A Location-Independent ASOCS Model

A Thesis

Presented to the

Department of Computer Science

Brigham Young University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

George Rudolph

June 1991

This thesis by George Rudolph is accepted in its present form by the Department of Computer Science of Brigham Young University as satisfying the thesis requirement for the degree of Master of Science.

Tony R. Martinez, Committee Chairman

Dan R. Olsen, Committee Member

Date

Larry C. Christensen, Graduate Coordinator

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

This thesis proposes a new Adaptive Self-Organizing Concurrent System (ASOCS) model [Mart86] called Location-Independent ASOCS (LIA) [Rudo89]. *The location-independent properties of LIA give it improved potential for implementation using current technology over previous ASOCS models.* This thesis focuses on an architectural model and execution as well as on learning algorithms for LIA. Theoretical background, an analysis of LIA, and a brief comparison with other ASOCS models also are presented. The thesis does not discuss details of a hardware implementation of LIA beyond the model level.

ASOCS is a class of artificial neural network (ANN) models. Most other ANN's use a fixed node topology. Learning algorithms modify node functions and weights on links, but nodes and links are not added or deleted [Rum86]. In such models, **topological adaptation** (adding and deleting of nodes and links) is left to the intuition of an external programmer. Two problems associated with this style of learning are the following:

1. There is no guarantee that a particular net (a fixed topology with an associated initial set of node functions and weights) can learn a given training set.
2. Assume that a given arbitrary net can learn a fixed, but arbitrary, training set. No practical bound can be given for how long the learning will take, because the training set is presented potentially many times to the net.

ASOCS models' learning algorithms, on the other hand, use topological adaptation. ASOCS models also exhibit bounded execution and learning times, because training examples are presented only once [Mart86]. Possible applications for ASOCS include, but are not limited to, symbolic computation, adaptive logic, fault detection systems, and robotics [Hell84], [Mart89], [McDon90].

LIA is a bridge between previous ASOCS models and current technology. The LIA model uses a different computational node in its net than previous ASOCS models. The nodes in an LIA net are logically and physically independent of one another, making the nodes location-independent. This is not true of previous models. Location independence allows greater flexibility in hardware implementation than previous models, because dynamic routing is not required and static resources can be used efficiently. LIA's flexibility makes it easier and more efficient to implement using current technology than previous models. This new model, with its increased potential for hardware implementation, is the major contribution of the thesis.

The thesis is organized as follows: Chapter 2 presents motivations and background for LIA. Chapter 3 discusses instances and instance sets as the basis for the LIA model. Chapter 4 presents an architecture and the execution and learning algorithms for LIA. Chapter 5 gives an example of LIA in learning and execution modes. Chapter 6 analyzes LIA, and briefly compares LIA with other ASOCS models. Chapter 7 provides details omitted from earlier chapters for simplicity. Chapter 8 presents the conclusions of the thesis. The appendix lists test results for a simulation of LIA using data provided in chapter 9 of [Mart86] (for the interested reader).

CHAPTER 2
MOTIVATION AND BACKGROUND FOR LIA

This chapter briefly gives some background and motivation for the LIA model. The chapter has five sections:

1. Conventional Parallelism
2. Parallelism through Artificial Neural Networks
3. ASOCS Artificial Neural Networks
4. Overview of LIA Learning and Execution Modes
5. LIA's Differences and Advantages

2.1 Conventional Parallelism

The main goals in creating parallel systems are increased computational speed and ease of programming. Much work in recent years has gone into extending the *conventional* Von Neuman (fetch-and-execute) computing paradigm to handle parallel schemes [Feng81], [Hayn82], [Hill85], [Hill86], [Hwang87a], [Hwang87b], [Walt87]. Machines that use the Von Neuman paradigm in parallel are classifed as **conventional parallel** machines. Examples of these are the Connection Machine [Hill85], the Supercomputer [Hwang87a], pipelined processors and systolic arrays. Research has shown that the so-called *Von Neuman bottleneck* which plagues the single-processor system is exacerbated rather than relieved when extended to highly parallel machines. The problem becomes one of communication bottlenecks and hotspots, rather than computation [Feng81], [Hayn82], [Hwang87a]. Many algorithms for conventional parallel machines have been designed to handle these problems, with varying degrees of success and failure.

2.2 Parallelism through Artificial Neural Networks

Another class of computing models is **artificial neural networks** (ANN's) [Rum86], [Lipp87]. The models in this class do not assume the physical separation of processor and memory as the Von Neuman paradigm does. These do not use the Von Neuman paradigm as the basis for computation, seeking to avoid problems, such bottle-necking and hotspots, associated with that paradigm. ANN's have two main goals:

- speed through massive parallelism
- ease of programming, or learning, through self-organization (having the machine accomplish as much of its own programming as possible)

Other goals include the following:

- fault tolerance
- taking advantage of emerging technologies
- solving new classes of problems

Ideas for parallel computing paradigms have been considered since the inception of computers. Only recently, however, have processor and memory technologies converged to the point that their physical separation is no longer forced. Only more recently have technologies begun to develop that allow massively parallel machines with thousands of millions of processors to exist.

ANN models typically are classified according to various characteristics of their **learning algorithms**. **Learning**, in this context, means modifying the network in a self-organizing manner. There are two main features that determine what a given ANN has learned:

1. network topology
2. weights on links between nodes and activation levels of nodes

ANN models can be divided into two broad classes:

1. ANN's whose learning algorithms modify weights on links between nodes and node activation levels, but whose topology is fixed
2. ANN's whose learning algorithms modify the topology as well as the other elements

Most of the current ANN research has dealt with models in the first class, the models that maintain a fixed topology. However, it is the second class, the **topologically adaptive** models, that are a focus of this thesis. Learning algorithms like that for the Backpropagation model [Rum86] have been devised to "automate" error corrections on weights. Such models are self-organizing in the sense that error-correction takes place without external intervention. Learning algorithms can make topological adaptation self-organizing also, potentially extending the ease with which ANN's can be used to solve problems.

Most current ANN models are **gradient-descent** models, because their learning algorithms depend on having smooth, continuous, hill-climbing gradient modifications and characteristics to work properly. Although topological adaptation is of interest in its own right, its use can also help solve a learnability problem that arises in gradient-descent models: given an arbitrary initial network structure with a fixed topology, and an arbitrary but fixed set of mappings of inputs to outputs, there is no guarantee that the network can learn the mapping. (This is analogous to the classic Halting Problem in conventional computing.)

Two questions now arise. First, how does an external programmer (arbitrarily) decide that a network cannot learn a mapping? Second, what modifications can be made to the network so that the network is capable of learning the desired mapping? The answers are arbitrary in the sense that more than one set of answers could lead to a correct solution. However, if an external programmer can arrive at solutions systematically, then a topologically self-organizing network should be able to do the same.

2.3 ASOCS Artificial Neural Networks

ASOCS models incorporate topological adaptation into the learning algorithm, and do not use a gradient-descent algorithm [Mart88b], [Mart90b]. ASOCS was designed to solve the problem of programming massively parallel networks of computing nodes, and to overcome problems with the gradient-descent learning algorithms common to other ANN models. Three main models had been proposed before LIA was developed, called AA1 [Mart88a], AA2 [Mart90c], and AA3 [Mart90a] ("AA" stands for "Adaptive Algorithm").

An ASOCS machine can be thought of as a black box that accepts n boolean inputs and produces m boolean outputs (figure 1). An ASOCS net, which stores information, is

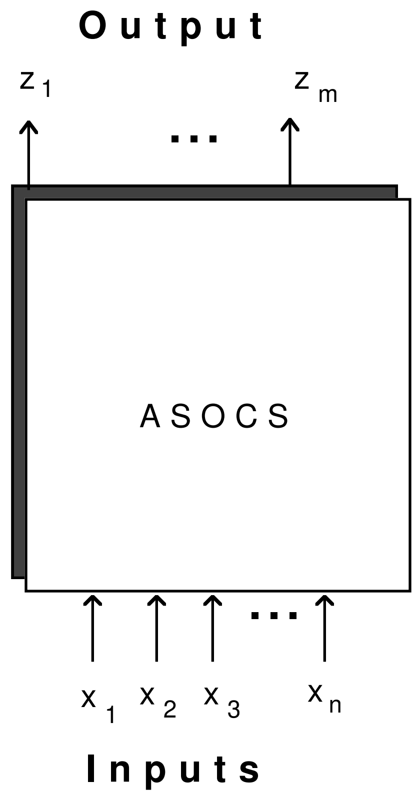

Figure 1. ASOCS Black Box Representation

composed of many simple digital nodes that operate asynchronously in parallel, and has two modes of operation--execution mode and learning mode. In execution mode, the system functions as a parallel hardware circuit, asynchronously mapping input data to output data. In learning mode, information is **incrementally** input into the net. The net then reconfigures itself to accommodate new (and possibly inconsistent) information. Through the learning process the system discovers critical features *on its own*. These features are then used to generalize and classify large input spaces.

Speed and boundedness (in both modes) are guaranteed by the algorithms and logarithmic interconnect between the nodes of the net. Self-organization comes because the net uses only the new information and information available locally to each node to decide how to reconfigure during learning.

2.4 Overview of LIA Execution and Learning Modes

Following is a brief overview of LIA execution and learning modes. This will help the reader understand the detailed explanation of the LIA model which constitutes the rest of this thesis. This overview has three parts:

1. instances
2. execution
3. learning

2.4.1 instances

An LIA net learns information in the form of an **instance**, and each node in an LIA net stores information in the form of an instance [Rudo89]. An instance is composed of two parts:

1. a conjunction of boolean input variables

2. one boolean output variable

An instance can be thought of as a cross between an inductive example and an expert-systems type rule [Mart90d]. The output variable of an instance specifies what value the net should output when the input variables are matched. Instances are written using letters, where each letter in an instance represents a distinct boolean variable. An unprimed letter indicates a **positive** value (high or true), and a primed letter indicates a **negative** value (low or false). The following are examples of instances:

1. AB ⇒ Z
2. A ⇒ Z'
3. D'EF ⇒ Z

The input variables are written on the left side of an instance, the output variable is on the right. Instance 1 specifies that when A and B are positive, Z should be positive. Instance 2 specifies that when A is positive Z should be negative. Instance 3 specifies that when D is negative and E and F are positive, Z should be positive.

The set of instances stored in a net at a given moment is called an **instance set**. An instance set represents an overall function which the net has learned. Like other ASOCS models, LIA learns instances **incrementally** (one-by-one over time). A new instance is presented *once* to a net for learning, is learned by the net, and the net waits for the next instance. This new instance may modify the overall function stored in the net in one of two ways:

1. by adding new information
2. by causing previously learned instances to become inconsistent, at which point nodes containing inconsistent instances are deleted.

Figure 2 shows one instance set stored in a LIA net. The following examples of execution and learning show how LIA supports instance learning.

2.4.2 execution

When input is presented to the LIA machine for execution, all nodes compare themselves with the input and those that match activate their output. If more than one node matches the input, they will always output the same value. Suppose the input A, B', C is presented to the net in figure 2. Node 1 matches, node 2 does not match because of C',

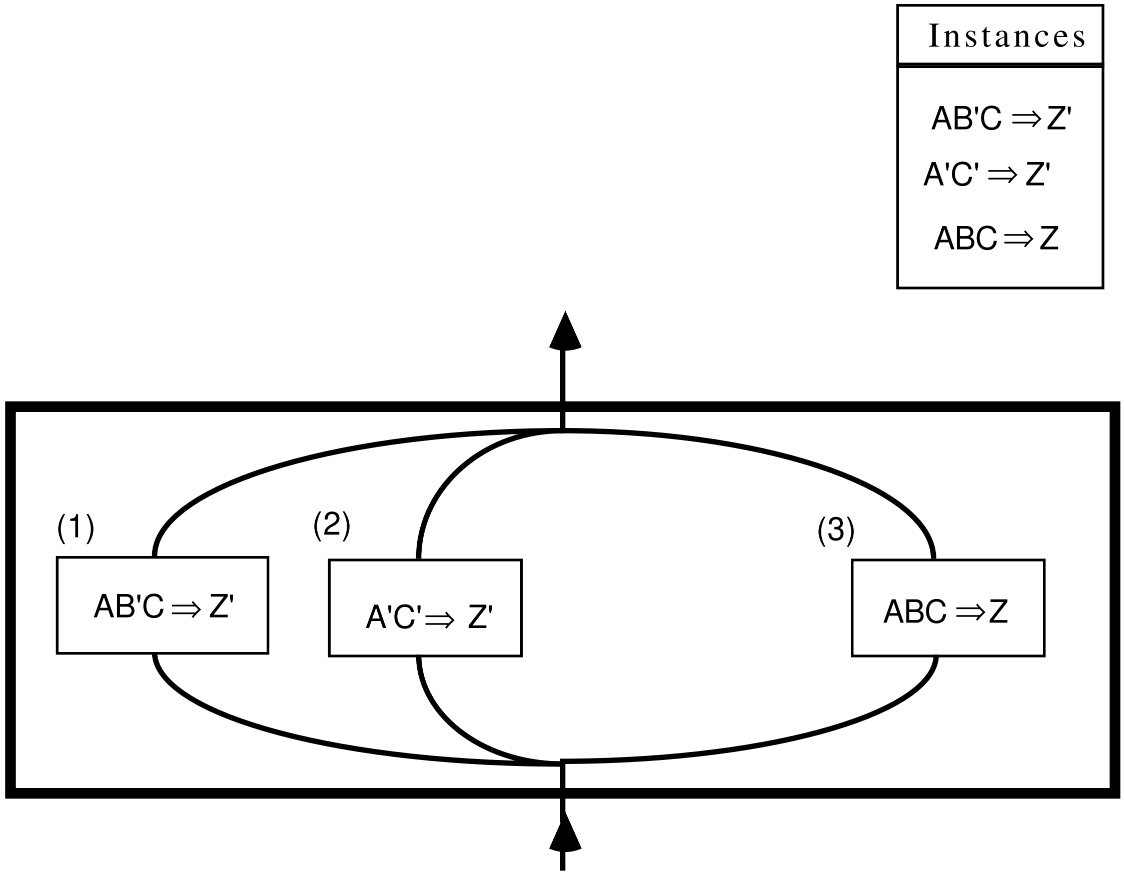

Figure 2. Example Net

node 3 does not match because of B. Thus node 1 activates its output, which is Z', and the network output becomes Z'.

2.4.3 learning

The **goal of learning** is to *store a consistent, efficient representation of instances (including generalization) which have been presented.* When an instance is presented to the machine for learning, both the input and the output are specified. Each node independently, asynchronously, and in parallel tests itself against the new instance to determine what adaptive action it should take.

If the new instance is covered by an instance already stored in the net, then the machine already outputs the correct answer for this instance, and no modification is necessary. The instance does not need to be added to the net. For example, if the new instance

AB'CD ⇒ Z'

is presented to the machine in figure 2 for learning, node 1 matches the input so the new instance is not added.

If the instance is not covered by the net, then the instance represents new information. Two possibilities exist:

1. All the instances already in the net are consistent with the new instance.
2. At least one instance already in the net is inconsistent with the new instance.

Consistent means that the network always outputs the same value for a given set of inputs. **Inconsistent** means that the network could output different values for the same set of inputs.

If the new instance is consistent, then the only necessary modification is to add the new instance to the net. For example, if the instance

$A'C \Rightarrow Z$

is presented to the machine of figure 2 for learning, none of the nodes matches the inputs, so a new node is added to the net that represents that instance. Figure 3 shows the modified instance set and the modified net.

If there is any inconsistency, then the net must be modified to resolve inconsistencies before the new instance is added to the net. Inconsistencies are resolved by the instances in the net that are inconsistent with the new instance while keeping those that are consistent. Assume the new instance

$AB' \Rightarrow Z$

is presented to the machine of figure 3 for learning. Node 1 is inconsistent with the new instance. Node 1 asserts that Z should be negative when the input is A is positive, B is negative, and C is positive. The new instance asserts that Z should be positive when A is positive, B is negative, and C is either positive or negative. Node 1 and the new instance are inconsistent when A is positive, B is negative and C is positive. None of the other nodes is inconsistent with the new instance. Thus, Node 1 is deleted and the new instance is added. Figure 4 shows the modified instance set and modified net.

Instances
$AB'C \Rightarrow Z'$
$A'C' \Rightarrow Z'$
$A'C \Rightarrow Z$
$ABC \Rightarrow Z$

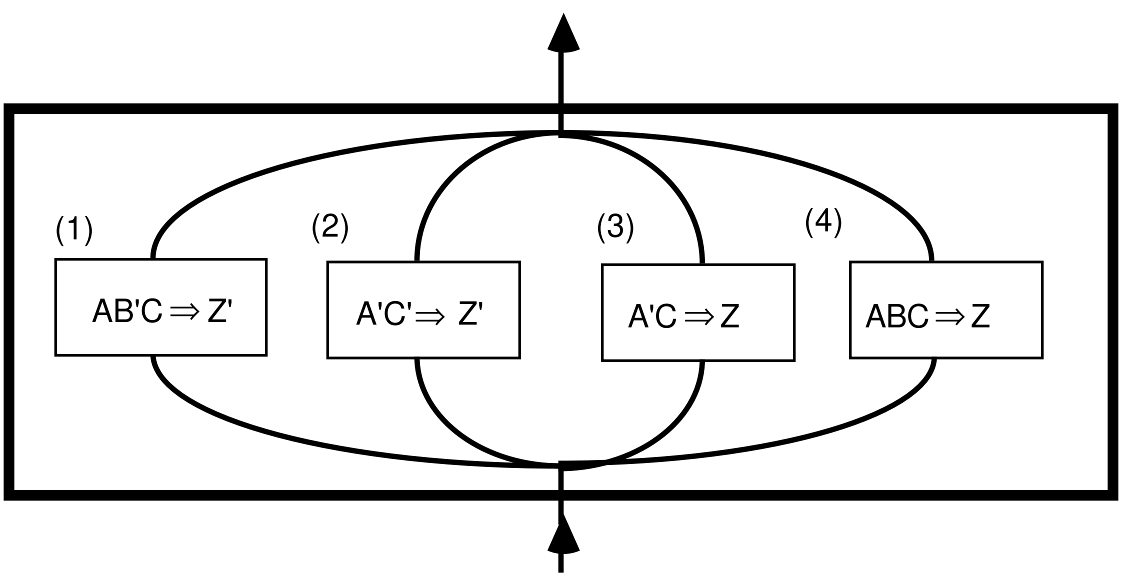

Figure 3. After instance A'C $\Rightarrow$ Z

Nodes in the net which neither match nor are inconsistent with a new instance make no changes.

Chapters 3 and 4 discuss both execution mode and learning (including handling inconsistencies) in more detail. The preceding discussion is only an overview.

