since they are the only Dnodes that differ in polarity from the new instance and are not discriminated from the new instance. Node 4 differs in polarity but is discriminated by the variable B. The variable set of node 8 is a superset of B so node 8 does polarity inversion becoming a positive Dnode. Node 7 is nondiscriminated with B so node 7 does DVA with S equal to $B - AC = B$. Since S contains only one variable the DVA procedure is executed only once creating two new parents for node 7 with variable sets ABC and $AB'C$. The network after incorporating the new instance is shown in Figure 3. Note that if S contained a second variable, node 11 would be split by adding two new parents.

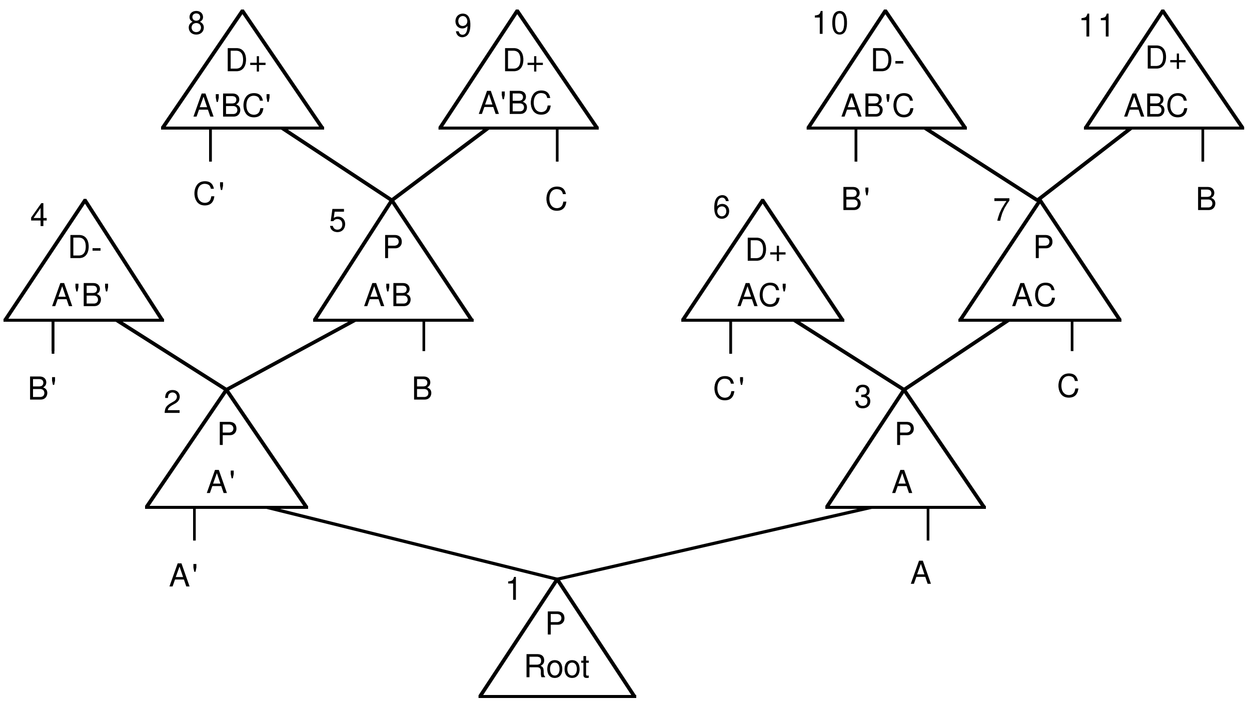

Figure 3 - Network after Adding Instance $B \rightarrow Z$

Next, self-deletion commences. Nodes 8 and 9 are sibling Dnodes with the same polarity so they delete making node 5 a positive Dnode. All sibling Dnodes are now opposite in polarity so self-deletion is completed. Note that if Node 4 were a positive Dnode, Nodes 4 and 5 would delete and node 2 would become a positive Dnode. The final network is shown in Figure 4.

