In *Journal of Artificial Neural Networks*, vol. **1**, no. 3, pp. 403-429, 1994.

Priority ASOCS

Tony R. Martinez
Douglas M. Campbell
Brent W. Hughes

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. An ASOCS can operate in either a data processing mode or a learning mode. During data processing mode, an ASOCS acts as a parallel hardware circuit. During learning mode, an ASOCS incorporates a rule expressed as a Boolean conjunction in a distributed fashion in time logarithmic in the number of rules. This paper proposes a learning algorithm and architecture for *Priority ASOCS*. This new ASOCS model uses rules with priorities. The new model has significant learning time and space complexity improvements over previous models.

Keywords: ASOCS, self-organizing, neural networks, parallel, rule-based, learning.

1. Introduction

Non-von Neumann architectures such as neural networks attack the word-at-a-time bottleneck of traditional computing systems [1]. Neural networks learn input-output mappings using highly distributed processing and memory [10,11,12]. Their numerous simple processing elements with modifiable weighted links permit a high degree of parallelism. A typical neural network has fixed topology. It learns by modifying weighted links between nodes.

A new class of connectionist architectures has been proposed called ASOCS (Adaptive Self-Organizing Concurrent Systems) [4,5]. ASOCS models support efficient computation through self-organized learning and parallel execution. Learning is done through the incremental presentation of rules and/or examples. ASOCS models learn by modifying their topology. Data types include Boolean and multi-state variables; recent models support analog variables. The model incorporates rules into an adaptive logic network in a parallel and self organizing fashion.

In processing mode, ASOCS supports fully parallel execution on actual inputs according to the learned rules. The adaptive logic network acts as a parallel hardware circuit during execution, mapping *n* input boolean vectors into *m* output boolean vectors, in a combinatoric fashion.

The overall philosophy of ASOCS follows the high level goals of current neural network models. However, the mechanisms of learning and execution vary significantly. The ASOCS logic network is topologically dynamic with the network growing to efficiently fit the specific application. Current ASOCS models are based on digital nodes. ASOCS also supports use of symbolic and heuristic learning mechanisms, thus combining the parallelism and distributed nature of connectionist computing with the potential power of AI symbolic learning. A proof of concept ASOCS chip has been developed [2].

ASOCS models have two significant advantages over other learning models:

a) they are guaranteed to learn any arbitrary set of legal rules;

b) learning time is both fast and bounded (linear with the depth of the network, and logarithmic with the number of nodes).

Application targets for ASOCS include adaptive logic devices, logical inference, robotics, embedded systems, and pattern recognition.

A number of formal ASOCS models have been proposed to date [6,7,8]. This paper presents a new model of ASOCS called *Priority ASOCS* [3,9]. The Prioritized ASOCS architecture provides a number of advantages over other ASOCS architectures:

- The number of nodes required never exceeds the number of rules presented to the system.
- The allocation of nodes is simpler and more parsimonious.
- Though not using a specific generalization method in the basic model, PASOCS allows for the efficient introduction of flexible and arbitrary generalization techniques.

Section 2 presents a simplified high level view of PASOCS to give the reader a basic understanding of the model. Section 3 gives essential definitions. Section 4 describes the learning algorithm in detail. Section 5 proposes a specific PASOCS architecture to support the learning algorithm. Section 6 presents a detailed example of PASOCS learning. Section 7 demonstrates the simple extension to allow multiple outputs. Section 8 concludes the paper.

While this paper presents a self-contained detailed discussion of PASOCS execution and learning, a more rigorous approach with proofs of all learning rules may be found in [3].

2. High Level Overview of Prioritized ASOCS

To give the reader the overall flavor of PASOCS processing, we informally describe a Prioritized ASOCS architecture. In sections 3 through 6 we give the technical detail of how the learning and processing are accomplished in a parallel and self-organizing fashion.

A Priority ASOCS is an ASOCS model which has the following attributes. PASOCS:

- processes inputs in the form of Boolean variables and outputs Boolean results;
- accepts rules made up of a conjunction of boolean inputs which imply a boolean output;
- learns new rules over time;
- automatically resolves rule conflicts;
- combines specific rules into more general rules where appropriate;
- maintains an associated priority with each rule.

We begin with the black box of figure 1, having four input variables {a,b,c,d} and the single output variable z. In general, an ASOCS maps *n* input variables into *m* output variables where the size of *m* and *n* do not need to be known a priori. The system allocates input and output lines as new variables are introduced in rules.

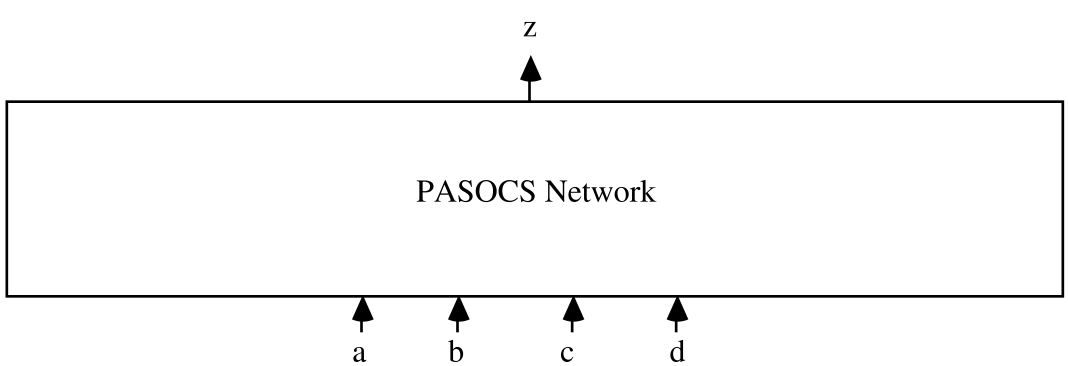

Figure 1 - PASOCS black box

Nodes and Rules: The black box of figure 2 contains nodes connected by a binary tree structure. Each node represents a rule. A rule is represented as a 3-tuple. The first element is a conjunction of boolean variables (the left hand side of the rule), the second element is the rule output (1 or -1), and the third element is the priority of the rule. The priority is a positive integer value with 1 being the lowest priority. The rule of node 1 in figure 2 has priority 2. It says to output (z=-1) whenever (a=1) and (b=1) and (c=0). Node 4 is currently empty.

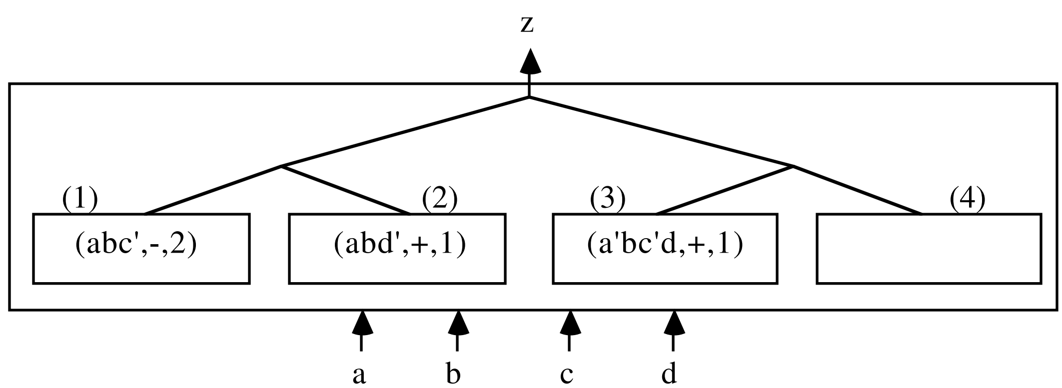

Figure 2 - Internal node representations

Execution: When the black box of figure 3 receives the input (a=1, b=1, c=0, d=0), it broadcasts these values down the binary tree structure to each node. Each node independently and in parallel compares itself to the input. If the input matches the rule, then the node passes up the binary tree a message consisting of the rule output and the rule priority. If the node does not match, no message is sent. *When two messages arrive at a junction in the tree, the one with higher priority (larger integer) is passed on while the other is discarded.* In the case of figure 3 and the given input, nodes 1 and 2 both fire. Since node 1 has priority 2, and node 2 has priority 1, the system outputs the result of node 1, which specifies an output of (z=-1).