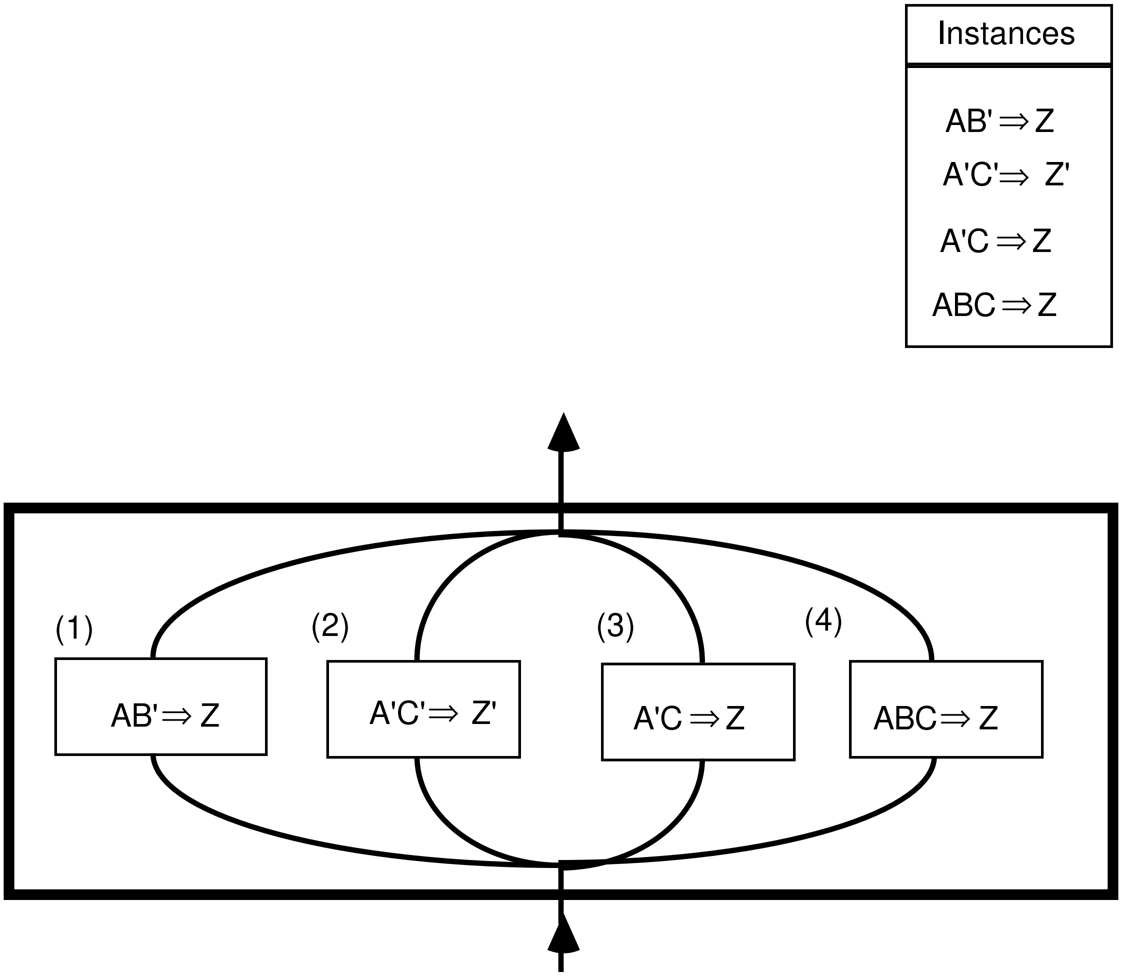

Figure 4. After Instance AB' ⇒ Z

2.5 LIA's Differences and Advantages

All of the differences between LIA and previous ASOCS models, and the advantages of LIA for hardware implementation, stem from the idea that *each LIA node represents a complete instance*. The differences are as follows:

1. In previous models, each node represents only a piece of an instance.
2. In previous models, a hierarchy of nodes is required to build up instances from pieces. In LIA, no hierarchy is needed since each node forms a complete instance.
3. *In previous models, the logical function and physical location of a node are important to the success of the network in storing instances. In LIA, the logical function of a node is independent of its physical location, because each node stores a complete instance.* Whereas a dynamic interconnect (topology) is required for previous ASOCS models, LIA can use a static topology.

In AA1 and AA2, any node may need to be connected to any other node in the network. Thus, these two models require some means for dynamic routing between two arbitrary nodes.

AA3 uses much less dynamic interconnect than AA1 and AA2, but has essentially the same problem with it (though to a lesser degree). AA3 uses a static binary tree interconnect. An arbitrary branch which grows beyond the depth of the tree needs to be connected to another (arbitrary) tree. This requires dynamic interconnect, since growth cannot be predicted a priori.

Figure 5 shows the dynamic routing problem. For AA1 and AA2, the question is, "How is node arbitrary node X dynamically connected directly to arbitrary node Y?". For AA3 the question is, "How is an arbitrary (end) node of tree X dynamically connected directly to the root node of arbitrary tree Y?".

Dynamic interconnect is the most intuitive way to implement AA1, AA2 and AA3. However, current technology does not support dynamic interconnect (on this scale) very well. Much of the simplicity and elegance of the models is lost in implementation details when these models are implemented using technology that does not fit well. The AA3 model is the most elegant of all the models. Yet, unbalanced tree branches may cause undesirable implementation problems. One problem in particular is that any branch that is significantly longer than the others causes the entire machine to be larger, because unused nodes on other branches remain unused. This problem could be solved if nodes could be created and destroyed as needed, instead of being used or unused.

Like AA3, LIA uses a static interconnect. Unlike AA3 however, the LIA nodes are logically and physically independent of one another, and therefore, any node may be programmed to represent any instance. Instances in a network may be moved around easily

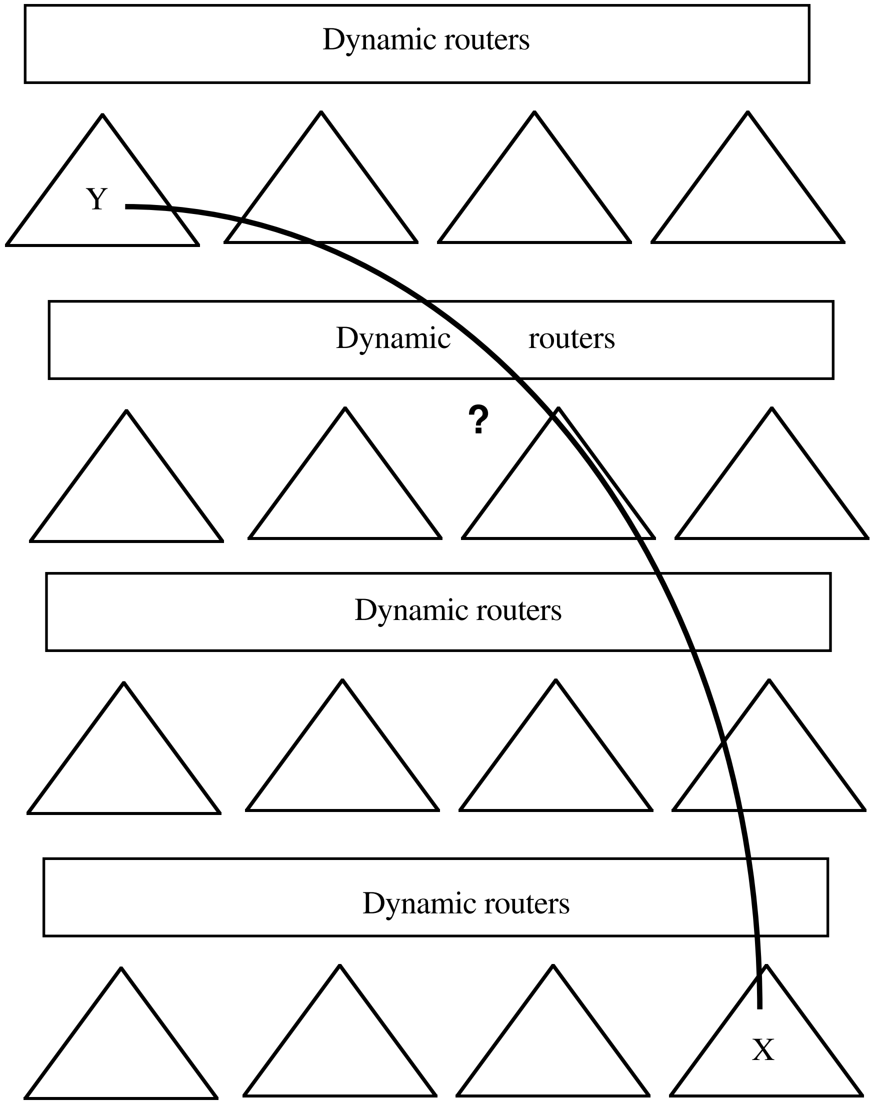

Figure 5. How is X Directly Connected to Y Dynamically?

in order to improve the efficiency of the representation. The number of unused nodes may be kept as small as desired, without affecting the network's function. This is the major advantage of the LIA model. This advantage leads toward improved implementation potential with current technology.

CHAPTER 3
INSTANCES AND INSTANCE SETS

This chapter discusses ideas and terms related to instance sets, and gives five theorems about how an instance set changes as new instances are added. The ideas presented here are important because the LIA model described in chapter 4 is based on the abstract instance set. This chapter has three sections:

1. definitions and explanations of LIA terms and ideas
2. five theorems describing how an instance set changes based on the relationship of old instances to a new instance
3. a four-step algorithm for modifying instance sets

3.1 Definitions and Explanations

Assume a logical universe composed of boolean input **variables**. This universe has a single distinguished boolean output variable **Z**, which cannot be an input variable. (Feedback variables are discussed in the ASOCS literature, but not in this thesis. Multiple output variables are discussed in Chapter 7. These two ideas are left out of the current discussion to simplify the explanations.) The input variables are indicated by capital letters: A, B, C, ... The value of any particular variable (i.e. A, B, or C, etc.) at any given moment is called its **polarity**. Polarity is defined to be **positive** or **negative**, which corresponds to high or low, true or false, respectively. Positive variables are denoted by the variable name, while negative variables are denoted by the name followed with a " ' " (prime). Thus "A" means that variable A is positive, and " A' " means that variable A is negative. The values of the variables at any given moment collectively are called the **state** of the universe. The polarities of the input variables are independent, but the polarity of Zdepends on the input variables.

The terms **subset**, **superset**, and **equal** are used throughout this thesis. These have meanings similar to those associated with sets, but two conventions need to be clarified. First, **subset** always means **proper subset**, **superset** always means **proper superset**, and **equal** covers the improper subset and improper superset cases. Second, when comparing sets of variables, the polarities are as important as which variables are in the sets. The two sets of variables, S_1 = {A, B, C'} and S_2 = {A, B, C}, are <u>not</u> equal. Even though they reference the same variables A, B, and C, S_1 has C' while S_2 has C. The same idea applies to the subset and superset set relationships.

An **instance** is a partial function from a set of input variables onto the output variable Z. It specifies what value Z must become when a given set of input variables is **matched**. Any non-negated variable is matched when the state is positive for that variable. Any negated variable is matched when the state is negative for that variable. Some state U **matches** an instance X iff (if and only if) all of the X's input variables are matched. For two instances X and Y, X **covers** Y iff every state that matches Y also matches X (this does not necessarily imply that Y covers X). An instance that is <u>not</u> matched says nothing about what value Z should have. Following are examples of instances (numbered for reference):

1. AB $\Rightarrow$ Z
2. AB'C $\Rightarrow$ Z
3. C'Y $\Rightarrow$ Z'

The variables on the left side are the input variables. The variable Z on the right side is the output variable. The instance

AB $\Rightarrow$ Z

asserts that when A and B are positive, Z must be set to positive. The instance

$C'Y \Rightarrow Z'$

asserts that when C is negative and Y is positive, then Z must be set to negative. Neither of these instances specify a value for Z when they are not matched.

Don't care variables do not appear on the left side of an instance, and have no effect on whether that instance asserts its output or not. For example, C and Y are "don't care" variables in the instance

$AB \Rightarrow Z.$

If A and B are positive,

$AB \Rightarrow Z$

asserts Z is positive regardless of the values of C and Y.

Using "don't care" variables means that a single instance can match a large number of states. For example, assume the input variables are {A,B,C,Y}. Of the 16 possible states for the input variables, the instance

$AB \Rightarrow Z$

is matched by four. In general, if there are m input variables, then there are 2^m possible states. An instance whose left side has length n is matched by $2^{(m-n)}$ states, because of the "don't care" variables.

An instance has **polarity** (like a variable). Instances that assert Z have **positive** polarity, while instances that assert Z' have **negative** polarity. Two instances with the same polarity are **concordant** with respect to each other; two with opposite polarities are **discordant** with respect to each other. Instance 1 and instance 2 above are concordant; instance 1 and instance 3 above are discordant.

A **discriminant variable** *for two instances* is an input variable that occurs negated in the input variables of one instance and non-negated in the input variables of the other. Two instances that contain a discriminant variable in their left sides are **discriminated** with respect to each other--the two instances are never simultaneously matched by any state. The variable B discriminates instances 1 and 2. The variable C discriminates instances 2 and 3. Instances 1 and 3 have no discriminant variable.

Consider instance 1,

$AB \Rightarrow Z,$

and instance 3,

$C'Y \Rightarrow Z'.$

If input variables A, B, and Y are positive and C is negative, then instance 1 asserts Z, while instance 3 asserts Z'. This is inconsistent. If two instances are discordant, and if there exists a state that matches both instances, then the two instances are **inconsistent**. The following lemma provides a test to check if two instances are inconsistent [Mart90b]:

Inconsistency Lemma. *Two instances are inconsistent iff they are discordant and have no discriminant variable.*

This lemma can also be used to test for consistency between instances. Two instances are **consistent** iff no state exists such that the instances can simultaneously assert opposite outputs (always true if they have a discriminant variable or they are concordant).

The union of a finite collection of consistent instances defines a partial function of input variables onto the output Z. It is the function defined by the set of instances, called the **instance set** (IS). Instances 1, 2 and 3 are a collection of instances, but are not an instance set, because instances 1 and 3 are inconsistent. Instances 4, 5 and 6, on the other hand, are an instance set because each pair of instances is consistent:

4. $AB \Rightarrow Z$
5. $AB'C \Rightarrow Z$
6. $B'C'Y \Rightarrow Z'$

Instance sets are always consistent, by definition. Two instance sets G_1 and G_2 are **equivalent** iff they assert the same output values for every possible state, that is, their partial functions are the same.

New instances can be added to an instance set **incrementally** over time, that is, one at a time, possibly changing the instance set and thus the partial function that the instance set defines. When a new instance is added to an instance set, it is compared against each of the instances already in the instance set to make sure each instance is consistent with the new instance. The new instance that is being compared against the others is called the **new instance** (NI). The others are called old instances, or instances previously added to the set, each one being an **old instance** (OI). When resolving inconsistencies between instances, precedence is given to the new instance. *The goal of changing the instance set is to add the new instance and modify only those old instances that are inconsistent (change only the information that is inconsistent).* Thus, it is necessary to distinguish between the new instance and the old instances so that we know how to modify the instance set. An algorithm for modifying instance sets is given in section 3.3.

In the process of adding new instances to an instance set, redundancies of various types may occur. For example, suppose we want to add the instance

AB ⇒ Z

to the following instance set:

7. AB ⇒ Z
8. AB'C ⇒ Z
9. B'C'Y ⇒ Z'

The instance

AB ⇒ Z

is consistent with all of the old instances, so it could be added to the instance set. But the new instance is exactly like instance 7: adding the new instance would be redundant.

The process of detecting and removing redundancies in an instance set is called **minimization**. There are many types of minimization. Minimization does not affect consistency--it is desirable because it helps keep the size of an instance set as small as possible. It also permits learning generalization in that special-case instances are replaced by more general instances. Some types of minimization can be done very quickly, while others take a long time [Mart86]. Instance set modification can be kept fast by compromising on parsimony. Fast and easy types of minimizations can be done at the same time as modifications for consistency. These are the subset, superset and equal minimizations. The slower types of minimizations can be done periodically as "background" processes, during times when the machine is not actively executing or learning. This description of the LIA model includes algorithms for doing the fast minimizations as part of "normal" learning. Chapter 7 discusses other minimizations.

3.2 Theorems

This section introduces some notation and presents five theorems about instances, and instance set modification. This section is not intended to be an exhaustive theoretical explanation, but only to provide a basis for the reader to understand Chapter 4. The notation used in this section is used throughout the rest of this thesis.

Let NI.V denote the input (left-hand) variables of a new instance. Let G be an instance set. Let *y* denote the polarity of a new instance NI, and let ~*y* denote the inverse of *y*. (Lowercase italicized letters denote any variable, unlike uppercase letters, which denote specific boolean variables.) Let OI.V denote the input (left-hand) variables of some instance OI in G. Each NI partitions G into 10 sets of instances, defined as follows:

1. OI is a *Concordant Subset* instance iff OI.V is a subset of NI.V, and OI is concordant (has the same polarity) with NI.

2. OI is a *Discordant Subset* instance iff OI.V is a subset of NI.V, and OI is discordant (has the opposite polarity) with NI.
3. OI is a *Concordant Equal* instance iff OI.V = NI.V, and OI is concordant with NI.
4. OI is a *Discordant Equal* instance iff OI.V = NI.V, and OI is discordant with NI.
5. OI is a *Concordant Superset* instance iff OI.V is a superset of NI.V, and OI is concordant with NI.
6. OI is a *Concordant Superset* instance iff OI.V is a superset of NI.V, and OI is discordant with NI.
7. OI is a *Concordant Discriminated* instance iff there is a discriminant variable for OI and NI, and OI is concordant with NI.
8. OI is a *Discordant Discriminated* instance iff there is a discriminant variable for OI and NI, and OI is discordant with NI.
9. OI is a *Concordant Overlap* instance iff OI is not a Subset, Equal, Superset, or Discriminated instance, and OI is concordant with NI.
10. OI is a *Discordant Overlap* instance iff OI is not a Subset, Equal, Superset, or Discriminated instance, and OI is discordant with NI.

All relationships between NI.V and OI.V are defined from the point of view of the OI.

Partition Theorem. *Let G be an IS, and let OI be some instance in G. Let NI be a new instance. The NI partitions G into the following 10 sets of instances: Concordant and Discordant Subset, Concordant and Discordant Equal, Concordant and Discordant Superset, Concordant and Discordant Discriminated, and Concordant and Discordant Overlap.*

<u>Proof:</u> To prove that these 10 sets form a partition, it suffices to show that every old instance in G belongs to exactly one of the 10 sets.

By definition, the Subset, Equal, Superset and Overlap sets are mutually disjoint, and the Discriminated and Overlap sets are mutually disjoint. Without loss of generality, let v belong to OI.V, and $\sim v$ belong to NI.V. NI and OI are Discriminated. Any input variable list never simultaneously contains a variable and its complement. Therefore, the presence of v in OI.V and $\sim v$ in NI.V proves that OI.V cannot be a subset, equal, or superset of NI.V.

Within each of the five sets, each old instance must be either concordant or discordant with NI, but not both. old instance cannot be a Concordant Subset and Discordant Subset simultaneously to NI. Thus NI partitions G into 10 sets of instances.

Consistency Theorem. *Let G_i be an IS and let NI be a new instance.*
Let C be the collection of all OI's in G_i such that each OI in C is Discordant Superset or Discordant and Equal with NI.
Let D be the collection of all OI's in G_i such that each OI in D is Discordant Subset or Discordant Overlap with NI. Compute the set

DiscrList = NI.V - OI.V, with each element inverted

for each OI in D.
Let D_1 be the collection of instances

{ $(w + OI.V) \Rightarrow \sim y$:
where w is any variable in the set DiscrList,
$\sim y$ is the polarity of OI }
computed for each OI in D.

The new instance set

$$G_{i+1} = (G_i - (C \cup D)) \cup D_1 \cup NI$$

resolves the inconsistencies due to C and D, and contains all of G_i that is consistent with NI.

Here are two examples that illustrate the Consistency Theorem, given before the proof because the proof is rather long. Consider the new instance

AB $\Rightarrow$ Z

confronting the old instance

ABC $\Rightarrow$ Z'.

The old instance is a Superset of the new instance because {A,B,C} is a superset of {A,B}. The new instance is matched by input states ABC and ABC', while the old instance is matched by only input state ABC. Since the new instance will always force Z to become Z when A, B, and C are matched, the old instance is useless. The old instance is deleted. Consider the new instance

ACD $\Rightarrow$ Z

confronting the old instance

AB $\Rightarrow$ Z'.

The old instance overlaps the new instance. (The new instance is matched by states ABCD and AB'CD. The old instance is matched by the states ABC'D', ABC'D, ABCD', and ABCD. They overlap in state ABCD.) Form the DiscrList = NI.V - OI.V = {C,D}. The Consistency Theorem says that the part of

AB $\Rightarrow$ Z'

that is consistent with

ACD $\Rightarrow$ Z

is given by the union of the two instances

C'AB $\Rightarrow$ Z'

D'AB $\Rightarrow$ Z'.

(These two instances are equivalent to combining ABC'D' with ABC'D and ABC'D' with ABCD').