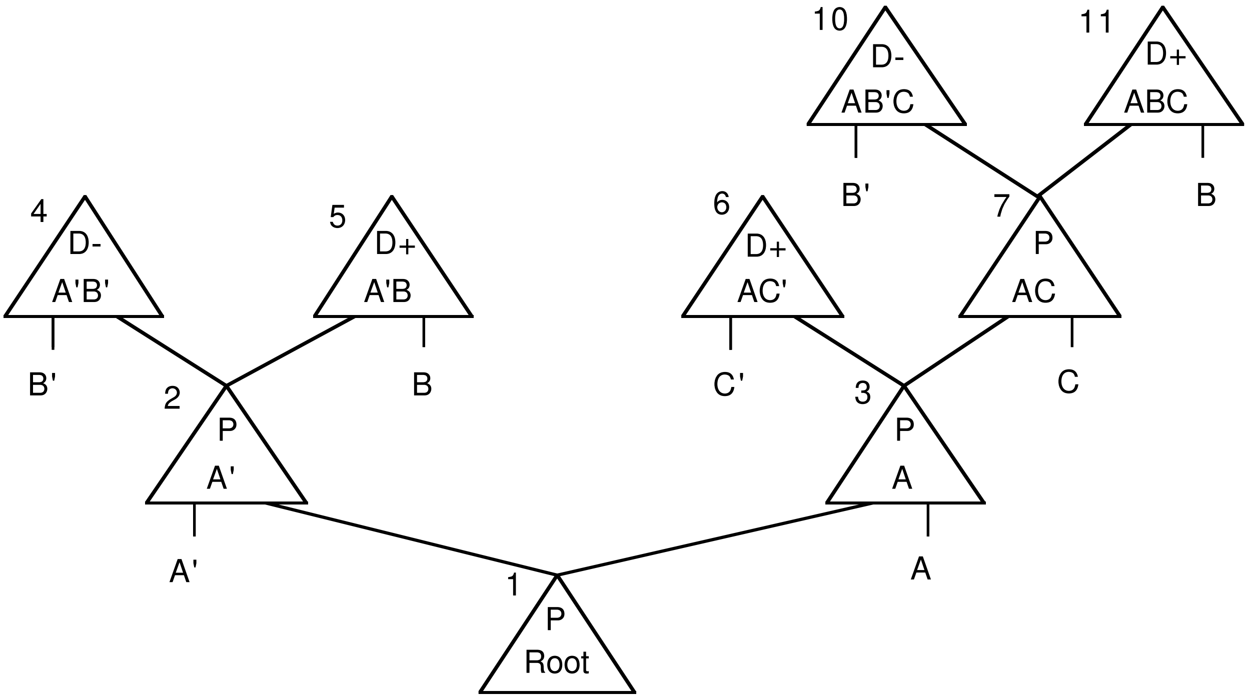

Figure 4 - Network after Self-Deletion

Note that $B \rightarrow Z$ is implemented by nodes 5, 6, and 11. With the three inputs, A, B, and C, there are four states of the environment that match the instance $B \rightarrow Z$: ABC, ABC', $A'BC$, and $A'BC'$. Node 11 covers the state ABC, node 6 covers the state ABC', and node 5 covers the states $A'BC$ and $A'BC'$. Node 6 was not modified by the learning algorithm but still participates in fulfilling the new instance. This distributed nature of the AA3 learning algorithm causes it to be unobvious. In addition since unmodified nodes may fulfill part of a new instance, it is many times unclear if an instance is completely fulfilled. This shows the need for a proof of the AA3 learning algorithm.

V. Boolean Domains

In order to show that an AA3 network correctly fulfills its instance set it is useful to be able to refer to subsets of the complete input environment. This section defines the input environment as the *domain* and subsets of the environment as *subdomains.* The method for naming subdomains and relating them to instances and nodes is also defined.

The *domain* of a network is the set of all possible combinations of the input variables and their complements. For example, suppose the environment has three variables A, B, and C. The domain then consists of all possible true/false combinations of the three variables as shown in Figure 5.