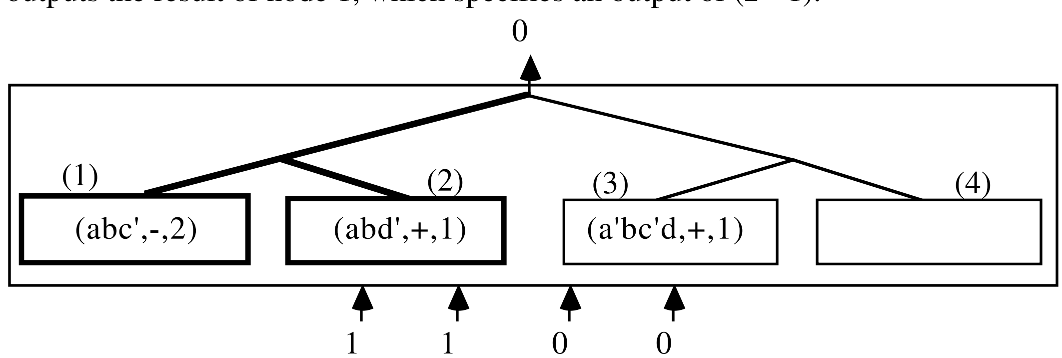

Figure 3 - PASOCS during input processing

Learning : When a new rule is presented to the system, it is broadcast down the tree to the nodes. Each node compares itself with the rule, and depending on how it matches, the node initiates an appropriate operation. The matching and operations occur independently and in parallel. These operations make up the learning algorithm. We present some examples to give a flavor for the basic learning algorithm. In the following, assume we wish to add the rule bd' => z, to the black box of figure 3. The following actions would take place. Though we present them sequentially, they would take place in parallel in an actual system.

Node Addition and Consistency Resolution: To resolve inconsistencies the system need only give the new rule a priority one greater than any rule it contradicts. The rule bd' => z is broadcast and all nodes which contradict the new rule (i.e. nodes which could be simultaneously matched for some input, but would cause the opposite output) send their priority up the tree. The highest priority arrives at the top and the new rule is added to the network with priority one higher than the highest inconsistent priority. In the case of figure 3, only node 1 (abc' = >z') is inconsistent with bd' = >z and it has priority 2. The system, therefore, adds bd' => z with priority 3, as shown in figure 4.

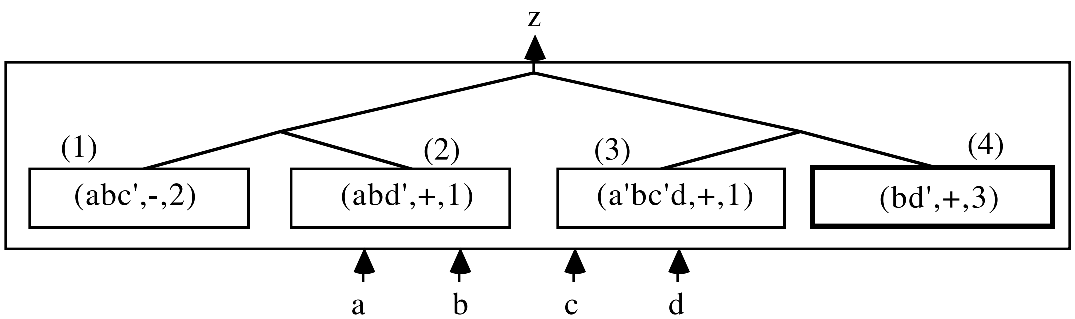

Figure 4 - Adding a new rule

Generalization: The new rule bd' => z has made node 2 redundant, since node 2 is matched in all cases where node 4 is matched. As part of the learning algorithm, the redundant node 2 deletes itself, as shown in figure 5.

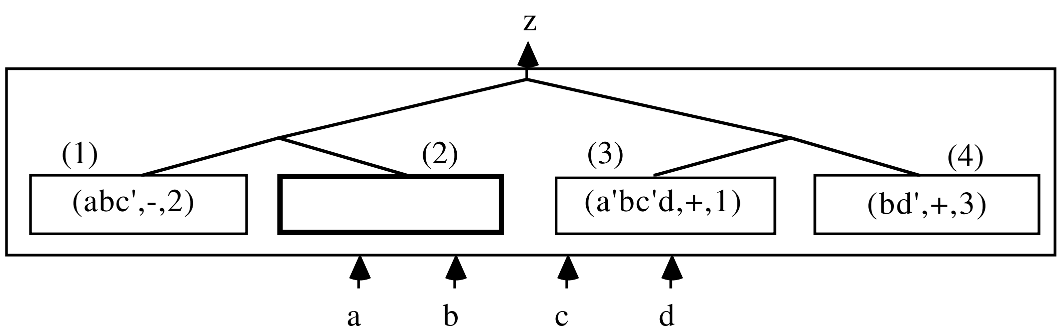

Figure 5 - After self deletion

The new rule bd' => z makes it possible for node 3 of figure 5 to generalize itself by removing the variable d. As part of the learning algorithm, node 3 minimizes itself, as shown in figure 6.

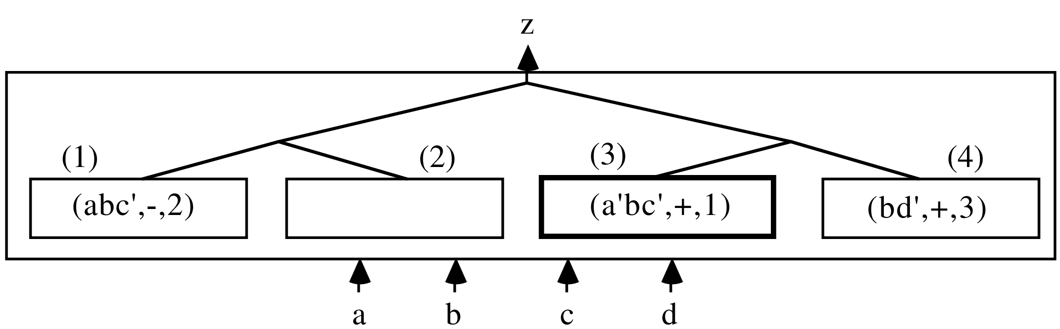

Figure 6 - After further generalization

This example gives a quick view of the style of learning mechanisms available in a PASOCS system. The body of the paper presents the full learning model and a potential architecture. We now conclude the list of PASOCS attributes begun earlier:

- In Prioritized ASOCS (PASOCS), whenever a new rule is learned, at most one node need be added to the system.
- PASOCS is guaranteed to learn any set of valid rules.

- PASOCS need only see a rule once when learning a rule set.
- PASOCS does not unlearn old rules when learning new rules, except in the case of conflict resolution.
- PASOCS can support multiple output variables. (See section 7)

Due to the localist node-per-rule nature of PASOCS, each node is independent of every other node. Thus any network topology can be chosen to best fulfill such needs as cost, speed, and fault tolerance. The particular topological instantiation proposed above is that of a binary tree. Due to this decision, we can state that the PASOCS:

- processes inputs in *O(log(n))* time, where *n* is the number of nodes (rules) in the network;
- learns individual rules in *O(log(n))* time, where *n* is the number of nodes (rules) in the network.

The PASOCS model presented above generalizes in the sense of simplifying rules. However, no mechanism is defined for what should be output when the input environment matches no learned rule. Other ASOCS models [6,8] use a distributed mechanism to accomplish automatic generalization for novel inputs.

Generalization in a learning system is at best a always a guess. The proper style of generalization is application dependent. Thus, one style of generalization is not sufficient to allow a learning system to support a broad spectrum of applications. Current connectionist models, including earlier ASOCS models, use one specific style of generalization which is implicit in the learning algorithm. We suggest that the type of generalization used be a self-organizing parameter of the learning system which can be discovered as learning takes place. This requires a) a model which allows flexible generalization styles, and b) mechanisms to guide the system into the best style of generalization for the problem being learned.