Proof: The first section of this proof proves that G_{i+1} resolves the inconsistencies due to C and D. The second section proves that G_{i+1} is consistent with the new instance. This is done in four parts:

1. G_{i+1} retains all of the instances in G_i that are consistent with the NI.
2. G_{i+1} retains all of C that is consistent with the NI, namely, none of C.
3. G_{i+1} retains all of D that is consistent with the NI, namely D_1.
4. The addition of D_1 cannot introduce new inconsistencies.

section 1. If C and D are deleted from G, then G_{i+1} does not include any instance in C or D. Therefore, no inconsistency caused by instances in C or D exists in G_{i+1}.

section 2, part 1. $(C \cup D)$ contains all of the instances that are inconsistent with the new instance, and only instances that are inconsistent. All the members of C or D are required to be discordant with the new instance in this theorem. None are concordant. None of the members of C or D contain discriminant variables for the new instance; if so, they could not be Subset, Equal, or Superset (Partition theorem). Thus the members of $C \cup D$ are inconsistent with the new instance (Inconsistency lemma). C and D are defined to contain all of the members of their respective sets from G_i. Therefore,

$G_i - (C \cup D)$

retains all instances in G_i that are Concordant or Discriminated with the new instance, i.e. those instances that are consistent.

section 2, part 2. G_{i+1} contains none of C. Since any OI.V for any OI in C is Superset with NI.V, every state that matches an OI.V in (and forces Z to be $\sim y$ by the old instance) automatically matches NI.V (and forces Z to be *y* by the new instance). Thus, every old instance in C is inconsistent with the new instance for every state that matches an old instance in C. Since the new instance has precedence, none of the old instances in C can be kept. They must be deleted from G_i. Thus

$G_i - (C \cup D)$

retains as much of C as is consistent with the new instance: G_{i+1} retains none of C.

section 2, part 3. G_i contains all of D that is consistent with the new instance. This is shown by construction. For every old instance in D, there is a corresponding OI.V and DiscrList. Thus, for every OI in D, we may do the following construction:

Let *w* belong to DiscrList, and *y* be the polarity of the new instance, and ~*y* be the polarity of OI (Discordant to the new instance). NI can be rewritten equivalently as

$\sim w + (\text{NI.V} - \sim w) \Rightarrow y$

for any *w*.

Since there is no discriminant variable for NI and OI, neither *w* nor ~*w* belong to OI.V. OI is equivalently rewritten as the union of the two instances

$w + \text{OI.V} \Rightarrow \sim y$ and

$\sim w + \text{OI.V} \Rightarrow \sim y$.

Because there is no discriminant variable for NI and OI, there exists one or more states that simultaneously matches both. This state or states, namely any that are equal to $\{\text{NI.V} \cup \text{OI.V}\}$ or contain them as a subset, force Z to be *y* by NI, but *Z to be* ~*y* by OI simultaneously.

$\sim w + \text{OI.V} \Rightarrow \sim y$

is the part of OI that is inconsistent with

$\sim w + (\text{NI.V} - \sim w) \Rightarrow y$

for any *w* in DiscrList since the two are discordant and there is no discriminant variable (Inconsistency lemma). On the other hand,

$w + (\text{OI.V}) \Rightarrow y'$

is consistent with

$\sim w + (\text{NI.V} - \sim w) \Rightarrow y$

for all *w* in DiscrList because they have discriminant variable *w*. Therefore, the part of OI that is consistent with NI is given by

{$w + \text{OI.V} \Rightarrow \sim$y:
where w is any variable in DiscrList,
and ~*y* is the polarity of OI}

This part is added to D_1 (for each OI in D), and thus D_1 is the part of D that is consistent with NI.

section 2, part 4. Adding D_1 to G_{i+1} cannot add new inconsistencies to G_{i+1}. For every instance X in D_1, there exists an OI in D which produced X. This OI in D has to be consistent with every instance in

$G_i - (C \cup D)$

either because OI is Concordant or because OI is Discordant but Discriminated with each instance in that set. Each X in D_1 is a copy of the OI in D which produced X with a variable added to X's input variable list. Therefore, each X must be concordant, or discordant but discriminated with the same instances as the OI that produced X. Therefore, adding D_1 to G_{i+1} cannot add inconsistencies to G_{i+1}.

This proof shows that G_{i+1} resolves inconsistencies due to C and D, and that G_{i+1} retains as much of G_i as is consistent with the new instance. This concludes the proof of the Consistency Theorem.

The process of creating new instances using discriminant variables and old instances is called **DVA (discriminant variable addition).** It is so named because the discriminant variables in new instances cause instances to discriminate properly the new information.

The Consistency Theorem shows how to produce a new instance set given a current instance set and a new instance that is to be added. The new instance set contains the new instance and as much of the old instance set as was consistent with the new instance. The number of instances in the new instance set may be greater than, less than, or equal to the number of instances in the old instance set. In the new instance set, all instances have equal precedence and order is inconsequential (since the instance set is consistent, no inconsistencies are possible and precedence is not necessary).

Covered Redundancy Theorem. *Let G_i be an IS. Let NI be a new instance, y be the polarity of NI, and let $(G \cup NI)$ be consistent. Let OI be an instance in G_i, such that OI is Concordant Subset or Concordant Equal with NI. Adding NI to G_i does not add any new information to G_i (the NI can be deleted or ignored).*

Proof: Every state that matches NI (and forces Z to be *y*) also matches OI (and forces Z to be *y*). If OI is Subset with NI, then NI is really a special case of OI; If OI is Equal to NI, then the two instances are the same. In either case, G_i already has an instance (namely OI) that covers NI. Thus, adding NI does not add new information to G_i.

To illustrate the Covered Redundancy Theorem, consider the new instance

$ABC \Rightarrow Z$

confronting an old instance

$AB \Rightarrow Z$.

The old instance matches states ABC and ABC', and the new instance matches state ABC. Since both the new instance and old instance assert Z for the state ABC, the new instance is redundant. It does not need to be added to the instance set. This is an example of subset minimization. Equal minimization is treated as a case of subset minimization.

Special Case Redundancy Theorem. *Let G_i be an IS, let NI be a new instance, and let y be the polarity of the NI. Let $(G_i \cup NI)$ be consistent. Let D be the collection of OI's in G_i such that OI is Concordant Superset with NI.*
Then the new instance set

$$G_{i+1} = (G_i - D) \cup NI$$

is equivalent to $(G_i \cup NI)$.

Proof. Every state that matches an OI in D (and forces Z to be *y*) also matches NI (and forces Z to be *y*). If OI is Superset with NI, then OI is really a special case of NI, and NI covers OI. Thus, for all states matched by instances in D, $(G_i \cup NI)$ and G_{i+1} will assert the same output value. For any instance in $(G_i - D)$, $(G_i \cup NI)$ and G_{i+1} are the same, and therefore assert the same output value. Therefore, G_{i+1} is equivalent to $(G_i \cup NI)$. This concludes the proof of the Special Case Redundancy Theorem.

To illustrate the Special Case Redundancy Theorem, consider the new instance

$AB \Rightarrow Z$

confronting the old instance

$ABC \Rightarrow Z$.

The new instance matches states ABC and ABC', and the old instance matches ABC.

Adding the new instance makes the old instance redundant because both assert Z for ABC (but the new instance does more). Thus, the new instance is added to, and the old instance deleted from, the instance set. This is an example of superset minimization.

The redundancy theorems provide a method for identfying and removing subset and superset redundancies in an instance set when a new instance is added. Eliminating redundancies from the instance set, reduces its size, and reduces the amount of computation needed at future steps.

Exclusive Conditions Theorem. *Let G be an IS, and let NI be a new instance. Exactly one of three conditions exists:*

1. There exists at least one OI in G that covers and is concordant with NI.
2. There exists at least one OI in G that is inconsistent with the NI.
3. The NI represents new consistent information.

Proof. The first part of this proof proves that at most one of these conditions always holds. Assume there exists an OI in G that covers and is concordant with NI. No OI in G can be inconsistent with NI, nor can the NI represent new information not already in the net, by definition. If the first were possible, then G could not be consistent. If the second were possible, then OI did not match NI. Therefore, conditions 2 and 3 cannot hold if condition 1 holds. By definition, if there exists at least one OI in G that is inconsistent with NI, then NI cannot represent new consistent information (and the converse is also true). Therefore, condition 3 cannot hold if condition 2 holds, and conversely.

The only other possible type of inconsistency, that OI_1 in G covers NI but OI_2 is simultaneously inconsistent with it is an impossible situation. Thus, at most one of these three conditions exists.

The second part of this proof proves that at least one of the three conditions always holds. The three conditions listed are the only possible choices. There are only two questions to answer,

1) Is there an old instance in G that covers and is consistent with the new instance?
2) Is there some old instance in G that is inconsistent with the new instance?

and these are yes or no questions. This makes a total of four possible ways to answer, one of which is (by definition) impossible:

1. Answering "yes" and "no", respectively, corresponds to condition 1.
2. Answering "no" and "yes", respectively, corresponds to condition 2.
3. Answering "no" and "no", respectively, corresponds to condition 3.
4. Finally, answering "yes" and "yes" is impossible, since G would then be inconsistent.

Therefore, exactly one of the three conditions listed above must hold. This concludes the proof of The Exclusive Conditions Theorem.

The theorems in this section form the basis for an instance set modification algorithm and our LIA learning algorithm.

3.3 The Four Steps of the Instance Set Modification Algorithm

An instance set modification algorithm builds a consistent instance set. The theorems in section 3.2 tell us how to modify an instance set. This section presents a four-step algorithm based on the theorems.

The instance set modification algorithm is organized to check for four things:

1. Old instances that cover the new instance (condition 1, The Exclusive Conditions Theorem).
2. Old instances that are inconsistent with the new instance (condition 2, The Exclusive Conditions Theorem).
3. New, consistent instances (condition 3, The Exclusive Conditions Theorem).
4. Redundant old instances (Theorems 3 and 4).

The four steps are the following:

1. If there exists an old instance in G_i that covers the new instance, then exit.
2. If there exists an old instance in G_i that is inconsistent with the new instance, then do discriminant variable addition for each such old instance.
3. If in G_i there exist redundant old instances, then delete these old instances.
4. Add the new instance to G_i.

The explanations in each step relate the steps to the classes defined by the Partition Theorem.

Step 1: If there exists an old instance in G_i that covers the new instance, then exit.

If there exists an old instance in G_i that covers the new instance (condition 1, The Exclusive Conditions Theorem), then the old instance is Concordant Subset or Concordant Equal with NI (Covered Redundancy and Exclusive Conditions Theorems). In these cases, the old instance already matches every state in which the new instance is matched, so G_i does not have to be changed. Thus, the new instance is ignored, and $G_{i+1} = G_i$.

Step 2: If there is an old instance in G_i that is inconsistent with the new instance, then do discriminant variable addition (DVA) for each such old instance.

An old instance is inconsistent with the new instance if the old instance is Discordant with the new instance and they have no discriminant variable (Inconsistency lemma). If an inconsistency exists (condition 2, The Exclusive Conditions Theorem), then four possible classes of inconsistencies exist (Partition and Consistency Theorems):

a. The old instance is Discordant Subset
b. The old instance is a Discordant Superset
c. The old instance is a Discordant Equal
d. The old instance is a Discordant Overlap

Classes a and d. If the old instance is Discordant Subset or Discordant Overlap (Partition Theorem) with the new instance, then we want to get rid of the inconsistency, but recover as much of the old instance as possible (and still be consistent with the new instance). DVA does this for each inconsistent Subset or Overlap OI (the Consistency Theorem, set D). The old instance replaces itself by a new set of instances, all of which contain discriminant variables with NI.

Classes b and c. The set DiscrList of discriminant variables for the Discordant Superset and Discordant Equal old instances is always empty (the Consistency Theorem, set C). This is because the new instance completely covers these old instances, thus there is nothing to recover. Doing DVA with these old instances amounts to simply deleting them from the instance set.

None of the actions for removing inconsistencies creates an instance that matches the new instance. The new instance needs to be added to G_i, but that is delayed until step 4, to simplify the algorithm (see step 4). Step 2 (this step) implements the equation

$$G_{i+1} = (G_i - (C \cup D)) \cup D_1.$$

This is the same as the one given in the Consistency Theorem (except that the NI is not yet added to G_{i+1}).

Step 3: If there exist redundant old instances in G_i, then delete these old instances.

Redundant, in this case, means Concordant Superset. If an old instance is Concordant Superset with the new instance, then the new instance covers the old instance (Partition and Special Case Redundancy Theorems, and condition 3, Exclusive Conditions Theorem). In this case, the old instance is a special case of the new instance, and is therefore redundant. The old instance is deleted from G_i.

(Any Discordant Superset old instances are deleted in step 2, therefore any remaining Superset old instances must be Concordant with the new instance. Thus, polarity does not need to be checked when executing this step.)

Step 4: Add the new instance to G_i.

This is the only step of the four in which no check has to be made. The only time the NI is not added to G_i to produce G_{i+1} is when step 1 is executed. If steps 2 or 3 are executed, then the new instance must be added. If none of steps 1, 2 or 3 are executed, the new instance still must be added to G_i (condition 3, the Exclusive Conditions Theorem), since no OI in G_i covers the new instance. Therefore, it is algorithmically simpler to wait until the end before adding the new instance.

Table 1 summarizes the adaptive actions defined by this algorithm for the partition given by the Partition Theorem:

		OI vs. NI CLASS				
		Subset	Equal	Superset	Overlap	Discriminated
POLARITY RELATIONS	Concordant	NI Covered	NI Covered	Delete OI	No Action	No Action
	Discordant	DVA	Delete OI	Delete OI	DVA	No Action

Table 1. LIA Adaptive Actions

The Discriminated instances and Concordant Overlap instances do not necessitate adaptive actions, by definition.

CHAPTER 4
A Location-Independent ASOCS Model

This chapter presents a potential model for LIA. The basis for the model is to implement the theoretical instance set construct described in chapter 3. This chapter has five sections:

1. the LIA node
2. the net topology
3. a block view of the LIA model
4. LIA in execution mode
5. LIA in learning (reconfiguring) mode

4.1 The LIA Node

Figure 6 shows a picture of an LIA node. A node is composed of an AND gate and a Control Unit (CU). The AND gate performs the AND function on the node's inputs. The CU controls adaptation of the node during learning.

Each node has memory for a variable list (the input variables it conjuncts), and a polarity flag. The size of the node's variable list is not limited. The polarity flag tells what polarity the output variable Z should become if the node is matched, or marks a node as unused (free).

The following example shows how a node corresponds to an instance. Assume a node stores the instance

$ABC \Rightarrow Z.$

The CU stores the input variables A, B, C and sets the polarity flag to positive. (The polarities of the variables in the node's variable list and of the polarity flag at any given time

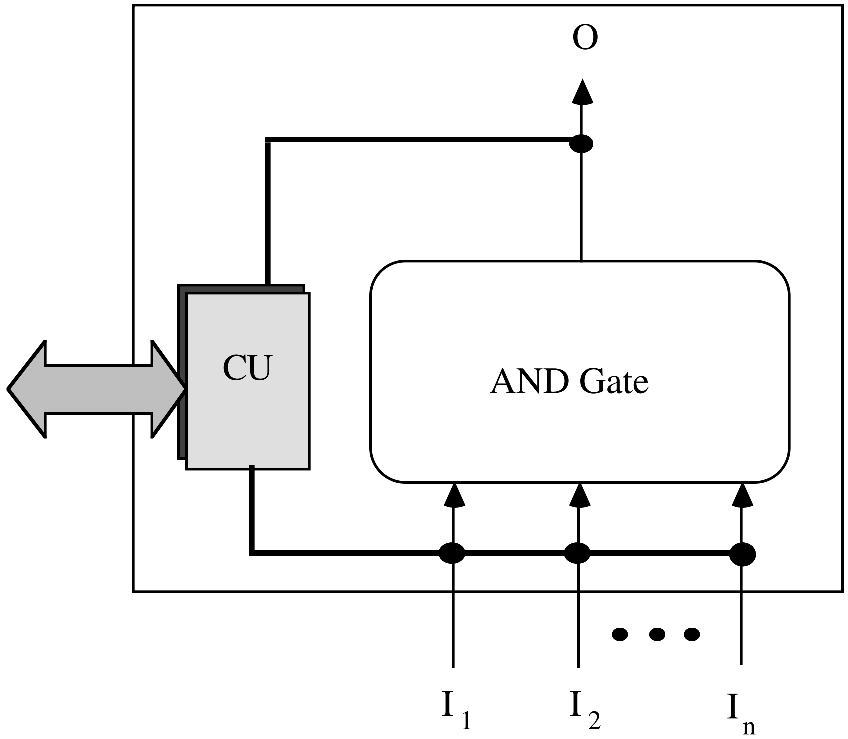

Figure 6. LIA Node

is called the **node state.**)

When a node receives input in execution mode, only the node's variable list is checked against the input for a possible match.

When an node receives input in learning mode, both the node's node's variable list and the polarity flag are checked against the the input to determine if (and how) the node state needs to be changed.

When an free node is allocated to store an instance during learning, it's state is changed in the following way: If the node is to fulfill the new instance, the new instance is copied into the new node's variable list and polarity flag. If the new node is allocated by DVA, its variable list is the original node's variable list concatenated with a discriminant variable. The new polarity flag is the same as the original node's polarity flag.

4.2 The Network Topology

The previous section shows how a node supports storing and learning instances. Unlike the instances in a theoretical instance set, each node cannot know what is the state of the boolean universe, or what is the newest instance, without communicating with its environment. Four types of communication must take place:

1. Nodes must receive broadcasts of information.
2. Nodes must be able to broadcast output, and that output must be gathered.
3. Nodes must be able to allocate unused nodes in the net in parallel with a minimum of contention.
4. Nodes must be able to (logically) self delete from the network.

This section discusses a binary tree topology for the LIA model net, and shows how the tree supports these communications.

The binary tree is a simple, regular, fast structure. The root node connects to two children. Each of the other nodes connects to one parent and zero, one or two children. Figure 7 shows one possible LIA configuration. Nodes are shown with a two-way interconnect between them. The root is at the top.

The nodes and interconnect together are called the **logic network** (net). In the LIA model, a net corresponds to an instance set. A net is a device that stores and executes instance sets. (**Execution** means when a net receives input from its environment, each node checks to see if it is matched by the environment. The net produces an output based on the results. Section 4.4 gives more detail on execution mode).

The following defines what it means for a network to store, or **fulfill**, an instance set:

Let K be an instance in an instance set G_i. Let S be a network <u>in execution mode</u>, and let N be a node in S. S **fulfills** G_i iff for every K, there exists an N such that the instance stored at N is Concordant Equal or Concordant Subset with K (that is, the instance stored at N covers and is concordant with K).

The previous section shows how a node corresponds to an instance. This section shows how a net corresponds to an instance set the net fulfills. The remainder of the thesis

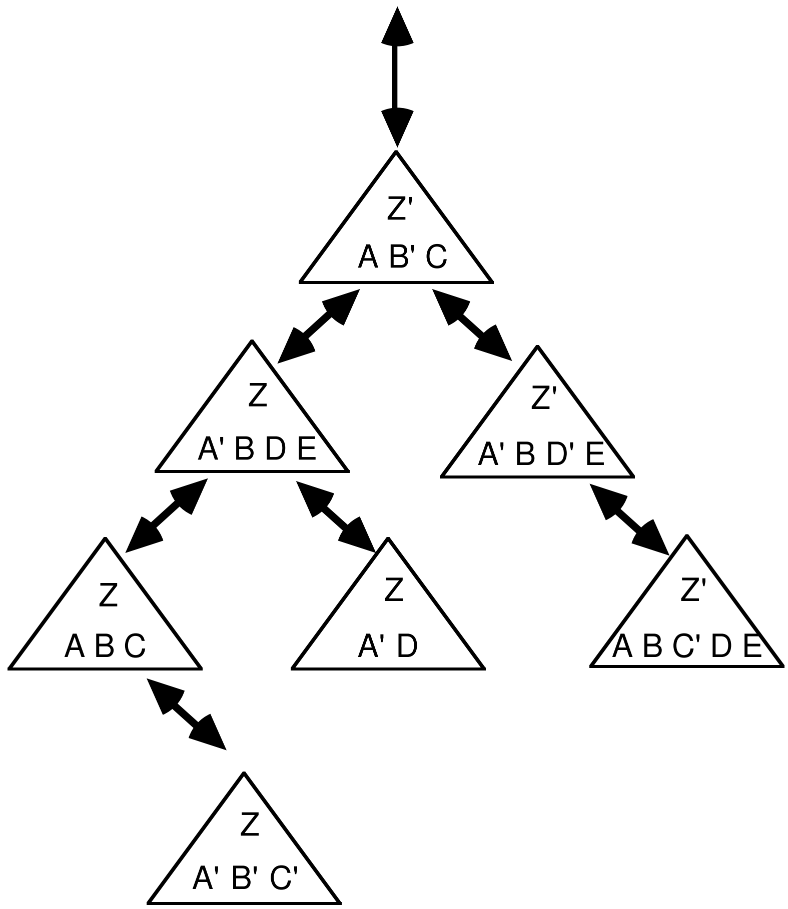

Figure 7. Example LIA Net

uses the same notation for nodes and nets as for instances and instance sets. That is, G_i refers to a net that fulfills an instance set as well as to the instance set itself. OI refers to a node in the net G_i, as well as to the instance OI in the instance set G_i stored in the node. The distinction is not as critical for LIA as for other ASOCS models, which implement the net differently [Mart86], but is still necessary. Although an exact correspondence exists between instances and nodes, an instance set is a theoretical construct, whereas an actual net is a machine implementation.