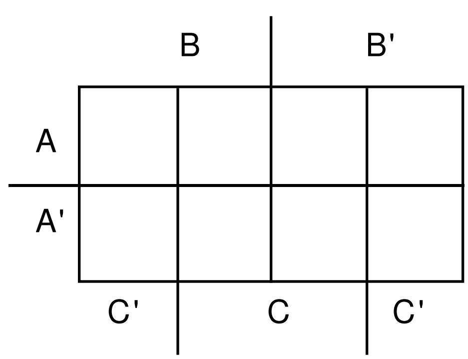

Figure 5 - Domain with Variables *A*, *B*, and *C*

We define a *subdomain* to be a subset of a domain that can be described by the conjunction of a set of variables; the subdomain being only the states of the environment where all variables in the set are matched. For example, if the variable *A′* is in the set, then the subdomain is restricted to the bottom row of Figure 5. *A* must be false for the variable set to be matched. On the other hand, if neither variables *V* nor *V′* are contained in the variable set describing a subdomain, the subdomain is not restricted in terms of *V*, and covers states of the environment where *V* is both negated and not negated. If the variables *B* and *B′* are not in the set, then the subdomain covers both sides of Figure 5. *B* is not specified in the variable set, so *B* can be either true or false in the environment and the variable set will still be matched. The variable set *A′B* describes the subdomain covering the two squares in the lower left of Figure 5. The variable *A′* restricts the subdomain to the bottom row and the variable *B* restricts the subdomain to the left half. The subdomain is not restricted in terms of the *C* variable and so covers both states of *C*.

A node in an AA3 network implements the network function for a subdomain. The subdomain of a node is defined by the variable set of the node. For example node 4 in Figure 2 has the subdomain shown in Figure 6 since the variable set of node 4 is *A′B′*.

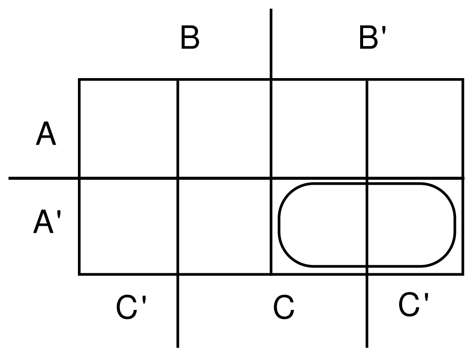

Figure 6 - Subdomain *A′B′*

Each instance input to an AA3 network defines the function for a subdomain. The subdomain of an instance is defined by the variable set of the instance. For example, the instance $A \rightarrow Z$ has the subdomain shown in Figure 7.

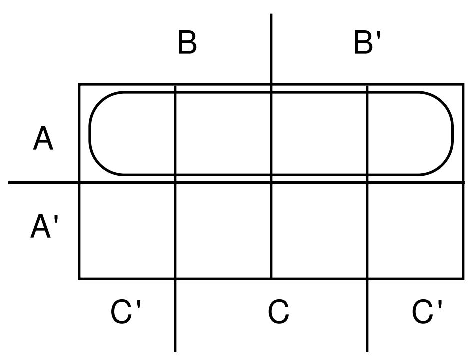

Figure 7 - Subdomain *A*

A subdomain can sometimes be divided into smaller subdomains. For the domain consisting of the three variables *A*, *B*, and *C*, the instance $A \rightarrow Z$ is equivalent to four instances.

$AB'C' \rightarrow Z$

$AB'C \rightarrow Z$

$ABC' \rightarrow Z$

$ABC \rightarrow Z$

The subdomain of a new instance can be divided into two parts, a *satisfied* part and a *contradicted* part. Either part may be empty. The satisfied part is already implemented correctly by the AA3 network. This is true because the network is a total function (shown later), meaning that it is defined for all inputs, and the network does not contradict the instance. The network outputs the wrong value for the contradicted part. For example, Figure 8 shows the domain and outputs of a network. The instance $B \rightarrow Z$ has the subdomain with a satisfied part and a contradicted part as shown.

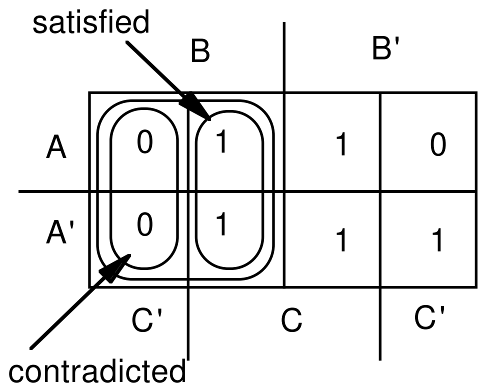

Figure 8 - Subdomain *B*

VI. AN AA3 NETWORK IS A TOTAL FUNCTION

In this section we show that an AA3 network is a consistent and total function. Consistent means that the network will not activate both the positive and the negative OR-planes simultaneously. Total means that the network will always have an output for any input; one of the two OR-planes will be active. The total nature of the network is important for understanding how the network correctly covers instances that are not contradictory. The learning algorithm performs no operation for instances or portions of instances that do not contradict the existing network. Since the network is total it will already output the correct value for such instances.