In PASOCS, the style of generalization can be both arbitrary and potentially dynamic. Generalization techniques being studied with the PASOCS model include hamming distance, percentage mismatch, weighted hamming, subsets of critical variables, confidence levels (allowing both noisy and training set style learning), and structural prototype generalization. Schemes which guide the automatic discovery of the best combination of generalization mechanisms are also part of current ASOCS research. Discussion of these particular concepts is beyond the scope of this paper.

3. Definitions

Propositional Rules: The atomic knowledge element in ASOCS is the instance (rule). Each instance is a (partial) function from a set of Boolean variables to a Boolean variable. An instance consists of a left hand side and a right hand side. Following are four examples of instances:

I. a' b' => z
II. a b' c => a'
III. a' b' => z'
IV. c d e => z

Instance Variables: The variables that appear in the left hand side of an instance are called *instance input variables*. The variable that appears in the right hand side of an instance is called the *instance output variable*.

Semantics of an Instance: Instance I forces z to become true whenever a and b are false. It says *nothing* about what z should be when a or b are true. Instance II forces a to become false whenever a and c are true and b is false. Instance III forces z to become false whenever a and b are false.

Matching and Firing: A set of Boolean values J matches (fires) an instance I if and only if:

for every literal v in the left hand side of I,
    if v is uncomplemented, then
        the variable v has the value true (1) in J
    else if v is complemented then
        the variable v has the value false (0) in J.

For example, the input a=0, b=0, c=1, d=1, and e=1 matches instances I, III, and IV above, but does not match instance II.

Polarity: An instance with a non-complemented output variable is said to have *positive polarity*; an instance with a complemented output variable is said to have *negative polarity*. Instances I and IV have positive polarity; instances II and III have negative polarity. Two instances with the same polarity are *concordant* with respect to each other. Two instances with opposite polarity are *discordant.*

Inconsistent: An instance I1 and an instance I2 are *inconsistent* if and only if 1) I1 and I2 are discordant, and 2) there exists a set of Boolean values J that fires both I1 and I2. Instances I and III are inconsistent: 1) they are discordant and 2) when a and b are false, instance I sets c to true, while instance III sets c to false.

Discriminant variable: Let I1 and I2 be two instances; let v be a Boolean variable. If v is an input variable of I1 and v' is an input variable of I2 (or vice versa), then v is said to be a *discriminant variable* for I1 and I2. Instances I and II have a as a discriminant variable. Neither b nor c is a discriminant variable for instances I and II.

The following theorem gives a simple computational test for inconsistency in terms of polarity and discriminant variables.

Theorem 1: Two instances are inconsistent if and only if they are of opposite polarity (discordant) and have no discriminant variable.

Prioritized Instances: A *prioritized instance* is an instance with an associated positive integer called its priority.

Instance Set: A set S of prioritized instances is called an *instance set*. Any subset of an instance set is also an instance set.

The sets S(J), S^+(J) and S^-(J): Let S be an instance set. Let J be a set of Boolean values. Let S(J) be the set of instances in S that J fires. Let S^+(J) be the set of instances in S(J) with positive polarity. Let S^-(J) be the set of instances in S(J) with negative polarity.

The MaxPriority function: Let S be an instance set. Define *MaxPriority(S)* as follows:

    if S is non-empty, then
        return the highest priority of any instance in S
    else if S is empty then
        return 0.

If S(J) is empty (no instance matches J), then MaxPriority(S(J)) = 0.

Consistency of an Instance Set: An instance set S is *consistent* if and only if for all sets of Boolean values J, and for all pairs of discordant instances I1 and I2 of S, if J fires I1 and I2, then priority(I1) $\neq$ priority(I2).

The following two lemmas relate consistency and the MaxPriority function.

Lemma 1: An instance set S is consistent if and only if for every set of Boolean values J, whenever MaxPriority(S(J)) is positive, then MaxPriority(S^+(J)) does not equal MaxPriority(S^-(J)).

Lemma 2: Let S be a consistent instance set and J be a set of boolean values. If MaxPriority(S(J)) is positive, then all instances in S that match J and have priority equal to MaxPriority(J) have the same polarity.

Execution: We now introduce the (partial) function MAP(S,J), which returns the polarity of the highest priority instance that is matched in the instance set. Let S be a consistent instance set and J be a set of Boolean variables. Define MAP(S,J) as follows:

MAP(S,J):
Set m to MaxPriority(S(J))
if m = 0, then MAP(S,J) is undefined
else MAP(S,J) is the polarity of any instance in S(J) with priority m.

Theorem 2: MAP(S,J) is a well defined partial function.

Proof: Let J be any set of Boolean values and let m be MaxPriority(S(J)). If m = 0, then MAP(S,J) is undefined. If m > 0, the set of all instances that have priority m is non-empty. By lemma 2, each such instance has the same polarity. Thus, when m > 0, MAP(S,J) is well defined. This concludes the proof of the theorem.

Note that when m=0 (MAP undefined) a generalization scheme such as mentioned in section 2 can be used to give an output.

Fulfillment: A network is said to *fulfill* an instance set S if and only if for every set of boolean inputs J, the network computes MAP(S,J). When MAP(S,J) is undefined, the network may generalize and set the output variable to true, false, or don't know.

Subsumption: Let I1 and I2 be instances. We say that I1 *subsumes* I2, if the variables of I1 are a subset of the variables of I2. If I1 subsumes I2, then whenever I2 fires, I1 must fire.

One-Difference Subset: Let I1 and I2 be two instances that share exactly one discriminant variable v. We say I1 is *one-difference subset* with I2 if the variables of I1 - {v,v'} are a proper subset of the variables of I2 - {v,v'}. For example, ab => z' is one-difference subset with ab'c => z.

One-Difference Equal: Let I1 and I2 be two instances that share exactly one discriminant variable v. We say I1 is *one-difference equal* with I2 if the variables of I1 - {v,v'} are the same as the variables of I2 - {v,v'}. For example, ab => z is one-difference equal with ab' => z.

One-Difference Superset: Let I1 and I2 be two instances that share exactly one discriminant variable v. We say I1 is *one-difference superset* with I2 if the variables of I1 - {v,v'} are a proper superset of the variables of I2 - {v,v'}. For example, a'bc => z' is one-difference superset with ab => z. Clearly, I1 is one-difference subset with I2 if and only if I2 is one-difference superset with I1. Circumstances determine which we emphasize.

4. Learning Algorithm

In this section we introduce a specific learning algorithm for priority ASOCS. A more rigorous presentation with proofs of all learning rules is be found in [3].

We assume that a network N has been created that fulfills an instance set IS. Suppose that a new instance (NI) is presented to the network. The network must reconfigure itself to fulfill the new instance set, which is the combination of the old IS and the NI. The learning algorithm defines how this is done in a self-organizing fashion.

When a NI is presented to the system, each OI independently and in parallel (see section 5) compares itself with the NI. Depending on how it matches the NI, the OI may

a) require no modification,
b) be deleted,
c) be modified,
d) cause the NI to be deleted
e) cause the NI to be modified.

The basic actions which take place in the learning algorithm are:

1) Resolve any inconsistencies caused by the NI (Section 4.1)
2) Generalize and simplify by:
    a) removing basic redundancies (Section 4.2)
    b) removing one-difference redundancies (Section 4.3)

Section 4.4 gives the actual learning algorithm, using the mechanisms presented in sections 4.1-4.3.

4.1 Resolution of New Instance Inconsistency

A NI may be inconsistent with one or more OI's in the IS. One solution would be to give the NI a priority greater than any priority in the IS. However, to restore consistency, it is sufficient to give the NI a priority one greater than the inconsistent OI having the highest priority. For example, assume the new instance a c => z' is added to the IS:

I.	a b => z	(2)
II.	b c' => z	(4)

(The priority of each instance is shown in parenthesis to the right of the instance. In general NI.P stands for the priority of the new instance and OI.P the priority of a specific old instance.)