Having discussed the net topology, each of the four types of communication are illustrated below.

4.2.1 broadcast and gather

Broadcast is used in both execution mode and learning mode. Figure 8 shows how a broadcast proliferates from the root node. Each node receives the message M and sends it to both children. In execution mode, the root node receives environmental input from the input binder. In learning mode, the root node receives new input and commands from the AU. Following is the basic information broadcast during learning mode:

- the new instance and a fulfill flag
- the inconsistency flag
- the “initiate DVA” flag
- the self-delete flag
- the new instance and an add flag

Gather is used in both execution and learning mode. Figure 9 shows how messages are gathered at each node, and ultimately, at the root node. If a node receives two messages at the same time, the two are combined, and only one sent to the parent. If messages arrive at a node at different times, they can be sent individually. The result is always be the same (because the net is consistent). In execution mode, a node gathers polarity flag values from its children (if any are sent), and sends one, or both, to its parent. The root node sends its gathered value to the output binder. In learning mode, the root sends its gathered values to the AU. Following is the basic information gathered during learning:

- a flag indicating that at least one node fulfills the new instance
- a flag indicating that at least one node is inconsistent with the current new instance
- a flag indicating that the new instance has been stored in a node

A net G_i *always* is consistent before a new instance is presented for learning. In order to decide that G_i already fulfills a new instance, the AU only has to get a response from one node. There may be more, but one is sufficient. Similarly, one

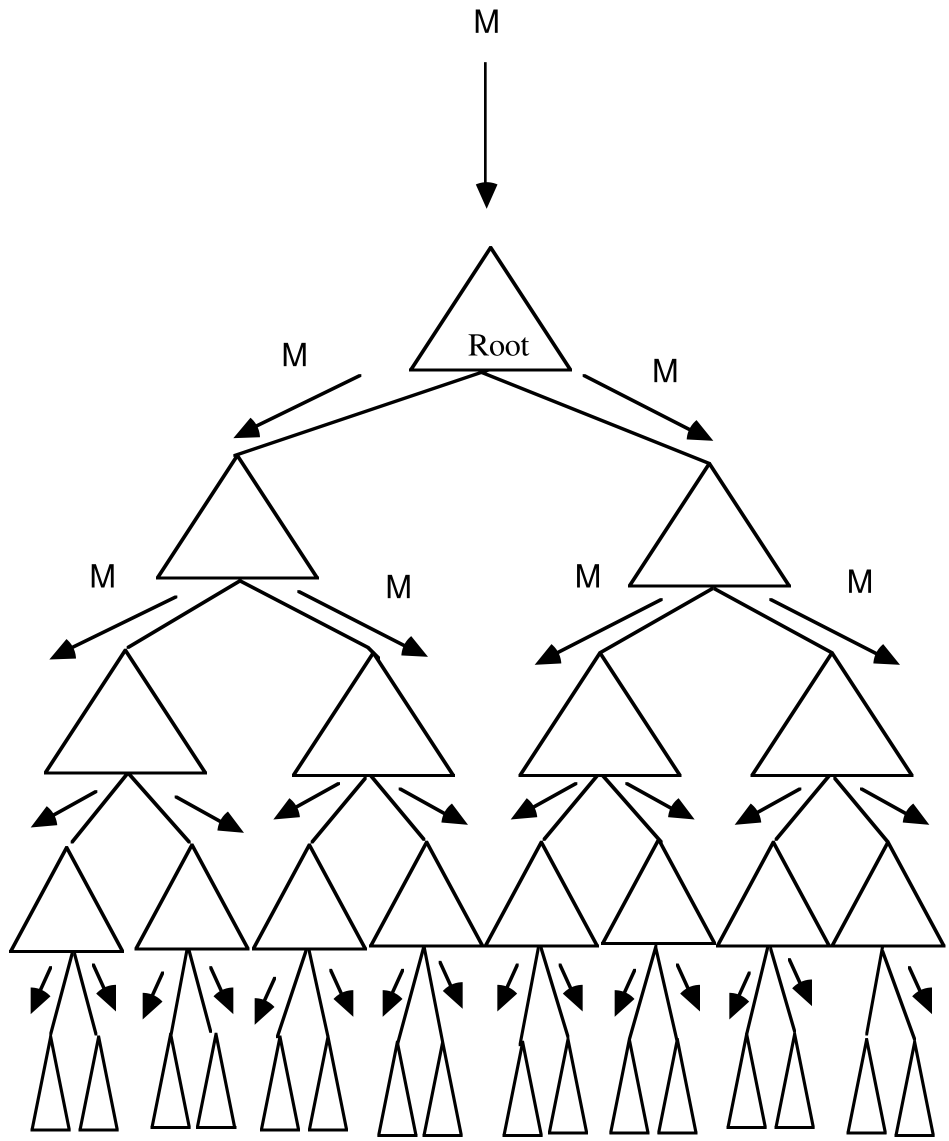

Figure 8. Parallel Broadcast

response indicating an inconsistency is enough to trigger a broadcast to initiate DVA. Finally, since one new instance is stored in one node, there is one response only from that new node. Because one response is sufficient in each case, efficient gathering is possible without loss of information. There is no contention as messages are passed up the tree.

Note that not all of these flags, for both broadcast and gather, are used every time a new instance is learned. Also, particular implementations may use more flags than these.

Here are two an examples of gathering operations--one in execution mode, the other

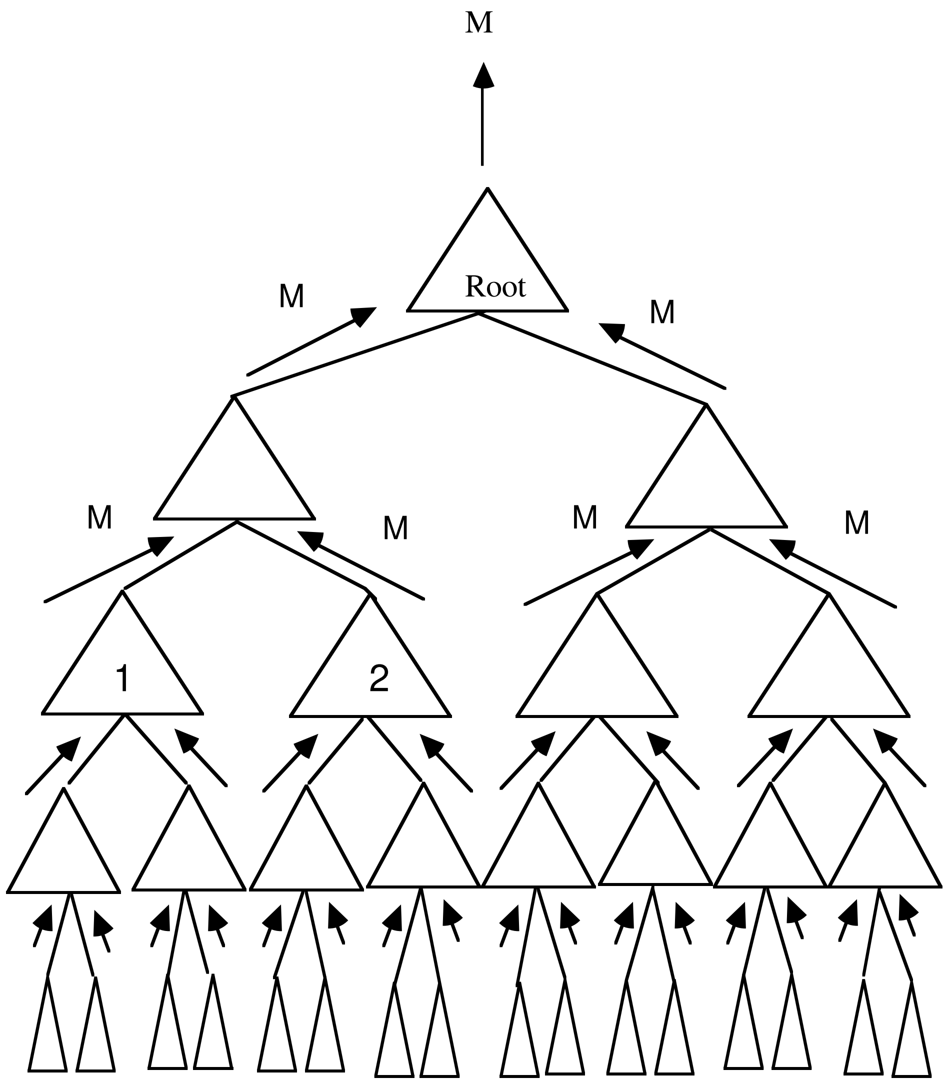

Figure 9. Parallel Gather

in learning mode. Assume that the net in figure 9 shows a net in execution mode. Nodes 1 and 2 both have positive polarity flags, and are simultaneously matched by some input from the environment. Both nodes send their polarity flags to the parent. The net always is consistent during execution. Therefore, knowing the polarity flag of one node is sufficient to know what the output of the net should be. Thus, the parent of nodes 1 and 2 passes a single positive polarity flag to its parent, instead of sending two. If polarity flags from two different nodes reach some parent at different times, then the parent sends both polarity flags up at different times. Timing does not affect the result.

Now assume that the net in figure 9 shows a net in learning mode, and that nodes 1 and 2 fulfill a new instance that has been presented. Each sends its response to the parent, which need send only one to the root.

4.2.2 allocating new nodes in parallel without contention

New nodes are added to the net only during DVA or when a new instance is added by the AU. Section 4.1 discusses how an instance is stored at a node. This section shows how the instance information gets to the node.

Figure 10 shows node allocation for DVA. The two nodes initiating DVA are marked as D_1 and D_2. N marks originally free nodes that are allocated by DVA. Allocated nodes that are not initiating DVA are blank. The arrows show the direction messages travel down the tree. The black line across the tree indicates the original frontier of allocated nodes in the tree.

Assume that D_1 and D_2 each add four new nodes/instances to the net. Instances created by DVA are placed in any free nodes in the initiating node's subtrees. Any allocated node receiving a DVA allocation message splits the message and sends half to its left child, and half to its right child. Assume that a single message is sent to the left child. While DVA for D_1 and D_2 is described separately here, they are not separate cases. The only difference between them is that originally D_1 has an allocated node below it, whereas all the nodes below D_2 are free.

D_1 splits the allocation of new nodes between its left and right children--each node receives a message to allocate two nodes. D_1's left child is already allocated, so the left child sends its children each a message to allocate one node. These children are already allocated, but their children are free. Given a choice between equally balanced children, assume the left child always is allocated. Since the messages are used up, node allocation stops for D_1's left child. D_1's right child is free, so the child uses one of the messages and becomes an allocated node. It sends a message to allocate one node to its left child;

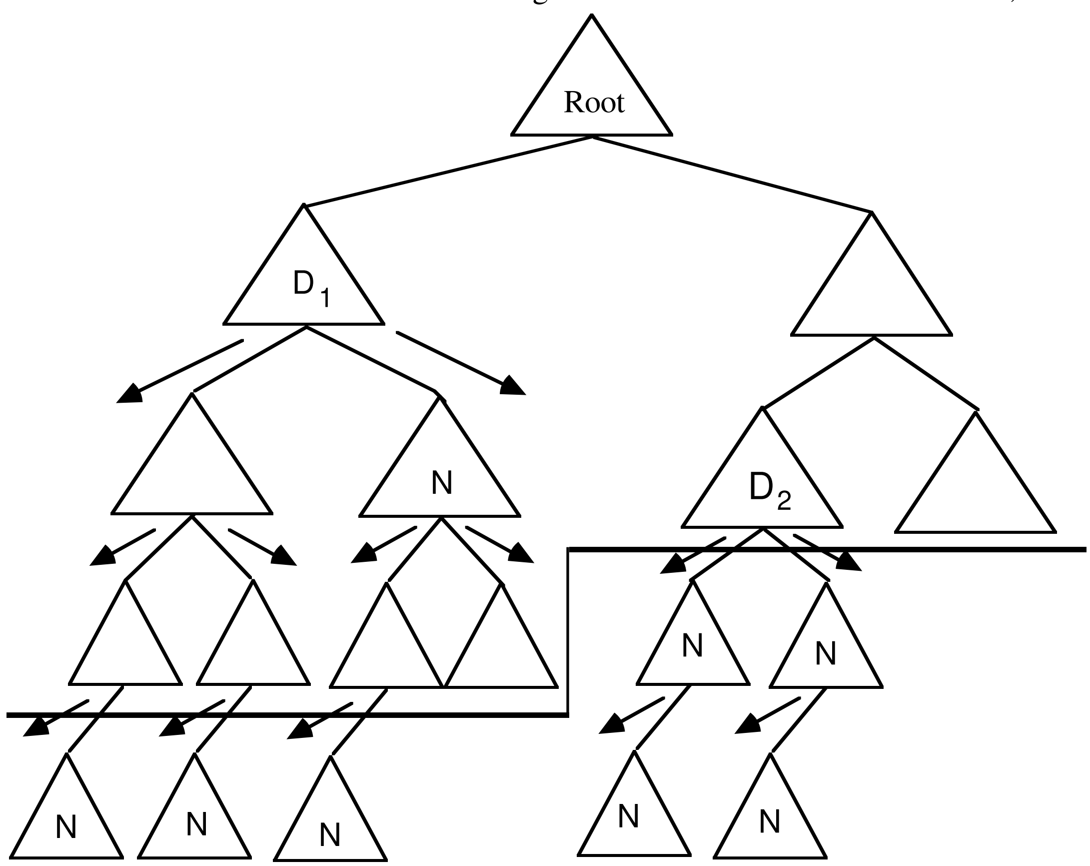

Figure 10. New Node Allocation (DVA)

none are sent to the right child. The left child is allocated, but its children are free. It's left child is allocated. Since all messages are used up, node allocation stops for D_1's right child. After D_1 has sent its messages, it self-deletes and becomes a free node.

D_2 splits the allocation of nodes between its children--each node receives a message to allocate two nodes. Both children are free, so each uses one of its messages. Each child has one remaining message, which is sent to the respective left children. Since these children are free, the nodes are allocated. As no more messages remain, node allocation stops for D_2. After D_2 sends its messages, it self-deletes and becomes a free node.

The node allocation was balanced in this particular example. However, a general worst-case scenario for node allocation by splitting the messages in half would create a linear tree down the left side. There are several possible methods for keeping the tree balanced, all dependent on implementation. One method is to have each node keep a count of the number of free nodes in each of its subtrees. When an allocation message is received, the message can be split up so that the subtrees remain as balanced as possible.

The same count which is used to balance local subtrees can also be used to balance the tree globally. There are two main advantages of balancing: One is decreased tree depth. The other is increased parallelism by distributing the nodes more evenly. These reorganizations make both processing and learning modes faster, increasing the efficiency of the system.

Since different subtrees below nodes are always distinct, new nodes can be allocated (in subtrees) in parallel without contention, down each branch. If an allocation message hits the bottom of the tree, or hits a subtree that is full, the message must be passed up to the parent. *If there is an unused in the network, the node is eventually found and allocated.* Thus, hardware is used efficiently. Some contention may occur if multiple messages are passed to a parent. This is kept to a minimum, since nodes are passed only as high as they need to go. The search time and contention for a free node could be further reduced by using the node counts mentioned above. In addition, protocols can be developed to overcome contention [Feng81].

When the new instance is added to the net by the AU, there is never any contention, since only one node is added.

4.2.3 logical self-deletion of redundant nodes

Redundant nodes only self-delete during learning mode. Communication is the same as for any other type of broadcast. The AU broadcasts a self-delete flag to the root, and the root broadcasts to its children, and so on. Assume that each node knows already whether it is supposed to self-delete or not (details are provided later in this chapter). When a node receives a self-delete flag, if it is supposed to self-delete, it sets its polarity flag marking itself as a free node, and passes the self-delete flag on to its children. If the node does not self-delete, it simply passes the flag on to its children. A node that self-deletes does not physically disconnect itself from the net--it marks itself as “unused” or free. From that point on, until it is reallocated, the node simply passes on any messages it receives.

Each of the four types of communication, broadcast, gather, node allocation and node self-deletion have been illustrated. These have shown how the tree topology supports instance learning and execution. The next section shows how the net (as a unit) communicates with the outside environment.

4.3 A Block View of the LIA Model

Figure 11 is a block diagram of the LIA model in *execution* mode. These parts are listed below with their functions:

- Input Binder: The input binder receives input from the environment and broadcasts to the root node.
- Output Binder: Receives input from the net and sets the value of Z accordingly.
- Logic Network: The net is composed of the nodes and the interconnect in between them. This is where instances are stored, and where the net's output is

computed. In this mode, the net acts like a parallel hardware circuit.

Figure 12 is a block diagram of the LIA model in *learning* mode. The parts shown in figure 12 function differently in learning mode than in execution mode.
The parts and their functions are listed below:

- Input Binder: Under the direction of the AU, the input binder allocates new input lines for new input variables.
- Adaption Unit (AU): The AU controls adaptation during learning. It receives instances and broadcasts them to the network, and directs allocation of new lines in the input binder.
- Logic Network: This is where nodes store instances, and hence, where the interesting part of learning takes place.

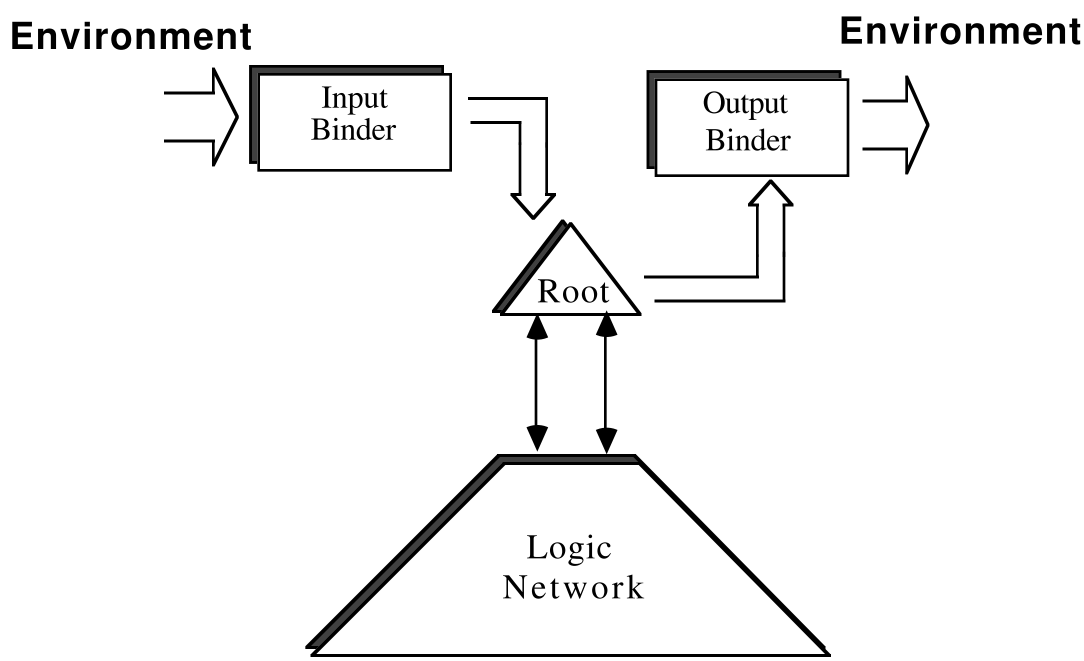

Figure11.LIA Execution Mode Block Diagram

The output binder is not active in learning mode because Z is the only output variable allowed. If multiple output variables are allowed (as discussed in Chapter 7), then lines are allocated for new outputs under the direction of the AU, the same as for new input variables.