Lemma 1.1 shows that for any two siblings in a network only one of the two siblings may be active at a time. This result is used in Lemma 1.2 to show inductively that only one Dnode can be active in any network. Finally, Theorem 1 shows that if only one Dnode is active then exactly one of the OR-planes will be active indicating that the network is a total function.

LEMMA 1.1 - Single Sibling Active

If two nodes *L* and *R* in an AA3 network are siblings with an active child *C*, exactly one of *L* and *R* will be active.

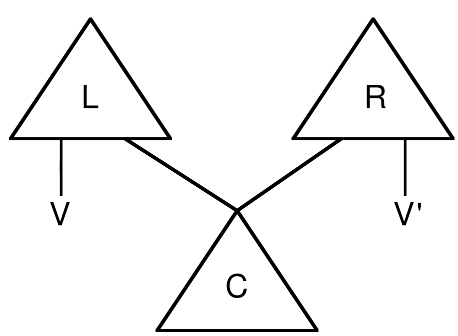

Figure 9 - AA3 Siblings

PROOF

The node *L* has as inputs 1) the output of *C*, and 2) some variable *V*. The node *R* has as inputs 1) the output of *C*, and 2) the complement of the variable *V*; written *V'*. Since all nodes in an AA3 network perform the AND function, both inputs to a node must be active for the node to be active. The variable *V* and its complement *V'* cannot both be active simultaneously, so both *L* and *R* cannot both be active simultaneously. On the other hand, one input to each node is the output of *C*, which is active as given in the statement of the lemma, and one of *V* or *V'* must be active, so one of *L* or *R* must be active. [end of lemma 1.1]

LEMMA 1.2 - Single Dnode Active

In an AA3 network exactly one Dnode is active at a time.

PROOF

We show by induction that exactly one node is active at each level in the tree from the root to the single active leaf or Dnode. The proof is by induction on the level, *K*, in the tree. The level of the root node is defined as 0 and is the basis case. The root node is always active and is the only node at level 0, so level 0 has exactly one node active and the basis is true.

Let the active node at level *K* be denoted by *N*. The inductive hypothesis allows us to assume that node *N* is the one and only active node at level K. All other nodes at level *K* must be inactive so any parents of those nodes will be inactive. *N* has two parents, *L* and *R*, that are siblings. Exactly one of *L* and *R* will be active by lemma 1.1, so exactly one node will be active at level *K*+1 and the induction step is true.

Node *N* at level *K* must be either a Pnode or a Dnode. If *N* is a Pnode then *N* has two parents and one of the parents must be active by lemma 1.1 so the induction must continue to level *K*+1. If *N* is a Dnode then *N* has no parents and no node can be active at level *K*+1 so the induction must terminate at the first Dnode encountered. [end of lemma 1.2]

THEOREM 1 - AA3 Consistency and Totality

An AA3 network is a consistent, total function.

PROOF

In an AA3 network, positive Dnodes connect to one OR-plane and negative Dnodes connect to a second OR-plane. Since exactly one Dnode is active, either the positive OR-plane will be active or the negative OR-plane will be active. Both OR-planes cannot be active, indicating an inconsistent state, since more than one Dnode cannot be active. Both OR-planes cannot be inactive, indicating no output for the given input or a non-total function, since one Dnode must be active. [end of theorem 1]

VII. An AA3 Network Fulfills the Instances Presented To It

This section proves that an AA3 network fulfills the instances presented to it. The proof shows that the network is correct for the first instance and that all changes to the network preserve its correctness. Theorem 1 is used to show that input subdomains that are not modified are still correctly covered by the network.