The NI and instance II are consistent since c is a discriminant variable. But instance I (with priority 2) is inconsistent with the NI since there is no discriminant variable. Therefore, if the NI is added with priority 3, the consistency of the instance set is restored.

4.2 Removing Basic Redundancies

This section presents two mechanisms for removing instances from the instance set.

M1. If the OI is subsumed by the NI, then the OI can be deleted from the IS.

Note that the OI deletion does not depend on the priority or polarity of the NI or OI. Let (*) represent an instance priority that can take on any value. We illustrate this mechanism by example. Assume that new instance a => z is added to the following instance set:

I.	a b => z	(*)
II.	a => z'	(*)
III.	a b' c => z'	(*)
IV.	b => z	(*)

Since instances I, II, and III are subsumed by the NI, mechanism M1 allows them to be deleted from the instance set.

M2. If 1) OI and NI are concordant, 2) O1 subsumes the NI, and 3) OI.P ≥ NI.P, then the NI can be deleted.

As noted in section 4.1, it suffices to set the priority of the NI to one greater than the inconsistent OI with the highest priority. Thus, if a concordant old instance O1 subsumes the NI, and if O1 has priority greater than the NI, then O1 already covers all cases where the NI could be matched. The NI therefore adds no new information to the IS, and can be discarded.

We illustrate mechanism M2 by example. Assume the new instance a b c => z is added to the instance set:

I. a b => z' (3)
II. a c => z (5)

The NI would be assigned the priority of 4 since it is inconsistent with instance I. Since the NI is subsumed by the concordant instance II and since instance II has a greater priority, mechanism M2 lets the NI be deleted. The new instance a b c => z adds no new information since instance II already covers that case.

On the other hand, if I1 is a concordant OI which subsumes the NI, and if I1 has priority lower than the NI, then there exists at least one discordant inconsistent instance I2 (having priority less than the NI, but greater than I1), to which the NI is an exception.

We illustrate this case by example. Assume the same new instance a b c => z is added to the instance set:

I. a b => z' (7)
II. a c => z (5)

The NI would be assigned the priority 8 since it is inconsistent with instance I. The NI is subsumed by the concordant instance II. But the assigned NI priority is greater than that of instance II. Therefore, M2 does not apply, and the NI is added to the instance set. The new instance abc => z is an exception to instance I when a=1, b=1, c=1. Instance II still contains information since it sets z to true when a=1, b=0, and c=1, a case not covered by either instance I or the NI.

4.3 Removing One-Difference Redundancies

In this section we state and illustrate two types of mechanisms. We first present M3.1, M3.2a, M3.2b, and M3.3, mechanisms for simplifying one difference *concordant* instances. Then we present M4a and M4b, mechanisms for simplifying one difference *discordant* instances.

Mechanisms M3.1-M3.3 permit the removal of discriminant variables when the NI is one-difference concordant with an OI.

M3.1) If the NI and OI are concordant, and the OI is one-difference superset with respect to the NI, then the discriminant variable can be deleted from the OI.

We illustrate mechanism M3.1. Assume the new instance a b => z is added to the instance set

I. a b' c => z (*)

Since I and NI are concordant and the instance I is one-difference superset with respect to the NI, mechanism 3.1 allows us to delete the discriminant variable b from I. The old instance a b' c => z becomes a c => z.

M3.2a) If the OI and NI are concordant, the OI is one-difference equal with respect to the NI, and OI.P $\leq$ NI.P, then the discriminant variable can be deleted from the OI.

We illustrate mechanism M3.2a. Assume the new instance a b c => z is added to the instance set

I. a b' c => z (3)
II. c d => z' (4)

The NI is inconsistent with instance II and is assigned priority 5. Since instance I and the NI are concordant one-difference equal, and the priority of the NI is greater than the priority of instance I, then by M3.2a the discriminant variable b' can be removed from instance I. The old instance a b' c => z becomes a c => z.

M3.2b) If the OI and NI are concordant, the OI is one-difference equal with respect to the NI, and OI.P ≥ NI.P, then 1) the discriminant variable can be deleted from the OI and 2) the NI itself can be deleted.

We illustrate mechanism M3.2b. Assume the new instance a b c => z is added to the instance set

I. a b' c => z (6)
II. c d => z' (4)

The NI is inconsistent with instance II and is assigned priority 5. Since instance I and the NI are concordant one-difference equal, and the priority of the instance I is greater than that of the NI, then by M3.2b 1) the discriminant variable b' can be removed from instance I and 2) the NI can be deleted. [Since instance I has higher priority than the NI, and since deleting the discriminant variable from instance I causes it to subsume the NI, the NI adds no new information.]

M3.3) If the OI and NI are concordant, the OI is one-difference subset with respect to the NI, and OI.P ≥ NI.P, then the discriminant variable can be deleted from the NI.

We illustrate mechanism M3.3. Assume the new instance a b c => z is added to the instance set

I. a b' => z (6)
II. c d => z' (4)
III. a c' => z (6)

The NI is inconsistent with instance II and is assigned priority 5. Since instance I and the NI are concordant, the NI is one-difference superset with respect to instance I, and the priority of the instance I is greater than that of the NI, then by M3.3 the discriminant variable b can be removed from the NI.

Note that instance III also fulfills the requirements to simplify the NI (namely, it is concordant one-difference subset with higher priority). However, simplification by M3.3 can be done with respect to only one OI. Either I or III could have been used for the simplification. If III had been chosen rather than I, the discriminant variable c would have been removed from the NI. We show how the restriction of at most one application of M3.3 is supported in an actual architecture in section 5.

We now present a case where M3.3 does not apply. Assume the new instance a b c => z is added to the instance set

I. a b' => z (3)
II. c d => z' (4)

The NI is inconsistent with instance II and is assigned priority 5. Instance I and the NI are concordant; the NI is one-difference superset with respect to instance I, but the priority of the NI is greater than that of instance I. In this case no removing action can take place since the NI is an exception to instance II. This makes sense since the NI's higher priority adds information.

We now state and illustrate two simplification mechanisms for one-difference *discordant* instances.

M4a) If the OI and NI are discordant, the OI is one-difference equal with respect to the NI, and OI.P ≤ NI.P, then the discriminant variable can be deleted from the OI.

We illustrate mechanism M4a. Assume the new instance a b => z is added to the instance set

I. a b' => z' (4)
II. a' b => z' (7)
III. c d => z' (4)

The NI is inconsistent with instance III and is assigned priority 5. Instance I and II are one-difference equal with respect to the NI. By M4a, the discriminant variable b' can be removed from instance I. [No change is made to instance II, since instance II has higher priority than the NI.]

M4b) If the OI and NI are discordant, the OI is one-difference superset with respect to the NI, and OI.P ≤ NI.P, then the discriminant variable can be deleted from the OI.

We illustrate mechanism M4b. Assume the new instance a b => z is added to the instance set

I. a' b c => z' (3)

II. a' b d => z' (7)

The NI is inconsistent with instance III and is assigned priority 5. Instance I and II are one-difference superset with respect to the NI. By M4a, the discriminant variable a' can be removed from instance I. [No change is made to instance II, since instance II has higher priority than the NI.]

4.4 Instance Set Learning Algorithm

Before giving the learning algorithm we define a mechanism for how the OI matches the NI. This matching is specified by the 3-tuple (Polarity, Discriminants, Set Relation) with each value taken from the following disjoint sets:

Polarity = *{Concordant, Discordant}*
Discriminants = *{NoDiscriminant, OneDiscriminant, Many}*
Set Relation = *{Subset, Equal, Superset, None}*

Let OI be an old instance and NI be a new instance. Their *matching tuple* is the three-tuple (polarity, discriminants, set relation). It is assigned as follows. The polarity is concordant, if the two instances are concordant, discordant otherwise.

The discriminants is NoDiscriminant, if the two instance have no discriminant variable, OneDiscriminant if the two instances share exactly one discriminant variable, and Many otherwise.