Chapter 3 describes an instance set as existing in a universe of boolean variables which has many possible states. The LIA model views its external universe as an **environment** composed of boolean features. A net stores those features which allow it to discriminate inputs [Mart86].

4.4 How the LIA Architecture Functions in Execution Mode

This section shows how the node and the rest of the model function in execution mode.

In each node, only the AND gate is active during execution mode. The node re

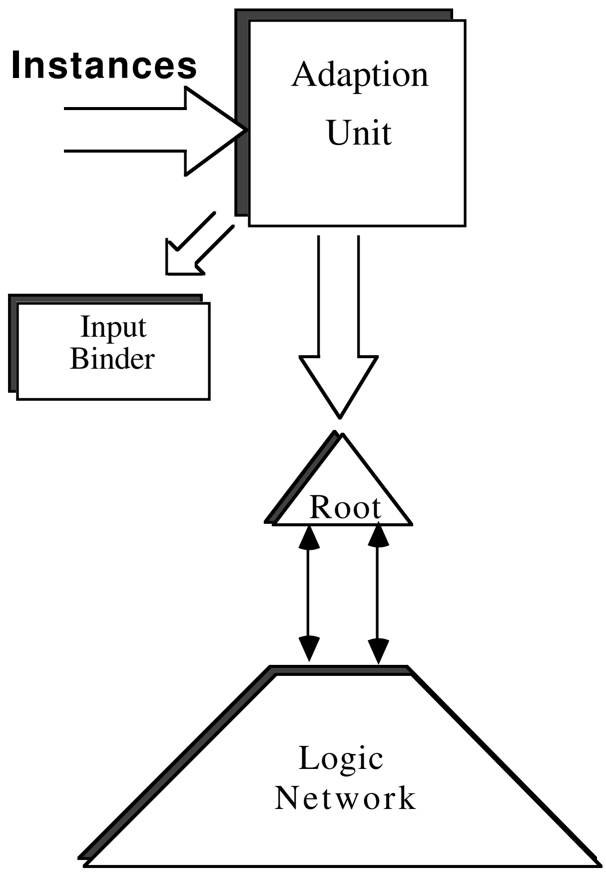

Figure 12. LIA Learning Mode Block Diagram

ceives the input from its parent, and sends the value of its polarity flag to its parent if the node's variables are matched by the input. If the node's variables are not matched, the node does not send its polarity flag's value. For example, suppose a node's state is variable list ABC with polarity flag Z. The input into the network from the environment is ABCD. Regardless of what other nodes may do, this node will send a "positive" to the output binder. Now suppose the input is AB'CE. The node's variables are not matched by the input, so the node does not send its polarity flag's value.

No reconfiguring or learning takes place in execution mode. The network as a whole acts like a set of parallel hardware gates, passing signals and producing an output. The other active parts of the architecture are I/O binders. The input binder broadcasts to the root node. The root passes the input to the nodes below it, and they to the nodes below them, etc. If a node is matched, it immediately sends its polarity flag to its parent. The parent passes the result on to its parent, on up to the root. The root node sends the value it receives or computes to the output binder. This value becomes the network output.

The net has four possible output values:

1. positive
2. negative
3. “don't know”
4. error

The output is positive if one or more positive instances are matched by the input. The output is negative if one or more negative instances match. The output is “don't know” if none of the instances match. An error occurs <u>only</u> when a node receives both a negative and positive signal, meaning that a positive and a negative instance were both matched. In the model, this is impossible. However, an error can occur in a real architecture if some part of the architecture is faulty. Detecting these errors is one way to detect faulty parts.

4.5 LIA in Learning Mode

In learning mode, the AU is active, and directs the modification of the input binder and the net. The AU broadcasts a new instance to the net. The AU sends the new instance to the root node, and the root node sends the new instance to its children. The input binder keeps track of any new variables introduced to the system. Each node tests itself against the new instance, possibly changing the node state, including deleting itself, if necessary. Since the net has only one output variable Z, the output binder does not need to change, and is therefore not active.

Inside each node, the CU has control of changing the state. The CU receives a new instance from the AU (possibly via other nodes), tests the node's current state against the new instance, and alters the state if necessary. In learning mode, every node receives the new instance, and every node operates independently of any other node in the network.

Learning means that a network changes over time to adapt to new information. A **learning algorithm** is an algorithm that specifies how a network should change. Pseudocode for the functions that each node executes at each step in the learning algorithm is listed below.

The learning algorithm presented in this section is the instance set modification algorithm of chapter 3 applied to a net rather than an instance set. The notation is the same as chapter 3, with one addition: NI.PF and OI.PF represent the polarity flag or PF. The four steps are as follows:

Step 1: Check for nodes that cover and are concordant with the new instance.
Step 2: Check for inconsistent nodes.
Step 3: Check for nodes which need to self-delete (redundant nodes).
Step 4: Add the new instance to the network.

The operation of the LIA learning algorithm is as follows:

Step 1: Check for nodes that cover and are concordant with the new instance.

The AU broadcasts the fulfill flag, NI.V and NI.PF.

Each node checks to see if it is Concordant Subset or Concordant Equal with the new instance.

If any node returns true to the AU (via its parent), then processing stops for this new instance, because there exists at least one node that covers and is concordant with the new instance. No modifications to the net are needed.

Pseudocode:

```
function fulfill: boolean;
begin
    Fulfill =
    Concordant(OI,NI) and
    ( (Subset(OI,NI) or Equal(OI,NI) );
end;
```

Step 2: Check for inconsistent nodes.

The AU broadcasts the inconsistency flag, NI.V and NI.PF.

Each node checks to see if it is Discordant and not Discriminated (Inconsistency lemma).

If any node reports true, the AU broadcasts a DVA command. All nodes that previously returned true add new nodes to the net by doing DVA; other nodes do not. Each node initiating DVA does the following:

1. Compute the set of variables NI.V - OI.V. Invert the polarity of each element of the set. This is the set DiscrList of possible discriminant variables (see Chapter 3).
2. For each member *w* of DiscrList do the following:

a. choose a *w* from DiscrList and delete it from DiscrList.
b. allocate a free node
c. make the new node's variable list OI.V concatenated with ~*w*.
d. make the new node's polarity opposite the new instance (same as the original conflicting old instance).

3. Delete OI (the original conflicting node) from the net.

If DiscrList is empty, as is the case with Superset or Equal old instances, no nodes are added to the net.

Pseudocode:

```
function inconsistency: boolean;
    begin
        Inconsistent =
        ( Discordant(OI,NI) and NOT(Discriminated(OI,NI)) )
    end;        {inconsistency function}
procedure DVA;
begin
    let DiscrList = NI.V - OI.V, with each element's polarity inverted;
    for each w in DiscrList do
    begin
        allocate(newnode);
        newnode.V = w + OI.V;                {concatenation}
        newnode.PF = ~(NI.PF);
    end;    {for each w}
    Delete OI from network;
end;                {DVA procedure}
```

Step 3: Check for nodes which need to self-delete (redundant nodes).

The AU broadcasts the self-delete flag, NI.V and NI.PF.

Each node checks to see if it is Concordant Superset with NI. Since any Discordant Superset old instances are deleted in Step 2 (inconsistencies), it is not necessary to check for Concordance at this point. Any node that is a Superset deletes itself from the net.

Pseudocode:

```
procedure self-delete;
begin
    if ( Superset(OI,NI) ) then
    Delete OI from the network;
end;
```

Step 4: Add the NI to the network.

The AU allocates a free node and copies the new instance into the node.

Pseudocode:

```
procedure AddNI;
    begin
        allocate(newnode);
        newnode.V = NI.V;
        newnode.PF = NI.PF;
    end;        { procedure AddNI }
```

CHAPTER 5
AN EXAMPLE OF LIA LEARNING AND EXECUTION

Section 1 of this chapter gives an example of how the LIA machine model operates in learning mode. Section 2 uses the net built in section 1 to show how execution mode works.

5.1 Learning Mode

Assume that the following eight instances are presented to an empty network in numbered order:

1. AB'C ⇒ Z'
2. A'BDE ⇒ Z'
3. BDE ⇒ Z'
4. ABC ⇒ Z
5. A'D ⇒ Z
6. BC'DEF ⇒ Z'
7. ABCD' ⇒ Z
8. ABCD ⇒ Z

This learning example illustrates all of the adaptive actions listed in Table 1 of Chapter 3.. The nodes are numbered in each figure. This is for reference only, and does not reflect any order nor precedence among the nodes.

Instance 1. AB'C ⇒ Z'. This is the first instance to be broadcast. At this point, no nodes are in the network.

The AU broadcasts the new instance and the fulfill flag. Because the network is empty, it does not fulfill the new instance. Therefore, the other three steps of the algorithm must be executed.

The AU broadcasts the inconsistency flag. There are no responses from the empty network. The same is true when the self-delete flag is broadcast.

The AU executes the AddNI procedure, so a new node is created to hold the new instance. Figure 13 shows the network.

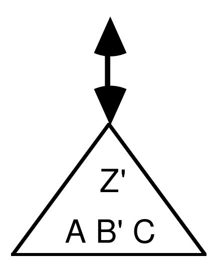

Figure 13. After Instance 1

Instance 2. A'BDE ⇒ Z'. The AU broadcasts the new instance and the fulfill flag. Node 1 compares itself to the new instance, and determines that it is Concordant Overlap. It does not cover the new instance. Therefore, the other three steps of the algorithm must be executed.

The AU broadcasts the inconsistency flag. Node 1 is not inconsistent with the new instance, because the two are concordant.

The AU broadcasts the self-delete flag. Node 1 does not self-delete because it is Concordant Overlap.

The AU executes the AddNI procedure, and the new instance becomes a left child to node 1. Figure 14 shows the new network.

Instance 3. BDE ⇒ Z'. The AU broadcasts the new instance and the fulfill flag. Node 1 is Discriminated, node 2 is Concordant Superset. Neither node covers the new

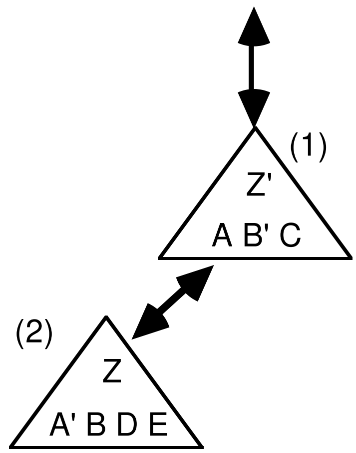

Figure 14. After Instance 2

instance, so the network does not fulfill it. Therefore, the other three steps of the algorithm must be executed.

The AU broadcasts the inconsistency flag. There are no inconsistencies.

The AU broadcasts the self-delete flag. Node 2, being Superset, self-deletes. The AU executes the AddNI procedure, and the new instance becomes the new left child to node 1. Figure 15 shows the network.

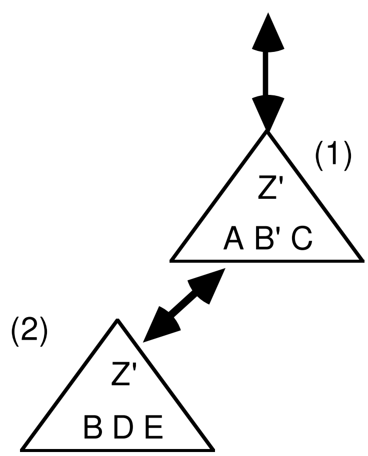

Figure 15. After Instance 3

Instance 4. ABC ⇒ Z. The AU broadcasts the new instance and the fulfill flag. Node 1 is Discriminated. Node 2 is Discordant Overlap. These nodes do not cover the new instance, so the network does not fulfill it. The other three steps of the algorithm must be executed.

The AU broadcasts the inconsistency flag. Node 2 reports that it is a DVA node. The AU receives the report and then sends out a flag to start DVA. Node 2 begins. It creates an DiscrList composed of {ABC} - {BDE} (DiscrList from the Consistency Theorem) with each member complemented, namely {A'C'}. Node 2 creates a left child and passes the child the message { {BDE}, {A'}, Z' }. It creates a right child, and passes the child the message { {BDE}, {C'}, Z' }. Then node 2 deletes itself from the network. The left child sets up its variable list and polarity, and represents the instance

A'BDE ⇒ Z'.

Its DiscrList is empty, since A' is used, and so there are no more DVA messages to send, and it stops processing. The right child concurrently sets up its variable list and polarity, and represents the instance

BC'DE ⇒ Z'.

Its DiscrList is empty, since C' is used, and so there are no more DVA messages to send, and it stops processing. DVA is complete for node 2, and for the net.

The AU broadcasts the self-delete flag. There are no superset nodes to self-delete.

The AU executes the AddNI procedure. The new instance becomes a right child of node 1 (to keep the tree balanced). Figure 16 shows the modified network.

Instance 5. ABCD ⇒ Z'. The AU broadcasts the new instance and the fulfill flag. Nodes 1, 4 and 5 are Discriminated. Node 3 is Discordant Subset. Node 3 covers the new instance, but is discordant. Therefore the other three steps of the algorithm must be executed.

The AU broadcasts the inconsistency flag. Node 3 reports that it is a DVA node. The AU receives the report and sends out a command to start DVA. Node 3 begins. Node 3's DiscrList will be {ABCD} - {ABC} with each element complemented, namely {D'}.

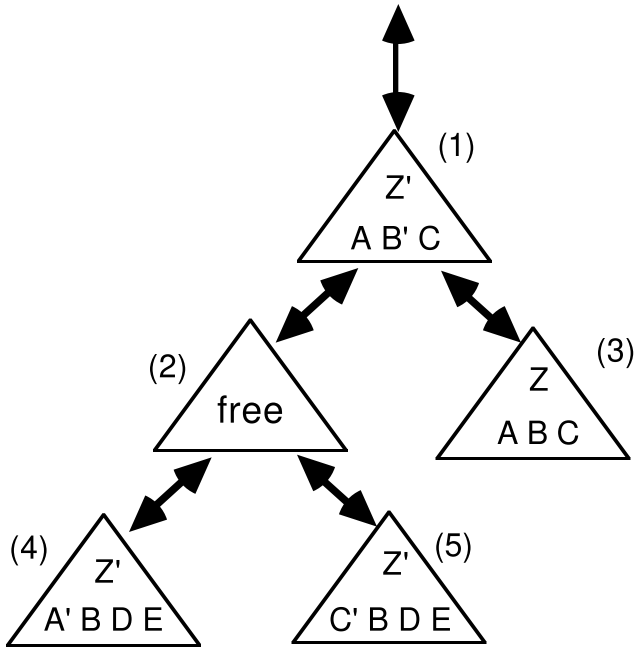

Figure 16. After Instance 4.

Node 3 creates a left child and passes it the message { {ABC}, {D'}, {Z} }. No right child is needed. Node 3 deletes itself from the net. The left child sets up its variable list and polarity, and represents the instance

ABCD' $\Rightarrow$ Z.

DVA is complete for node 3, and for the net.

The AU broadcasts the self-delete flag. There are no superset nodes, therefore none self-delete.

The AU executes the AddNI procedure, and the new instance becomes Node 2. The new network is shown in figure 17.

Instance 6. BC'DEF $\Rightarrow$ Z'. The AU broadcasts the new instance and the fulfill flag. Nodes 1, 2, and 6 are Discriminated. Node 4 is Concordant Overlap. Node 5 is Concordant Subset.

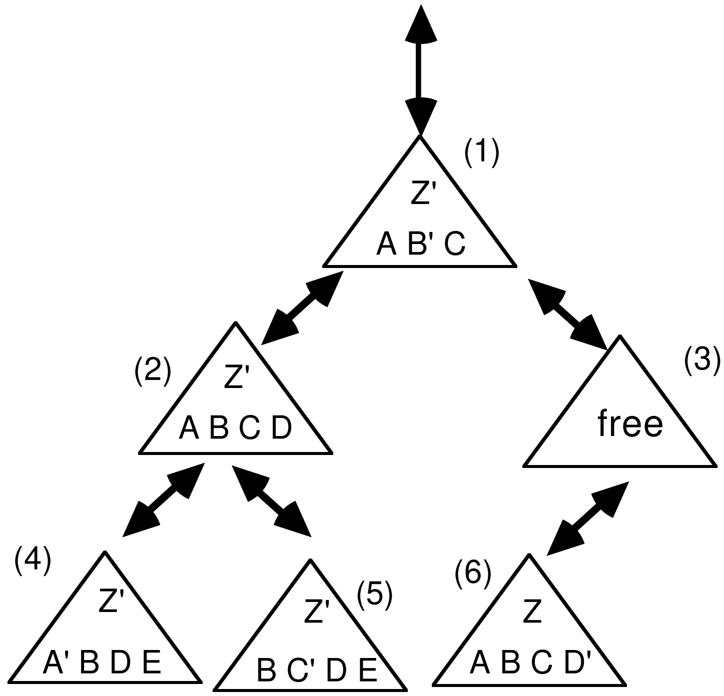

Figure 17. After Instance 5

Node 5 reports back to the AU that it covers and is concordant with the new instance, and processing stops for this new instance since it is already fulfilled. No modifications are necessary. The network remains as in figure 17.

Instance 7. ABCD' ⇒ Z. The AU broadcasts the new instance and the fulfill flag. Nodes 1, 2, 4, and 5 are Discriminated. Node 6 is Concordant Equal, exactly the same as the new instance. Node 6 reports back to the AU that it covers and is concordant with the new instance, and processing stops for this new instance. No modifications are necessary. The network remains as in figure 17.

Instance 8. ABCD ⇒ Z. The AU broadcasts the new instance and the fulfill flag. Nodes 1, 4, 5 and 6 are Discriminated. Node 2 is Discordant Equal. Node 2 covers the new instance, but is discordant, so the other three steps of the algorithm must be executed.

The AU broadcasts the inconsistency flag. Node 2 responds, being Discordant Equal. The AU sends out a flag to start DVA. Since DiscrList for Node 2 is empty, Node 2 deletes itself without adding any children to the network.

The AU broadcasts the self-delete flag. None of the nodes self-delete.

The AU executes the AddNI procedure. The new instance becomes Node 2. Figure 18 shows the final network.

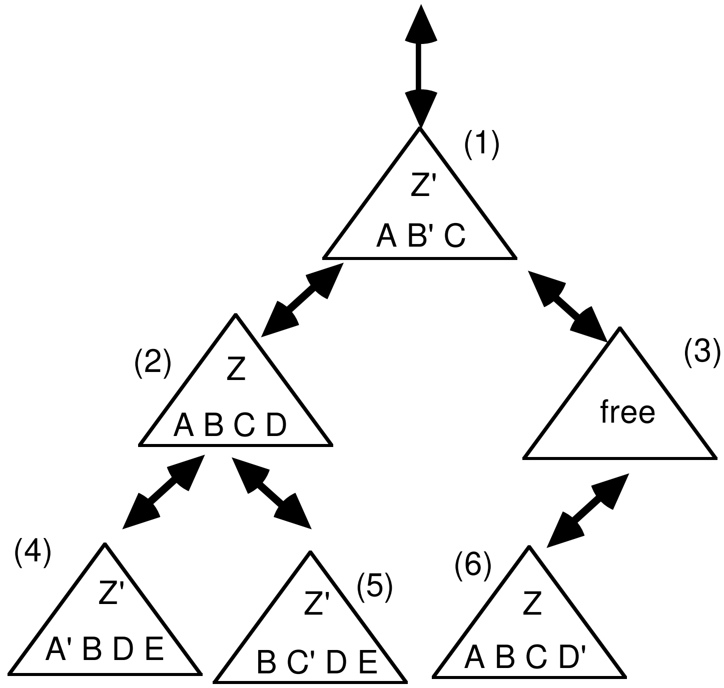

Figure 18. Final Network

There are no more instances to add. Each one of the adaptive actions discussed in chapter 3 have been shown.