When a new instance is incorporated into a network, zero or more nodes may change in parallel according to the learning algorithm, but all changes are of two types: polarity inversion and DVA. Polarity inversion occurs when an existing node is superset or equal with respect to the new instance. Lemma 2.1 shows that polarity inversion is correct in this case since the subdomain of the node is contained within the subdomain of the new instance. Lemma 2.3 is used to show that DVA is correct since it only changes the contradicted part of existing nodes. When DVA occurs a new node is created whose variable set is the union of the variable sets of the old node and the new instance. Lemma 2.2 is used in Lemma 2.3 to show that this new node covers the subdomain that is the intersection of the subdomains of the new instance and the old node.

LEMMA 2.1 - Superset Subdomain Containment

If the variable set of a node is superset or equal with respect to the variable set of an instance, then the subdomain of the node is completely contained within the subdomain of the instance.

PROOF

Let the variable set for the node be V_n and the variable set for the instance be V_i. Let the subdomain of the node be S_n and the subdomain of the instance be S_i. V_n is either a superset or equal to V_i. Let D be $V_n - V_i$. If V_n is equal to V_i then D is empty and S_n is equal to S_i. Otherwise the instance can be rewritten as several separate instances with the following variable sets:

$V_i\, D_1'\, D_2' \ldots D_k'$
$V_i\, D_1\, D_2' \ldots D_k'$
$\ldots$
$V_i\, D_1\, D_2 \ldots D_k$

Each member of D is included exactly once in each instance using all combinations of polarity. The last instance matches V_n so the original instance covers at least as much as the node, and the subdomain of V_i is a superset of the subdomain of V_n. [end of lemma 2.1]

For example, let the variable set for the node be *ABCD* and let the variable set for instance be *AB*.

$$AB = ABCD + ABCD' + ABC'D + ABC'D'$$

So the subdomain of the instance includes the subdomain of *ABCD* as well as the subdomains of the other terms listed.

LEMMA 2.2 - Subdomain Intersection

Let *A* and *B* be two subdomains. Let V_A and V_B be the variable sets that define the subdomains *A* and *B*. Let *C* be the subdomain that is the intersection of *A* and *B*. The variable set that defines *C*, V_C, is equal to V_A union V_B.

PROOF

The proof is by contradiction. There are two cases where the statement can be contradicted, either V_C is missing a variable that is contained in the union, or V_C contains an additional variable not contained in the union. In both cases it is shown that *C* would not equal the intersection of *A* and *B*.

Case 1) Suppose V_C does not contain a variable *V* in V_A union V_B. The subdomain *C* then covers both polarities of *V*, but the complement of *V* is outside of one of the original subdomains. Since *V* is in V_A union V_B, *V* must be contained in one of V_A or V_B. Suppose *V* is contained in V_A. Then *A* only covers *V* and not the complement of *V*, so the intersection of *A* and *B* cannot contain the complement of *V*.

Case 2) Suppose V_C contains an extra variable *V* not in V_A union V_B. Then *C* covers only one polarity of the variable *V* and not its complement, but the complemented position is contained in both *A* and *B* (since the variable is not listed) and so should be contained in the intersection. [end of lemma 2.2]

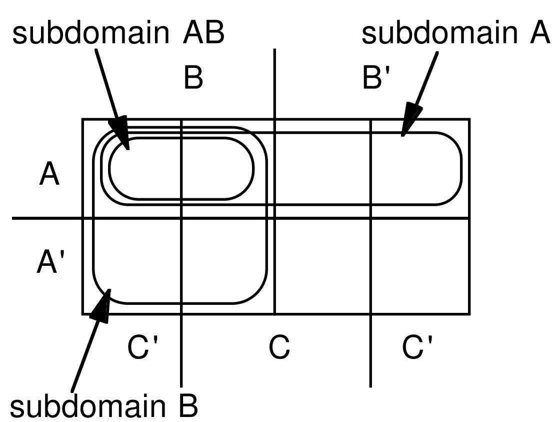

Figure 10 - Subdomain Intersection

For example, the intersection of the two subdomains *A* and *B* in Figure 10 is *AB*. Note that the subdomain *A* is equivalent to the union of four subdomains.

AB'C'
AB'C
ABC'
ABC

Subdomain *B* is equivalent to the union of four subdomains.