If discriminants is NoDiscriminant, then set relation is:

Subset: if the variables of the OI are a proper subset of the variables of the NI

Equal: if the variables of the OI are equal to the variables of the NI.

Superset: if the variables of the OI are a proper superset of the variables of the NI.

None: otherwise

If discriminants is OneDiscriminant then set relation is:

Subset: if except for the discriminant variable, the variables of the OI are a proper subset of the variables of the NI

Equal: if, except for the discriminant variable, the variables of the OI are equal to the variables of the NI.

Superset: if except for the discriminant variable, the variables of the OI are a proper superset of the variables of the NI.

None: otherwise

If discriminants is Many, then set relation is None.

For example, the new instance a b => z would cause the following old instances to have the following associated matching tuples.

a b c => z	{Concordant, NoDiscriminant, Superset}
a b => z'	{Discordant, NoDiscriminant, Equal}
a' b c => z'	{Discordant, OneDiscriminant, Superset}

d e => z {Concordant, NoDiscriminant, None}

We restate M1-M4 in terms of these triples.

M1: if OI is {*, NoDiscriminant, Superset or Equal} with the NI, then the OI can be deleted.

M2: If OI is {Concordant, NoDiscriminant, Subset or Equal} with the NI and OI.P ≥ NI.P, then the NI can be deleted.

M3.1: If OI is {Concordant, OneDiscriminant, Superset} with the NI, then the OI's discriminant variable can be deleted.

M3.2a: If OI is {Concordant, OneDiscriminant, Equal} with the NI and OI.P ≤ NI.P, then the OI's discriminant variable can be deleted.

M3.2b: If OI is {Concordant, OneDiscriminant, Equal} with the NI and OI.P ≥ NI.P, then 1) the OI's discriminant variable can be deleted, and 2) the NI can be deleted.

M3.3: If OI is {Concordant, OneDiscriminant, Subset} with the NI and OI.P ≥ NI.P, then the NI's discriminant variable can be deleted.

M4a: If OI is {Discordant, OneDiscriminant, Superset} with the NI and OI.P ≤ NI.P, then the OI's discriminant variable can be deleted.

M4b: If OI is {Discordant, OneDiscriminant, Equal} with the NI and OI.P ≤ NI.P, then the OI's discriminant variable can be deleted.

Having discussed the matching notation and the mechanisms of sections 4.1-4.3, we present the basic learning algorithm. The italicized text gives the high level description of each step. The plain text gives an algorithmic instantiation of the steps.

1. *Present the new instance (NI) to the system.*
   Each OI assigns its matching tuple with respect to the NI.
2. *Determine the priority of the NI*
   If an OI is {Discordant, NoDiscriminant, *}, then it presents its priority to the system, else the OI presents priority 0 to the system.
   Let the highest such priority be called TopPriority.
   The NI's priority, (NI.P), is set to TopPriority + 1.
3. *Determine whether the NI is redundant. If so, then the NI is <u>not</u> added to the instance set and no further processing occurs; otherwise, continue at step 4.*

   If an OI is {Concordant, NoDiscriminant, Subset or Equal} and OI.P ≥ NI.P, then return without adding the NI to the instance set. (M2)
4. *Delete subsumed OIs.*

   If an OI is {*, NoDiscriminant, Superset or Equal}, then the OI self-deletes. (M1)
5. *Generalize concordant one-difference superset OIs.*

   If an OI is {Concordant, OneDiscriminant, Superset}, then
   the OI removes the discriminant variable from its variable list. (M3.1)
6. *Generalize concordant one-difference equal OIs.*

   If an OI is{Concordant, OneDiscriminant, Equal}, and OI.P ≤ NI.P, then
   the OI removes the discriminant variable from its variable list. (M3.2a)

   If an OI is {Concordant, OneDiscriminant, Equal}, and OI.P ≥ NI.P, then
   1) the OI removes the discriminant variable from its variable list and 2) the NI is deleted. (M3.2b)
7. *Generalize the NI by concordant one-difference subset OIs.*

Let T = {OI: OI is {Concordant, OneDiscriminant, Subset} and OI.P ≥ NI.P}. If T is non-empty, then choose any instance I of T and remove the discriminant variable of I from the NI (M3.3).

8. *Generalize discordant one-difference OIs.*
   If an OI is {Discordant, OneDiscriminant, Superset or Equal}, and OI.P ≤ NI.P, then the OI removes the discriminant variable from its variable list. (M4a and M4b)

9. *Add the NI to the instance set.*

   Add the NI with its assigned priority NI.P.

For simplicity, we have presented the above steps as a sequential algorithm. Many of the operations can take place in parallel and we present a parallel version of the algorithm after overviewing the PASOCS architecture in section 5.

5. Prioritized ASOCS Architectural Model

In this section we give a simple prioritized ASOCS architecture, its associated execution algorithm, and its associated learning algorithm. Emphasis has been placed on ease of explanation rather than efficiency.

5.1 Interconnect and Node Requirements

The Interconnect: We choose a binary tree for our basic topology. Consider the n-layered tree with $N = 2^{n-1}$ leaf nodes and N-1 junction (internal) nodes shown in figure 7. A separate message controller exists outside the tree, called the *adaption unit (AU).* Communication flows up and down the tree in *O(log n)* where *n* is the number of nodes in the network.

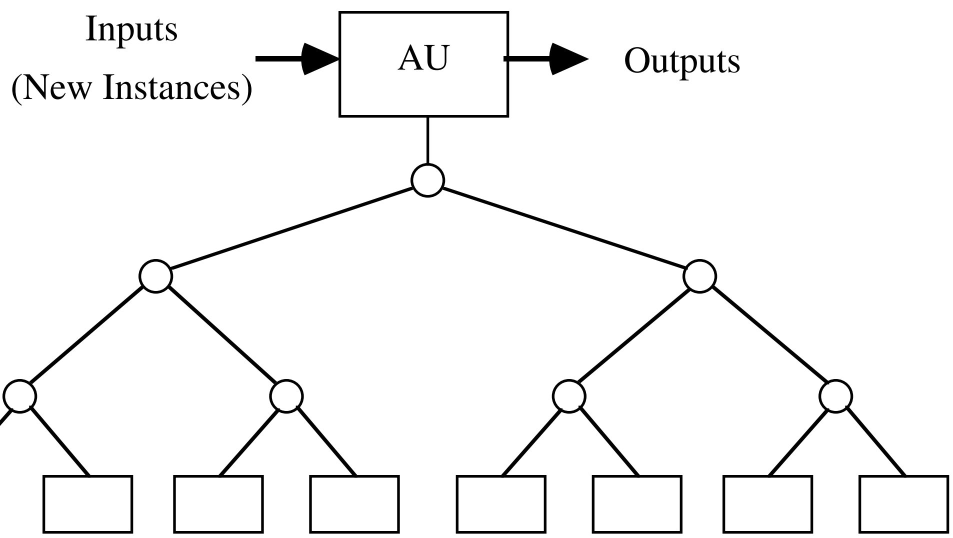

Figure 7 - PASOCS Network

Leaf Nodes: PASOCS nodes (the nodes that store the rules) are the leaf nodes of the tree.

Junction Nodes: The internal nodes of the binary tree are junction nodes. They can transmit a message, store a messages, and compare a message against a stored value. Junction nodes have no rule bearing capacity in this particular architectural instantiation.

Downward Messages: To send a message to all PASOCS nodes, the controller creates a downward message and sends it to the top junction node. When a junction node receives a downward message, it duplicates the message, and broadcasts it to each of its children. When a PASOCS (leaf) node receives a downward message, it reads and executes the message.

Upward messages: To send a message to the controller, a PASOCS node creates an upward message, attaches its internal priority, and sends the message to its junction (parent) node. Each junction node checks messages from its children, determines which message has higher priority, sends the higher priority message up the tree, and discards the other message. In case of a tie, one message is arbitrarily chosen and sent.