5.2 Execution Mode

This section illustrates how the net operates in execution mode, using the network of figure 18.

The network G is surrounded by an external environment with which it communicates. G interprets its environment as a set of boolean features. The state of the environment at any particular moment can be described using boolean variables, in the same manner as the boolean universe of chapter 3 is described. The set U is the set of variables defined for the environment, and the state of the environment is E. Assume that for a given E, every variable in U has a defined value.

Each node's variable list (OI.V) is a subset of U in some state E. OI.V is matched by E iff OI.V is a subset of, or equal to, E.

In execution mode, E flows into input binder, and is broadcast to the root node. The root node sends E to its children, and so on. As each node receives E, it tests itself to see if it is matched by E. Any nodes that are matched immediately send their polarity flags to their respective parents. The parents pass the flags to their parents, up to the root node. The root sends its result to the output binder. If none of the nodes is matched by E, the output binder activates the "don't know" signal.

Given a net in execution mode (such as the one in figure 18) and E, one of three conditions is possible:

- At least one node is matched by E. The output is positive or negative (but not both), and the "don't know" and error signals are negative.
- None of the nodes matches E. The "don't know" signal is positive, the output and error signals negative.
- The error signal is positive, the output and "don't know" signals negative. This case only occurs when the output binder receives both a positive and negative signal from the root (i.e. the net). Since the net is consistent after the end of

learning mode, either some node or some other part of the hardware is faulty. This never occurs in the model; it is possible only for a real architecture. Methods of fault tolerance and error recovery are beyond the scope of this thesis.

Note that no learning ever happens in execution mode. An example for positive, negative and "don't know" output is now given using figure 18. Each node receives E in parallel from its parent.

E is ABCD'E. E flows into the input binder, which passes it to node 1. Node 1 is not a Subset nor Equal nodes, and therefore, is not matched. Nodes 2, 4, and 5 are Discriminated. Node 6 is Subset, so it sends "positive" to node 3. Node 3 passes "positive" to node 1, which is the root node. No other nodes respond, so "positive" is sent to the output binder.

E is AB'CDE. Node 1 is a Subset node, and sends "negative" to the output binder. Nodes 2, 4, 5, and 6 are Discriminated, and do not respond. The result is a negative output for the network.

E is A'B'C'D'E'. None of the nodes is matched by E. Nodes 1, 2, 4, 5, and 6 are Discriminated. None of the nodes sends a polarity flag to its parent. The output binder does not receive any signal from the root node, and so activates the "don't know" output.

CHAPTER 6
ANALYSIS AND COMPARISON OF LIA

This chapter discusses the features and limitations of LIA, and presents a brief comparison of LIA with AA1, AA2, and AA3. The PRS (Prioritized Rule System ASOCS) model [Hugh89] is discussed at the end of this chapter, but only relative to DVA.

6.1 Advantages and Limitations of LIA

The main motivation for LIA is to design an ASOCS model that is easily implementable using current technology. AA1, AA2 and AA3 all require dynamic interconnect topologies, but current technology does not easily support dynamic interconnect. LIA has the following characteristics that are advantageous for potential implementation using current technology:

- dynamic interconnect is not needed
- instances stored in a net may be arbitrarily rearranged in any order without affecting the network function
- any desired interconnect topology may be used

As technology advances, it should become easier to implement the other three models. LIA is designed to "bridge the gap" between previous ASOCS models and current technology. This section focuses two on ideas:

1. Why AA1, AA2 and AA3 require dynamic interconnect, and the problems with implementing them with current technology.
2. LIA's advantages and limitations.

AA1, AA2 and AA3 all require dynamic interconnect because instances are stored in a net as hierarchies of two-input nodes. A single two-input node cannot store all the information about an instance that has more than two variables in its variable list. To fulfill such an instance, more than one two-input node is needed. The nodes must be connected in some hierarchical order. Figure 19 shows one way the instance

$$ABCD \Rightarrow Z$$

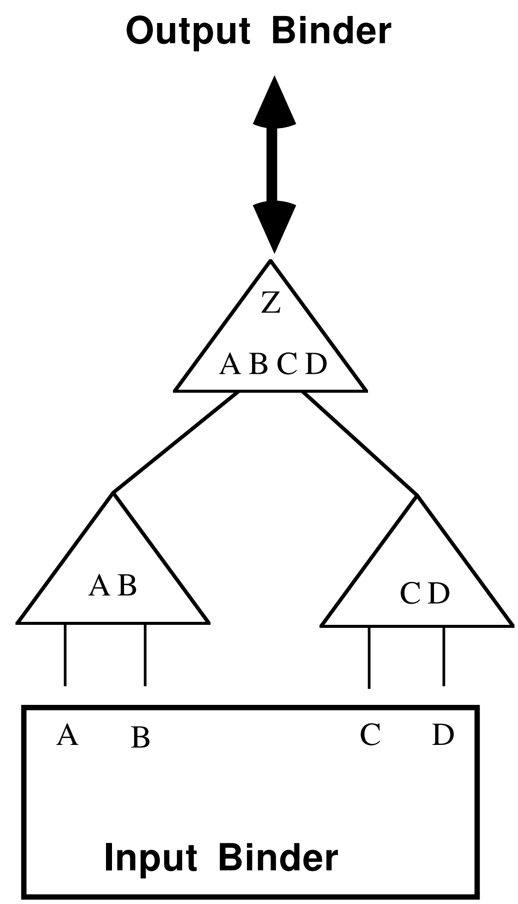

Figure 19. Instance ABCD $\Rightarrow$ Z Stored In An AA2 Net

could be stored in an AA2 net. Each node in the figure performs the AND function on its inputs. Thus, Node 2 is the conjunction AB (both high). Node 3 is the conjunction CD (both high). When all four variables are high, the outputs of both nodes 2 and 3 are high, so both inputs to Node 1 will be high. Node 1 sends a "positive" to the output binder, as indicated by the Z at the top of Node 1. Thus, Node 1 logically represents the conjunction ABCD, and the instance

ABCD $\Rightarrow$ Z.

The need for a hierarchical interconnect for AA1 and AA3 can be similarly demonstrated.

In AA1 and AA2, a particular physical node may potentially need to be connected to any other physical node in the network to build up and fulfill an instance [Mart86]. Dynamic interconnect is the most natural, intuitive way to implement the interconnect. As figure 20 shows, the question then becomes "How is Node X directly routed to Node Y dynamically?"

Dynamic interconnect is often expensive and complex to implement with current technology. AA3 is an attractive alternative to AA1 and AA2 because AA3 uses a static binary tree interconnect between nodes in a single tree, with dynamic interconnect between clusters of trees.

Figure 21 shows an example of a single AA3 binary tree. The root is at the bottom, the subtrees extend toward the top. Assume that all the nodes that are showing are allocated--free nodes are not shown. Each of the nodes (other than the root) receives input its parent, and a variable in the input binder. The connection to the variable input is dynamic, but the connection to the parent is fixed.

It is impossible to know *a priori* which branches will grow longer than others. If the instances in an instance set are nearly the same length, then an AA3 tree stays fairly full (and balanced). However, if a few instances are much longer than the others, then some branches may grow much longer than others (see figure 21). If a branch grows too long and runs out of room, then more nodes may have to be added to the tree on that branch. The most practical solution is to dynamically connect the end of one static tree to the top of another. Refer back to figure 20, but now consider each triangle to be a binary tree of nodes instead of a single node: AA3, because of the dynamic interconnect between trees, shares the implementation problems of AA1 and AA2, though the problems are not as great.

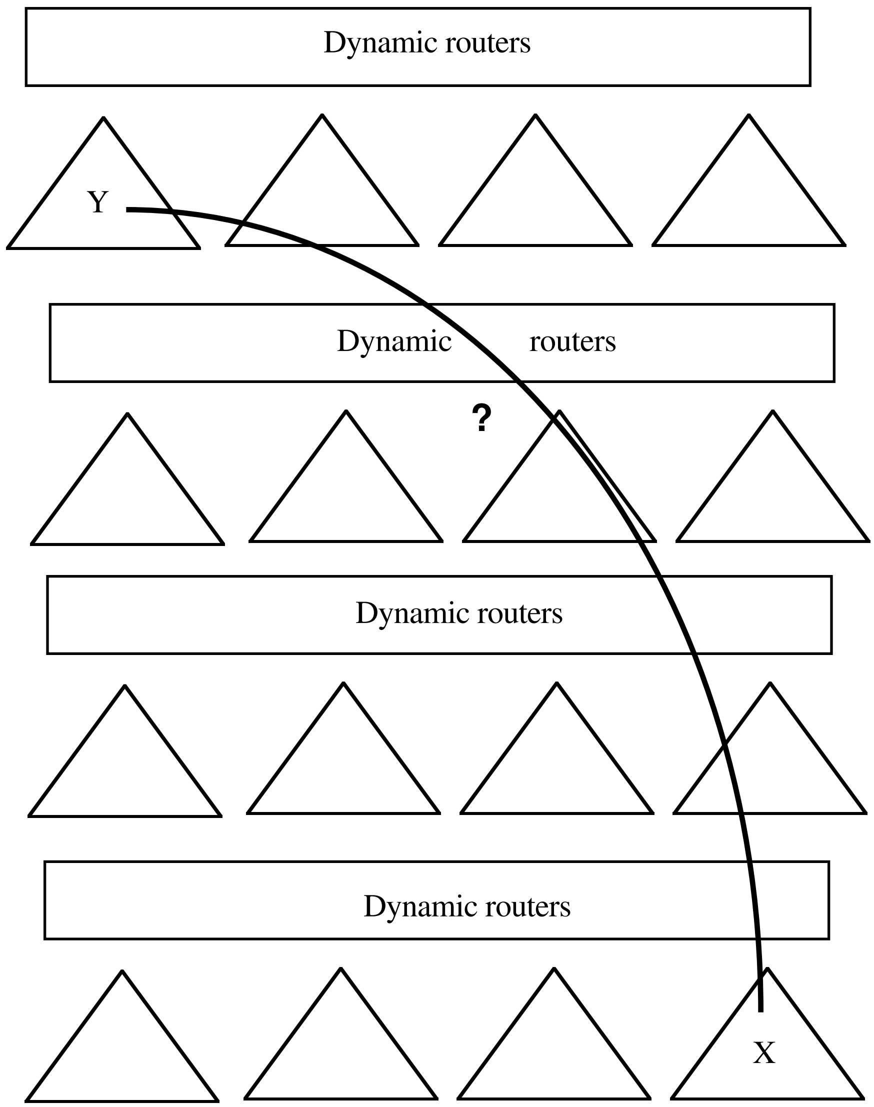

Figure 20. How is X Directly Connected to Y Dynamically?

LIA avoids implementation problems with dynamic interconnect by using an m-input node (instead of a two-input node) with static interconnect. "m" is the number of variables defined in the system. In the LIA model, each node stores a complete instance (Chapter 4 discusses the LIA architecture in detail). Since each node stores a complete instance, all the information about a particular instance is available at the single node where it is stored. *This eliminates the need for a logical and physical hierarchy among the nodes.* Each node is logically and physically independent of any other node. This has the follow

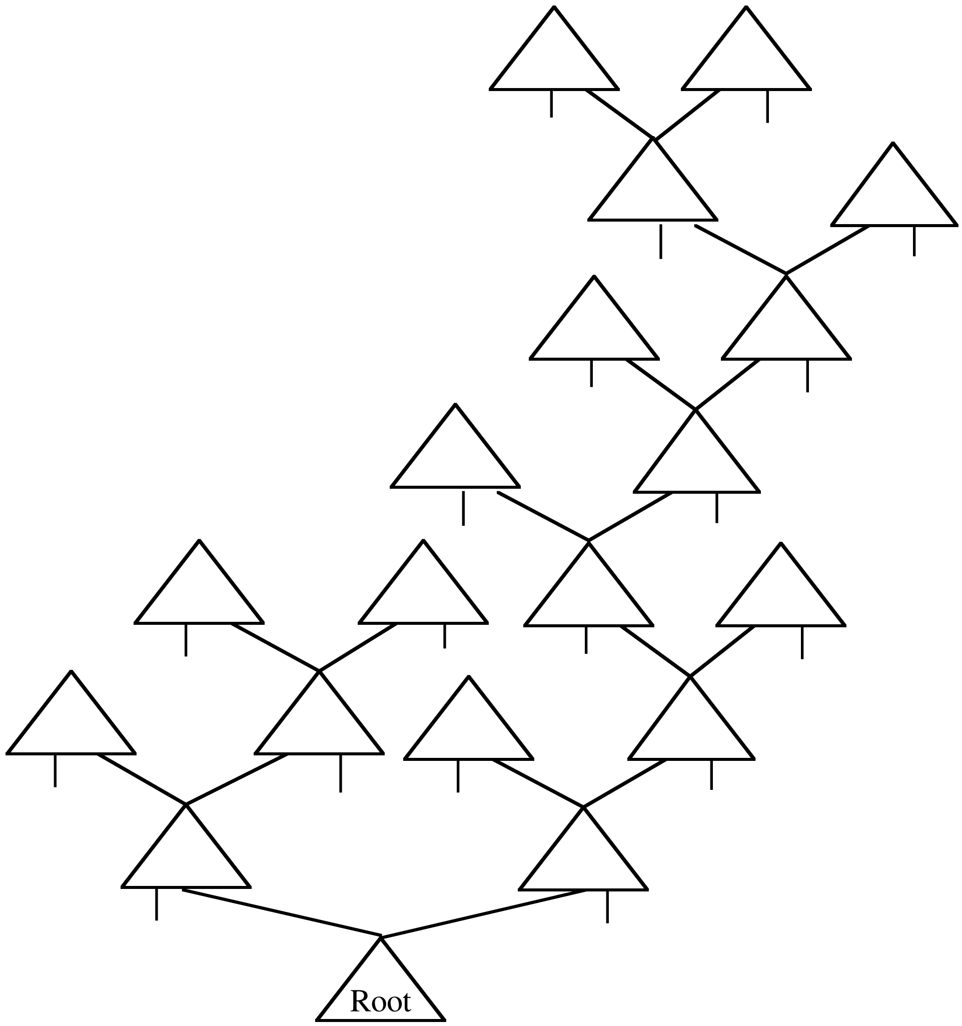

Figure 21. AA3 Binary Tree

ing implications, which are advantageous for hardware implementation:

- Dynamic interconnect is not needed.
- Instances stored in a net may be arbitrarily rearranged in any order without affecting the network function.
- Any desired interconnect topology may be used.

Figure 22 illustrates how LIA nodes are logically independent of one another, how nodes may be rearranged arbitrarily, and how physical position is not important (hence the name Location-Independent ASOCS). Instances 1 and 2 can be switched, and the network remains functionally unaffected. Dynamic routing is not needed because arbitrary nodes are never forced to communicate with each other.

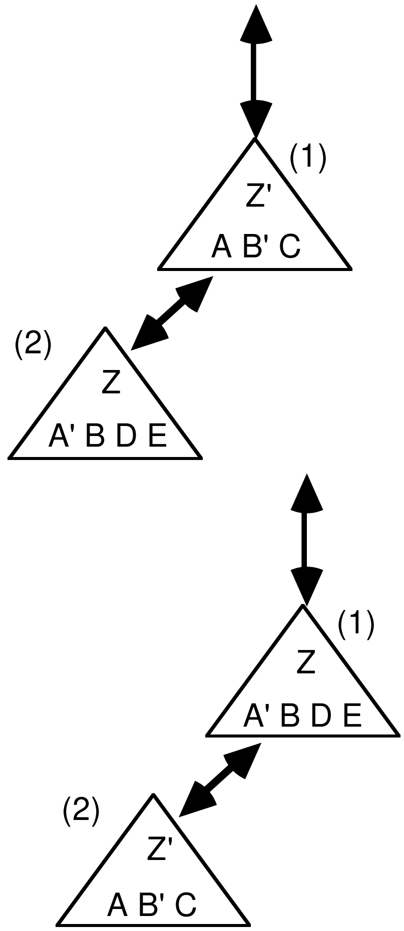

Figure 22. Switched Nodes

Because the nodes are logically independent, the topology of the physical interconnect is unimportant to the functional success of the network. Thus, any desirable interconnect may be used for communication. Chapter 4 describes a binary tree interconnect, but a simple bus, nearest-neighbor mesh, or a hypercube might also be used. (Different topologies will have different longest paths and other characteristics, however.) A static binary tree (such as Chapter 4 describes) can be reorganized (or balanced) periodically to optimize the longest path and make efficient use of free nodes and computing time.

The flexibility in implementation that LIA offers does not come without a price. Each LIA node must have the potential to receive input from the environment for all variables defined for the system at any given moment. Since the number of defined variables is assumed to be dynamic, the number of inputs to a node must also be dynamic. An m-input AND gate, where m is typically greater than two, is logically more complex than a simple two-input AND gate. (Physical implementations for a node are not considered in any detail in this thesis.)

6.2 Comparison With Other ASOCS Models

This section gives a brief comparison of LIA with AA1, AA2 and AA3. The comparison is summarized in Table 2. (As mentioned at the beginning of the chapter, PRS, though shown in the table, is discussed here only in connection with DVA at the end of the chapter.) LIA appears to compare well with AA1, AA2 and AA3 based on the contents of

Table 2. However, because LIA's nodes and topology are different, any but the most general comparison is useless. (Again, the main strength of LIA is that it is implementable using current technology.)

Of the categories listed in Table 2, section 6.1 discusses the following three:

1. interconnect topologies
2. LIA stores an instance in each node (i.e. Node-Per-Instance), whereas AA1, AA2 and AA3 do not.
3. LIA uses an m-input node, where m is the number of defined input variables, whereas AA1, AA2 and AA3 all use two-input nodes.

These ideas are not discussed further here. However, the remainder of the categories are discussed here. These are the following:

4. amount of memory required at each node
5. average number of nodes per instance
6. depth
7. use of DVA

	LIN	AA1	AA2	AA3	PRS
Interconnect Topology	Static Binary Tree [1]	Dynamic Hierarchy Required	Dynamic Hierarchy Required	Static Binary Tree (Hierarchy) Required	Static Binary Tree [1]
Node Per Instance	yes	no	no	no	yes
Node Type	m-inputs [2]	2-inputs	2-inputs	2-inputs	m-inputs [2]
Memory at Node	2m bits required [2]	5n bits required [3]	2m bits required [2,5]	2m bits required [2]	2m bits required [2]
Avg. Number of Nodes Per Instance	1	1	2	2	1
Depth	$\log_2(n)$ [3, 4]	n^x $0 \leq x < 1.5$ [3]	$\leq m$ [2]	$\leq m$ [2]	$\log_2(n)$ [3, 4]
Use DVA	yes	yes	yes	yes	no

[1] any desirable interconnect topology may be used instead of a binary tree

[2] m is the number of defined input variables

[3] n is the number of instances stored in the net (or the IS)

[4] varies with the interconnect topology used

[5] AA2 can be implemented with no memory at the node [Chang87].

Table 2. Comparative Summary

All of the models require 2m bits at the node except AA1, which requires 5n bits. m is the number of defined input variables, n is the number of stored instances. Each input variable has three-state logic--positive, negative and "don't know". (These statistics do not count the output variable Z.)

LIA always uses 1 node per instance (a different type of node), for all instance sets. AA2 uses about 2 nodes per instance, AA1 and AA3 may use less than or more than 1 node per instance. In the latter three models, the amount varies.

The depth for LIA (assuming the tree is balanced) is always $\log_2(n)$, where n is the number of nodes in the net. The depth for AA2 and AA3 is always ≤ m, where m is the number of defined input variables. AA1 varies widely. Which depth is shorter depends on the exact instance set in question.