A'BC'
A'BC
ABC'
ABC

The intersection of the above two sets of subdomains is *ABC'*, *ABC* which is equivalent to the subdomain *AB*.

LEMMA 2.3 - DVA Correctness

Let *N* be a node and *I* be a new instance with opposite polarity and no discriminant variable. Let S_n be the subdomain of *N*, S_i be the subdomain of *I*, and *S* be the intersection of S_n and S_i. AA3 DVA: 1) changes the function for the subdomain *S* to match *I*, 2) leaves the function for the subdomain S_n–*S* unchanged.

PROOF

Let V_n be the variable set for *N* and V_i be the variable set for *I*. Let *D* be the set difference V_i–V_n. Starting with the node *N*, DVA creates two new nodes for each member of *D*. Each iteration adds a variable from the set *D* to the set V_n until the final iteration creates a node *F* with variable set V_n union *D* = V_n union V_i. The polarity of *F* is set to match *I* and by lemma 2.2 the subdomain of *F* is equal to *S*. This proves the first claim of the lemma.

The second claim of the lemma follows by noting that all other nodes created by the DVA are set to the polarity of *N*. By lemma 1.2 no Dnode other than *N* can cover S_n before DVA. The Dnodes created by DVA must therefore cover all of S_n after DVA. One of these nodes is *F* and it covers *S*, so the remaining nodes must cover S_n–*S*. [end of lemma 2.3]

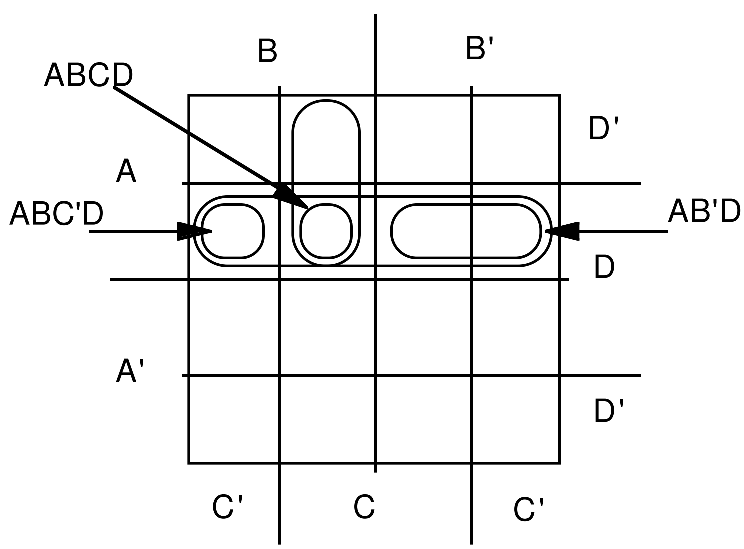

Figure 11 - DVA Correctness

For example, given node *AD* and instance *ABC*, the first DVA iteration produces two nodes from *AD*; *ABD* and *AB'D*. The second iteration splits *ABD* into two nodes; *ABCD* and *ABC'D*. The variable set *ABCD* is the union of *AD* and *ABC*. The map in Figure 11 shows that *ABCD* covers the intersection and the remaining nodes cover the rest of the original node's subdomain.

THEOREM 2 - AA3 Instance Fulfillment

An AA3 network fulfills the instances presented to it.

PROOF

It will be shown that 1) an initial AA3 network consisting of a single root node fulfills the first instance, and 2) all changes to the network caused by the addition of instances preserve its correctness.

1) An initial AA3 network consists of one Dnode. The Dnode has no inputs and is set to output the same polarity as the first instance without regard to the state of the environment. Since the node outputs the same polarity as the first instance no matter what the state of the environment, the first instance is fulfilled.