Adaption Unit (AU): This unit communicates directly with the root node of the network. During execution, the AU receives the input variables and sets the output variables. During learning, the AU receives new instances, sends messages to the network, and receives messages from the network.

PASOCS Nodes: The PASOCS nodes are the leaf nodes of the tree. Each PASOCS leaf node contains:

1) a variable list (the list of input variables over which the node does a conjunction),

2) a priority field,

3) a polarity flag,

4) a matching triple: (Polarity, Discriminants, Set Relation).

Initialization: All leaf node priorities are set to 0 initially, indicating that the leaf node is free. An *active* leaf node has a positive priority. Each active leaf node in the tree represents an instance.

5.2 Execution Algorithm

During execution, a set of Boolean values J is input to the AU and broadcast down the tree. Each leaf node executes the same simple algorithm:

If the input J matches a leaf node's variable list, then the leaf node sends the upward message consisting of the node's polarity and priority.

The highest priority message bubbles to the top of the tree in log time. If the winning message has non-zero priority and positive polarity, then the system outputs true. If the winning message has non-zero priority and negative polarity, then the system outputs false. If the winning message has zero priority, then the output is system dependent; it could output a "?" or a generalized value.

5.3 Learning Algorithm Functions

This section defines the basic operations which can be initiated by the adaption unit (AU) during learning. The (AU) initiates six types of messages: *BroadcastNI, GetPriority, SendPriority, Update, FindAvailableNodes,* and *CreateNode.*

BroadcastNI(NI):

BroadcastNI transmits the NI's variable list and the NI's polarity allowing each leaf node to assign its matching tuple. The logic for a leaf node executing the BroadcastNI message is: if active, then assign one flag from each the following three disjoint sets:

{Concordant, Discordant}
{NoDiscriminant, OneDiscriminant, Many}
{Subset, Equal, Superset, None}

otherwise, do nothing.

The flag's values indicate the OI's relationship to the NI, in the same manner as presented in section 4.4.

GetPriority():

GetPriority allows the AU to determine the highest priority of any node which contradicts the NI. The logic for a leaf node executing the GetPriority message is: If the node is active and

{Discordant, NoDiscriminant,*}, then it sends an upward message of the node's priority, otherwise it sends an upward message of 0. Each junction node passes on the highest priority and the root node passes the highest priority to the AU.

SendPriority(NI.P):

SendPriority lets the AU inform the network of the assigned priority of the new instance NI.P. (The AU sets NI.P to one greater than the value returned by GetPriority.) The logic for a leaf node executing the SendPriority(NI.P) message is: if active, then store a copy of NI.P, otherwise do nothing.

Update()

Update allows the nodes to do independent modification using mechanisms M1-M4 as introduced in section 4. These actions allow the system to become consistent with the updated instance set. The logic for a leaf node executing the Update message is contained in section 5.5. Each node generates an upward message during Update as explained in section 5.4. The AU accepts a single highest priority message as part of the Update.

FindAvailableNodes()

FindAvailableNodes marks unused nodes. The logic for a leaf node executing the FindAvailableNodes message is: If the node is active or inactive, send node.P (the node's priority) in an upward FindAvailableNode message. The logic for a junction node executing an upward FindAvailableNode message is: choose the *lowest* priority of either child and mark which child sent the lowest priority (ties are broken arbitrarily). The AU receives the lowest priority in the network. If the priority is positive, then there are no inactive nodes left in the network and the AU signals an error.

CreateNode(NI)

CreateNode causes a new node to be allocated for the NI. If the result of FindAvailableNodes is a priority of 0, then the AU initiates the CreateNode message, sending the NI as a parameter. The logic for a junction node executing the CreateNode message is: forward the message to the marked child. The message is guaranteed to arrive at a single inactive leaf node. The logic for the leaf node executing the CreateNode message is: record the NI's variable list, polarity, and priority. At this point the node is active and represents the NI.

5.4 A Parallel PASOCS Learning Algorithm

The basic PASOCS learning algorithm was presented in section 4.4. Section 5 has proposed a specific architecture on which to implement the algorithm. This section illustrates how the architecture supports the learning algorithm of section 4.4 in a parallel and distributed fashion. The architectural learning algorithm uses the operations defined in section 5.3. The basic algorithm is shown in italics, with each step followed by comments.

1. *BroadcastNI(NI);*

   The BroadcastNI message is sent down the tree and each active node assigns its matching three-tuple.

2. *GetPriority();*

   The GetPriority message is sent down the tree. Each contradictory node sends its priority in an upward message. The AU receives P, the highest contradictory priority. The AU assigns NI.P the value P + 1.

3. *SendPriority(NI.P);*

The SendPriority message is sent down the tree with the value NI.P. Each active leaf node stores NI.P.

4. *Update;*

   Each node does independent adaptation depending on NI.P and its matching tuple. The independent updates are discussed in section 5.5. Some actions in Update require modification of the NI. To support this, all active leaf nodes send one of the following three upward messages during update:

   DeleteNI: Causes the NI to be deleted (not added).

   GeneralizeNI(DV): The discriminant variable passed in the message is deleted from the NI.

   NoChange: No NI modification takes place.

   A junction node passes on only one message. At a junction, DeleteNI has priority over GeneralizeNI, which has priority over NoChange. Ties are broken arbitrarily.

5. *If Update message = DeleteNI, then exit algorithm,*

   else If Update message = GeneralizeNI(DV), then remove the discriminant variable DV from the NI.

6. *FindAvailableNodes().*

   Inactive nodes are marked preparatory to CreateNode.

7. *CreateNode(NI).*

   An inactive node is allocated for the NI.

After the learning algorithm is finished, the system can either add another NI or can go to execution mode.

5.5 Update

During the Update phase (step 4 of section 5.4), each node executes an independent and parallel update, depending on the state of NI.P and the matching triple. The parallel actions correspond to the sequential actions 3-9 of section 4.4 All node actions occur simultaneously.

If the node is {Concordant, NoDiscriminant, Subset or Equal} and node.P ≥ NI.P, then send an upward DeleteNI message. [In this case the NI is already fulfilled and will be deleted. (M2)],

else if the node is {, NoDiscriminant, Superset or Equal}, then self-delete (set priority to 0) and send the upward NoChange message*. (M1),

else if the node is {Concordant, OneDiscriminant, Superset}, then self-generalize by removing the discriminant variable from the node variable list and send the upward NoChange message. (M3.1),

else if the node is {Concordant, OneDiscriminant, Equal}, then self-generalize by removing the discriminant variable. (If node.P ≥ NI.P then send the upward DeleteNI message, else send the upward NoChange message.) [The node always removes the discriminant variable in this case, regardless of its priority. Also, if node.P ≥ *NI.P*, the NI will be deleted. (M3.2a and M3.2b)],

else if the node is {Concordant, OneDiscriminant, Subset} and node.P ≥ NI.P, then send the upward GeneralizeNI(DV) message. [If the GeneralizeNI message reaches the AU, the NI will be generalized by removing the discriminant variable. (M3.3)], *else if the node is {Discordant, OneDiscriminant, Superset} and node.P ≤ NI.P, then self-generalize by removing the discriminant variable from the node variable list and send the upward NoChange message.* (M4a and M4b),

else send the upward NoChange message.

6. PASOCS Example

This section illustrates how PASOCS learns. An inactive leaf node (priority=0) is shown as empty. We assume the network begins without any active nodes and that the following seven instances are entered in the given order:

A. abd' => z
B. abc' => z'
C. abc'd => z'
D. a'bc'd => z
E. bc' => z
F. a'b'd' => z
G. ab'd' => z'

A. Since there are no instances in the system initially, the system assigns abd' => z the priority 1 and adds it to the network. (See figure 8).