One problem that LIA shares with AA1, AA2 and AA3 is related to DVA and is called the **DVA Growth Problem**. LIA does not attempt to solve this problem; the PRS model avoids this problem by not using DVA to resolve inconsistencies. This discussion does two things:

1. place an upper bound on the change in size of an instance set (or LIA net) due to inconsistencies with a new instance (i.e. the change in size when DVA is executed), and define the DVA Growth Problem
2. show how the PRS model resolves inconsistencies without using DVA

As shown earlier in the thesis, DVA is the operation that makes an instance set consistent with a new instance (the Consistency Theorem). Assume there is an instance set G, a new instance whose variable list has length n, and that m old instance's in G that are inconsistent with the new instance. For each inconsistent old instance, there are <u>at most</u> n possible discriminant variables in the new instance (the Consistency Theorem). If there are m inconsistent old instance's, then the maximum number of new instances added to the instance set to resolve inconsistencies is mn. Given that m instances are also deleted (the inconsistent ones) and the new instance added, the total <u>change</u> in instance set size is never greater than

$$mn - m + 1.$$

Thus,

$$mn - m + 1$$

is an absolute upper bound on the change in size of an instance set due to DVA.

In most cases, this means that the size of an instance set is much bigger than the size of an equivalent (but possibly inconsistent) group of instances that have been presented. That is, the use of DVA often causes a small number of instances to fragment into a large number because of inconsistencies, yet the two sets describe the same logical space (are equivalent). This problem is the DVA growth problem.

The PRS (prioritized rule system) ASOCS model is designed to eliminate the DVA growth problem. It was created as an outgrowth of this research on the LIA model. PRS nodes also fulfill complete instances like LIA, but a priority is kept at each node in addition to the instance information. During instance set modification, the new instance always is given a priority higher than any inconsistent OI, and then added to the instance set. Since the node with the highest priority always "wins" (gets to assert its output), this makes the instance set consistent. This method guarantees that the size of the instance set will grow by <u>at most</u> one instance for every new instance (and may get smaller if other nodes can be deleted). Since DVA is not used, the DVA growth problem is avoided.

In summary, the logical and physical flexibility that LIA offers as compared to models AA1, AA2, and AA3 make it potentially easier to implement using current technology. The advantages are the following:

- Dynamic interconnect is not needed.
- Instances stored in a net may be arbitrarily rearranged in any order without affecting the network function.
- Any desired interconnect topology may be used.

LIA's flexibility comes from the fact that a single node stores an instance.

The main cost is that the LIA m-input node is logically (if not physically) more complex than the two-input node used by AA1, AA2 and AA3. LIA uses DVA to resolve conflicts, and therefore has the DVA growth problem, as do AA1, AA2, and AA3. (The PRS model shares many of LIA's features, but avoids problems with DVA.)

CHAPTER 7
ADDITIONAL DETAILS IN THE LIA MODEL

Preceding chapters discuss the basic LIA model. This chapter presents additional details which were left out of earlier discussions to simplify the explanations. Four topics are discussed:

1. handling multiple outputs
2. one-difference minimization
3. background minimization
4. working with limited resources

The description of these ideas is not as formal as are chapters 3 and 4. The intent is to acquaint the reader with these ideas rather than to give a formal description.

7.1. Handling Multiple Output Variables

Chapter 3 defines an instance to have only one output variable Z. In general however, an instance can specify other output variables as well. That is, the following are all valid instances:

AB ⇒ Z

CD ⇒ B'

AC ⇒ Y.

Variables Z, B, Y or any other output variables are logically distinct from, and independent of, one another. The only output asserted by an instance is the one listed on the right side.

AB ⇒ Z

affects only Z; it says nothing about the output value of any other variable.

Consistency and minimization apply only to instances with the same output variable. Two instances with different output variables cannot be discordant, because the output values never conflict. Thus, they cannot be inconsistent. For example,

AB ⇒ Z

is not inconsistent with

CD ⇒ B'

because Z and B' are values for two different outputs. These two instances cannot be minimized with each other because the outputs are independent. Such a reduction would be erroneous. Thus, during instance set modification, it is necessary only to check for inconsistencies and minimization between instances that have the same output variable as the new instance.

With multiple output variables, the network hardware can be used more efficiently by "sharing" nodes. Consider the following three instances

AB ⇒ Z'

AB ⇒ C

AB ⇒ X

in which the variable lists are all the same. In a network, one node can store all three instances, instead of three separate nodes. That is, a single node could store

AB ⇒ Z'CX.

Architecturally, the change to the LIA node is simple. Each node has a separate polarity flag for each output variable. During execution, a node sends all of its polarity flags to its parent when it is matched. During learning, the AU broadcasts the output variable's name along with the value. Learning proceeds as explained, with one difference. When an instance is deleted from the net, the variable is removed from the node's polarity flag list; a node is not deleted unless its polarity flag list is empty.

7.2 One-Difference Minimization

The LIA learning algorithm of chapters 3 and 4 is simplified by leaving out a description of one-difference minimization. One-difference minimization is done for each NI at the same time as checking for inconsistencies and redundant Superset instances, during learning. This section defines one-difference minimization for instance sets, presents three theorems about one-difference minimization (without the proofs) and gives algorithms for doing it. The notation is the same as that in Chapter 3. Those interested in more detail should consult [Mart86].

7.2.1 definitions for one-difference minimization

Two instances are **one-difference** instances *with each other* iff they are Concordant and they have exactly one discriminant variable. There are three mutually exclusive classes of one-difference pairs of instances. Examples are shown here with the corresponding class in parentheses:

1. ABC ⇒ Z and ABC' ⇒ Z (One-DiffEqual)
2. ABC ⇒ Z and AB' ⇒ Z (One-DiffSubset)
3. ABC ⇒ Z and A'BCD ⇒ Z (One-DiffSuperset)

The following are *not* one-difference pairs. The reason is given in parentheses:

4. ABC ⇒ Z and ABC' ⇒ Z' (not concordant)
5. ABC ⇒ Z and ABCD ⇒ Z (not discriminated)
6. ABC'D ⇒ Z and ABCD' ⇒ Z (more than one difference)

As shown above, there are three classes of one-difference pairs:

1. One-DiffEqual
2. One-DiffSubset
3. One-DiffSuperset

The definitions are the following:

Let w be the discriminant variable (either positive or negative), and $\sim w$ be it's inverse.

1. OI is One-DiffEqual iff OI and NI are Concordant, OI and NI have exactly one discriminant variable w or $\sim w$, and $\{OI.V - w\} = \{NI.V - \sim w\}$.
2. OI is One-DiffSubset iff OI and NI are Concordant, OI and NI have exactly discriminant variable w or $\sim w$, and $\{OI.V - w\}$ is a subset of $\{NI.V - \sim w\}$.
3. OI is One-DiffSuperset iff OI and NI are Concordant, OI and NI have exactly one discriminant variable w or $\sim w$, and $\{OI.V - w\}$ is a Superset of $\{NI.V - \sim w\}$.

One-Difference instances are a special subset of the Discriminated class defined in the Partition Theorem. Thus, one-difference minimization takes advantage of instances that take no action during instance set modification as defined in Chapter 3. The Partition Theorem can be extended to show that these additional classes are mutually exclusive, but this is left to the interested reader.

7.2.2 one-difference theorems

Three one-difference theorems are presented here, but the proofs are omitted since they are almost identical to those in section 3.2.

One-Difference Equal Redundancy Theorem. *Let G_i be an IS. Let NI be a new instance, y be the polarity of NI, and let $(G_i \cup NI)$ be consistent.*

Let D be the set of all old instances in G_i such that each OI in D is One-DiffEqual with NI. Let E be the set of instances

{ (OI.V - w) $\Rightarrow$ y :
OI is in D,
w is the discriminant variable for OI and NI, and y is the polarity of OI }.

The new instance set

$G_{i+1} = (G - D) \cup E$

is equivalent to G $\cup$ NI.

Suppose the new instance is

ABC $\Rightarrow$ Z

and one old instance is

ABC' $\Rightarrow$ Z.

The OI is modified to become

AB $\Rightarrow$ Z.

Note that the modified old instance covers both the original old instance and the new instance. Thus, the new instance does not need to be added.

One-Difference Covered Redundancy Theorem. *Let G be an IS. Let NI be a new instance, y be the polarity of NI, and let (G $\cup$ NI) be consistent. Let D be the set of all OI's in G such that each OI in D is One-DiffSubset with NI. Choose exactly one instance X from the set D. Let N' be the instance*

{ (NI.V - w) $\Rightarrow$ y :
w is the discriminant variable for X with NI,
and y is the polarity of NI }.

The new instance set

$G' = G \cup N'$

is equivalent to G $\cup$ NI.

For example, if the NI is

ABC $\Rightarrow$ Z

and an old instance

AB' $\Rightarrow$ Z

is chosen for minimization, the new instance is modified to become

AC $\Rightarrow$ Z.

One-Difference Special Case Redundancy Theorem. *Let G be an IS. Let NI be a new instance, y be the polarity of NI, and let (G $\cup$ NI) be consistent. Let D be the set of all OI's in G such that each OI in D is One-DiffSuperset with NI. Let R be the set of instances*

{ (IO.V - w) $\Rightarrow$ y :
OI is in D,
w is the discriminant variable for OI and NI,
and y is the polarity of OI }.

The new instance set

$G_{i+1} = (G - D) \cup R \cup NI$

is equivalent to $G \cup NI$.

Assume an instance set composed of the following instances:

1. AB'CD $\Rightarrow$ Z
2. A'BD $\Rightarrow$ Z
3. A'B'C $\Rightarrow$ Z

Let the new instance be

AB $\Rightarrow$ Z.

The new instance set (with the new instance added) is the following:

1. ACD $\Rightarrow$ Z
2. BD $\Rightarrow$ Z
3. A'B'C $\Rightarrow$ Z
4. AB $\Rightarrow$ Z

Note that instance 3 is not a one-difference instance, and therefore cannot be minimized.

7.2.3 one-difference instance set modification algorithms

The following seven-step algorithm is the instance set modification of Chapter 3 with the one-difference checks added. The one-difference steps, 2, 5 and 6, are in italics. Only these three steps are explained below.

Step 1. If there exists an old instance in G_i that covers and is concordant with the new instance, then exit.

Step 2. If there are any One-DiffEqual old instances in G_i, then modify each such old instance and exit.

Step 3. If there exist any old instances in G_i that are inconsistent with the new instance, then do DVA for each such old instance.

Step 4. If redundant old instances exist in G_i, then delete these old instances.

Step 5. If there are any One-DiffSubset old instances in G_i, then choose exactly one such old instance and modify the new instance.

Step 6. If there are any One-DiffSuperset old instances in G_i, then modify each such old instance.

Step 7. Add the (possibly modified) new instance to G_i.

"Modify" in this case means "drop the one-difference variable". Pseudocode for each of the steps is given below:

Step 2: If there any One-DiffEqual old instances in G_i, then modify each such old instance.

When the one-difference variable is removed from the old instance, the modified old instance covers both its original instance and the new instance. Thus, the new instance does not need to be added. The pseudocode is as follows:

```
for each OI that is One-DiffEqual with NI
begin
        let w be the discriminant variable;
        OI.V = OI.V - w;
end;
exit;
```

If this step is executed, the new instance does not need to be added to the instance set, as explained.

Step 5: If there are any One-DiffSubset old instances in G_i, choose exactly one such old instance and modify the new instance.

If there are one or more One-DiffSubset old instances, then the new instance is minimizable with exactly one old instance, but cannot be minimized with any others to retain the same partial function [Mart86]. The new instance is modified always in this case: If *w* is the discriminant variable, NI.V is replaced by NI.V - *w*. The pseudocode is as follows:

select an OI;
let w be the discriminant variable;
NI.V = NI.V - *w*;
goto step 6.

The modified new instance must be added back into the instance set, but this will be done in step 7.

Step 6: If there are any One-DiffSuperset old instances in G_i, then modify each such old instance.

A one-difference variable for two instances is a discriminating variable for those instances (see Chapter 3). However, since a One-DiffSuperset old instance is always Concordant with the new instance, discrimination between the two is unnecessary, and redundant. Thus, each such old instance can delete it's one-difference variable. Pseudocode for step 6 is as follows:

for each OI that is One-DiffSuperset with NI
begin
    let w be the discriminant variable;
    OI.V = OI.V - *w*;
end;
goto step 7.

7.3 Background Minimization

Two types of minimization are done only as "background" minimizations, i.e. only when a net is in execution mode, or when the net is not actively learning new information from the external environment. These two types are **one-difference rebroadcast** and **DVA rebroadcast**. They are called "rebroadcast" because the net broadcasts (to itself) the instances stored in the net. Some instances were previously broadcast to the net, while others are the result of modifications. However, each instance contains information that the net has previously learned--no new information is being added to the net.

The goal is to reduce the size of the net. However, rebroadcasting large numbers of instances in sequential order would significantly slow down learning time. These two minimizations are therefore done as **background minimization**, in order to keep normal learning very fast. One-Difference rebroadcast is described first, then DVA rebroadcast, followed by an algorithm for doing background minimization.

M-way minimization is a form of minimization beyond the pair-wise comparisons described in this thesis. It is mentioned only to introduce the concept as a possible form of background minimization.

7.3.1 one-difference rebroadcast minimization

Assume an instance set has the two instances

$ABC \Rightarrow Z$

$ABD \Rightarrow Z$

and a new instance

$ABC' \Rightarrow Z$

is presented. The first old instance is One-DiffEqual, and is modified to become

$AB \Rightarrow Z.$

The NI is covered by this modified instance, and so is not added. The new instance set has the following two instances:

$AB \Rightarrow Z$

$ABD \Rightarrow Z$

The second instance is now a Concordant Superset of the first instance, and could be deleted from the instance set. However, detecting redundancies of this kind requires broadcasting each modified instance (in turn) to the network after the original new instance. If newly modified instances are broadcast to the network immediately after the original new instance, learning one new instance could take a significant amount of time. Each new broadcast could potentially cause further changes, necessitating further broadcasts. Martinez calls this cascading effect **minimization ripple** [Mart86].

To avoid the potential slow-down in learning time due to minimization ripple, newly modified instances are not immediately broadcast to the net. Instead, potential redundancies are eliminated in a background minimization process called **one-difference rebroadcast**. During one-difference rebroadcast, every node's instance is broadcast (in turn) to the net, as if it were a new instance. Modified instances (if there are any) are broadcast to the network (in sequence) right after the new instance that caused the modifications. Since the object of background minimization is to minimize the net (or instance set), there is no attempt to avoid minimization ripple.

7.3.2 DVA rebroadcast minimization

When several nodes execute DVA during learning, many DVA nodes are added to the net. No DVA node may conflict with another node in the net (Consistency Theorem, definition of consistency). However, it is possible that some node already in the net covers a DVA node, or that other minimization could occur. Detecting and removing redundant DVA nodes is desirable, but doing so during normal learning slows down the algorithm. Thus this process, called **DVA rebroadcast**, is left as a background process.

How does DVA rebroadcast slow down the learning algorithm? Assume that each potential DVA instance is to be rebroadcast to the net immediately after the new instance that caused DVA. The AU must gather all the DVA instances from the net and sequentially broadcast each one as if it were a new instance. The number of DVA instances added at any time is potentially large, therefore broadcasting each DVA instance could take a long time.

During DVA rebroadcast, each node's instance is broadcast as if it were a new instance. This may cause, for any given instance, any of the previously described minimizations to occur.

7.3.3 a background minimization algorithm

The following is an algorithm for background minimization that includes both one-difference rebroadcast and DVA rebroadcast. Although the two types of minimization are described separately above, only one algorithm is needed to do both types. The redundancies can be checked for and eliminated simultaneously, just as in the normal learning algorithm. This is because background minimization is checking for exactly the same types of redundancies as the normal learning algorithm. Background minimization differs from normal learning in the following three ways:

1. The instances come from the net rather than from the external environment.
2. No inconsistencies between instances are possible.
3. Minimization ripple is used rather than ignored.

The algorithm is presented for an instance set, but applies equally to a net:

While there is at least one old instance in G_i that is either a modified old instance or an old instance that has not been broadcast to the net:

Step 1. Select a modified old instance from G_i. If there are none, select an old instance that has not yet been broadcast to the net. If there are none, end background minimization. Otherwise, make the selected old instance the new instance.

Step 2. If an old instance exists in G_i that covers the new instance, then stop (for this new instance).
Step 3. If there are any One-DiffEqual old instances in G_i, then modify each such old instance and stop (for this new instance).
Step 4. If redundant old instances are in G_i, then delete these old instances.
Step 5. If any One-DiffSubset old instances are in G_i, then choose exactly one such old instance and modify the new instance.
Step 6. If any One-DiffSuperset old instances are in G_i, then modify each such old instance.
Step 7. Add the (possibly modified) new instance to G_i.

The instance set is consistent always both before and after background minimization. Since no new information is being added, there is no need to check for inconsistencies. Thus, this step is left out of the algorithm. Except for this, and for step 1, this algorithm is the same as that described in section 7.2. Therefore, only step 1 is described below.

Step 1. Select a modified old instance from G_i. If there are none, select an old instance that has not yet been broadcast to the net. If there are none, end background minimization. Otherwise, make the selected old instance the new instance.

"Making an old instance the new instance" consists of the following steps:
1. The AU selects a node in the net.
2. The AU creates a new instance out of the node's vlist and polarity flag.
3. The node deletes itself from the net.

7.3.4 M-way Minimizations

M-way minimization compares groups of three (or more) instances together in the same manner as the pair-wise comparison. The purpose is to find smaller network configurations than pair-wise comparisons can produce. This could be implemented as a background process, if desired. Although the idea has been discussed in ASOCS literature, it has not been made a part of any learning algorithm. This type of minimization is not discussed further here, but see [Mart86].

7.4 Limited Resources

Limited resources may become an issue when using a physical network in an application. In a physical network, the number of nodes and the number of inputs lines available for input variables is limited always . That is, the tree has fixed depth, and a fixed number of input lines. The fixed size has two important implications that are discussed in this section:

1. If the number of instances in an application grows beyond the size of the network, more nodes have to be added. If the number of inputs grows beyond the number of input lines, more lines have to be added. What assumptions then, can be made about instances, instance sets and the size of a network used to store them?
2. What is the effect does limiting the size of the net have on the learning algorithm?

Two assumptions are made about instances, instance sets, and the size of a network stores them:

1. For n inputs there are 2^{2^n} possible mappings, but only 2^n are realizable at any one time. For example, there are four mappings possible on one input variable:

$A \Rightarrow Z$

$A \Rightarrow Z'$

$A' \Rightarrow Z$

$A' \Rightarrow Z'$.

If the mapping

$A \Rightarrow Z$

is chosen, then a mapping for A' can still be chosen, but

$A \Rightarrow Z'$

cannot be chosen. A cannot map to Z and Z' simultaneously.

2. Certain problems require exponential hardware to solve, such as storing every possible mapping on n inputs. These are not the class of problems ASOCS, and neural nets in general, target. Target problems involve some subset of the total possible mappings, and require only a reasonable amount of hardware to solve.

[Mart86] and [Mart89] discuss these assumptions in more detail.

In a tree architecture, there is the possibility of bottlenecks anytime distinct messages have to be passed up a tree. However, protocols could be developed to overcome contention [Feng81]. Another problem is the space-time trade-off. As the tree approaches being full, learning time increases from logarithmic toward linear, due to the necessity for messages to traverse the tree. One solution to minimize that degradation is to keep a count of free nodes in each subtree at each node. Nodes use their counts to direct messages across the tree to the closest free nodes.

100% efficiency in utilizing resources is not expected, only a good balance between time and resources. Additional minimizations, balanced trees and efficient ways of using limited resources increase the efficiency of the model.

CHAPTER 8
CONCLUSION

This concluding chapter to the thesis has two sections. The first section gives an overview of the thesis, and discusses what has been accomplished. The second section discusses future research.

8.1 Overview and Accomplishments

This thesis presents the architecture and learning algorithm for a new ASOCS computing model, LIA (Location-Independent ASOCS). Its main strength is that it is more easily implementable using current technology than previous ASOCS models.

A review of instances and instance sets is presented. The heart of the LIA model is the LIA node. LIA nodes store complete instances rather than pieces of instances. This feature distinguishes LIA from AA1, AA2, AA3, and gives LIA its flexibility while retaining the self-organizing features of ASOCS. Flexibility comes because the nodes are logically independent of each other; nodes may be easily rearranged in any order to increase efficiency. A corollary to that is that any convenient interconnect topology may be used for communication--a hierarchical structure is not required. The description in this thesis used a binary tree structure because it is simple, and fast. All of the differences in the architectures and learning algorithms between LIA and previous ASOCS models stem from the differences in node structure.