2) There are three ways that an AA3 network can be modified:

- polarity inversion
- DVA
- self-deletion

We will show that a) Polarity inversion and b) DVA both change only subdomains that contradict the new instance and that c) Self-deletion does not change the network function. The subdomain of the new instance that is not contradicted by any node in the network must be satisfied because the network is a total function and if the network does not disagree in output with the new instance then it must agree in output. Nodes that agree in output with a new instance do not change. Nodes that disagree in output with a new instance but have a discriminant variable do not change because the subdomains of the node and the instance do not overlap.

a) Polarity inversion is done when a node is superset or equal with respect to the new instance. By lemma 2.1, the subdomain of the node is completely contained within the subdomain of the new instance. Polarity inversion changes the defined function for the node's subdomain to agree with the new instance. No part of the domain outside of the subdomain of the new instance is modified. Thus, only changes needed to satisfy the new instance are made.

b) When a node is subset or nondiscriminated with respect to the new instance then the subdomain of the node intersects with the subdomain of the new instance. By lemma 2.3, DVA changes the function for the intersection of the two subdomains. The network then agrees with the new instance in the subdomain of intersection. The rest of the subdomain of the node is unchanged. Thus, only the part of the domain needed to satisfy the new instance is changed. (Recall that newer instances take priority over old, so the network is correct to change a contradicting node within the subdomain of the new instance to agree with the new instance.)

c) Self deletion does not change the network function. The self delete process allows two sibling Dnodes of the same polarity to be removed from the network. The variable set of one sibling is of the form AX while the variable set of the other sibling is of the form $A'X$, where A is the direct input variable and X is the variable set of the child node. Since the two Dnodes are of the same polarity they are both connected to the same OR-plane. The functions of the two nodes therefore can be combined by the OR function giving $AX + A'X$. The function performed by the child of the two Dnodes is X since all AA3 nodes perform the AND function of their child and their direct input variable. By standard Boolean identities, $AX + A'X \equiv X$. Therefore, when the child node performing the function X is made a Dnode of the same polarity as the two removed nodes, the function of the network does not change. [end of theorem 2]

VIII. CONCLUSION

We have proven that an AA3 network correctly fulfills the instances presented to it. The network will always output the correct value when an instance is matched by the input. When no instance is matched, meaning that the network has not been trained for the current input, the network will extend the function defined by the instances and output a generalized value. The network is always a consistent and total function.

The AA3 learning algorithm has been explained. Each node is sent the polarity and variable set of the new instance. Nodes that conflict with the new instance then either invert their polarity or perform DVA to resolve the conflict. Nodes then independently consider whether self-deletion is possible.

Though at times not intuitive, the AA3 algorithm is guaranteed to learn any Boolean function correctly and to learn the function in time bounded by the log of the number of nodes in the network. Once the function is learned, the network will then execute the mapping like a parallel hardware circuit, in time also bounded by the log of the number of nodes in the network.

REFERENCES

[1] G. Hinton, T. Sejnowski, and D. Ackley, "Boltzmann machines: Constraint satisfaction networks that learn", Tech. Rep. CMU-CS-84-119, CMU, Pittsburgh, PA, 1984.

[2] J. S. Judd, "On the complexity of loading shallow neural networks," *Journal of Complexity,* vol. 4, pp. 177-192, 1988.

[3] T. Kohonen, *Self-Organization and Associative Memory*, New York: Springer Verlag, 1984.

[4] T. R. Martinez, "Adaptive self-organizing logic networks," Ph.D. Dissertation, Tech. Rep. CSD 860093, Univ. Calif., Los Angeles, May 1986.

[5] T. R. Martinez and J. J. Vidal, "Adaptive parallel logic networks," *Journal of Parallel and Distributed Computing*, vol. 5, no. 1, pp. 26-58, Feb. 1988.

[6] T. R. Martinez, "Adaptive self-organizing concurrent systems," in *Progress in Neural Networks*, vol. 1, O. Omidvar, Ed. Norwood, NJ: Ablex Publishing, 1990, ch.5, pp. 105-126.

[7] T. R. Martinez and D. M. Campbell, "A self-adjusting dynamic logic module," *Journal of Parallel and Distributed Computing*, vol. 11, no. 4, pp. 303-313, 1991.

[8] T. R. Martinez and D. M. Campbell, "A self-organizing binary decision tree for incrementally defined rule-based systems", *IEEE Transactions on System, Man, and Cybernetics*, vol. 21, no. 5, pp. 1231-1238, 1991.

[9] D. Rumelhart and J. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1, Cambridge, MA: MIT Press, 1986, pp. 318-362.

[10] D. Rumelhart and D. Zipser, "Feature discovery by competitive learning," *Cognitive Science,* vol. 9, pp. 75-112, 1985.