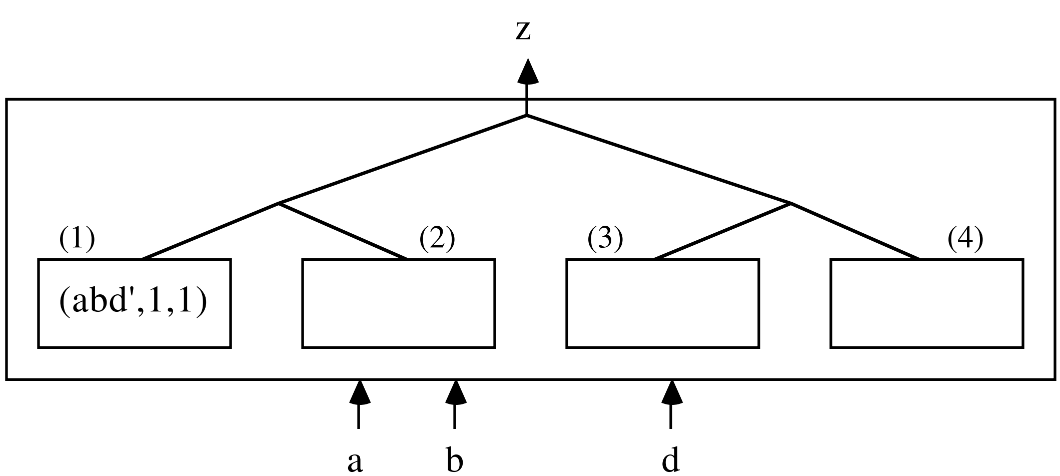

Figure 8 - After First Instance

B. Since abc' => z' is inconsistent with abd => z of node 1 with priority 1, the system assigns abc' => z' the priority 2. No other action takes place and the NI is added to the network. (See figure 9)

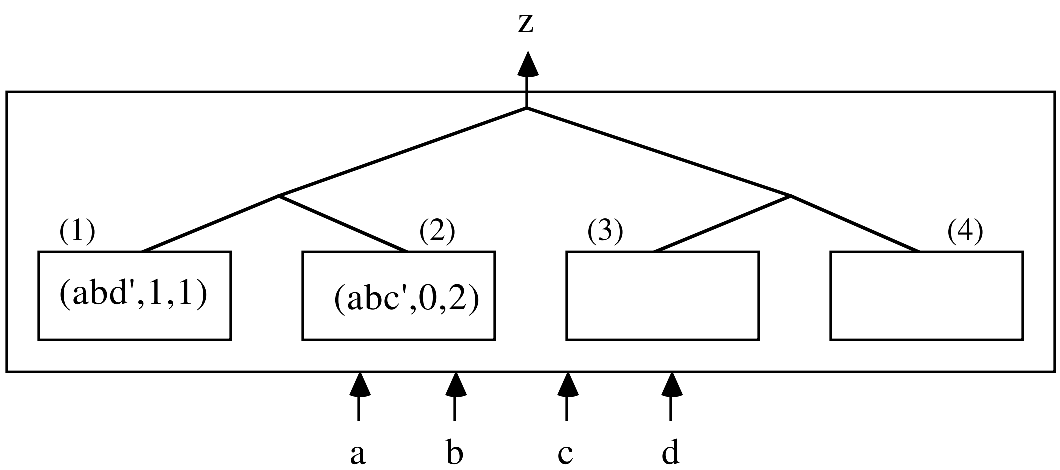

Figure 9 - Modified Network

C. Since abc'd => z' is consistent with every node in the network, the system assigns abc'd => z' the priority 1. Since node 2 (abc' => z') is {Concordant, NoDiscriminant, Subset} with respect to the NI abc'd => z', and since the priority of node 2 is greater than the priority of the NI, the NI is deleted by a DeleteNI message during Update. The network remains as in figure 9.

D. Since a'bc'd => z is consistent with every node in the network, the system assigns a'bcd' => z priority 1. No other action takes place and the NI is added to the network. (See figure 10)

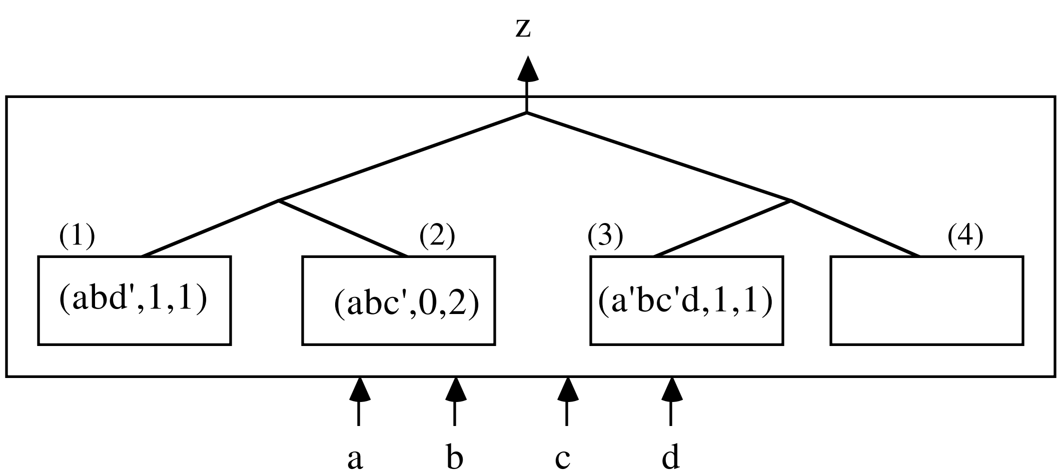

Figure 10 - Modified Network

E. Since bd' => z is inconsistent with abc' => z' (node 2) with priority 2, the system assigns bd' => z the priority 3. Node 1 (abd' => z) is {Concordant, NoDiscriminant, Superset} with respect to the NI. Thus, it is subsumed by the NI and self-deletes by setting its node.P to 0.

Node 3 (a'bc'd => z) is {Concordant, OneDiscriminant, Superset}. Thus node 3 generalizes itself by removing the discriminant variable d. (See figure 11).

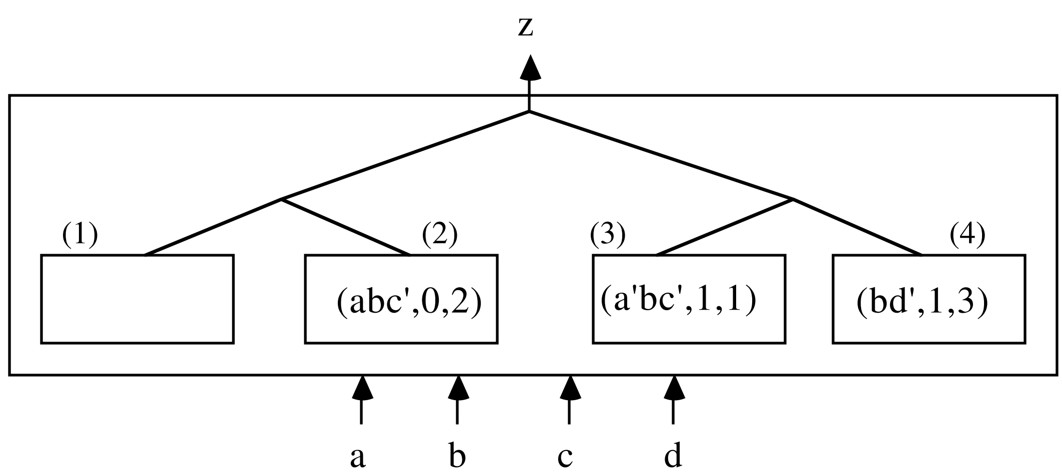

Figure 11 - Modified Network

F. Since a'b' => z is consistent with every node in the network, the system assigns it the priority 1. Since node 3 (a'bc' => z) is {Concordant, OneDiscriminant, Superset} with respect to the NI, node 3 generalizes itself by removing the discriminant variable b. (See figure 12)