Finally, this research for this thesis provided impetus for the PRS ASOCS model. The PRS model features a method of conflict resolution that does not use DVA.

8.2 Future Research

The main motivation behind the LIA model is to develop an ASOCS model that is implementable using current technology. The next projected steps in LIA research are the following:

1. model LIA using current IC technology (such as VLSI)
2. build an LIA chip

APPENDIX

This appendix contains test results for LIA for the instance set given in Chapter 9 of [Mart86]. The LIA simulation's main purpose is to demonstrate the LIA algorithms presented in the thesis by duplicating the results in [Mart86]. That is, beginning with the same initial group of instances, LIA can generate the same "intermediate" and final instance sets as AA1, AA2, and AA3 (although the internal representation is quite different). In order to achieve these results, the one-difference minimizations and DVA rebroadcast background processes (described in Chapter 7) are run after each NI is learned by the current net. (An *equivalent* final instance set can be generated by learning all the instances first, then running the background minimizations last. This test is not shown in the appendix.)

The following instances are the input to the simulation (in numerical order). In this set, the maximum number of input variables is 12, so the length of the longest variable list of an instance is also 12. (The instances below are written using lowercase letters so that the lists will be easier to read.)

1. f'l'a'i'k'hcd ⇒ z
2. eb ⇒ z
3. b ⇒ z'
4. c'h'ib' ⇒ z'
5. fi'c ⇒ z
6. igac'e'hj ⇒ z
7. fk' ⇒ z
8. j'ib'a'ke ⇒ z'
9. c'i ⇒ z'
10. k'djb'fl'h'eci' ⇒ z
11. be ⇒ z'
12. k'die'lj'h'f'cba' ⇒ z
13. h' ⇒ z'
14. b'kla ⇒ z
15. cf'gk'ld'eb'ah'j'i ⇒ z'
16. cb' ⇒ z'
17. id' ⇒ z'
18. b'fgkl'c'aihjd'e' ⇒ z'
19. fbk'l'aj'c ⇒ z
20. h'j'f'l'gi ⇒ z'
21. j'f'gb'idc'k'l'a' ⇒ z'
22. ed'j'gb' ⇒ z
23. cd'e'i'j'kg'a' ⇒ z'
24. c ⇒ z'
25. hj'dgb'lae'fk ⇒ z

26. ab'ec'l' ⇒ z'
27. a'i'kg'c' ⇒ z'
28. hl'a'fk'i' ⇒ z
29. g'al ⇒ z'
30. ga'lkf' ⇒ z
31. ijk'b'eahdf'cgl ⇒ z'
32. c'ae'i'h'j'kd'lg'f'b ⇒ z'
33. jlh'f'ga'cbi ⇒ z'
34. fkla'dcgj'h'i ⇒ z'
35. d'c'ga'f'bk'l'ihj' ⇒ z
36. jh'kfl'e'dac'g'ib' ⇒ z
37. jl ⇒ z
38. b'jk'el'i ⇒ z
39. aj'kl'g'he'd'f'b ⇒ z'
40. b'a'el'i'hjcf' ⇒ z'
41. j ⇒ z'
42. j'efgkba' ⇒ z'
43. l'a'c'j'fi'bk ⇒ z
44. f'bg'j' ⇒ z
45. bd'i'ca'el'g'j ⇒ z
46. l'cajf'g'i'hed ⇒ z'
47. ic'j'leg'b'h'a ⇒ z
48. l'j'i'da'k'f'h'gb ⇒ z'
49. fika' ⇒ z'
50. k'j'i' ⇒ z'

The instance set listed below is the result after the fiftieth instance is learned. Since the instance set is consistent, there is no order nor precedence among the instances.

d'c'a'f'bk'l'ihj' ⇒ z
hdgb'lae'fkj' ⇒ z
b'klac'i'gj' ⇒ z
b'klac'dgj' ⇒ z
ga'lkf'j' ⇒ z
l'a'c'j'fi'bk ⇒ z
bd'i'ca'el'g'j ⇒ z
ic'j'leg'b'h'a ⇒ z
f'bg'j'k ⇒ z
f'bg'j'i ⇒ z

ed'j'gb'c'a'f'k $\Rightarrow$ z
ed'j'gb'c'a'f'i $\Rightarrow$ z
ed'j'gb'c'a'i'k $\Rightarrow$ z
ed'j'gb'c'a'k'i $\Rightarrow$ z
ed'gb'c'lj'f'k $\Rightarrow$ z
ed'gb'c'lj'f'i $\Rightarrow$ z
ed'gb'c'lj'i'k $\Rightarrow$ z
ed'gb'c'lj'k'i $\Rightarrow$ z
egb'c'lj'ak $\Rightarrow$ z
ed'gb'c'lj'ai $\Rightarrow$ z
il'e'b' $\Rightarrow$ z'
ia'e'f $\Rightarrow$ z'
ia'df $\Rightarrow$ z'
ie'fl' $\Rightarrow$ z'
belf $\Rightarrow$ z'
cd'lf $\Rightarrow$ z'
cd'if $\Rightarrow$ z'
cblf $\Rightarrow$ z'
ca'lf $\Rightarrow$ z'
ca'if $\Rightarrow$ z'
celf $\Rightarrow$ z'
ceif $\Rightarrow$ z'
ibf $\Rightarrow$ z'
h'l'e'b' $\Rightarrow$ z'
a'kg'b' $\Rightarrow$ z'
bkfl $\Rightarrow$ z'
h'a'e'fl $\Rightarrow$ z'
h'a'e'fb' $\Rightarrow$ z'
h'a'dfl $\Rightarrow$ z'
h'a'dfb' $\Rightarrow$ z'
h'l'db' $\Rightarrow$ z'
il'df $\Rightarrow$ z'
il'db' $\Rightarrow$ z'
il'dg $\Rightarrow$ z'
il'e'kg $\Rightarrow$ z'
ik'dg $\Rightarrow$ z'
h'k'e'f $\Rightarrow$ z'
h'k'e'b' $\Rightarrow$ z'

h'k'e'g $\Rightarrow$ z'

h'k'dg $\Rightarrow$ z'

cd'af $\Rightarrow$ z'

cd'ab' $\Rightarrow$ z'

cd'ag $\Rightarrow$ z'

cg'lb' $\Rightarrow$ z'

cg'af $\Rightarrow$ z'

cg'ab' $\Rightarrow$ z'

cbaf $\Rightarrow$ z'

cbag $\Rightarrow$ z'

ck'lf $\Rightarrow$ z'

ck'lb' $\Rightarrow$ z'

ck'lg $\Rightarrow$ z'

ck'af $\Rightarrow$ z'

ck'ab' $\Rightarrow$ z'

ck'ag $\Rightarrow$ z'

bf'ag $\Rightarrow$ z'

h'bk'g $\Rightarrow$ z'

id'e'f $\Rightarrow$ z'

id'e'ab' $\Rightarrow$ z'

id'e'ag $\Rightarrow$ z'

ch'ab' $\Rightarrow$ z'

ch'ag $\Rightarrow$ z'

ch'k'b' $\Rightarrow$ z'

ch'k'g $\Rightarrow$ z'

cf'g'b' $\Rightarrow$ z'

cf'ab' $\Rightarrow$ z'

cf'ag $\Rightarrow$ z'

cf'k'b' $\Rightarrow$ z'

cf'k'g $\Rightarrow$ z'

beaf $\Rightarrow$ z'

beag $\Rightarrow$ z'

belk'g $\Rightarrow$ z'

ckl'b' $\Rightarrow$ z'

ckl'g $\Rightarrow$ z'

ceab' $\Rightarrow$ z'

ceag $\Rightarrow$ z'

cik'b' $\Rightarrow$ z'

$cik'g \Rightarrow z'$

$ik'e'ag \Rightarrow z'$

$ik'e'b' \Rightarrow z'$

$ik'e'lg \Rightarrow z'$

$bf'k'dg \Rightarrow z'$

$bf'dl'g \Rightarrow z'$

$bf'k'lg \Rightarrow z'$

$bf'k'i'g \Rightarrow z'$

$bf'i'l'g \Rightarrow z'$

$ibag \Rightarrow z'$

$ibk'lg \Rightarrow z'$

$ig'cb' \Rightarrow z'$

$ael'f \Rightarrow z'$

$ael'b' \Rightarrow z'$

$ael'g \Rightarrow z'$

$cl'ab' \Rightarrow z'$

$cl'ag \Rightarrow z'$

$ch'l'b' \Rightarrow z'$

$ch'l'g \Rightarrow z'$

$cf'l'b' \Rightarrow z'$

$cf'l'g \Rightarrow z'$

$cil'b' \Rightarrow z'$

$cil'g \Rightarrow z'$

$ial'f \Rightarrow z'$

$ial'b' \Rightarrow z'$

$ial'g \Rightarrow z'$

$bkl'f'g \Rightarrow z'$

$bkl'ig \Rightarrow z'$

$h'l'af \Rightarrow z'$

$h'l'ab' \Rightarrow z'$

$h'l'ag \Rightarrow z'$

$h'l'e'f'g \Rightarrow z'$

$h'l'e'ig \Rightarrow z'$

$bkaf \Rightarrow z'$

$bkag \Rightarrow z'$

$h'bl'f'g \Rightarrow z'$

$h'bl'ig \Rightarrow z'$

$h'bag \Rightarrow z'$

h'l'df'g $\Rightarrow$ z'

ch'fb' $\Rightarrow$ z'

ch'fd $\Rightarrow$ z'

ch'fi $\Rightarrow$ z'

ch'fe' $\Rightarrow$ z'

ch'fl $\Rightarrow$ z'

ch'fg $\Rightarrow$ z'

ch'fj' $\Rightarrow$ z'

cd'kfb' $\Rightarrow$ z'

cd'kfe' $\Rightarrow$ z'

cd'kfg $\Rightarrow$ z'

cd'kfj' $\Rightarrow$ z'

ca'kfb' $\Rightarrow$ z'

ca'kfd $\Rightarrow$ z'

ca'kfe' $\Rightarrow$ z'

ca'kfg $\Rightarrow$ z'

ca'kfj' $\Rightarrow$ z'

cekfb' $\Rightarrow$ z'

cekfd $\Rightarrow$ z'

cefa $\Rightarrow$ z'

cekfg $\Rightarrow$ z'

cekfj' $\Rightarrow$ z'

jb' $\Rightarrow$ z'

jd $\Rightarrow$ z'

ji $\Rightarrow$ z'

jc' $\Rightarrow$ z'

ja $\Rightarrow$ z'

je' $\Rightarrow$ z'

jl $\Rightarrow$ z'

jg $\Rightarrow$ z'

bkfcd $\Rightarrow$ z'

bkfce' $\Rightarrow$ z'

bkfcg $\Rightarrow$ z'

bkfcj' $\Rightarrow$ z'

h'bfl $\Rightarrow$ z'

ckl'fd $\Rightarrow$ z'

cl'fi $\Rightarrow$ z'

cl'fa $\Rightarrow$ z'

ckl'fe' ⇒ z'

ckl'fj' ⇒ z'

kg'cb' ⇒ z'

kg'cfd ⇒ z'

g'cfi ⇒ z'

kg'cfe' ⇒ z'

kg'cfj' ⇒ z'

h'g'b'i' ⇒ z'

h'g'b'c ⇒ z'

h'g'b'l' ⇒ z'

h'g'b'e' ⇒ z'

h'g'b'a' ⇒ z'

ik'fe' ⇒ z'

ik'dfh ⇒ z'

ik'db'h ⇒ z'

ik'db'a' ⇒ z'

ik'fc ⇒ z'

ik'g'fh ⇒ z'

ik'g'b'h ⇒ z'

ig'b'a' ⇒ z'

h'k'dfi' ⇒ z'

h'k'dfl' ⇒ z'

h'k'dfb ⇒ z'

h'k'dfa' ⇒ z'

h'k'db'i' ⇒ z'

h'k'db'a' ⇒ z'

ig'fl' ⇒ z'

id'g'fh ⇒ z'

ig'fa' ⇒ z'

ig'b'l' ⇒ z'

id'g'b'e' ⇒ z'

id'g'b'h ⇒ z'

g'alfi' ⇒ z'

g'alfe' ⇒ z'

g'alfb ⇒ z'

g'alfh ⇒ z'

g'alb'i' ⇒ z'

g'alb'e' ⇒ z'

g'alb'h $\Rightarrow$ z'

kg'lfi' $\Rightarrow$ z'

g'lfc $\Rightarrow$ z'

kg'lfe' $\Rightarrow$ z'

kg'lfh $\Rightarrow$ z'

kg'lfa' $\Rightarrow$ z'

kg'lb'i' $\Rightarrow$ z'

kg'lb'e' $\Rightarrow$ z'

kg'lb'h $\Rightarrow$ z'

h'g'afi' $\Rightarrow$ z'

h'afc $\Rightarrow$ z'

h'g'afe' $\Rightarrow$ z'

h'afb $\Rightarrow$ z'

h'g'ife' $\Rightarrow$ z'

h'g'k'fc'i' $\Rightarrow$ z'

h'g'k'fc'l' $\Rightarrow$ z'

h'k'fc'b $\Rightarrow$ z'

h'g'k'fc'a' $\Rightarrow$ z'

h'g'fli' $\Rightarrow$ z'

h'g'fle' $\Rightarrow$ z'

h'g'fla' $\Rightarrow$ z'

h'g'k'fj'l' $\Rightarrow$ z'

h'k'fj'b $\Rightarrow$ z'

h'g'k'fj'a' $\Rightarrow$ z'

fika' $\Rightarrow$ z'

k'j'i' $\Rightarrow$ z'

Instances	# of Nodes	Depth	Max Width
1	1	1	8
2	2	2	8
3	2	2	9
4	3	2	9
5	6	3	8
6	23	5	8
7	25	5	8
8	26	5	8
9	15	4	8
10	15	4	8
11	18	5	8
12	41	6	10
13	19	5	8
14	26	5	8
15	26	5	8
16	25	5	6
17	27	5	7
18	27	5	7
19	51	6	7
20	51	6	7
21	51	6	7
22	98	7	7
23	102	7	7
24	38	6	6
25	48	6	10
26	51	6	10
27	52	6	10
28	73	7	10
29	76	7	10
30	102	7	10
31	102	7	10
32	102	7	10
33	107	7	10
34	107	7	10
35	118	7	11
36	170	8	12
37	175	8	11
38	201	8	11
39	201	8	11
40	201	8	11
41	100	8	11
42	100	7	11
43	113	7	11
44	160	7	10
45	194	8	10
46	194	8	10
47	222	8	10
48	222	8	10
49	228	8	10
50	226	8	10

Table 3. LIA Statistics

The number of instances stored in the net is always the same as the number of nodes. "Depth" is always $\log_2$(number of nodes). "Max Width" is the length of the longest variable list among the nodes in the net.

"Depth" and "Max Width" are provided only for convenience. Depth can always be calculated from the number of nodes. Since the maximum number of input variables is 12, as stated previously, the length of any node's variable list cannot exceed 12 (but may be less).

Selected Bibliography

[Chang87]. Chang, J., and J. J. Vidal. "Inferencing in Hardware." *Proceedings of the MCC University Research Symposium*, Austin TX. (July 1987).

[Feng81]. Feng, Tseyun. "A Survey of Interconnection Networks." *Computer, Vol.14, #12.* pp. 12-27. (December 1981).

[Hayn82]. Haynes, L.S., R.L. Lau, D.P. Siewiorek, D.W. Mizell. "A Survey of Highly Parallel Computing." *Computer, Vol. 15, #1.* pp. 9-24. (January 1982).

[Helly84]. Helly, J.J., W.V. Bates, M. Culter, and S. Kelem. "A Representational Basis for the Development of a Distributed Expert System for Space Shuttle Flight Control." *NASA Technical Memorandum 58258.* (May 1984).

[Hill85]. Hillis, W.D. *The Connection Machine.* Cambridge, Mass.: MIT Press, 1985.

[Hill86]. Hillis, W.D., Guy L. Steele, Jr. "Data Parallel Algorithms. "*Communications of the ACM, Vol. 29, #12.* pp. 1170-1183. (December 1986).

[Hugh89]. Hughes, Brent. *Prioritized Rule Systems.* BYU, Master's Thesis. (1989).

[Hwang87a]. Hwang, Kai. "Advanced Parallel Processing With Supercomputer Architectures." *Proceedings of the IEEE, Vol. 75, #10.* pp. 1348-1379. (1987).

[Hwang87b]. Hwang, K., J. Ghosh, R. Chowkwanyun. "Computer Architectures for Artificial Intelligence Processing." *Computer, Vol. 20*, #1. pp. 19-27. (January 1987).

[Lipp87]. Lippman, R.P. "An Introduction to Computing with Neural Nets." IEEE ASSP Magazine, Vol. 3, #4, pp. 4-22. (April 1987).

[Mart86]. Martinez, T.R. *Adaptive Self-Organizing Logic Networks.* UCLA, Ph.D. Dissertation. (May 1986).

[Mart87]. Martinez, T.R. "Models of Parallel Adaptive Logic." *Proceedings of the 1987 IEEE Systems Man and Cybernetics Conference.* pp.290-296. (October 1987).

[Mart88a]. Martinez, T.R., J.J. Vidal. "Adaptive Parallel Logic Networks. "*Journal of Parallel and Distributed Computing, Vol. 5, #1.* pp. 26-58. (1988).

[Mart88b]. Martinez, T.R. "Digital Neural Networks." *Proceedings of the 1988 IEEE Systems Man and Cybernetics Conference.* pp. 681-684. (August 1988).

[Mart89]. Martinez, T.R. "Neural Network Applicability: Classifying the Problem Space." *Proceedings of the IASTED International Symposium on Expert Systems and Neural Networks.* pp. 41 -44. (August 1989).

[Mart90a]. Martinez, T.R. "Adaptive Self-Organizing Concurrent Systems." *Progress in Neural Networks.* pp.105-126. Ablex Publishing, 1990.

[Mart90b]. Martinez, T.R., D.M. Campbell. *A Self-Organizing Binary Decision Tree for Incrementally Defined Rule-Based Systems.* In preparation. (1990).

[Mart90c]. Martinez, T.R., D.M. Campbell. "A Self-Adjusting Dynamic Logic Module." To appear in the *Journal of Parallel and Distributed Processing*. (1991).

[Mart90d]. Martinez, T.R. "Consistency and Generalization of Incrementally Trained Connectionist Models." *Proceedings of the International Symposium on Circuits and Systems.* pp. 706-709. (1990).

[McDon90]. McDonald, K. C., T.R. Martinez, D.M. Campbell. "Application of Adaptive Learning and Parallelism to Computer Network Routing". Submitted. (1990).

[Rudo89]. Rudolph, G., T.R. Martinez. “DNA: Towards an Implementation of ASOCS.” *Proceedings of the IASTED International Symposium on Expert Systems and Neural Networks.* pp. 12-15. (August, 1989).

[Rum86]. Rumelhart, D., J. McClelland, et. al. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1.* Cambridge, Mass.:MIT Press, 1986.

[Walt87]. Waltz, D.L. "Applications of the Connection Machine." *Computer, Vol. 20, #1.* pp. 85-96. (January 1987).

A Location-Independent ASOCS Model

George Rudolph

Department of Computer Science

M. S. Degree, June 1991

ABSTRACT

This thesis proposes a new Adaptive Self-Organizing Concurrent System (ASOCS) model [Mart86] called Location-Independent ASOCS (LIA). An architectural model and execution and learning algorithms for LIA are described. Theoretical background, an analysis of LIA, and a brief comparison with other ASOCS models are also presented.

The LIA architecture is based on a computing node that localizes more information at each node than previous ASOCS models' nodes. This localization allows information to be stored independent of its location in a network, and allows information to be easily shuffled between nodes. Whereas previous ASOCS are more naturally implementable using dynamic interconnects between nodes, LIA can use a static interconnect. Thus, the location-independent characteristics of LIA give it improved potential for implementation using current technology over previous ASOCS models. This thesis does not discuss details of a hardware implementation of LIA beyond the model level.

Committee Approval:

Tony R. Martinez, Committee Chairman

Dan R. Olsen, Committee Member

Larry C. Christensen, Graduate Coordinator