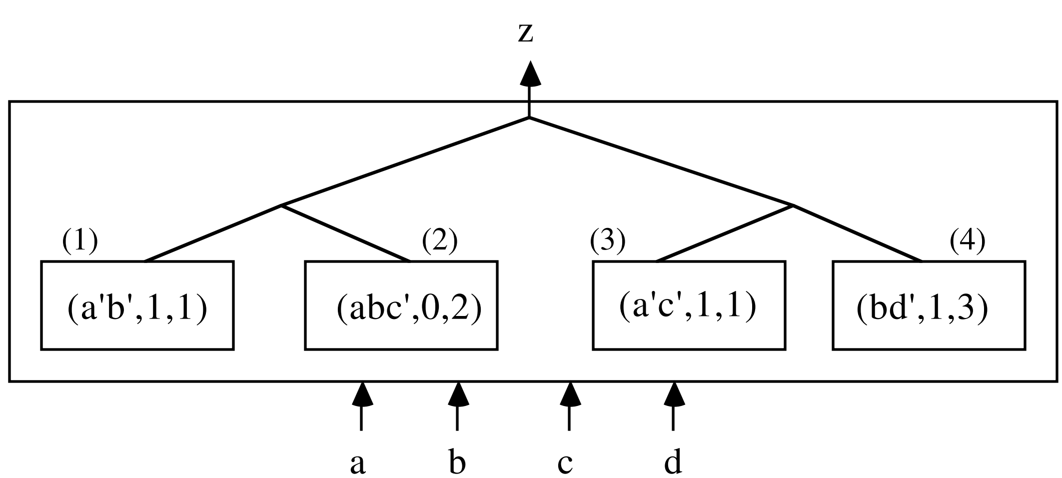

Figure 12 - Modified Network

G. Since ab'c' => z' is consistent with every node in the network, the system assigns it the priority 1. Because node 2 (abc' => z') is {Concordant, OneDiscriminant, Equal} with respect to the NI, it generalizes itself by removing the discriminant variable b. Also, since the priority of node 2 is greater than the priority of the NI, it sends the DeleteNI message during update. The DeleteNI message lets the AU discard the NI. The final network is shown in figure 13

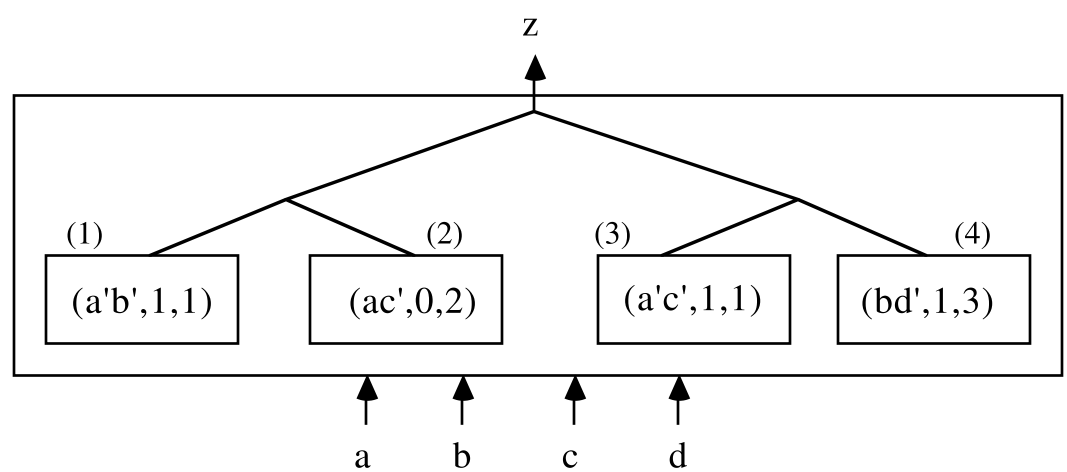

Figure 13 - Final Network

7. Multiple Outputs

Up to this point the PASOCS discussion has been limited to a single output variable. We want to extend the system to permit multiple outputs, for example:

I. a b => x
II. a c' => y'
III. b c => z
IV. a b' d => z
V. a b' d => x'

Of course, one implementation could use a separate PASOCS for each output variable. However, more efficient use of hardware can be attained if nodes share responsibility for multiple outputs.

One extension augments the node's variable list and learning flag set with a separate polarity and priority for each output variable it represents. Thus, a node may have positive polarity for x, negative polarity for y, and be undefined for other variables. Figure 14 shows a node representing the output variables z and x for instances IV and V above. Assume they have the priorities 5 and 3 respectively.

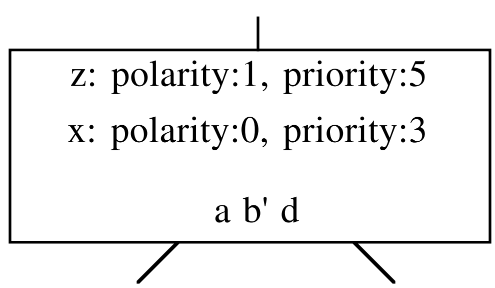

Figure 14 - Multiple Output Node

During execution, if the node of figure 14 is matched, then its upward message contains information for both output variables. In this case the message could be {(z,1,5), (x,0,3)}.

During learning, the BroadcastNI message is augmented with the new instance's output variable. For example, if the new instance ab' => x is entered, then when the node of figure 14 receives the message BroadcastNI *and* x, the node assigns its matching triple to {Discordant, NoDiscriminant, Superset}. Therefore, by M4b, it can delete its x output variable reference.

However, the node still retains its variable z values. When all output variable fields for a node are deleted, then the node itself can self-delete.

As a second example, assume we have the original node of figure 14 and the new instance ab'd => y is added to the network. The node of figure 14 can send an upward message indicating that it matches the variable list of the NI. Then, if a later *CreateNode* function is issued by the AU, the node can simply add the new polarity and priority for variable y to its variable fields, with no new node allocation required.

Though informal, this section gives the basic flavor of an extension to support multiple outputs.

8. Conclusion

We have introduced the Priority ASOCS model and have described one possible PASOCS architecture. The significant features of PASOCS include a) guaranteed learning of any set of rules and b) fast execution and learning times (*O(log(n)),* where *n* is the number of rules). The advantages of PASOCS over previous ASOCS models are 1) at most one node is added for each instance learned and 2) the execution and learning time bounds are based on the number of rules, rather than the number of nodes as in other models. Current work with PASOCS and other ASOCS models is concentrating on a) improved generalization for inputs not explicitly represented in training, b) extension to analog and higher-order data, c) integration of rules and examples, d) applications, and e) physical implementation studies.

9. Bibliography

1. Backus, J., Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, **21** (1978), 613-641.

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

3. Hughes, B., Prioritized Rule Systems, MS Thesis, Brigham Young University, 1990.

4. Martinez, T. R., Adaptive Self-Organizing Logic Networks, PhD Dissertation, University of California Los Angeles, Los Angeles, California, 1986.

5. Martinez, T. R., Adaptive Self-Organizing Concurrent Systems, *Progress in Neural Networks*, **vol. 1**, Ch. 5, 105-126, Ablex Publishing, 1990.

6. Martinez, T. R. and Vidal, J.J., Adaptive Parallel Logic Networks, *Journal of Parallel and Distributed Computing*, **vol. 5**, 26-58, 1988.

7. Martinez, T. R. and Campbell, D. M., A Self-Adjusting Dynamic Logic Module, *Journal of Parallel and Distributed Computing*, **vol. 11**, no. 4, pp. 303-313, 1991.

8. Martinez, T. R. and Campbell, D. M., A Self-Organizing Binary Decision Tree for Incrementally Defined Rule Based Systems, *IEEE Transactions on System, Man, and Cybernetics,* **vol. 21**, 1991.

9. Martinez, T. R., Consistency and Generalization of Incrementally Trained Connectionist Models, Proceedings of the *International Symposium on Circuits and Systems*, pp. 706-709, 1990.

10. Minsky, M. and Papert, S., *Perceptrons, an Introduction to Computational Geometry*. MIT Press, Cambridge, MA 1969.

11. Rosenblatt, F., *Principles of Neurodynamics*. Spartan Books, Washington D. C. 1962.

12. Rumelhart, D. and McClelland J., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* **1**, MIT Press, Cambridge, MA 1986.